Struktura organizacyjna i architektura systemów komputerowych

Autorzy: Linda Null, Julia Lobur
Tłumaczenie: Michał Dadan (wstęp, rozdz. 1 –; 5), Piotr Pilch (rozdz. 6 –; 10), Andrzej Grażyński (rozdz. 11, dod. A –; C)
Tytuł oryginału: The Essentials of Computer Organization and Architecture
Format: B5, stron: 672

Komputery już dawno stały się zjawiskiem powszechnym i nie są już traktowane jak magiczne skrzynki. Praktycznie wszyscy znają już możliwości ich praktycznego wykorzystania. W dobie intuicyjnych systemów operacyjnych, technologii plug-and-play i postępującego uproszczenia wszelkich operacji związanych z komputerami wiedza o architekturze i organizacji systemów komputerowych może wydawać się potrzebna jedynie wąskiej grupie specjalistów-sprzętowców. Jest jednak inaczej. Ogólna znajomość tego, co kryje się pod „maską" komputera potrzebna jest każdemu użytkownikowi komputera. Dzięki niej programista zrozumie, z czego wynikają błędy w działaniu programu, twórca systemów czasu rzeczywistego zoptymalizuje wykorzystanie procesora przez system, a osoba decydująca się na zakup nowego sprzętu we właściwy sposób zinterpretuje „obiektywne" testy przytaczane przez producentów w materiałach reklamowych.

Struktura organizacyjna i architektura systemów komputerowych to przystępne omówienie organizacji i architektury współczesnych komputerów. Książka, stworzona zgodnie z założeniami komitetu ACM-IEEE Computing Curricula 2001, nadaje się idealnie jako podręcznik dla kursu wprowadzającego w tą tematykę. Zawarte w niej zagadnienia zilustrowane są licznymi przykładami zaczerpniętymi z rzeczywistego świata, co dodatkowo ułatwia ich zrozumienie.

- Historia rozwoju komputerów.
- Sposoby przedstawiania danych, notacje i kody.
- Logika binarna i algebra Boole'a.
- Organizacja pracy systemu komputerowego, cykle maszynowe, magistrala, lista rozkazów, asembler.
- Tryby adresowania.
- Przechowywanie danych w pamięci komputera.
- Narzędzia programistyczne i systemy operacyjne.
- Alternatywne architektury komputerów.
- Analiza wydajności systemów komputerowych.
- Sieci komputerowe.

Doskonale dobrane proporcje pomiędzy objętością tekstu i poziomem szczegółowości oraz opisywanie wyłącznie istotnych aspektów zagadnienia powodują, że książka stanowi doskonałe źródło naprawdę przydatnej wiedzy.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

5.

Bliższe spojrzenie na architektury zbioru rozkazów 205

6.

Pamięć 237

7.

Systemy wejścia-wyjścia i magazynowania danych 277

8.

Oprogramowanie systemowe 357

A

B

C

4

MARIE: wprowadzenie do budowy prostego komputera

„Jeżeli do uzyskania wyniku musisz posłużyć się jakimś instrumentem, budując go, nie pozwól sobie na to, aby był on zbyt skomplikowany”.

— Leonardo da Vinci

4.1. Wstęp

W dzisiejszych czasach zaprojektowanie komputera to zadanie dla dobrze wyszkolonego informatyka. Niemożliwością jest, aby w książce takiej jak ta (ani w podstawowym kursie o architekturze komputera) przedstawić wszystko, co jest niezbędne do zbudowania współczesnego komputera. Tym niemniej w niniejszym rozdziale przyjrzymy się bardzo prostemu komputerowi, jakim jest MARIE (ang. *A Machine Architecture that is Really Intuitive and Easy*), co w języku polskim oznacza naprawdę prostą i intuicyjną architekturę komputera. Następnie krótko omówimy komputery Intel i MIPS — dwie popularne architektury, będące odzwierciedleniem filozofii projektowania CISC oraz RISC. Celem niniejszego rozdziału jest wyjaśnienie, jak naprawdę działa komputer. Dlatego też starałyśmy się, zgodnie z zacytowaną radą Leonarda da Vinci, przedstawić budowę komputera w możliwie najmniej skomplikowany sposób.

4.1.1. Podstawy budowy i organizacji procesora

Z rozdziału 2. („Reprezentacja danych w systemach komputerowych”) wiemy już, że komputer musi operować na danych zapisanych w systemie binarnym. Z rozdziału 3. z kolei dowiedzieliśmy się, że pamięć wykorzystywana jest do przechowywania zarówno danych, jak i poleceń programów (również zapisanych w systemie binarnym). Programy muszą być w jakiś sposób wykonywane, a dane przetwarzane. Jest to zadanie *procesora* (ang. *central processing unit*, *CPU*), który jest odpowiedzialny za wczytywanie poleceń programu, rozkodowanie każdego z nich oraz wykonanie działań we wskazanej kolejności. Aby zrozumieć, jak pracują komputery, należy najpierw zaznajomić się z ich różnorodnymi podsystemami i interakcjami pomiędzy nimi. Zanim przedstawimy prostą architekturę komputera, przyjrzymy się najpierw mikroarchitekturze występującej na poziomie sterowania współczesnych komputerów.

We wszystkich komputerach znajduje się procesor. Może on być podzielony na dwie części. Pierwszą z nich jest *ścieżka danych*, czyli sieć jednostek przechowujących dane (rejestrów) oraz jednostek logiczno-arytmetycznych (do wykonywania różnych operacji na danych), połączonych ze sobą magistralami (zdolnymi do przenoszenia danych z miejsca na miejsce), których pracą sterują zegary. Drugim składnikiem procesora jest *jednostka sterująca* — moduł odpowiedzialny za ustalanie kolejności wykonywanych operacji i gwarantujący, że dane znajdą się we właściwym miejscu o właściwym czasie. Łącznie oba te składniki wykonują zadania powierzone procesorowi: wczytywanie instrukcji, rozkodowywanie ich i wykonywanie ich we wskazanej kolejności. Działanie komputera zależy bezpośrednio od sposobu zaprojektowania ścieżki danych i jednostki sterującej. Dlatego poniżej zostaną one szczegółowo omówione.

Rejestry

Rejestry są wykorzystywane w systemach komputerowych w charakterze miejsc, w których przechowywane są różnego rodzaju dane, takie jak adresy, liczniki rozkazów lub dane niezbędne do wykonania określonego programu. Mówiąc prościej, *rejestr* jest fizycznym urządzeniem, które przechowuje dane binarne. Aby dostęp do nich był możliwie szybki, rejestry umiejscowione są w procesorze. W rozdziale 3. dowiedziałeś się, że do zbudowania rejestrów można użyć przerzutników D. Jeden przerzutnik D odpowiada rejestrowi przechowującemu 1 bit. Zatem do przechowywania wartości wielobitowych niezbędny jest zespół przerzutników D. Na przykład aby zbudować rejestr szesnastobitowy, musielibyśmy połączyć ze sobą 16 przerzutników D. Na rysunku licznika binarnego, który zamieściłyśmy w rozdziale 3., można było zauważyć, że wspomniane przerzutniki muszą być zsynchronizowane tak, aby działały jednocześnie. Z każdym impulsem zegara dane wchodzą do rejestru i nie mogą być zmienione, dopóki nie nastąpi kolejny impuls (są tam więc przechowywane).

Przetwarzanie danych na komputerze jest zazwyczaj wykonywane na słowach binarnych o określonej wielkości, które są przechowywane w rejestrach. Dlatego większość komputerów ma rejestry o określonej pojemności. Popularne rozmiary rejestrów to 16, 32 i 64 bity. Liczba rejestrów w komputerze różni się w zależności od architektury, ale zazwyczaj jest potęgą liczby dwa, a najpopularniejsze z nich to 16 i 32. Rejestry zawierają dane, adresy lub informacje sterujące. Niektóre z nich są wyspecjalizowane, a więc mogą zawierać tylko dane, tylko adresy lub tylko informacje sterujące. Inne są bardziej ogólne i mogą przechowywać w różnych momentach zarówno dane, adresy, jak i informacje sterujące.

Dane są w rejestrach zapisywane, z nich odczytywane i przenoszone z rejestru do rejestru. Do rejestrów nie odwołujemy się tak samo, jak do pamięci (przypominamy, że każde słowo w pamięci ma swój niepowtarzalny adres, liczony od zera). To sama jednostka sterująca odwołuje się do nich i nimi manipuluje.

We współczesnych systemach komputerowych istnieje wiele rodzajów wyspecjalizowanych rejestrów: rejestry do przechowywania informacji, rejestry do zmieniania wartości, rejestry do porównywania wartości oraz rejestry do liczenia. Są też rejestry zwane „brudnopisami", które przechowują tymczasowe wartości, rejestry indeksowe do kontrolowania pętli, rejestry do zarządzania stosami informacji dla procesów, rejestry stanu do przechowywania informacji o stanie lub trybie pracy (i występowaniu takich zdarzeń, jak przepełnienie, przeniesienie czy osiągnięcie warunków zerowych) oraz rejestry ogólnego przeznaczenia, czyli rejestry dostępne dla programisty. W większości komputerów znajdują się zespoły rejestrów, przy czym każdy z tych zespołów jest zespołem wyspecjalizowanym. Na przykład w architekturze Pentium istnieje zespół rejestrów danych i zespół rejestrów adresowych. Niektóre architektury mają bardzo duże zespoły rejestrów, które mogą być w nowoczesny sposób wykorzystywane do przyspieszenia procesu wykonywania operacji. (Temat ten omówimy po przedstawieniu zaawansowanych architektur w rozdziale 9.).

ALU

Jednostka arytmetyczno-logiczna (*ALU*) wykonuje operacje logiczne (takie, jak na przykład porównanie) i arytmetyczne (na przykład dodawanie i mnożenie) wymagane podczas wykonywania programu. Przykład prostej jednostki ALU widzieliśmy już w rozdziale 3. Ogólnie mówiąc, jednostka ALU ma dwa wejścia danych i jedno wyjście. Operacje wykonywane w ALU mają często wpływ na bity w *rejestrze stanu* (zadaniem tych bitów jest informowanie o wystąpieniu określonych sytuacji, takich jak na przykład przepełnienie). Jednostka ALU „wie", które operacje wykonywać, ponieważ jest kontrolowana przez sygnały płynące z jednostki sterującej.

Jednostka sterująca

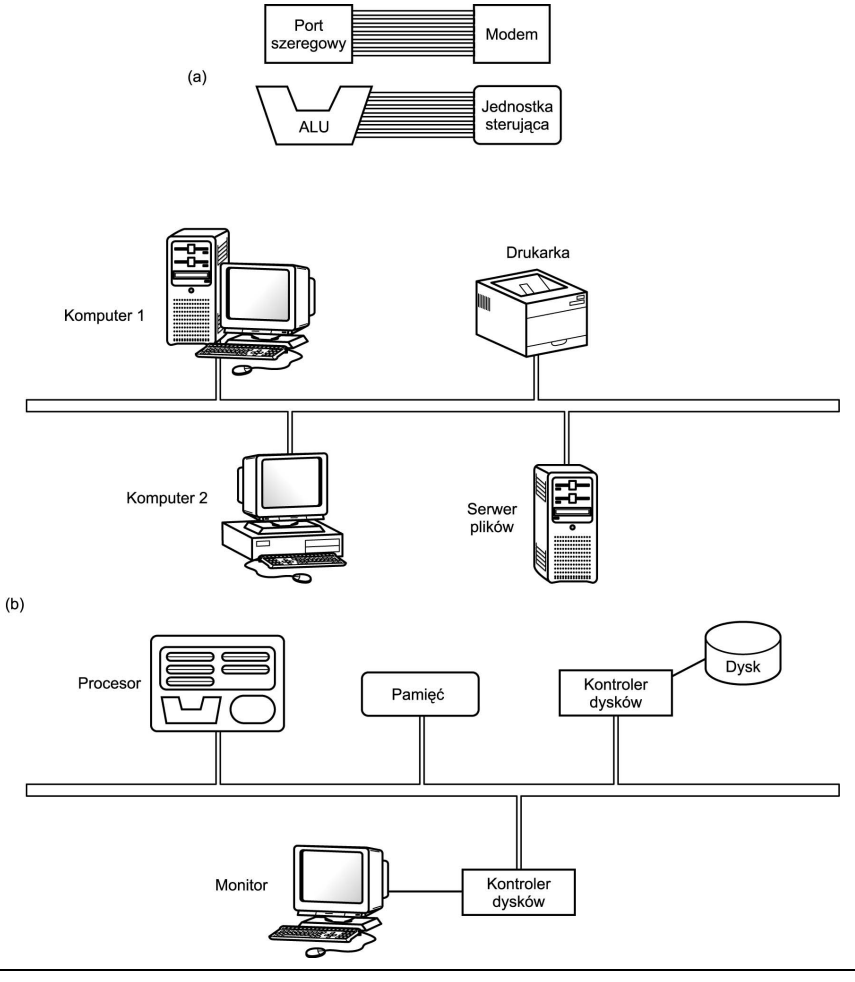
Jednostka sterująca jest „policjantem" czy też „kierującym ruchem" procesora. Monitoruje ona wykonywanie wszystkich instrukcji oraz transfer wszystkich informacji. Wydobywa instrukcje z pamięci, rozkodowuje je i dba o to, aby znalazły się w odpowiednim czasie we właściwym miejscu. Informuje też jednostkę ALU, których rejestrów ma ona użyć, obsługuje przerwania oraz — w celu wykonania pożądanej operacji — włącza odpowiedni zestaw obwodów elektrycznych w ALU. Aby odnaleźć następną instrukcję, jaką należy wykonać, jednostka sterująca wykorzystuje rejestr *licznika rozkazów*, natomiast aby „wiedzieć" na bieżąco o przepełnieniach, przesunięciach, „zapożyczeniach", itp. — używa rejestru stanu. Więcej szczegółowych informacji o jednostce sterującej prezentujemy w podrozdziale 4.7.

4.1.2. Magistrala

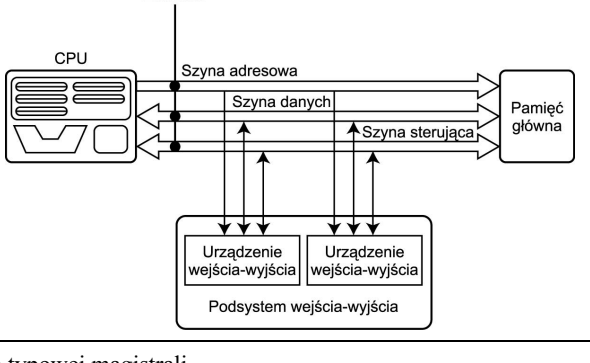
Procesor komunikuje się z innymi podsystemami poprzez tzw. szynę (zwaną też magistralą). *Magistrala* jest zespołem przewodów działających jako wspólna ścieżka danych, łącząca różnorodne podsystemy. Składa się z wielu linii, co pozwala na równoległy przepływ bitów. Szyny są niedrogim, ale też wszechstronnym sposobem na łatwe włączenie nowych urządzeń do systemu i połączenie ich ze sobą. W danej chwili czasu szyna może być wykorzystywana tylko przez jedno urządzenie, niezależnie od tego, czy jest to rejestr, jednostka ALU, pamięć czy cokolwiek innego. Pomimo to współużytkowanie szyny często staje się „wąskim gardłem" w komunikacji. Na prędkość szyny wpływa zarówno jej szerokość, jak i liczba urządzeń, przez które jest współużytkowana. Dość często urządzenia te podzielone są na dwie kategorie: urządzenia *master* i urządzenia *slave*, przy czym urządzenie *master* to takie, które inicjuje działanie, natomiast urządzenie *slave* odpowiada na żądania urządzenia *master*.

Magistrala może być dwupunktowa, co oznacza, że łączy ona dwa konkretne podzespoły (tak, jak to zostało przedstawione na rysunku 4.1a), lub też może być *wspólną ścieżką* danych, która łączy kilka urządzeń, zmuszając je do jej współużytkowania (nazywamy ją magistralą *wielopunktową*, patrz rysunek 4.1b).

W związku ze wspomnianym współużytkowaniem niezwykle ważny jest *protokół magistrali* (zestaw reguł użytkowania). Rysunek 4.2 pokazuje typową magistralę, składającą się z linii danych, linii adresowych, linii sterujących i linii zasilania. Często linie magistrali odpowiedzialne za przenoszenie danych nazywane są *szyną danych*. Linie te zawierają faktyczne informacje, które muszą zostać przeniesione z jednego miejsca do innego. *Linie sterujące* wskazują, które z urządzeń ma pozwolenie na użycie szyny i w jakim celu (np. zapis lub odczyt danych z pamięci bądź z urządzenia wejścia-wyjścia). Linie sterujące przesyłają też komunikaty potwierdzające odbieranie żądań dostępu do magistrali, żądań obsługi przerwań i impulsów synchronizacyjnych zegara). *Linie*



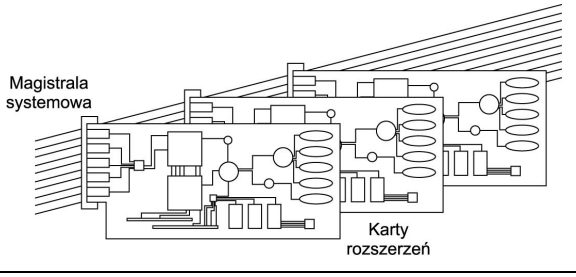
RYSUNEK 4.1. a) szyny dwupunktowe; b) szyna wielopunktowa



RYSUNEK 4.2. Budowa typowej magistrali

adresowe wskazują miejsce (np. w pamięci), w którym dane powinny zostać zapisane lub z którego należy je odczytać. *Linie zasilania* zapewniają niezbędne zasilanie elektryczne. Typowe operacje w przypadku magistrali to wysyłanie adresu (do odczytu lub zapisu), transfer danych z pamięci do rejestru (odczytanie pamięci) oraz transfer danych z rejestru do pamięci (zapis w pamięci). Dodatkowo magistrale wykorzystywane są do operacji zapisu i odczytu danych z urządzeń peryferyjnych. Każdy rodzaj transferu ma miejsce w cyklu magistrali, to znaczy w okresie czasu pomiędzy dwoma impulsami zegara szyny.

Ze względu na różne rodzaje szyn transportujących informacje oraz ze względu na różnorodność urządzeń, które te szyny wykorzystują, również magistrale możemy podzielić na kilka rodzajów. Magistrale *procesor-pamięć* są krótkimi, bardzo szybkimi szynami, ściśle dopasowanymi do zastosowanego w komputerze systemu pamięci. Stosowane są w celu maksymalnego powiększenia przepustowości (szybkości przesyłu danych) i zazwyczaj mają dosyć specyficzną budowę. *Magistrale wejścia-wyjścia* są zazwyczaj dłuższe niż poprzednie i łączą wiele typów urządzeń o różnej przepustowości. Budowa tych szyn jest kompatybilna z różnymi architekturami komputerów. *Magistrala systemowa* (rysunek 4.3) jest właściwie wbudowana w płytę główną komputera. Łączy ona procesor, urządzenia wejścia-wyjścia oraz pamięć (tak więc wszystkie te urządzenia współdzielą jedną szynę). Wiele komputerów ma pewną hierarchię magistral, stąd też występowanie dwóch magistral (na przykład szyny procesor-pamięć i szyny wejścia-wyjścia) lub ich większej liczby wewnątrz jednego systemu nie jest niczym nadzwyczajnym. Systemy o dużej wydajności często używają wszystkich trzech rodzajów magistral.



RYSUNEK 4.3. Magistrala systemowa

W przypadku komputerów PC stosujemy nieco odmienną terminologię. Komputery te mają wewnętrzną szynę zwaną *magistralą systemową*, która łączy procesor, pamięć i wszystkie inne wewnętrzne podsystemy. Szyny zewnętrzne (określane często mianem *szyn rozszerzeń*) łączą urządzenia zewnętrzne, peryferyjne, gniazda rozszerzeń oraz porty wejścia-wyjścia z resztą komputera. W większości komputerów PC znajdują się również *szyny lokalne*, czyli szyny danych, które łączą urządzenie peryferyjne bezpośrednio z procesorem. Są to bardzo szybkie magistrale i mogą być wykorzystane do połączenia tylko ograniczonej liczby podobnych urządzeń. Magistrale rozszerzeń są wolniejsze, ale umożliwiają bardziej uniwersalne połączenia. Rozdział 7. bardzo szczegółowo opisuje te zagadnienia.

Magistrale to w rzeczywistości niewiele więcej aniżeli wiązki przewodów, których działanie usankcjonowane jest pewnymi standardami odnośnie połączeń, taktowania, przesyłania impulsów i wykorzystywanego protokołu przesyłu danych. Szyny *synchroniczne* są zsynchronizowane w taki sposób, że transakcje zachodzą w nich tylko w czasie trwania impulsu (taktu) zegara (kolejność zdarzeń kontrolowana jest przez zegar). Każde urządzenie zsynchronizowane jest z częstotliwością, z jaką zegar wysyła impuls. Wspomniany wyżej czas trwania impulsu zegara stanowi odwrotność jego *częstotliwości*. Na przykład jeżeli częstotliwość taktowania zegara magistrali wynosi 133 MHz,

to czas trwania jednego taktu wynosi 1/133 000 000, czyli 7,52 ns. Ponieważ zegar steruje transakcjami, jakakolwiek *rozbieżność zegarów* (odchylenie) może potencjalnie spowodować problemy, co oznacza że szyna musi być możliwie najkrótsza, aby odchylenie zegara nie mogło być zbyt duże. Ponadto czas trwania cyklu magistrali nie może być krótszy niż okres czasu potrzebny do przetransportowania danych przez szynę. A zatem długość magistrali stawia pewne ograniczenia zarówno dla częstotliwości taktowania magistrali, jak i dla czasu trwania cyklu magistrali.

W przypadku magistral *asynchronicznych* to linie sterujące koordynują operacje. Jednocześnie w celu zapewnienia właściwego odliczania czasu zastosowany musi być tu skomplikowany *protokół uzgadniania*. Na przykład aby możliwe było odczytanie słowa danych z pamięci, w protokole tym konieczne byłoby wykonanie mniej więcej następujących kroków:

(1) `ReqREAD`: ta linia sterująca magistrali jest uaktywniana i w tym samym czasie na odpowiedniej linii szyny umieszczony jest adres danych zapisanych w pamięci.
(2) `ReadyDATA`: ta linia sterująca magistrali jest aktywowana, gdy podsystem pamięci przygotuje dane dla magistrali i umieści je na linii danych.
(3) `ACK`: ta linia sterująca magistrali używana jest w celu potwierdzenia wykonania operacji `ReqREAD` lub `ReadyDATA`.

Dzięki sterowaniu transakcjami za pośrednictwem specjalnego protokołu (a nie zegara) nowoczesne magistrale coraz lepiej dzielą swój czas między korzystające z nich urządzenia i mogą współpracować z większą ich liczbą.

Aby wykorzystać magistralę, urządzenie musi sobie ją zarezerwować, ponieważ tylko jedno urządzenie może używać szyny w tym samym momencie. Jak już zostało to wspomniane, urządzenia pracujące w trybie bus master mogą same zainicjować transfer informacji (kontrolować magistralę), podczas gdy urządzenia typu slave są aktywowane przez nadrzędne wobec nich urządzenia master i odpowiadają jedynie na żądania odczytu lub zapisu danych (tak więc tylko urządzenia master mogą zarezerwować magistralę). Obydwa rodzaje urządzeń działają w zgodzie z protokołem komunikacyjnym i muszą wywiązywać się ze swych zadań w bardzo ściśle określonych przedziałach czasowych. W bardzo prostym systemie (takim, jaki prezentujemy w następnym podrozdziale) jedynym urządzeniem mogącym pracować w trybie master jest procesor. Pozwala to uniknąć chaosu na magistrali, ale zauważmy, że przy takim rozwiązaniu procesor jest włączony w przebieg każdej operacji wykorzystującej szynę.

W systemach z więcej niż jednym urządzeniem master wymagany jest arbitraż magistrali. Mechanizmy arbitrażu magistrali muszą przyznawać priorytet określonym urządzeniom master i jednocześnie gwarantować, że urządzenia o niższym priorytecie nie zostaną całkowicie zablokowane. Mechanizmy *arbitrażu magistrali* możemy podzielić na cztery kategorie:

(1) **Arbitraż szeregowy:** metoda ta wykorzystuje linię sterującą „przydziel magistralę", która jest „przekazywana" od urządzenia o najwyższym priorytecie do tego o najniższym. Nie ma tu jednak sprawiedliwości i możliwa jest sytuacja, w której urządzenia o niższym priorytecie zostaną zablokowane i magistrala nigdy nie zostanie im udostępniona. Metoda ta jest więc prosta, ale niesprawiedliwa.
(2) **Scentralizowany arbitraż równoległy:** każde urządzenie ma linię sterującą żądającą dostępu do magistrali, a centralny arbiter wybiera, które urządzenie go otrzyma. W przypadku arbitrażu tego typu mogą wystąpić „zatory" komunikacyjne.
(3) **Zdecentralizowany arbitraż oparty na samodzielnym wyborze:** schemat ten podobny jest do arbitrażu zcentralizowanego, ale zamiast centralnej jednostki podejmującej decyzję urządzenia same określają, które z nich ma wyższy priorytet i powinno otrzymać dostęp do magistrali jako pierwsze.

(4) **Zdecentralizowany arbitraż z wykrywaniem kolizji:** każde urządzenie może zażądać dostępu do magistrali. Jeśli zaś magistrala wykryje jakiekolwiek kolizje (wiele równoczesnych żądań), urządzenie to będzie musiało wystosować żądanie jeszcze raz. (Ten typ arbitrażu stosowany jest w sieciach Ethernet).

Rozdział 7. zawiera więcej szczegółowych informacji na temat magistrali i ich protokołów.

4.1.3. Zegary

Każdy komputer ma wewnętrzny zegar, determinujący szybkość wykonywania poleceń. Zegar synchronizuje również wszystkie podzespoły systemu. Poprzez nadanie impulsu ustawia on tempo dla wszystkich operacji, które mają miejsce w systemie — w podobny sposób, jak metronom czy też dyrygent orkiestry. Procesor wykorzystuje zegar do regulowania swojej pracy, kontrolując szybkość wykonywania przez bramki logiczne określonych operacji. Wykonanie każdej instrukcji zabiera procesorowi określoną liczbę impulsów zegara. Stąd też czas wykonania polecenia jest często mierzony w *cyklach zegara* — czyli czasie, jaki upływa między jego dwoma impulsami — a nie w sekundach. *Częstotliwość zegara* (czasami nazywana też jego szybkością) mierzona jest w megahercach. Jak już wiemy z rozdziału 1., 1 MHz oznacza milion cykli na sekundę (tak więc 1 herc to 1 cykl na sekundę). *Czas trwania cyklu zegara* (inaczej *okres zegara*) jest po prostu odwrotnością częstotliwości zegara. Na przykład 800-megahercowy komputer ma czas trwania cyklu zegara równy 1/800 000 000 s, a więc 1,25 ns. Jeśli zaś cykl zegara trwa w komputerze 2 ns, to wiemy, że jest on taktowany częstotliwością 500 MHz.

Większość komputerów jest synchroniczna, to znaczy główny impuls zegara jest w nich nadawany w równych odstępach czasu (a jego wartość zmienia się cyklicznie z 0 na 1 i z powrotem). Rejestry muszą czekać na impuls zegara, zanim załadują nowe dane. Rozsądnie byłoby przypuszczać, że jeśli przyspieszymy zegar, komputer będzie pracował szybciej. Istnieją jednak ograniczenia co do tego, jak krótki może być cykl zegara. Kiedy zegar wysyła impuls i do rejestrów ładowane są nowe dane, dane wynikowe rejestru z dużym prawdopodobieństwem się zmienią. Te zmienione wartości muszą przejść przez wszystkie układy w komputerze, zanim dotrą do wejścia następnego zespołu rejestrów, w których będą przechowywane. Cykl zegara musi być na tyle długi, aby zmienione dane mogły dotrzeć do następnego zespołu rejestrów. Jeżeli byłby on zbyt krótki, niektóre wartości mogłyby nie dotrzeć do rejestrów. Spowodowałoby to powstanie pewnej niespójności, a tego chcemy za wszelką cenę uniknąć. Stąd też najkrótszy cykl zegara musi trwać przynajmniej tyle, ile wynosi maksymalne opóźnienie przesyłu danych z każdego zespołu rejestrów wynikowych do rejestrów wejściowych. Co stanie się, jeśli skrócimy odległość pomiędzy rejestrami, aby ograniczyć opóźnienie w rozprzestrzenianiu się danych? Moglibyśmy tego dokonać poprzez dodanie rejestrów pomiędzy rejestrami wynikowymi i odpowiadającymi im rejestrami wejściowymi. Przypomnijmy jednak, że rejestry nie mogą zmienić wartości, zanim nie pojawi się impuls zegara. W efekcie otrzymamy więc zwiększoną liczbę cykli zegara. Na przykład polecenie, które pierwotnie wymagało 2 cykli zegara, będzie teraz wymagało trzech albo czterech cykli (lub nawet więcej — w zależności od tego, gdzie umieścimy dodatkowe rejestry).

Większość poleceń komputera wymaga 1 lub 2 cykli zegara, ale niektóre mogą trwać nawet 35 lub więcej. Poniższy wzór pokazuje zależność między sekundami a cyklami zegara:

$$czas\_procesora = \frac{liczba\_sekund}{program} = \frac{liczba\_instrukcji}{program} * \frac{średnia\_liczba\_cykli}{instrukcję} * \frac{liczba\_sekund}{cykl}$$

Ważne jest, abyśmy zauważyli, że architektura komputera ma ogromny wpływ na jej wydajność. Dwa komputery o tych samych prędkościach zegarów niekoniecznie będą kończyły wykonywanie poleceń w ciągu tej samej liczby cykli. Na przykład operacja mnożenia wykonywana przez starszy model Intel 286 wymagała 20 cykli zegara, ale już w przypadku nowoczesnego Pentium może być wykonana w ciągu jednego cyklu, co wskazuje na to, że nowy komputer jest 20 razy szybszy niż 286, nawet gdyby były one taktowane takim samym zegarem. W zasadzie mnożenie wymaga więcej czasu aniżeli dodawanie, obliczenia na liczbach zmiennoprzecinkowych trwają dłużej niż na liczbach całkowitych, a sięgnięcie do pamięci zabiera więcej czasu niż do rejestrów.

Generalnie, jeśli odwołujemy się do terminu *zegar*, mamy na myśli zegar systemowy, czyli nadrzędny zegar regulujący funkcjonowanie procesora i innych komponentów. Jednakże niektóre magistrale mają swoje własne zegary. *Zegary magistral* są zazwyczaj wolniejsze od zegarów procesora, co powoduje problem „wąskiego gardła".

Poszczególne podzespoły komputera mają narzucone granice czasowe, w których muszą się zmieścić z wykonywaniem swych operacji. Ich producenci gwarantują, że nawet w najgorszej sytuacji sprzęt będzie się trzymał narzuconych „terminów". Kiedy połączymy razem wszystkie te podzespoły w szereg, w którym jeden z nich musi zakończyć swoje zadanie po to, aby następny mógł prawidłowo działać, ważne jest, abyśmy zdawali sobie sprawę ze wspomnianych granic — wówczas będziemy mogli prawidłowo zsynchronizować urządzenia. Tym niemniej wiele osób forsuje granice niektórych podsystemów, próbując polepszyć wydajność systemu. Jedną z metod przez nie stosowanych jest tzw. *przetaktowywanie*.

Mimo iż możliwe jest przetaktowanie wielu podzespołów, najczęściej zabiegowi temu poddaje się procesor. Podstawowe założenie tego pomysłu to ustawienie zegara procesora i (lub) zegara magistrali powyżej górnej granicy określonej przez producenta. Mimo tego, że taki zabieg może zwiększyć wydajność systemu, trzeba uważać, żeby go nie rozsynchronizować lub nawet gorzej — nie przegrzać procesora. Można też przetaktować magistralę systemową, co spowoduje przetaktowanie wszystkich korzystających z niej podzespołów. Może to polepszyć wydajność komputera, ale może też zniszczyć podłączone do niej urządzenia.

4.1.4. Podsystem wejścia-wyjścia

Urządzenia wejścia-wyjścia (*I/O*) pozwalają nam na komunikowanie się z systemem komputerowym. Wejście-wyjście to transfer danych pomiędzy pamięcią podstawową a różnymi urządzeniami peryferyjnymi. Urządzenia wejścia — takie jak klawiatury, myszy, czytniki kart, skanery, systemy rozpoznawania głosu oraz ekrany dotykowe — pozwalają nam wprowadzać dane do komputera. Z kolei urządzenia wyjścia — takie jak monitory, drukarki, plotery oraz głośniki — pozwalają nam uzyskać informacje z komputera.

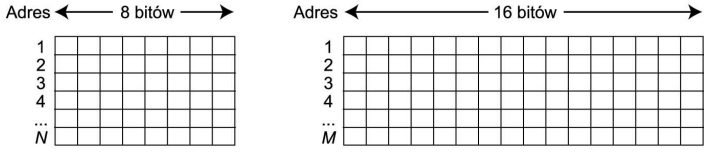
Urządzenia te nie są bezpośrednio połączone z procesorem. W każdym komputerze znajduje się *interfejs*, który zajmuje się transferem danych. Przekształca on sygnały magistrali systemowej na format akceptowany przez dane urządzenie i odwrotnie. Procesor zaś komunikuje się z urządzeniami zewnętrznymi poprzez rejestry wejścia i wyjścia. Taka wymiana danych wykonywana jest na dwa sposoby. Przy *wejściu-wyjściu odwzorowywanym pamięciowo* rejestry interfejsu pojawiają się na mapie pamięci komputera i nie ma w tym przypadku właściwie żadnej różnicy pomiędzy dostępem do pamięci a dostępem do urządzenia wejścia-wyjścia. Jest to wyraźna zaleta, jeśli uwzględniamy prędkość, z tym że takie rozwiązanie zajmuje określoną ilość pamięci w komputerze. W przypadku *wejścia-wyjścia opartego na rozkazach* procesor ma wyspecjalizowane rozkazy realizujące operacje I/O. Pomimo że takie rozwiązanie nie zużywa pamięci, wymaga ono stosowania

specyficznych rozkazów wejścia-wyjścia, co oznacza, że można je stosować tylko w przypadku tych procesorów, które je obsługują. Bardzo istotną rolę w operacjach wejścia-wyjścia odgrywają przerwania. Są one wydajnym sposobem informowania procesora o dostępności operacji wejścia-wyjścia.

4.1.5. Organizacja pamięci i adresowanie

W rozdziale 3. przedstawiłyśmy przykład raczej niewielkiej pamięci. Nie omówiłyśmy jednak jeszcze szczegółowo tego, jak pamięć jest rozplanowana i jak się do niej odwołujemy. Bardzo ważne jest, abyś rozumiał te dwie kwestie, zanim będziemy kontynuować bieżący temat.

Możemy wyobrazić sobie pamięć jako macierz bitów. Szerokość każdego wiersza, implementowanego w postaci rejestru, jest zazwyczaj taka sama, jak długość słowa maszynowego komputera. Każdy rejestr (nazywany częściej *komórką pamięci*) ma niepowtarzalny adres. Adresy pamięci zazwyczaj zaczynają się od zera i rosną. Rysunek 4.4 przedstawia tę zasadę.



RYSUNEK 4.4. a) N 8-bitowych komórek pamięci, b) M 16-bitowych komórek pamięci

Prawie zawsze adres jest przedstawiony w postaci liczby całkowitej bez znaku. Przypomnijmy sobie z rozdziału 2, że 4 bity to półbajt, a 8 bitów to bajt. Normalnie pamięć jest *adresowana bajtowo*, co oznacza, że każdy indywidualny bajt ma swój niepowtarzalny adres. Niektóre komputery mogą mieć rozmiar słowa większy niż pojedynczy bajt — na przykład komputer może operować 32-bitowymi słowami (co oznacza, że w danej chwili czasu określona instrukcja może modyfikować 32 bity na raz), ale ciągle używać architektury adresowanej bitowo. W sytuacji, kiedy słowo składa się z kilku bajtów, bajt o najniższym adresie decyduje o adresie całego słowa. Jest też możliwe, żeby komputer był *adresowany słowowo*, co oznacza, że każde słowo (niekoniecznie każdy bajt) ma swój własny adres. Tym niemniej większość komputerów adresowana jest bajtowo (nawet jeśli mają 32-bitowe lub większe słowa). Adres pamięci jest zazwyczaj przechowywany w postaci pojedynczego słowa maszynowego.

Jeśli powyższy wywód o komputerach używających adresowania bajtowego i słów o różnej długości nieco Cię zmylił, przydatna może okazać się poniższa analogia. Pamięć podobna jest do ulicy pełnej bloków mieszkalnych. Każdy budynek (słowo) ma wiele mieszkań (bajtów), Zarazem każde mieszkanie ma swój własny adres. Wszystkie mieszkania ponumerowane (adresowane) są w kolejności od 0 do ogólnej liczby mieszkań w kompleksie. Budynki stanowią grupę mieszkań. W komputerach taką samą rolę pełnią słowa. Słowa są podstawową jednostką używaną w różnorodnych rozkazach. Na przykład możesz odczytać lub zapisać słowo w pamięci — nawet na komputerze adresowanym bajtowo.

Jeśli architektura komputera jest adresowana bajtowo, a słowo w rozkazie jest większe niż jeden bajt, należy przeprowadzić *wyrównanie*. Na przykład jeśli chcemy odczytać 32-bitowe słowo na komputerze adresowanym bajtowo, należy upewnić się, że: (1) słowo to zostało zapisane na naturalnej

granicy wyrównywania oraz że (2) dostęp do niego zaczyna się właśnie na tej granicy. W przypadku słów 32-bitowych jest to osiągane poprzez wprowadzenie wymagania, aby każdy adres był wielokrotnością 4. Niektóre architektury pozwalają na „niewyrównany dostęp" — wówczas pożądany adres nie musi zaczynać się na naturalnej granicy.

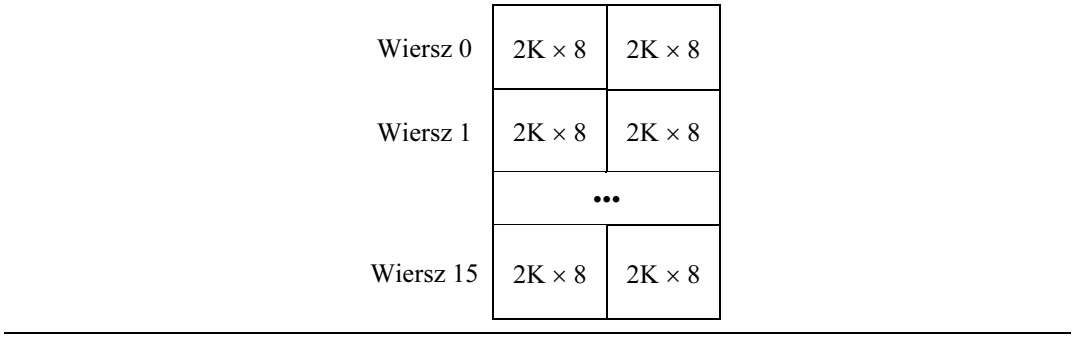
Pamięć zbudowana jest z chipów pamięci o dostępie bezpośrednim (RAM). (O pamięci mówimy szczegółowo w rozdziale 6.). Pamięć jest często opisywana za pomocą skrótu L × W (długość × szerokość). Na przykład 4M × 16 oznacza, że pamięć ma 4M długości (to znaczy ma $4M = 2^2 \times 2^{20} = 2^{22}$ słowa) i jest szeroka na 16 bitów (każde słowo ma 16 bitów). Szerokość (druga liczba z pary) przedstawia rozmiar słowa. Aby zaadresować tę pamięć (mając na myśli adresowanie słowa), musimy być w stanie niepowtarzalnie zidentyfikować 2^{22} różnych składników, co oznacza, że potrzebujemy 2^{22} różnych adresów. Ponieważ adresy są liczbami binarnymi bez znaku, należy policzyć od 0 do $(2^{22} - 1)$ w systemie binarnym. Ilu to wymaga bitów? Cóż, aby policzyć w systemie binarnym od 0 do 3 (dla łącznie 4 składników), potrzebujemy 2 bitów. Aby policzyć w systemie binarnym od 0 do 7 (łącznie 8 składników), potrzebujemy 3 bitów. Aby policzyć w systemie binarnym od 0 do 15 (łącznie 16 składników), potrzebujemy 4 bitów. Zauważyłeś już tutaj pewien wzór? Czy potrafisz wypełnić brakujące wartości w tabeli 4.1?

TABELA 4.1. Obliczanie wymaganej liczby bitów adresowych

Suma składników	2	4	8	16	32
Suma wyrażona jako potęga liczby 2	2^1	2^2	2^3	2^4	2^5
Liczba bitów	1	2	3	4	??

Prawidłowa odpowiedź to 5 bitów. W zasadzie, jeśli komputer ma 2^N adresowalnych jednostek pamięci, będzie potrzebował N bitów do niepowtarzalnego zaadresowania każdego bajtu.

Pamięć główna jest zazwyczaj większa niż jeden chip RAM. Konsekwentnie, aby otrzymać wymagany rozmiar pamięci, chipy te połączone są w pojedynczy moduł pamięci. Załóżmy na przykład, że potrzebujesz zbudować pamięć 32K × 16, a masz jedynie chipy RAM o rozmiarach 2K × 8. Mógłbyś połączyć 16 wierszy i 2 kolumny chipów razem, tak jak to zostało pokazane na rysunku 4.5.



RYSUNEK 4.5. Pamięć jako zbiór chipów RAM

Każdy wiersz chipów odnosi się do 2K słów (zakładając ze komputer jest adresowalna przez bajty), ale wymaga to dwóch chipów do pokrycia całej szerokości. Adresy dla tej pamięci muszą mieć 15 bitów (jest $32K = 2^5 \times 2^{10}$ słów do zapisania). Jednakże każda para chipów (czyli każdy

wiersz) wymaga tylko 11 linii adresowych (każda para chipów mieści tylko 2^{11} słów). W tej sytuacji potrzebny byłby dekoder do rozkodowania 4 bitów adresu z lewej strony, aby określić, na której parze chipów zlokalizowany jest dany adres. Kiedy już odpowiednia para chipów zostanie zlokalizowana, pozostałe 11 bitów będzie stanowiło dane wejściowe do innego dekodera, który znajdzie dokładny adres danych wewnątrz tej pary chipów.

Pojedynczy, współdzielony moduł pamięci wymaga szeregowania dostępu. Problem ten można rozwiązać, stosując *przeplatanie pamięci*, polegające na rozbijaniu jej na kilka modułów (lub inaczej banków). W przypadku *przeplotu niskopozycyjnego* bank wybierany jest na podstawie zawartości mniej znaczących bitów adresu, zaś w przypadku *wysokopozycyjnego* — na podstawie bardziej znaczących. Przeplot wysokopozycyjny jest bardziej intuicyjny, ponieważ adresy są w nim rozprowadzane tak, aby każdy moduł zawierał adresy o kolejnych numerach, tak jak 32 adresy przedstawione na rysunku 4.6.

Moduł 0	Moduł 1	Moduł 2	Moduł 3	Moduł 4	Moduł 5	Moduł 6	Moduł 7
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

RYSUNEK 4.6. Wysokopozycyjny przeplot pamięci

W pamięci o przelocie niskopozycyjnym następujące po sobie słowa pamięci umieszczane są w osobnych modułach. Rysunek 4.7 ukazuje przeplot niskopozycyjny na przykładzie 32 adresów.

Moduł 0	Moduł 1	Moduł 2	Moduł 3	Moduł 4	Moduł 5	Moduł 6	Moduł 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

RYSUNEK 4.7. Niskopozycyjny przeplot pamięci

Jeśli odpowiednia magistrala używa przeplotu niskopozycyjnego, odczyt lub zapis zawartości jednego modułu pamięci może być rozpoczęty, zanim zakończy się odczyt lub zapis zawartości innego modułu (a zatem odczyt i zapis mogą na siebie zachodzić).

Zagadnienia dotyczące pamięci, które omówiłyśmy, są bardzo istotne i będziemy do nich wracać w wielu miejscach w dalszej części tej książki, zwłaszcza w rozdziale 6., w którym znajdziesz bardziej szczegółowy opis pamięci. Podstawowe zasady, o których należy pamiętać, to: (1) adresy pamięci to wartości binarne bez znaku (mimo to często zapisujemy je w postaci szesnastkowej, ponieważ tak łatwiej jest się nimi posługiwać) oraz (2) liczba składników, jakie musimy zaadresować, określa liczbę bitów pojawiających się w adresie. Pomimo tego, że zawsze moglibyśmy używać w adresie większej liczby bitów niż wymagana, rzadko stosuje się takie rozwiązanie, pamiętając o tym, że minimalizacja jest w przypadku budowy komputera podstawową zasadą.

4.1.6. Przerwania

Przedstawiłyśmy podstawowe informacje o sprzęcie komputerowym wymagane do dokładnego zrozumienia jego budowy. Omówiłyśmy procesor, magistrale, jednostkę sterującą, rejestry, zegary, urządzenia wejścia-wyjścia oraz pamięć. Jednakże istnieje jeszcze jedna zasada, o której koniecznie należy powiedzieć, a która dotyczy tego, w jaki sposób wszystkie te urządzenia współdziałają z procesorem.

Przerwania są to zdarzenia, które zakłócają normalny tryb wykonywania programów. Mogą one być inicjowane w różnych sytuacjach, między innymi na skutek:

- żądań urządzeń wejścia-wyjścia;
- błędów arytmetycznych (np. dzielenia przez zero);
- niedoboru lub nadmiaru arytmetycznego;
- nieprawidłowego działania sprzętu (np. błędu parzystości pamięci);
- świadomych działań użytkownika (np. podczas wyszukiwania błędów w kodzie źródłowym programu);
- błędu strony (dokładne omówienie w rozdziale 6.);
- nieprawidłowego polecenia (zazwyczaj związanego z nieprawidłowym użyciem wskaźników);
- wielu innych powodów.

Czynności podejmowane w celu *obsłużenia* wszystkich wymienionych rodzajów przerwań potrafią być bardzo różne. Wynika to z faktu, że poinformowanie procesora o tym, że żądanie I/O dobiegło końca, różni się przecież od zakończenia wykonywania programu z powodu błędu dzielenia przez zero. Jednak obie te sytuacje są obsługiwane za pośrednictwem przerwań, ponieważ wymagają one zmiany normalnego trybu wykonywania programu.

Przerwanie może zostać zainicjowane przez użytkownika lub przez sam komputer. Może być ono *maskowalne* (dające się wyłączyć lub zignorować) bądź *niemaskowalne* (przerwanie o wysokim priorytecie, którego nie można zignorować i którego zauważenie trzeba potwierdzić). Może ono też pojawiać się wewnątrz instrukcji lub między jedną instrukcją a drugą, być synchroniczne (występować za każdym razem w tym samym miejscu w programie) lub asynchroniczne (pojawiać się niespodziewanie). Może też prowadzić do zakończenia wykonywania programu lub chwilowego zawieszenia jego wykonywania do czasu, aż przerwanie to nie zostanie obsłużone. Bardziej szczegółowe omówienie przerwań znajdziesz w podrozdziale 4.3.2 i w rozdziale 7.

Teraz, gdy przyjrzeliśmy się już z grubsza podstawowym komponentom, bez których komputer nie mógłby działać, zapoznamy Cię z prostą, choć funkcjonalną architekturą MARIE.

4.2. MARIE

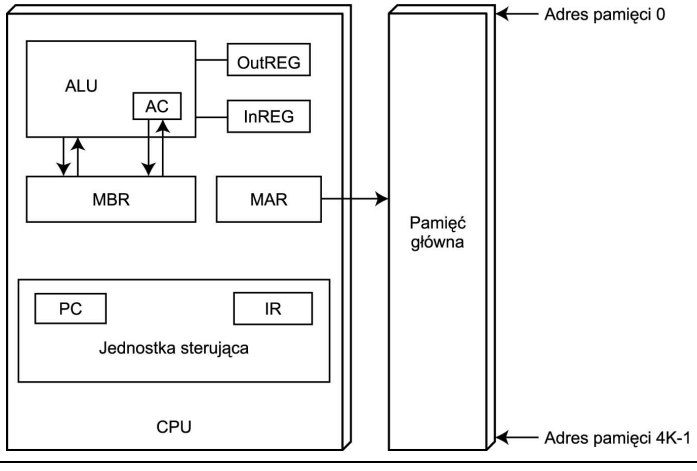
MARIE (ang. *A Machine Architecture that is Really Intuitive and Easy* — naprawdę prosta i intuicyjna architektura komputera) jest prostą architekturą składającą się z pamięci (przechowującej programy i dane) oraz procesora (składającego się z jednostki ALU i kilku rejestrów). Zawiera ona wszystkie funkcjonalne podsystemy niezbędne do stworzenia prawdziwego, działającego komputera. MARIE pomoże nam zilustrować zagadnienia omawiane w tym i trzech poprzednich rozdziałach. Jej architekturę opisujemy w kolejnych podrozdziałach.

4.2.1. Architektura

MARIE ma następujące parametry:

- Notacja dwójkowa, zapis w kodzie dopełnieniowym.
- Przechowywanie programu, stała długość słowa.
- Adresowanie słowowe (a nie bajtowe).
- 4K pamięci głównej (co oznacza, że na każdy adres przypada 12 bitów).
- 16-bitowe dane (słowa mają 16 bitów).
- 16-bitowe rozkazy — 4 bity na kod operacji i 12 na adres.
- 16-bitowy akumulator (AC).
- 16-bitowy rejestr rozkazów (IR).
- 16-bitowy rejestr bufora pamięci (MBR).
- 12-bitowy licznik rozkazów (PC).
- 12-bitowy rejestr adresów pamięci (MAR).
- 8-bitowy rejestr wejściowy.
- 8-bitowy rejestr wyjściowy.

Rysunek 4.8 przedstawia architekturę MARIE.



RYSUNEK 4.8. Architektura MARIE

Zanim przejdziemy dalej, należy podkreślić jedną ważną kwestię dotyczącą pamięci. W rozdziale 8. przedstawiłyśmy prostą budowę pamięci przy użyciu przerzutników D. Podkreślamy jeszcze raz, że każde miejsce w pamięci ma swój niepowtarzalny adres (zapisany w systemie binarnym) i każde miejsce może przechowywać wartość. Czasami Czytelnikom adresy mylą się z wartościami przechowywanymi pod tymi adresami. Aby uniknąć nieporozumienia, wyobraźmy sobie budynek poczty. Znajdują się w nim skrzynki z różnymi adresami lub numerami. Wewnątrz skrzynek są listy. Aby je otrzymać, trzeba znać numer skrzynki pocztowej. To samo dotyczy danych lub rozkazów, które trzeba odczytać z pamięci. Wszelkie operacje na wartości dowolnego miejsca w pamięci możemy wykonywać dopiero po podaniu jego konkretnego adresu. Przekonamy się, że jest wiele różnych sposobów na jego określenie.

4.2.2. Rejestry i magistrale

Jak pokazałyśmy na rysunku 4.8, rejestry są znajdującymi się wewnątrz procesora komórkami pamięci, w których można przechowywać dane. Wszystkie operacje związane z przetwarzaniem danych (działania arytmetyczne, decyzje logiczne itd.) wykonywane są przez jednostkę arytmetyczno-logiczną (ALU). Podczas wykonywania programów rejestry wykorzystywane są do ściśle określonych celów: Przechowują one wartości tymczasowe, dane, które w pewien sposób przetwarzamy, albo też wyniki prostych operacji. Bardzo często instrukcje jawnie odwołują się do poszczególnych rejestrów. Przekonasz się o tym, gdy będziemy opisywały zbiór rozkazów MARIE w podrozdziale 4.2.3.

MARIE ma następujące rejestry:

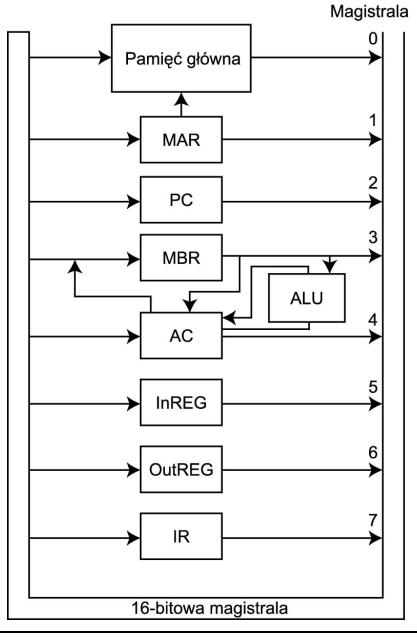
- **AC:** *akumulator*, który przechowuje wartości danych. Jest to *rejestr uniwersalny* przechowujący dane, które procesor potrzebuje przetworzyć. Większość współczesnych komputerów ma wiele rejestrów uniwersalnych.
- **MAR:** *rejestr adresu pamięci*, który przechowuje adres pamięci danych, do których się odnosimy.
- **MBR:** *rejestr bufora pamięci*, który przechowuje albo dane właśnie odczytane z pamięci, albo dane gotowe do zapisania.
- **PC:** *licznik rozkazów*, który przechowuje adres następnej instrukcji, którą program ma wykonać.
- **IR:** *rejestr rozkazów*, przechowujący następny rozkaz, jaki należy wykonać.
- **InREG:** *rejestr wejściowy*, przechowujący dane z urządzenia wejściowego.
- **OutREG:** *rejestr wyjściowy*, który przechowuje dane dla urządzenia wyjściowego.

Rejestry MAR, MBR, PC oraz IR przechowują bardzo specyficzne informacje i nie mogą być użyte do innych celów aniżeli te wymienione. Na przykład nie moglibyśmy przechowywać arbitralnych wartości danych z pamięci w liczniku rozkazów. Konieczne jest użycie do tego celu albo rejestru MBR, albo akumulatora. MARIE zawiera też *rejestr stanu* (lub inaczej *rejestr flagowy*), który przechowuje informacje o zaistnieniu pewnych okoliczności, takich jak błąd przepełnienia w jednostce ALU. Ponieważ jednak chcemy utrzymać klarowność wywodu, nie będziemy umieszczały tego rejestru na żadnym z rysunków.

MARIE jest bardzo prostym komputerem z ograniczonym zestawem rejestrów. Współczesne procesory mają wiele rejestrów uniwersalnych, często nazywanych *rejestrami widocznymi dla użytkownika*, które pełnią funkcję podobną do akumulatora. Współczesne komputery mają też inne rejestry, które pełnią na przykład rolę przesuwników lub które przechowują tylko część pewnej większej wartości (zajmującej kilka rejestrów).

MARIE nie może przesyłać danych czy rozkazów z lub do rejestrów bez udziału magistrali. Przyjmujemy, że w architekturze tej mamy do czynienia ze współdzieloną magistralą wykorzystywaną przez wszystkie urządzenia. Każde z nich ma swój numer identyfikacyjny, który musi być ustawiony, zanim będzie ono mogło odwoływać się do wspólnej szyny. Przewidziałyśmy też pewne ścieżki danych, które przyspieszają wykonywanie programów. Zastosowałyśmy na przykład osobną ścieżkę komunikacyjną między rejestrem MAR a pamięcią (rejestr MAR zapewnia dane wejściowe dla linii adresowych pamięci, tak aby procesor „wiedział", skąd ma odczytać zawartość pamięci lub gdzie ma dokonać zapisu) oraz oddzielną ścieżkę prowadzącą z rejestru MBR do akumulatora. Istnieje również specjalna ścieżka z rejestru MBR do jednostki ALU, pozwalająca na to, aby dane z MBR mogły być wykorzystywane w operacjach matematycznych. Informacje mogą również płynąć z akumulatora poprzez ALU i z powrotem do akumulatora, bez potrzeby wysyłania ich przez wspólną magistralę. Zaletą wykorzystywania tych dodatkowych ścieżek jest to, że można za ich pomocą wysłać informacje poprzez wspólną magistralę w trakcie tego samego cyklu zegara, co z kolei sprawia, że oba zdarzenia mogą zachodzić równocześnie.

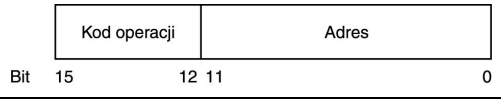
Rysunek 4.9 pokazuje ścieżkę danych (ścieżkę, którą płyną informacje) w MARIE.



RYSUNEK 4.9. Ścieżka danych w MARIE

4.2.3. Architektura zbioru rozkazów

MARIE ma bardzo prosty zbiór rozkazów, choć daje on duże możliwości. *Architektura zbioru rozkazów* (*ISA*) określa rozkazy, jakie komputer może wykonywać, oraz format każdego z nich. ISA jest w szczególności interfejsem pomiędzy oprogramowaniem a fizycznym sprzętem. Niektóre ISA zawierają nawet setki rozkazów. Wspominałyśmy już wcześniej, że każdy rozkaz MARIE składa się z 16 bitów. Najbardziej znaczące 4 bity, od 12 do 15, tworzą tzw. *kod operacji*, który określa rozkaz, jaki należy wykonać (możemy więc posługiwać się 16 rozkazami). 12 najmniej znaczących bitów, tj. bity od 0 do 11, tworzą adres, co oznacza, że jesteśmy w stanie zaadresować pamięć o rozmiarze $2^{12} - 1$ bajtów. Format instrukcji MARIE przedstawiony jest na rysunku 4.10.



RYSUNEK 4.10. Format rozkazu MARIE

Na większość architektur ISA składają się rozkazy dotyczące przetwarzania danych, ich przenoszenia oraz kontrolowania kolejności wykonania programu. Zbiór rozkazów MARIE składa się z poleceń przedstawionych w tabeli 4.2.

Polecenie `Load` (ang. *Ładuj*) pozwala nam przenosić dane z pamięci do procesora (poprzez rejestr MBR oraz akumulator). Wszystkie dane (a więc wszystko, co *nie* jest rozkazem) zapisane w pamięci muszą przejść najpierw przez rejestr MBR, a następnie albo przez akumulator, albo

TABELA 4.2. Zbiór rozkazów MARIE

Numer instrukcji		Instrukcja	Znaczenie
Dwójkowo	**Szesnastkowo**		
0001	1	`Load X`	Załaduj zawartość spod adresu `X` do akumulatora
0010	2	`Store X`	Zachowaj zawartość akumulatora pod adresem `X`
0011	3	`Add X`	Dodaj zawartość spod adresu `X` do akumulatora i zachowaj wynik w akumulatorze
0100	4	`Subt X`	Odejmij zawartość spod adresu `X` od akumulatora i zachowaj wynik w akumulatorze
0101	5	`Input`	Wprowadź wartość z klawiatury do akumulatora
0110	6	`Output`	Wyślij wartość przechowywaną w akumulatorze na ekran
0111	7	`Halt`	Zakończ program
1000	8	`Skipcond`	Pomiń następny rozkaz, jeżeli spełniony jest określony warunek
1001	9	`Jump X`	Wczytaj wartość `X` do licznika rozkazów

przez jednostkę ALU — w tej architekturze nie ma innej możliwości. Zauważ, że w poleceniu `Load` nie musimy precyzować, że chcemy użyć akumulatora, ponieważ ten rejestr jest wykorzystywany *domyślnie*. Inne polecenia odwołują się do rejestru akumulatora w podobny sposób. Polecenie `Store` pozwala nam na przesuwanie danych z procesora z powrotem do pamięci. Instrukcje `Add` i `Subt` odpowiednio dodają i odejmują od wartości zapisanych w akumulatorze wartości danych odnalezione pod adresem `X`. Dane zlokalizowane pod adresem `X` kopiowane są do rejestru MBR, gdzie są przechowywane, dopóki nie zostanie zakończone działanie arytmetyczne. Polecenia `Input` i `Output` pozwalają MARIE komunikować się ze światem zewnętrznym.

Wejście i wyjście danych to skomplikowane operacje. We współczesnych komputerach wejście i wyjście realizowane jest za pomocą bajtów zapisanych w kodzie ASCII. Oznacza to, że jeśli wpiszemy na klawiaturze liczbę 32, to w rzeczywistości zostanie to odczytane jako znaki ASCII „3” i „2”. Obydwa te znaki muszą być przekształcone na wartość numeryczną 32, zanim zostaną zapisane w akumulatorze. Ponieważ koncentrujemy się na tym, jak funkcjonuje komputer, założymy, że wartości danych wejściowych wprowadzone za pomocą klawiatury są automatycznie przekształcane na prawidłowe wartości liczbowe. Dotykamy tu bardzo ważnej kwestii: skąd komputer „wie”, czy wartość na wejściu lub wyjściu ma być traktowana jako liczba czy jako kod ASCII, skoro wszystko na wejściu i wyjściu ma format ASCII? Otóż komputer „domyśla się” tego z kontekstu, w jakim wartość ta jest używana.

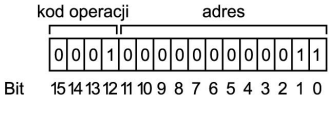
W przypadku MARIE zakładamy, że na wejściu i wyjściu są tylko dane numeryczne. Pozwalamy również na wprowadzanie wartości w systemie dziesiętnym i zakładamy, że zostaną one „w magiczny sposób” przekształcone na system binarny i że w takiej postaci będą zachowane. W rzeczywistości, jeżeli komputer ma pracować prawidłowo, trzeba zaimplementować odpowiednie mechanizmy pomocnicze.

Instrukcja `Halt` (ang. *Zatrzymaj*) powoduje zakończenie wykonywania bieżącego programu. Polecenie `Skipcond` pozwala na stosowanie skoków warunkowych, które są wykorzystywane na przykład w pętlach `while` i wyrażeniach `if`. Gdy wykonywana jest ta instrukcja, procesor musi sprawdzić wartość wyrażenia znajdującego się w akumulatorze. Dwa z bitów adresowych (załóżmy, że zawsze będziemy odwoływali się do dwóch bitów położonych najbliżej pola określającego kod operacji, to znaczy do bitów 10 i 11) określają warunek, który trzeba sprawdzić. Jeśli obydwa bity mają wartość 00, tłumaczymy to jako „pomiń, jeśli wartość w akumulatorze jest ujemna”. Jeśli oba

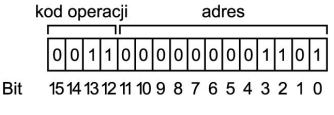
przedstawiają wartość 01 (bit jedenasty ma wartość 0, a dziesiąty 1), tłumaczymy to następująco: „pomiń, jeśli wartość w akumulatorze równa jest 0”. Wreszcie, jeśli dwa bity adresowe przedstawiają wartości 10 (lub inaczej 2), należy to rozumieć jako „pomiń, jeśli akumulator ma wartość większą niż 0”. Przez „pomiń” rozumiemy po prostu „skocz do następnej instrukcji”. Taki skok jest realizowany w taki sposób, że zawartość licznika rozkazów jest zwiększana o 1, co powoduje pominięcie kolejnej instrukcji, która nigdy nie zostanie wczytana. Bezwarunkowe polecenie `Jump` również wpływa na zawartość rejestru rozkazów. Sprawia ono, że bieżąca zawartość rejestru PC zostaje zastąpiona wartością `X`, będącą adresem następnej instrukcji, którą należy wczytać.

Ponieważ chcemy utrzymać architekturę komputera oraz zbiór rozkazów w jak najprostszej postaci, a zarazem przekazać Ci informacje potrzebne do zrozumienia, jak działa komputer, pominęłyśmy kilka przydatnych rozkazów. Jak się wkrótce przekonasz, przedstawiony przez nas skromny zbiór poleceń wciąż ma wiele możliwości. Jak tylko zapoznasz się z zasadami działanie tego komputera, rozszerzymy zestaw instrukcji, aby ułatwić Ci pisanie programów.

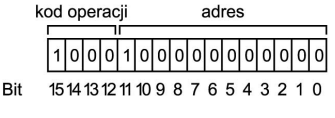
Przetestujmy format rozkazów używany w MARIE. Załóżmy, że mamy następujący 16-bitowy rozkaz:



Pierwsze 4 bity z lewej strony oznaczają kod operacji lub inaczej rodzaj instrukcji, jaką należy wykonać. 0001 to zapis binarny liczby 1, co oznacza polecenie `Load`. Pozostałe 12 bitów wskazuje na adres wartości, które będziemy wczytywać. Jest to adres 3 w pamięci głównej. Wykonanie tej instrukcji powoduje, że wartości znalezione w pamięci głównej pod adresem 3 będą skopiowane do akumulatora. Rozpatrzmy inny rozkaz:



Pierwsze 4 bity z lewej strony, 0011, przechowują wartość 3, odpowiadającą instrukcji `Add`. Bity adresowe wskazują na adres 00Dh (13 dziesiętnie). Idziemy zatem do głównej pamięci, pobieramy wartość danych znajdujących się pod adresem 00Dh i dodajemy ją do wartości w akumulatorze. Wartość w akumulatorze zmieni się zatem w sumę obu danych. Innym przykładem będzie:



Kod operacji występujący w tym rozkazie wskazuje na instrukcję `Skipcond`. Bity dziesiąty i jedenasty (lub jedenasty i dziesiąty, zależy z której strony patrzeć) przechowują wartość binarną 10, czyli 2 w systemie dziesiętnym. Należy to rozumieć jako „pomiń, jeśli wartość w akumulatorze jest większa lub równa 0”. Jeśli wartość w akumulatorze jest mniejsza niż zero, rozkaz ten jest ignorowany i przechodzimy po prostu do kolejnego polecenia. Jeśli wartość w akumulatorze jest większa lub równa zero, rozkaz ten nakazuje zwiększenie licznika rozkazów o 1, powodując w ten sposób zignorowanie kolejnej instrukcji (pamiętaj o tym, czytając następny podrozdział o cyklach rozkazów).

W przykładach tych poruszono pewną interesującą kwestię. Będziemy pisać programy, używając tego ograniczonego zestawu poleceń. Czy wolałbyś pisać program za pomocą poleceń `Load`, `Add` i `Halt`, czy raczej ich binarnych odpowiedników 0001, 0011 oraz 0111? Większość ludzi wolałaby używać zamiast wartości binarnych łatwych do zapamiętania nazw poleceń. Polecenia dwójkowe MARIE nazywane są *instrukcjami maszynowymi*, natomiast odpowiadające im, łatwe do zapamiętania nazwy to *polecenia w języku asemblera*. Między tymi dwoma rodzajami poleceń występuje zależność typu „jeden do jednego", co oznacza, że każde polecenie asemblera tłumaczone jest na dokładnie jedną instrukcję maszynową. Gdy wprowadzimy już program w języku asemblera (posługując się na przykład poleceniami wymienionymi w tabeli 4.2), będziemy potrzebowali programu zwanego asemblerem, który przekształci go na kod maszynowy. Asemblery omówimy w podrozdziale 4.5.

4.2.4. Notacja „rejestr-przesunięcie"

Widzieliśmy już, że systemy cyfrowe składają się z wielu podzespołów, w tym z jednostek ALU, rejestrów, pamięci, dekoderów i jednostek sterujących. Jednostki te są ze sobą połączone magistralami, które umożliwiają przesyłanie danych we wnętrzu komputera. Zbiór rozkazów architektury MARIE, przedstawiony w poprzednim podrozdziale, składa się z instrukcji maszynowych przeznaczonych dla poszczególnych podzespołów, realizujących określone partie programu. Wszystkie te polecenia wydają się być bardzo proste, jednak jeśli przyjrzymy się temu, co faktycznie dzieję się na poziomie każdego urządzenia, okaże się, że na każdą instrukcję składa się kilka operacji. Na przykład polecenie `Load` wczytuje zawartość danego miejsca w pamięci do rejestru akumulatora. Jeżeli przyjrzymy się temu, co w trakcie jego wykonywania dzieje się na poziomie poszczególnych podzespołów, to zobaczymy, że w rzeczywistości wykonywana jest ogromna liczba „minirozkazów". Najpierw adres zawarty w rozkazie musi być wczytany do rejestru MAR. Następnie dane zapisane w pamięci pod danym adresem muszą być załadowane do rejestru MBR. Później rejestr MBR musi być załadowany do akumulatora. Te minirozkazy nazywane są *mikrooperacjami* i to od nich zależą wszystkie podstawowe operacje, jakie można wykonywać na danych przechowywanych w rejestrach.

Notacja „rejestr-przesunięcie" (*RTN*) to zbiór oznaczeń i symboli używanych do opisywania przebiegu mikrooperacji. Oznaczenia M[X] używamy do wskazania na dane faktycznie przechowywane pod adresem `X` w pamięci, zaś ← do zaznaczenia przesyłania informacji. W rzeczywistości transfer danych z jednego rejestru do drugiego zawsze wymaga przesłania ich z rejestru źródłowego do magistrali, a następnie z magistrali do rejestru przeznaczenia. Jednakże ze względu na przejrzystość naszego wywodu będziemy już ignorowali transfery poprzez magistralę, zakładając, iż rozumiesz, że mają one miejsce. Teraz zapiszemy w notacji „rejestr-przesunięcie" przebieg każdej instrukcji występującej w architekturze ISA MARIE.

Load X

Przypomnijmy sobie, że polecenie to wczytuje zawartość pamięci pod adresem `X` do akumulatora. Jednakże adres `X` musi być najpierw umieszczony w rejestrze MAR. Następnie dane w miejscu M[MAR] (lub inaczej pod adresem `X`) zostaną przesunięte do MBR. Na koniec dane te zostaną umieszczone w akumulatorze.

```
MBR ← X
MBR ← M[MAR], AC ← MBR
```

Ponieważ IR musi użyć magistrali do skopiowania wartości X do MAR, zanim dane z miejsca X będą mogły być wstawione do MBR, operacja ta wymaga dwóch cykli magistrali. Dlatego też obie te operacje są wykonywane na oddzielnych liniach — pozwala to na podkreślenie, że nie mogą one wystąpić w ciągu tego samego cyklu. Jednakże, ponieważ mamy specjalne połączenie pomiędzy MBR i akumulatorem, transfer danych z MBR do akumulatora może mieć miejsce zaraz po tym, jak dane zostaną umieszczone w rejestrze MBR, bez konieczności czekania na magistralę.

Store X

Polecenie to zapisuje zawartość akumulatora do pamięci, umieszczając ją pod adresem X:

```
MAR ← X, MBR ← AC
M[MAR] ← MBR
```

Add X

Wartość przechowywana pod adresem X zostanie dodana do wartości zapisanej w akumulatorze. Operacja ta realizowana jest w następujący sposób:

```
MAR ← X
MBR ← M[MAR]
AC ← AC + MBR
```

Subt X

Polecenie podobne do Add. Odejmuje ono wartość przechowywaną pod adresem X od wartości zapisanej w akumulatorze i umieszcza wynik z powrotem w akumulatorze.

```
MAR ← X
MBR ← M[MAR]
AC ← AC - MBR
```

Input

Wszelkie dane wejściowe pochodzące z urządzeń wejściowych są najpierw kierowane do rejestru wejściowego, a dopiero z niego są przenoszone do akumulatora.

```
AC ← InREG
```

Output

Polecenie to powoduje, że zawartość akumulatora przenoszona jest do rejestru wyjściowego, skąd jest ostatecznie wysyłana na urządzenie wyjściowe.

```
OutREG ← AC
```

Halt

Na rejestrach nie są w tym przypadku wykonywane żadne operacje. Po napotkaniu tej instrukcji procesor po prostu wstrzymuje wykonywanie programy.

Skipcond

Jak sobie zapewne przypominasz, polecenie to wykorzystuje bity 10 i 11 w polu adresu do ustalenia, do czego należy przyrównać wartość znajdującą się w akumulatorze. W zależności od kombinacji bitów sprawdzamy, czy akumulator ma wartość ujemną, równą zero, czy też większą od zera. Jeśli podany warunek jest spełniony, następna instrukcja jest pomijana. Dokonywane jest to poprzez zwiększenie zawartości licznika rozkazów o 1.

```
if IR[11-10] = 00 then          {jeśli bity 10. i 11. w IR mają oba wartość 0}
   If AC < 0 then PC ← PC+1
else If IR[11-10] = 01 then     {jeśli bit 11. = 0, a bit 10. = 1}
   If AC = 0 then PC ← PC + 1
else If IR[11-10] = 10 then     {jeśli bit 11. = 1, a bit 10. = 0)
   If AC > 0 then PC ← PC + 1
```

Jeśli bity na pozycji dziesiątek i jedenastej mają wartość 1, występuje błąd. Można jednak sformułować dodatkowy warunek, w którym taka konfiguracja wartości będzie miała jakieś znaczenie.

Jump X

Polecenie to powoduje bezwarunkowy przeskok do podanego adresu `X`. Stąd też, aby możliwe było wykonanie tego polecenia, wartość `X` musi być wczytana do licznika rozkazów.

```
PC ← X
```

W rzeczywistości jednak wartość `X` zapisana jest na 12 mniej znaczących bitach rejestru rozkazów (IR[11-0]). Operację tę można więc precyzyjniej opisać jako:

```
PC ← IR[11-0]
```

Wyczuwamy jednak, że oznaczenie `PC ← X` jest łatwiejsze do zrozumienia i połączenia z pozostałymi instrukcjami, stąd używać będziemy właśnie tego zapisu

Notacja „rejestr-przesunięcie” jest symbolicznym sposobem zapisania tego, co dzieje się w systemie, kiedy jest wykonywane dane polecenie. Notacja ta (RTN) zależy od ścieżki danych w tym znaczeniu, że jeśli wiele mikrooperacji musi współdzielić magistralę, to polecenia te muszą być wykonywane sekwencyjnie jedno po drugim.

4.3. Przetwarzanie rozkazów

Teraz, kiedy mamy już podstawowy język, za pomocą którego możemy komunikować się z komputerem, należy tylko dokładnie omówić, w jaki sposób wybrany program jest wykonywany. Wszystkie komputery działają według podstawowego cyklu maszynowego — „pobierz, dekoduj, wykonaj”.

4.3.1. Cykl „pobierz, dekoduj, wykonaj”

Cykl „pobierz, dekoduj, wykonaj” to zbiór czynności, które komputer przeprowadza w celu wykonania programu. Procesor pobiera polecenie (przesyła je z pamięci głównej do rejestru rozkazów), rozkodowuje je (określa kod operacji i wczytuje dane potrzebne do jej wykonania) oraz wykonuje je (przeprowadza operacje wskazane przez instrukcję). Zauważ, że dużą część tego cyklu zajmuje kopiowanie danych z jednego miejsca do drugiego. Kiedy program jest wczytywany, adres jego pierwszej instrukcji musi zostać umieszczony w liczniku rozkazów. Kolejne kroki cyklu, które mają miejsce w określonych cyklach zegara, zostały przedstawione poniżej. Zauważ, że kroki 1. i 2. tworzą fazę „pobierz”. Krok 3. to faza rozkodowywania, a krok 4. to faza wykonania.

(1) Kopiuj zawartość licznika programu do rejestru MAR: `MAR ← PC`.
(2) Idź do pamięci głównej i wczytaj rozkaz znaleziony pod adresem zapisanym w rejestrze MAR, umieszczając go w IR; dodaj do licznika rozkazów 1 (licznik programu wskazywał będzie teraz na następne polecenie w programie): `IR ← M[MAR]`, a następnie `PC ← PC+1`. (Uwaga: ponieważ MARIE jest komputerem adresowanym słowowo, wartość licznika rozkazów jest zwiększana o 1, co powoduje, że w rejestrze tym pojawia się adres nowego słowa. Gdyby MARIE była adresowana bajtowo, licznik rozkazów musiałby zostać powiększony o 2, ponieważ każda instrukcja składałaby się z dwóch bajtów. Na komputerze adresowalnym bajtowo, o słowach 32-bitowych, licznik programu musiałby być powiększony o 4).
(3) Skopiuj pierwsze 12 bitów od prawej z rejestru IR do rejestru MAR; rozkoduj cztery pierwsze bity od lewej, aby poznać kod operacji: `MAR ← IR[11-0]`, a następnie zdekoduj `IR[15-12]`.
(4) Jeśli to konieczne, użyj adresu w rejestrze MAR, aby z tego miejsca pamięci pobrać dane i umieścić je w rejestrze MBR (i w razie potrzeby też w akumulatorze). Następnie wykonaj polecenie `MBR ← M[MAR]` i wykonaj rzeczywistą instrukcję.

Cykl ten został przedstawiony na rysunku 4.11.

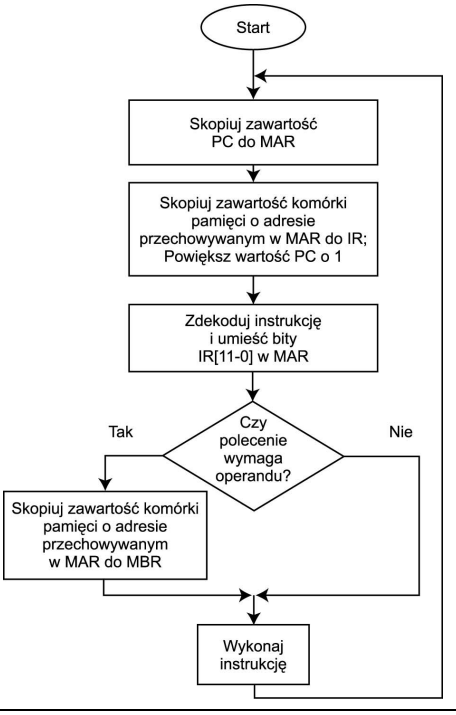
Zauważ, że mimo iż współczesne komputery mają duże zbiory rozkazów, długie polecenia i wiele pamięci, potrafią wykonywać miliony instrukcji w mgnieniu oka.

4.3.2. Przerwania a obsługa I/O

Wejściu i wyjściu poświęcony jest rozdział 7. Jednak już w tym momencie zdecydowałyśmy się omówić niektóre podstawowe pojęcia, aby upewnić się, że rozumiesz już cały proces wykonywania programu.

MARIE ma dwa rejestry do obsługi operacji wejścia i wyjścia. Rejestr wejściowy przechowuje dane przesyłane z urządzeń wejściowych do komputera, a rejestr wyjściowy przechowuje dane gotowe do wysłania do urządzeń wyjściowych. Bardzo ważne jest umiejętne posługiwanie się tymi rejestrami w czasie. Jeżeli na przykład wprowadzasz dane wejściowe za pośrednictwem klawiatury i piszesz bardzo szybko, komputer musi być w stanie przeczytać każdy znak, który trafia do rejestru wejściowego. Jeśli inny znak wpłynie do tego rejestru, zanim komputer przekształci obecny znak, ów bieżący znak zostanie utracony. Bardziej prawdopodobne jest, że skoro procesor jest bardzo szybki, a zapis na klawiaturze bardzo wolny, procesor może odczytać ten sam znak z rejestru wejściowego wielokrotnie. Konieczne jest uniknięcie obu tych sytuacji.

MARIE rozwiązuje te problemy, używając *wejścia-wyjścia sterowanego przerwaniami*. (Szczegółowe omówienie różnych typów wejścia-wyjścia można znaleźć w rozdziale 7.). Kiedy procesor wykonuje instrukcję wejścia lub wyjścia, jest o tym informowane odpowiednie urządzenie



RYSUNEK 4.11. Cykl „pobierz, dekoduj, wykonaj”

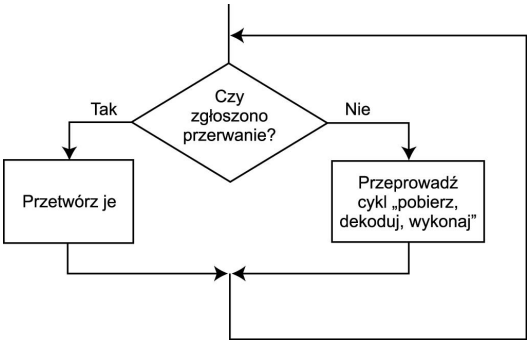
I/O. Dopóki nie przygotuje się ono do współpracy z procesorem, CPU wykonuje inne czynności. W tym czasie urządzenie wysyła do procesora sygnał przerwania. Procesor obsługuje wówczas to przerwanie, po którym kontynuuje normalny cykl „pobierz, dekoduj, wykonaj”. Proces ten wymaga:

- Sygnału (przerwania) od urządzenia wejścia-wyjścia do procesora, informującego o tym, że wprowadzanie lub przyjmowanie danych zostało zakończone.
- Środków, które pozwolą procesorowi zawiesić wykonywanie operacji „pobierz, dekoduj, wykonaj” i zająć się chwilową obsługą przerwania.

Większość komputerów sprawdza, czy przypadkiem nie wystąpiło przerwanie, na początku każdego zwykłego cyklu zegarowego. Jeśli takie zdarzenie miało miejsce, to przerwanie jest obsługiwane, po czym komputer kontynuuje standardowy cykl „pobierz, dekoduj, wykonaj”. W przeciwnym przypadku procesor zwyczajnie kontynuuje wykonywanie programu. Przebieg wykonywania poszczególnych instrukcji prześledziłyśmy na rysunku 4.12.

Zazwyczaj urządzenie wejścia lub wyjścia wysyła sygnał przerwania, używając specjalnego rejestru, zwanego rejestrem stanu lub rejestrem flagowym. Jest w nim wówczas ustawiany specjalny bit, który sygnalizuje wystąpienie przerwania. Może on być włączany na przykład zaraz po tym, jak użytkownik skończy wprowadzać dane z klawiatury. Procesor sprawdza wartość tego bitu na początku każdego cyklu zegara. Jeśli bit jest ustawiony, procesor przerywa swoją pracę i wczytuje dane; jeśli nie jest ustawiony — procesor, nie przerywając pracy, dalej wykonuje normalny cykl „pobierz, dekoduj, wykonaj”, przetwarzając polecenia aktualnie wykonywanego programu.

Kiedy procesor zauważy ustawiony bit, wykonuje określoną procedurę, stosownie do rodzaju zgłoszonego przerwania. Pamiętaj, że przerwania wejścia-wyjścia nie są jedynymi, jakie mogą wystąpić w trakcie wykonywania programu. Czy kiedykolwiek wprowadziłeś kombinację klawiszy



RYSUNEK 4.12. Zmodyfikowany cykl przetwarzania instrukcji, w którym sprawdzamy, czy nie zgłaszano żadnych przerwań

Ctrl+Break lub *Ctrl+C*, aby przerwać wykonywanie programu? Jest to kolejny przykład nadania sygnału przerwania. Wyróżniamy przerwania zewnętrzne, generowane przez zdarzenia pochodzące z zewnątrz (takie jak błędy odczytu bądź zapisu danych, czy też brak zasilania), wewnętrzne, wygenerowane przez jakąś wyjątkową sytuację zaistniałą w programie (taką jak na przykład dzielenie przez zero, błąd przepełnienia lub naruszenie ochrony), oraz przerwania programowane, generowane na skutek wywoływania określonych instrukcji (takich jak polecenie nakazujące przenieść program z warstwy użytkownika na warstwę jądra systemu operacyjnego).

Bez względu na to, jaki typ przerwania wystąpi, efekt przez niego wywołany będzie zawsze ten sam. Po tym, jak procesor przyjmie żądanie obsługi przerwania, określany jest (zazwyczaj sprzętowo) adres procedury obsługi, po czym jest ona wywoływana (w podobny sposób, jak procedury), a procesor przerywa chwilowo te operacje, które wykonywał do tej pory i czeka na jej zakończenie. Dopiero wówczas może przestawić się z powrotem na wykonywanie programu, którym zajmował się przed zgłoszeniem przerwania. Oczywiście musi powrócić dokładnie do tego samego miejsca, w którym skończył pracę. Dlatego też przed wywołaniem procedury obsługi przerwania konieczne jest zachowanie zawartości licznika programu, zawartości wszystkich innych rejestrów oraz stanu wszystkich operacji z danej chwili. Po zakończeniu obsługi przerwania procesor przywraca wszystko do pierwotnego stanu i zaczyna na nowo wczytywać, dekodować i wykonywać rozkazy.

4.4. Prosty program

Przedstawimy teraz prosty program napisany dla MARIE. W podrozdziale 4.6 znajdziesz kilka innych przykładów, ilustrujących zdolności tej niewielkiej architektury. Może ona nawet być użyta do wykonywania programów zawierających procedury, różnego rodzaju pętle czy instrukcje wyboru.

Nasz pierwszy program dodaje do siebie dwie liczby (obie zapisane w głównej pamięci) i przechowuje ich sumę w pamięci. (Na razie pominiemy operacje wejścia-wyjścia).

Tabela 4.3 przedstawia program zapisany w języku asemblera, który wykona powyższe operacje oraz odpowiadający mu program, zapisany w języku maszynowym. Faktyczny program w języku asemblera znajduje się w kolumnie *Instrukcja*. Wiemy już, że cykl „pobierz, dekoduj, wykonaj” zaczyna się od wczytania pierwszego rozkazu programu, który znajdujemy, wczytując adres pierwszego polecenia do licznika rozkazów. Dla uproszczenia załóżmy, że nasze programy dla MARIE są zawsze wczytywane do obszaru pamięci zaczynającego się od komórki o adresie 100 (szesnastkowo).

TABELA 4.3. Program dodający do siebie dwie liczby

Adres w systemie szesnastkowym	Instrukcja	Binarna zawartość komórek pamięci	Szesnastkowa zawartość komórek pamięci
100	Załaduj 104	0001000100000100	1104
101	Dodaj 105	0011000100000101	3105
102	Zachowaj 106	0010000100000110	2106
103	Zatrzymaj	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000	0000

Instrukcje wymienione w kolumnie *Binarna zawartość komórek pamięci* to kod w języku maszynowym.

Często łatwiej jest odczytywać wartości szesnastkowe niż dwójkowe, dlatego przedstawiłyśmy zawartość pamięci również w tej postaci.

Program ten wczytuje wartość 0023_{16} (dziesiętnie 35) do akumulatora, a następnie dodaje do niej wartość $FFE9_{16}$ (dziesiętnie −23), znajdującą się pod adresem 105. W wyniku tej operacji w akumulatorze otrzymamy wartość 12. Polecenie `Store` nakazuje zachować tę wartość w pamięci pod adresem 106. Po zakończeniu wykonywania programu binarną zawartością komórki 106 staje się 0000000000001100, co w systemie szesnastkowym oznacza 000C, a w dziesiętnym 12. Rysunek 4.13 pokazuje zawartość rejestrów w trakcie wykonywania programu.

Ostatnia instrukcja RTN w części c umieszcza wynik sumowania w odpowiednim miejscu w pamięci. „Dekoduj IR[15-12]" oznacza po prostu, że polecenie musi być rozkodowane, aby wiadomo było, co powinno zostać wykonane. Proces dekodowania instrukcji może być realizowany sprzętowo lub za pośrednictwem mikrokodu. Oba sposoby zostały bardziej szczegółowo opisane w podrozdziale 4.7.

Zauważ, że występuje bezpośrednia korelacja pomiędzy językiem asemblera a poleceniami języka maszynowego. Zdecydowanie ułatwia to przekształcenie jednego języka w drugi. Za pomocą przedstawionych w niniejszym rozdziale tablic poleceń powinieneś być w stanie własnoręcznie zasemblować każdy z naszych przykładowych programów. Dlatego właśnie od tej pory używać będziemy jedynie kodu w języku asemblera. Zanim jednak przedstawimy więcej przykładowych programów, musimy omówić proces asemblacji, czyli tłumaczenia programu na język maszynowy.

4.5. Omówienie asemblerów

W przypadku programu przedstawionego w tabeli 4.3 przekształcenie instrukcji języka asemblera, dajmy na to `Load 104`, na kod w języku maszynowym (110416) jest całkiem proste. Ale po co właściwie zajmować się tymi przekształceniami? Dlaczego po prostu nie używać kodów maszynowych? Mimo iż bezpośrednie wprowadzanie do komputera rozkazów maszynowych pozwala osiągnąć bardzo dużą wydajność, człowiekowi trudno jest rozumieć program zapisany w postaci następujących po sobie zer i jedynek. Wolimy słowa i symbole aniżeli długie liczby. Dlaczego nie mielibyśmy więc stworzyć programu, który będzie przeprowadzał tę konwersję za nas? Na szczęście takie programy już istnieją. Nazywamy je *asemblerami*.

a) wczytaj 104:

Krok	RTN	PC	IR	MAR	MBR	AC
(wartości początkowe)		100	----	----	----	----
Pobierz	MAR ← PC	100	----	100	----	----
	IR ← M[MAR]	100	1104	100	----	----
	PC ← PC + 1	101	1104	100	----	----
Dekoduj	MAR ← IR[11-0]	101	1104	104	----	----
	(Dekoduj IR[15-12])	101	1104	104	----	----
Pobierz operand	MBR ← M[MAR]	101	1104	104	0023	----
Wykonaj	AC ← MBR	101	1104	104	0023	0023

b) dodaj 105:

Krok	RTN	PC	IR	MAR	MBR	AC
(wartości początkowe)		101	1104	104	0023	0023
Pobierz	MAR ← PC	101	1104	101	0023	0023
	IR ← M[MAR]	101	3105	101	0023	0023
	PC ← PC + 1	102	3105	101	0023	0023
Dekoduj	MAR ← IR[11-0]	102	3105	105	0023	0023
	(Dekoduj IR[15-12])	102	3105	105	0023	0023
Pobierz operand	MBR ← M[MAR]	102	3105	105	FFE9	0023
Wykonaj	AC ← AC + MBR	102	3105	105	FFE9	000C

c) zapisz 106

Krok	RTN	PC	IR	MAR	MBR	AC
(wartości początkowe)		102	3105	105	FFE9	000C
Pobierz	MAR ← PC	102	3105	102	FFE9	000C
	IR ← M[MAR]	102	2106	102	FFE9	000C
	PC ← PC + 1	103	2106	102	FFE9	000C
Dekoduj	MAR ← IR[11-0]	103	2106	106	FFE9	000C
	(Dekoduj IR[15-12])	103	2106	106	FFE9	000C
Pobierz operand	(niekonieczne)	103	2106	106	FFE9	000C
Wykonaj	MBR ← AC	103	2106	106	000C	000C
	M[MAR] ← MBR	103	2106	106	000C	000C

RYSUNEK 4.13. Śledzenie przebiegu wykonywania programu, który dodaje do siebie dwie liczby

4.5.1. Jakie funkcje właściwie pełni asembler?

Zadaniem asemblera jest przekształcanie kodu zapisanego w języku asemblera (który składa się z łatwych do zapamiętania „słów") na program w języku maszynowym, mający w całości postać binarną (jest to ciąg zer i jedynek). Asemblery wczytują program napisany przez programistę

w języku asemblera (który stanowi tak naprawdę symboliczną reprezentację liczb dwójkowych) i zamieniają go na jego odpowiednik w postaci ciągu rozkazów dwójkowych, czyli na kod maszynowy. Asembler wczytuje *kod źródłowy* i tworzy z niego *plik obiektowy* (kod maszynowy).

Zastąpienie kodów operacji (kodów rozkazów) prostymi nazwami alfanumerycznymi bardzo ułatwia programowanie. Ponadto, zamiast odwoływać się bezpośrednio do adresów pamięci, możemy posługiwać się łatwymi do zapamiętania *etykietami*, co jeszcze bardziej ułatwia pisanie programów. Jeżeli na przykład nasz program ma za zadanie dodać do siebie dwie liczby, możemy wskazać ich położenie poprzez podanie ich etykiet, co zwalnia nas z obowiązku zapamiętywania adresów wszystkich operandów danej instrukcji. Pokazałyśmy to w tabeli 4.4.

TABELA 4.4. Przykład zastosowania etykiet

Adres		**Instrukcja**	
	100	Load	X
	101	Add	Y
	102	Store	Z
	103	Halt	
X,	104	0023	
Y,	105	FFE9	
Z,	106	0000	

Gdy w danej instrukcji zamiast fizycznego adresu pojawia się etykieta, asembler i tak zastępuje ją faktycznym adresem. W większości asemblerów możliwe jest stosowanie etykiet. W przypadku każdego z nich istnieją osobne wymagania co do sposobu zapisu instrukcji, w tym umieszczania etykiet. Długość etykiety może być na przykład ograniczona do trzech znaków, a asembler może dodatkowo wymagać, aby znajdowała się ona w pierwszym polu instrukcji. W przypadku MARIE każda etykieta musi być zakończona średnikiem.

Etykiety są bardzo ciekawym rozwiązaniem z punktu widzenia programisty, ale wymagają wiele pracy od asemblera, ponieważ w celu dokonania translacji musi on przejrzeć cały kod programu dwukrotnie. Za pierwszym razem buduje on tak zwaną *tablicę symboli*. W przypadku omówionego już przykładu zawierałaby ona trzy symbole: X, Y i Z. Ponieważ asembler czyta kod od góry do dołu, nie może on za jednym podejściem przetłumaczyć całego programu zapisanego w języku asemblera na język maszynowy, ponieważ na podstawie samej etykiety nie może on ustalić, gdzie znajduje się ta część danej instrukcji, która zawiera dane. Jeżeli jednak zbuduje tablicę symboli, to analizując kod po raz drugi, będzie już mógł uzupełnić brakujące informacje.

Gdy asembler będzie czytał przytoczony powyżej program po raz pierwszy, utworzy następującą tablicę symboli:

X	104
Y	105
Z	106

Zacznie też tłumaczyć instrukcje, ale będą one jeszcze niekompletne:

1			X
3			Y
2			Z
7	0	0	0

Przy drugim „przejściu" asembler wykorzystuje tablicę symboli do wstawienia w odpowiednie miejsca adresów i utworzenia odpowiadających im instrukcji języka maszynowego. Tak więc za drugim razem asembler „wie" już, że instrukcja zlokalizowana jest pod adresem 104 i wstawi w miejsce `X` wartość 104. Podobnie będzie z instrukcjami `Y` i `Z`, co da w wyniku:

1	1	0	4
3	1	1	5
2	1	0	6
7	0	0	0

Ponieważ większości ludzi odczytywanie liczb zapisanych w systemie szesnastkowym sprawia trudność, większość asemblerów pozwala wprowadzać dodatkowo wartości dwójkowe i dziesiętne. Zazwyczaj stosuje się przy tym jakąś *dyrektywę asemblera* (wskazówkę dla asemblera, która nie jest tłumaczona na kod maszynowy), która określa, w jakim systemie zapisana jest dana liczba. W języku maszynowym MARIE liczby dziesiętne będziemy oznaczali dyrektywą `DEC`, a szesnastkowe `HEX`. Przepiszmy więc program z tabeli 4.4 do nowej postaci (tabela 4.5).

TABELA 4.5. Przykład wykorzystywania dyrektyw w charakterze stałych

Adres		**Instrukcja**	
	100	Load	X
	101	Add	Y
	102	Store	Z
	103	Halt	
X,	104	DEC	35
Y,	105	DEC	-23
Z,	106	HEX	0000

Zamiast wymagać faktycznych wartości dwójkowych (zapisanych szesnastkowo), możemy wprowadzić wartość dziesiętną, posługując się dyrektywą `DEC`. Asembler rozpozna tę wskazówkę i odpowiednio przekształci wpisaną liczbę przed umieszczeniem jej w pamięci. Jeszcze raz podkreślmy, że dyrektywy nie są tłumaczone na język maszynowy. Służą one jedynie do poinstruowania asemblera, jakie działania ma podejmować.

Innym rodzajem dyrektywy, który występuje we wszystkich językach programowania, jest *znacznik komentarza*. Jest to specjalny znak, który informuje asembler (lub kompilator), że należy zignorować cały tekst, który po nim występuje. W przypadku MARIE takim separatorem jest ukośnik (`/`). Wszelkie znaki wpisane od miejsca, w którym on występuje, do końca wiersza będą ignorowane.

4.5.2. Po co używać asemblera?

Głównym celem, dla którego zapoznajemy Cię z językiem asemblera MARIE, jest chęć pokazania Ci zależności, jakie zachodzą między architekturą a językiem programowania. Zrozumienie tego, jak pisze się programy w języku asemblera, ogromnie ułatwia zrozumienie architektury (i na odwrót). Tak więc nie tylko poznajesz podstawy architektury komputera, ale też dowiadujesz się, jak dokładnie działa procesor i nabierasz szerszego oglądu tej konkretnej architektury, do której piszesz programy. Istnieje wiele sytuacji, w których znajomość asemblera jest przydatna.

Większość programistów zgadza się co do tego, że zazwyczaj 10% kodu programu wykorzystuje 90% czasu procesora. W przypadku aplikacji, które muszą być wykonywane tak szybko, jak to możliwe, często konieczne jest optymalizowanie tych 10%. Zazwyczaj wykonują to kompilatory, które wczytują kod zapisany w języku wysokiego poziomu (takim jak C++) i zamieniają go na język asemblera (który jest następnie tłumaczony na kod maszynowy). Kompilatory rozwijane są już od bardzo wielu lat i w większości przypadków świetnie spełniają swoją rolę. Jednak od czasu do czasu programiści muszą obejść któreś z ograniczeń stawianych przez języki wysokiego poziomu i samodzielnie wpłynąć na kształt kodu asemblerowego. W ten sposób można przyspieszyć działanie programu i zmniejszyć jego rozmiary. Mieszane podejście, w którym większa część programu powstaje w języku wysokiego poziomu, a pewna wydzielona część jest przepisywana w asemblerze, pozwala programiście wykorzystywać najlepsze cechy obu języków.

Czy istnieją zatem sytuacje, w których opłaca się pisać cały program w asemblerze? Jeżeli rozmiar lub czas reakcji programu ma kluczowe znaczenie, często język ten jest najlepszym wyborem. Dzieje się tak dlatego, że kompilatory często nie pozwalają oszacować, jak długo będzie trwać dana operacja — stąd programistom trudno jest ocenić, z jaką prędkością będą działały ich programy. Język asemblera stawia programistę bliżej architektury i daje mu tym samym większą kontrolę nad komputerem. W sytuacji, gdy programista chce wykonać operację, która w językach wysokiego poziomu nie jest po prostu możliwa, asembler staje się jedynym rozwiązaniem.

Świetnym przykładem urządzeń, w których zarówno rozmiar programu, jak i czas reakcji mają kluczowe znaczenie, są *systemy wbudowane*. Są to komputery, które integruje się z innymi urządzeniami, o innym obszarze zastosowań. Systemy osadzone muszą szybko reagować na czynniki zewnętrzne i często umieszczane są w urządzeniach o ograniczonych zasobach. Mają one wykonywać albo tylko jedną instrukcję, albo bardzo ograniczony ich zbiór. Istnieje spore prawdopodobieństwo, że jednego z takich systemów używasz na co dzień. Systemy osadzone znaleźć można między innymi w elektronice użytkowej (kamerach cyfrowych i analogowych, telefonach komórkowych, palmtopach, interaktywnych grach), produktach AGD (takich jak zmywarki, kuchenki mikrofalowe czy pralki), samochodach (zwłaszcza w systemach sterujących pracą silnika i w mechanizmach ABS), urządzeniach medycznych (takich jak skanery CAT czy monitory pracy serca) i urządzeniach przemysłowych (sterujących określonymi procesami).

W przypadku systemów osadzonych jakość oprogramowania ma kluczowe znaczenie. Musi ono reagować na żądania w ściśle określonym czasie i mieć zadane rozmiary. Właśnie dlatego tak często są one pisane w asemblerze.

4.6. Rozszerzanie naszego zbioru rozkazów

Mimo iż zbiór rozkazów MARIE jest na tyle rozbudowany, że możemy napisać dowolny program, dodanie kilku nowych instrukcji ułatwiłoby programowanie. Na kod operacji zarezerwowaliśmy 4 bity, co oznacza, że możemy posługiwać się 16 różnymi instrukcjami, a na razie mamy ich tylko 9. Rozszerzmy więc nasz zbiór rozkazów o instrukcje wymienione w tabeli 4.6.

TABELA 4.6. Rozszerzony zbiór rozkazów MARIE

Numer instrukcji (szesnastkowo)	Instrukcja	Znaczenie
0	JnS X	Ustaw PC na adres X i skocz do X + 1
A	Clear	Umieść w AC same zera
B	AddI X	Dodaj „niebezpośrednio”: przejdź do adresu X. Użyj wartości przechowywanej pod *X* w charakterze adresu operandu, który ma zostać dodany do AC
C	JumpI X	Wykonaj „niebezpośredni” skok: przejdź do adresu X. Użyj wartości przechowywanej pod X w charakterze adresu lokalizacji, do której należy skoczyć

Instrukcja JnS (ang. *jump-and-store*) pozwala nam zachować wskaźnik do instrukcji powrotnej i ustawić PC na inną instrukcję. Dzięki temu będziemy mogli wywoływać procedury i inne podprogramy, po których wykonaniu sterowanie będzie wracało do miejsca w programie, w którym nastąpiło wywołanie. Instrukcja Clear umieszcza w akumulatorze same zera. Pozwala to zaoszczędzić cykle zegara, które byłyby potrzebne, gdybyśmy chcieli wczytywać operand 0 z pamięci.

Instrukcja AddI (podobnie jak JumpI) wykorzystuje inny *tryb adresowania*. We wszystkich dotychczas poznanych instrukcjach zakładano, że wartość przechowywana w części instrukcji przeznaczonej na dane była *bezpośrednim adresem* potrzebnego operandu. Instrukcja AddI wykorzystuje tryb *adresowania pośredniego* (więcej na temat trybów adresowania powiemy w rozdziale 5.). Zamiast traktować wartość znajdującą się w lokalizacji *X* tak, jak gdyby była ona adresem, przyjmujemy, że jest to wskaźnik do nowego miejsca w pamięci, w którym znajdują się dane, które chcemy wykorzystać w naszej instrukcji. Na przykład jeżeli mamy instrukcję AddI 400, to musimy najpierw przejść do lokalizacji 400 i odczytać znajdującą się tam wartość. Powiedzmy, że będzie to 240. Będzie to oznaczało, że należy przejść do adresu 240 i stąd pobrać właściwy operand dla naszej instrukcji. Można więc powiedzieć, że rozszerzyliśmy nasz język o możliwość stosowania wskaźników.

W notacji „rejestr-przesunięcie” nasze nowe instrukcje wyglądają następująco:

JnS

```
MBR ← PC
MAR ← X
M[MAR] ← MBR
MBR ← X
AC ← 1
AC ← AC + MBR
PC ← AC
```

Clear

```
AC ← 0
```

AddI X

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← M[MAR]
AC ← AC + MBR
```

JumpI X

```
MAR ← X
MBR ← M[MAR]
PC ← MBR
```

W tabeli 4.7 podsumowałyśmy cały zbiór rozkazów MARIE.

Przyjrzyjmy się teraz kilku przykładom, w których korzystałyśmy ze wszystkich dostępnych instrukcji.

Przykład 4.1. Oto przykład, w którym wykorzystujemy pętlę do zsumowania pięciu liczb:

```
    Adres  Instrukcja  Komentarze
      100  Load        Addr /Wczytaj adres pierwszej dodawanej liczby
      101  Store       Next /Zachowaj ten adres jako nasz wskaźnik Next
      102  Load        Num  /Wczytaj liczbę elementów do zsumowania
      103  Subt        One  /Zmniejsz wartość o 1
      104  Store       Ctr  /Zachowaj tę wartość w Ctr. Będzie ona kontrolowała pętlę
      105  Clear            /Wyczyść AC
Loop, 106  Load        Sum  /Wczytaj Sum do AC
      107  AddI        Next /Dodaj wartość, na którą wskazuje lokalizacja Next
      108  Store       Sum  /Zachowaj tę sumę
      109  Load        Next /Wczytaj Next
      10A  Add         One  /Zwiększ ją o 1, tak aby wskazywała na następny adres
      10B  Store       Next /Zachowaj ją w naszym wskaźniku Next
      10C  Load        Ctr  /Wczytaj zmienną sterującą pętli
      10D  Subt        One  /Odejmij jeden od zmiennej sterującej pętli
      10E  Store       Ctr  /Zapisz tę nową wartość w zmiennej sterującej pętli
      10F  Skipcond    000   /Jeżeli zmienna sterująca < 0, pomiń następną instrukcję
      110  Jump        Loop /W przeciwnym przypadku przejdź do Loop
      111  Halt             /Zakończ program
Addr, 112  Hex         118  /Liczby, które mają być zsumowane, zaczynają się od adresu 108
Next, 113  Hex         0    /Wskaźnik do następnej liczby do zsumowania
Num,  114  Dec         5    /Liczba wartości, które trzeba dodać
Sum,  115  Dec         0    /Suma
Ctr,  116  Hex         0    /Zmienna sterująca pętli
One,  117  Dec         1    /Wykorzystywana do zwiększania i zmniejszania wartości o 1
      118  Dec         10   /Wartości, które mają być zsumowane
      119  Dec         15
      11A  Dec         20
      11B  Dec         25
      11C  Dec         30
```

TABELA 4.7. Pełny zbiór rozkazów MARIE

Kod operacji	Instrukcja	RTN
0000	JnS X	MBR ← PC
		MAR ← X
		M[MAR] ← MBR
		MBR ← X
		AC ← 1
		AC ← AC + MBR
		PC ← AC
0001	Load X	MAR ← X
		MBR ← M[MAR], AC ← MBR
0010	Store X	MAR ← X, MBR ← AC
		M[MAR] ← MBR
0011	Add X	MAR ← X
		MBR ← M[MAR]
		AC ← AC + MBR
0100	Subt X	MAR ← X
		MBR ← M[MAR]
		AC ← AC - MBR
0101	Input	AC ← InREG
0110	Output	OutREG ← AC
0111	Halt	
1000	Skipcond	If IR[11-10] = 00 then
		if AC < 0 then PC ← PC + 1
		Else If IR[11-10] = 01 then
		If AC = 0 then PC ← PC + 1
		Else If IR[11-10] = 10 then
		If AC > 0 then PC ← PC + 1
1001	Jump X	PC ← IR[11-0]
1010	Clear	AC ← 0
1011	AddI X	MAR ← X
		MBR ← M[MAR]
		MAR ← MBR
		MBR ← M[MAR]
		AC ← AC + MBR
1100	JumpI X	MAR ← X
		MBR ← M[MAR]
		PC ← MBR

Mimo iż komentarze wydają się wszystko wyjaśniać, prześledźmy przykład 4.1. Przypomnij sobie, że w tablicy symboli przechowywane są pary [etykieta, lokalizacja]. Instrukcja `Load Addr` staje się instrukcją `Load 112`, ponieważ fizycznie `Addr` mieści się w pamięci pod adresem 112. Następnie wartość 118 (przechowywana pod adresem `Addr`) jest zapisywana pod zmienną `Next`. Jest to wskaźnik, który pozwala nam „przejść” przez pięć wartości, które chcemy dodać (zlokalizowanych pod adresami 118, 119, 11A, 11B i 11C). Zmienna `Ctr` zapamiętuje liczbę wykonanych iteracji

pętli. Ponieważ warunkiem zakończenia pętli jest to, aby wartość ta była ujemna, zaczynamy od zmniejszenia jej o 1. Następnie do AC wczytywana jest suma, której początkowa wartość wynosi 0. Rozpoczyna się wykonywanie pętli, w której `Next` pełni rolę adresu danych, które chcemy dodać do AC. Wyrażenie `Skipcond` zatrzymuje pętlę, gdy `Ctr` ma wartość ujemną, przeskakując bezwarunkowy skok do początku pętli. Z chwilą napotkania instrukcji `Halt` program jest zatrzymywany.

Przykład 4.2 pokazuje, jak możesz wykorzystać instrukcje `Skipcond` i `Jump` w celu dokonania wyboru. Mimo iż w tym przypadku prezentujemy konstrukcję `if/else`, po niewielkich modyfikacjach będziesz w stanie zaimplementować polecenie `if/then`, a nawet `case` (`switch`).

Przykład 4.2. Ten przykład ilustruje dokonywanie wyboru za pośrednictwem konstrukcji `if/else`. W szczególności implementuje on następującą konstrukcję:

```
if X = Y then
   X := X x 2
else
   Y := Y - X;
```

```
     Adres  Instrukcja      Komentarze
If,    100  Load     X      /Wczytaj pierwszą wartość
       101  Subt     Y      /Odejmij wartość Y i zachowaj wynik w AC
       102  Skipcond 400    /Jeśli AC = 0, pomiń następną instrukcję
       103  Jump     Else   /Jeśli AC nie jest równe 0, skocz do części Else
Then,  104  Load     X      /Ponownie wczytaj X, aby można było podwoić jego wartość
       105  Add      X      /Podwój wartość X
       106  Store    X      /Zapisz nową wartość
       107  Jump     Endif  /Przeskocz nad częścią Else do końca If
Else,  108  Load     Y      /Rozpocznij część Else, wczytując Y
       109  Subt     X      /Odejmij X od Y
       10A  Store    Y      /Zapisz w Y wartość Y – X
Endif, 10B  Halt             /Zakończ program
X      10C  Dec      12     /Wartości X i Y
Y      10D  Dec      20
```

Przykład 4.3 pokazuje, że instrukcje `JnS` i `JumpI` pozwalają na stosowanie podprogramów (procedur). Ten program zawiera wyrażenie `END`, które jest kolejnym przykładem dyrektywy asemblera. Informuje ona asembler, w którym miejscu kończy się program. Inne możliwe dyrektywy to na przykład wyrażenia, które informują asembler, gdzie można znaleźć pierwszą instrukcję programu, w jaki sposób należy skonfigurować program oraz czy bloki kodu są procedurami.

Przykład 4.3. Ten przykład ilustruje wykorzystanie prostej podprocedury do podwojenia dowolnej liczby:

```
100  Load   X     /Wczytaj pierwszą liczbę, którą należy podwoić
101  Store  Temp  /Użyj Temp w charakterze parametru, za pośrednictwem którego zostanie
                    przekazana wartość do Subr
102  JnS    Subr  /Zachowaj adres powrotny, skocz do procedury
103  Store  X     /Zapisz pierwszą podwojoną liczbę
104  Load   Y     /Wczytaj drugą liczbę, którą należy podwoić
105  Store  Temp  /Użyj Temp w charakterze parametru, za pośrednictwem którego zostanie
                    przekazana wartość do Subr
```

```
       106  JnS    Subr  /Zachowaj adres powrotny, skocz do procedury
       107  Store  Y     /Zapisz drugą podwojoną liczbę
       108  Halt         /Zakończ program
X,     109  Dec    20
y,     10A  Dec    48
Temp,  10B  Dec    0
Subr,  10C  Hex    0     /Przechowaj tu adres powrotny
       10D  Clear        /Wyczyść AC, ponieważ był on zmodyfikowany przez JnS
       10E  Load   Temp  /Podprocedura podwajająca liczby
       10F  Add    Temp  /AC przechowuje teraz podwojoną wartość Temp
       110  JumpI  Subr  /Wróć do wywołującego kodu
            END
```

Wykorzystując prosty zbiór rozkazów MARIE, powinieneś być w stanie zaimplementować dowolne wyrażenie znane z języków programowania wysokiego poziomu, takie jak pętle czy konstrukcje `while`. Jest to zresztą tematem ćwiczeń zamieszczonych na końcu tego rozdziału.

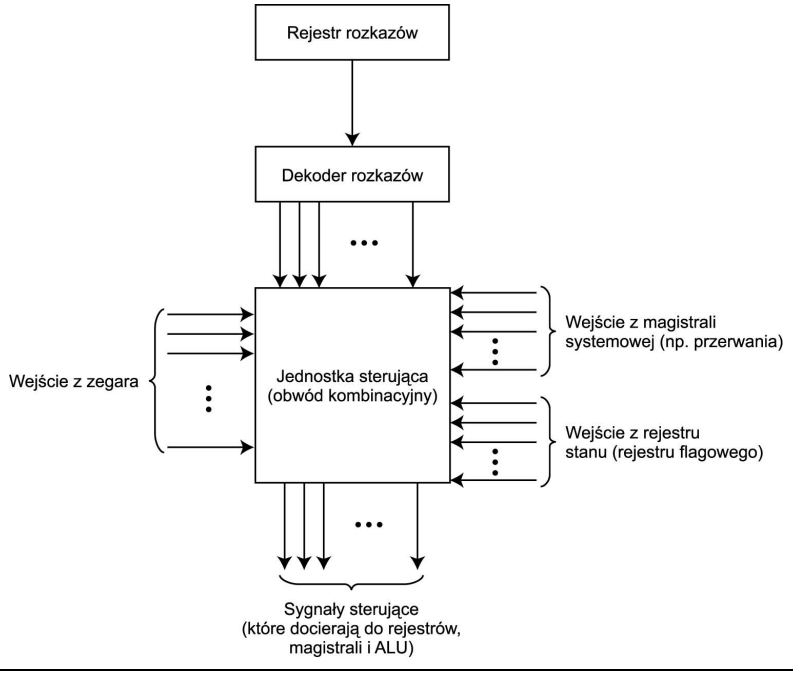
4.7. Dekodowanie — czysto sprzętowe czy realizowane za pośrednictwem mikrokodu?

Jak naprawdę działa jednostka sterująca? Dotychczas przyjmowaliśmy na wiarę, że funkcjonuje ona tak, jak opisywałyśmy, wiedząc jedynie, że dla każdej instrukcji zmusza ona CPU do prawidłowego wykonywania określonej sekwencji kroków. W rzeczywistości, aby wszystko przebiegało jak należy, konieczne jest wykorzystywanie sygnałów kontrolnych, wysyłanych na wejścia różnych cyfrowych podzespołów (przypomnij sobie podzespoły, które omawiałyśmy w rozdziale 3.). Na przykład wykonując instrukcję `Add` asemblera MARIE, zakładamy, że komputer wykona dodawanie, ponieważ sygnały sterujące pracą ALU są ustawiane na operację „dodaj”, a wynik zostanie umieszczony w AC. ALU ma wiele linii sterujących, które określają, jaką operację należy wykonać. Należy teraz odpowiedzieć sobie na pytanie, w jaki sposób do wszystkich tych linii są wysyłane sygnały.

Aby mieć pewność, że linie sterujące zostały skonfigurowane prawidłowo, możesz zastosować jedną z dwóch technik. Po pierwsze, możesz fizycznie połączyć wszystkie linie sterujące z instrukcjami języka maszynowego. Instrukcje rozbijane są wówczas na pola, a ich poszczególne bity są ze sobą łączone za pośrednictwem różnych podzespołów wykonujących operacje z zakresu logiki cyfrowej, które kierują pracą linii sterujących. Jest to tak zwane *sterowanie skonfigurowane na stałe*, czyli czysto sprzętowe (patrz rysunek 4.14).

Jednostka sterująca jest zaimplementowana sprzętowo. Może się ona składać na przykład z prostych bramek NAND, przerzutników i liczników. Potrzebujemy też specjalnego obwodu cyfrowego, który wykorzystuje w charakterze wejścia bity pochodzące z pól instrukcji przechowujących kody operacji, bity z rejestru stanu (rejestru flagowego), sygnały płynące z magistrali i sygnały zegara. Na wyjściu powinien on umieszczać sygnały sterujące pracą różnych podzespołów komputera. Na przykład do dekodowania kodu operacji można by użyć dekodera „4 do 16”. Wykorzystując zawartość rejestru IR i informację o stanie ALU, jednostka ta kontroluje rejestry, działanie ALU, wszystkie przesuwniki i dostęp do magistrali.

Zaletą czysto sprzętowej implementacji jednostki sterującej jest to, że jest ona bardzo szybka. Natomiast jej wadą jest złożoność oraz bardzo trudne projektowanie jej i modyfikowanie. Trudności te wynikają stąd, że zbiór rozkazów i logika sterująca są ze sobą bezpośrednio powiązane za pośrednictwem specjalnych obwodów. Gdy ktoś zaprojektuje komputer z czysto sprzętowym sterowaniem,



RYSUNEK 4.14. Czysto sprzętowa jednostka sterująca

a później zdecyduje się rozszerzyć jego zbiór instrukcji (tak, jak to uczyniliśmy z MARIE), będzie musiał wymienić jego fizyczne podzespoły. Wiąże się to z ogromnymi kosztami, ponieważ nie dość, że trzeba wówczas wyprodukować nowe chipy, to jeszcze wypadałoby zlokalizować i wymienić stare.

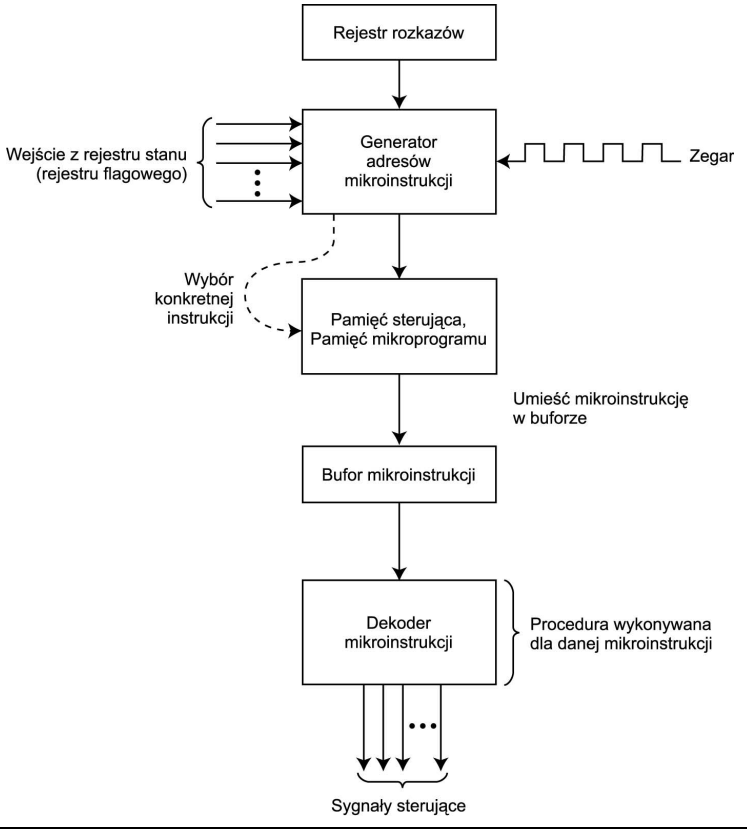
Istnieje też inne podejście, zwane *mikroprogramowaniem*, w którym sterowaniem zajmuje się specjalne oprogramowanie (patrz rysunek 4.15).

Wszystkie instrukcje maszynowe są wprowadzane do specjalnego programu, zwanego *mikroprogramem*, który zamienia je na odpowiednie sygnały sterujące. Mikroprogram pełni więc tak naprawdę rolę interpretera, napisanego w *mikrokodzie* i przechowywanego w oprogramowaniu *firmware* (w pamięci ROM, PROM lub EPROM, którą często nazywamy *pamięcią sterującą*). Program ten zamienia instrukcje maszynowe składające się z zer i jedynek na sygnały sterujące.

Zasadniczo na każdą instrukcję maszynową przypada w tym programie jedna procedura. Zaletą takiego podejścia jest to, że jeśli zbiór rozkazów wymaga modyfikacji, wystarczy uaktualnić mikroprogram. Nie są natomiast wymagane żadne fizyczne zmiany na poziomie sprzętowym.

Mikroprogramowanie daje dużą elastyczność, jest stosunkowo proste i pozwala tworzyć bardzo zaawansowane zbiory rozkazów. Jest to też podejście, które pozwala nam zadecydować, co chcemy zaimplementować sprzętowo, a co programowo. Jeżeli potrzebna Ci funkcja nie jest realizowana sprzętowo (na przykład w Twoim komputerze nie ma instrukcji realizującej mnożenie), możesz zaimplementować ją za pomocą mikrokodu. Wadą tego rozwiązania jest to, że wszystkie instrukcje muszą przechodzić przez dodatkowy poziom interpretacji, co spowalnia wykonywanie programu. Poza kosztami czasowymi ponosi się tu też koszty zakupu wyspecjalizowanych narzędzi, które umożliwiają programowanie. Bardziej szczegółowe porównanie rozwiązań czysto sprzętowych i tych opartych na mikrokodzie zamieściłyśmy w rozdziale 9.

Należy pamiętać, że niezależnie od tego, które rozwiązanie wybierzemy, czas wykonywania poszczególnych operacji musi być tak krótki, jak to tylko możliwe. Jednostka sterująca odpowiada za sygnały synchronizacyjne, które kontrolują wszystkie operacje przesyłania danych i inne działania



RYSUNEK 4.15. Sterowanie za pomocą mikroprogramu

komputery. Sygnały te są generowane sekwencyjnie z wykorzystaniem prostego licznika dwójkowego. W danej architekturze sygnały synchronizacyjne mogą wyglądać na przykład tak: T_1, T_2, T_3, T_4, T_5, T_6, T_7 oraz T_8. To one decydują o tym, kiedy komputer może wykonywać poszczególne czynności. Pobranie instrukcji może nastąpić tylko wtedy, gdy aktywny jest sygnał T_1, zaś pobranie operandu tylko wtedy, gdy aktywny jest T_4. Wiemy, że rejestry mogą zmieniać stan tylko w rytm impulsów zegara, ale dodatkowe ograniczone polega na tym, że może to następować jedynie po pojawieniu się konkretnego sygnału synchronizacyjnego. W rozdziale 3. omawialiśmy przykład pamięci, która zawierała linię sterującą Write Enable. Moglibyśmy na przykład wykonać logiczną operację AND tej pamięci i sygnału synchronizacyjnego, dzięki czemu zmiany zawartości pamięci mogłyby zachodzić jedynie w określonych odstępach czasu.

4.8. Przykładowe architektury prawdziwych komputerów

Architektura MARIE została zaprojektowana przede wszystkim tak, aby była jak najprostsza. Chciałyśmy przedstawić na jej przykładzie pewne podstawowe zagadnienia związane z architekturą komputerów, bez przytłaczania Cię masą mniej znaczących szczegółów. Mimo to architektura MARIE i jej asembler są wystarczająco zaawansowane, aby poradzić sobie z dowolnym problemem,

który rozwiązuje się dziś na komputerach o nowoczesnej architekturze, w językach wysokiego poziomu, takich jak C++, Ada czy Java. Wydaje nam się jednak, że nie byłbyś zbyt szczęśliwy, gdybyś musiał pisać program przeznaczony do tak mało wydajnej architektury, zwłaszcza, że pisanie go byłoby żmudne, podobnie jak usuwanie ewentualnych błędów. Wydajność MARIE byłaby dużo większa, gdybyśmy umieścili w CPU więcej rejestrów, w których można by przechowywać dane. Ułatwienie korzystania z nich programistom to już inna sprawa. Załóżmy, że programista MARIE chce na przykład korzystać z procedur przyjmujących parametry. Choć MARIE pozwala na stosowanie procedur (sterowanie może być kierowane do różnych partii kodu, które mogą być następnie wykonywane i z których następuje powrót do miejsca, w którym je wywołano), nie zawiera ona żadnego mechanizmu, który umożliwiałby przekazywanie parametrów. Oczywiście można pisać programy bez parametrów, ale wiemy, że dzięki zastosowaniu parametrów są one wydajniejsze (zwłaszcza jeśli chodzi o ponowne wykorzystywanie kodu), łatwiej się je pisze i można szybciej usuwać z nich błędy.

Aby umożliwić korzystanie z parametrów, MARIE musiałaby zawierać *stos* — strukturę danych przechowującą listę elementów, do których można się dostać tylko z jednej strony. Przypomina to stos talerzy w kuchni. Układasz talerze jeden na drugim, a zdejmujesz je (zazwyczaj) począwszy od najwyższego. Właśnie dlatego stosy nazywamy strukturami typu *ostatni na wejściu, pierwszy na wyjściu*. (Krótkie omówienie ważniejszych struktur danych znajduje się w dodatku A na końcu książki).

Jeżeli ograniczymy sposób dostępu do danych, będziemy mogli emulować stos, korzystając z określonej części głównej pamięci komputera. Jeżeli na przykład umówimy się, że komórki pamięci o adresach od 0000 do 00FF będą pełnić rolę stosu, którego wierzchołkiem będzie komórka 0000, to *dodawanie* do niego elementów będzie musiało odbywać się od góry, podobnie zresztą jak ich *zdejmowanie*. Jeżeli umieścimy na stosie wartość 2, będzie ona przechowywana w komórce 0000. Jeżeli następnie dorzucimy wartość 6, trafi ona do komórki 0001. Jeżeli zdejmiemy teraz ze stosu jeden element, otrzymamy 6. Specjalny wskaźnik stosu śledzi miejsce, do którego należy wprowadzać (bądź z którego należy zdejmować) każdy kolejny element.

MARIE ma wiele cech wspólnych z architekturami współczesnych komputerów, ale nie oddaje w pełni ich struktury. W kolejnych dwóch podrozdziałach omówimy rzeczywiste architektury i ich cechy, które — zgodnie z radą Leonarda da Vinci — zostały pominięte w MARIE. Zaczniemy od architektury Intela (rodziny x86 i Pentium), a następnie przejdziemy do architektury MIPS. Wybraliśmy je ze względu na to, że w pewnym sensie są one do siebie podobne, ale powstawały na bazie zupełnie innych założeń. Wszystkich członków rodziny x86 Intela nazywamy komputerami CISC (ang. *Complex Instruction Set Computer*), zaś komputery z rodziny Pentium i MIPS to urządzenia RISC (ang. *Reduced Instruction Set Computer*).

Komputery CISC mają większą liczbę instrukcji o zmiennej długości i skomplikowanej charakterystyce. Wiele z tych instrukcji jest naprawdę złożonych. Często wykonują one za jednym zamachem wiele operacji (możliwa jest na przykład sytuacja, w której *pojedyncza* instrukcja asemblera realizuje całą pętlę). Główny problem z komputerami CISC polega na tym, że pewien określony podzbiór instrukcji CISC w znaczący sposób spowalnia działanie całego komputera. Projektanci postanowili więc wrócić do mniej skomplikowanej architektury, w której mały (ale wyczerpujący) zbiór rozkazów jest zaimplementowany czysto sprzętowo, a instrukcje te wykonują się niezwykle szybko. Oznacza to, że odpowiedzialność za tworzenie wydajnego kodu dla ISA zostaje przesunięta na kompilator. Komputery, które wykonano w duchu tej filozofii, nazywamy komputerami RISC.

Skrót RISC, oznaczający komputer o zredukowanym zbiorze rozkazów, nie jest do końca adekwatny. To prawda, że liczba instrukcji została w tej architekturze zredukowana, ale powinniśmy pamiętać, że główny nacisk w komputerach RISC jest położony na to, aby instrukcje były jak najprostsze, dzięki czemu będą mogły być bardzo szybko wykonywane. Przy takim układzie każda instrukcja realizuje tylko jedną operację, a wszystkie rozkazy mają taką samą długość i mogą być składane tylko na kilka określonych sposobów. Ponadto wszystkie operacje arytmetyczne muszą

być wykonywane pomiędzy rejestrami (dane zapisane w pamięci nie mogą występować w charakterze operandów). Praktycznie wszystkie nowe zbiory rozkazów, powstałe po roku 1982 (niezależnie od docelowej architektury), są już zgodne z założeniami RISC bądź pewnego połączenia RISC i CISC. Obie te architektury omówimy dokładniej w rozdziale 9.

4.8.1. Architektury Intela

Firma Intel Corporation opracowała wiele różnych architektur, spośród których niektóre mogą być Ci znane. Pierwszy popularny chip Intela, 8086, wszedł na rynek w 1979 roku i był stosowany w komputerach IBM PC. Obsługiwał on 16-bitowe dane i operował na 20-bitowych adresach, co oznacza, że mógł zaadresować milion bajtów pamięci. W wielu pecetach stosowano bliskiego krewnego 8086 — 8-bitowy chip 8088, który był znacznie tańszy. Procesor 8086 był rozbity na dwie części — *jednostkę wykonawczą*, zawierającą rejestry ogólnego przeznaczenia i ALU, oraz *jednostkę interfejsu magistrali*, która zawierała kolejkę instrukcji, rejestry segmentowe i wskaźnik instrukcji.

8086 zawierał cztery 16-bitowe rejestry ogólnego przeznaczenia, nazwane AX (główny akumulator), BX (rejestr bazowy używany do rozszerzonego adresowania), CX (rejestr licznika) i DX (rejestr danych). Każdy z nich był podzielony na dwie części. Bardziej znacząca połowa była nazywana „wysoką", od ang. *high* (i oznaczana AH, BH, CH i DH), zaś mniej znacząca — „niską", od ang. *low* (i oznaczana AL., BL, CL i DL). Wiele instrukcji 8086 wymaga użycia konkretnego rejestru, ale generalnie rejestry mogły być też wykorzystywane do innych celów. 8086 zawierał też trzy rejestry wskaźnikowe: wskaźnik stosu (SP), który określał przesunięcie względem jego początku, wskaźnik bazowy (BP), za pomocą którego odwoływano się do parametrów umieszczanych na stosie i wskaźnik rozkazów (IP), który przechowywał adres następnej instrukcji (podobnie jak PC w MARIE). Dostępne były też dwa rejestry indeksowe — SI (indeks źródłowy), używany jako wskaźnik źródła w operacjach na łańcuchach, oraz DI (indeks docelowy), wykorzystywany w tych samych operacjach jako wskaźnik miejsca przeznaczenia. Ponadto 8086 zawierał *rejestr flagowy*, którego poszczególne bity oznaczały różne okoliczności, takie jak przepełnienie, parzystość, przerwanie przeniesienia itp.

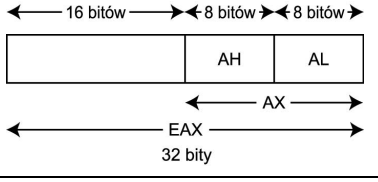
Program napisany w języku asemblera 8086 był dzielony na segmenty, czyli specjalne *obszary*, z których każdy przechowywał informacje innego rodzaju. Mieliśmy więc segment kodu (w którym umieszczany był program), segment danych (do którego trafiały dane) i segment stosu (przechowujący stos programu). Aby dostać się do informacji przechowywanych w którymś z tych segmentów, trzeba było podać przesunięcie, liczone względem jego początku. Dlatego też potrzebne były wskaźniki segmentów, które przechowywały adresy poszczególnych obszarów. Były to: rejestr segmentu kodu (CS), rejestr segmentu danych (DS) i rejestr segmentu stosu (SS). Istniał też czwarty rejestr segmentowy, nazywany dodatkowym rejestrem segmentowym (ES), który wykorzystywano przy wykonywaniu pewnych operacji na łańcuchach, które wymagały specjalnego adresowania pamięci. Wszystkie adresy podawano w postaci *segment:przesunięcie* (*xxx*:*yyy*), gdzie *xxx* było wartością przechowywaną w rejestrze segmentowym, a *yyy* było przesunięciem.

W roku 1980 Intel wprowadził na rynek chip 8087, który rozszerzał zbiór rozkazów 8086 o operacje na liczbach zmiennoprzecinkowych i wprowadzał 80-bitowy stos. Na rynek wprowadzono wiele układów, które miały praktycznie taką samą ISA, jak 8086, w tym procesor 80286 z 1982 roku, który potrafił zaadresować 16 milionów bajtów, oraz 80386 z roku 1985, który mógł zaadresować do 4 miliardów bajtów. 80386 był chipem 32-bitowym — pierwszym z rodziny określanej mianem IA-32 (od ang. *Intel Architecture, 32-bit*). Gdy Intel przeszedł z produkcji 16-bitowych procesorów 80286 na 32-bitowe chipy 80386, projektanci chcieli, aby nowa architektura była

zgodna ze starą, to znaczy żeby na nowszych, szybszych procesorach dawało się uruchamiać programy napisane dla starych komputerów. Na przykład programy działające na 80286 powinny też działać na 80386. Z tych względów Intel postanowił pozostawić tę samą podstawową architekturę i zbiór rejestrów (ale w każdym kolejnym modelu wprowadzano nowe funkcje, więc nie gwarantowano zgodności „w przód").

Konwencja nazewnicza obowiązująca dla rejestrów 80386, które zostały poszerzone z 16 do 32 bitów, przewidywała stosowanie przedrostka „E" (od ang. *extended* — rozszerzony). Tak więc zamiast rejestrów AX, BX, CX i DX mieliśmy EAX, EBX, ECX i EDX.

W ten sam sposób nazywano wszystkie pozostałe rejestry. Mimo to programiści mogli w dalszym ciągu używać starych nazw, takich jak AX, AL czy AH. Rysunek 4.16 pokazuje — na przykładzie rejestru AX — jak to było możliwe.



RYSUNEK 4.16. Rejestr EAX, rozbity na części

Procesory 80386 i 80486 były urządzeniami 32-bitowymi, o 32-bitowych szynach danych. W 80486 dołożono szybką *pamięć podręczną* (omawianą szczegółowo w rozdziale 6.), która znacznie podniosła wydajność procesora.

Rodzina *Pentium* zaczyna się od procesora o takiej samej nazwie (Intel przestał oznaczać rodziny procesorów liczbami, ponieważ nie mógł zastrzec ich jako znaków towarowych), który miał 32-bitowe rejestry i 64-bitową szynę danych i był zaprojektowany w technice *superskalarnej*. Oznacza to, że CPU zawierał kilka ALU i mógł wykonywać więcej niż jedną instrukcję w każdym cyklu zegara (były one wykonywane równolegle). W Pentium Pro wprowadzono dodatkowo przewidywanie skoków, zaś w Pentium II pojawiła się technologia MMX, która miała wspomagać obsługę multimediów, ale nie odniosła zbyt dużego sukcesu. Z kolei w Pentium III ulepszono obsługę grafiki 3D (za pośrednictwem nowych instrukcji zmiennoprzecinkowych). W początkowym okresie Intel wykorzystywał w swych procesorach klasyczną architekturę CISC. W nowszych procesorach Pentium II i III zastosowano rozwiązanie mieszane, w którym pozostawiono co prawda architekturę CISC, ale zastosowano jądra RISC, które tłumaczyły instrukcje CISC na RISC. Intel podążał cały czas za bieżącymi trendami, odchodząc od rozkazów CISC coraz bardziej w kierunku RISC.

Siódmą generację procesorów Intela zapoczątkował układ *Intel Pentium 4* (P4). Procesor ten różni się pod wieloma względami od swych poprzedników, ale dokładne omówienie jego budowy wykracza poza zakres tej książki. Wystarczy powiedzieć, że pracuje on przy częstotliwościach zegara na poziomie 1,4 GHz i wyższych, zbudowany jest z co najmniej 42 milionów tranzystorów i implementuje tak zwaną mikroarchitekturę Netburst. (Przed pojawieniem się P4 wszystkie procesory z rodziny Pentium miały tę samą *mikroarchitekturę*, czyli architekturę na poziomie poniżej zbioru instrukcji). Wykorzystano w niej wiele innowacyjnych technologii, takich jak hiperpotokowość (o potokach piszemy w rozdziale 5.), 400-megahercową (lub szybszą) magistralę systemową oraz wiele usprawnień pamięci podręcznej i operacji zmiennoprzecinkowych. Uczyniło to z P4 procesor świetnie nadający się do zastosowań multimedialnych.

W 2001 roku pojawił się pierwszy 64-bitowy procesor Intela, który zapoczątkował rodzinę IA-64. Procesor ten, o nazwie *Itanium*, obsługuje język programowania oparty na rejestrach i bardzo bogaty zbiór instrukcji. Zaimplementowano w nim też sprzętową emulację zbioru rozkazów IA-32/x86. Procesor ten ma 4 jednostki stało- i 2 zmiennoprzecinkowe, wiele pamięci podręcznej rozmieszczonej na 4 poziomach (które omówimy w rozdziale 6.), 128 rejestrów zmiennoprzecinkowych i wiele różnych rejestrów, odpowiedzialnych za wczytywanie instrukcji przy aktywnym przewidywaniu skoków. Itanium jest w stanie zaadresować do 16 GB pamięci.

Język asemblera danej architektury zdradza nam wiele informacji na jej temat. Aby porównać architekturę MARIE z architekturą Intela, wróćmy do przykładu 4.1, w którym zaprezentowałyśmy program MARIE wykorzystujący pętlę do zsumowania pięciu liczb. Przepiszemy teraz ten program w języku asemblera x86 (przykład 4.4). Zwróć uwagę na dodanie dyrektywy oznaczającej segment danych (DATA) i segment kodu (CODE).

Przykład 4.4. Program wykorzystujący pętlę do zsumowania pięciu liczb, napisany pod kątem procesora Pentium

```
        .DATA
Num1    EQU 10                  ; Num1 jest inicjalizowany wartością 10
        EQU 15                  ; Inicjalizowane są wszystkie słowa występujące po Num1
        EQU 20
        EQU 25
        EQU 30
Num     DB  5                   ; Inicjalizuj licznik pętli
Sum     DB  0                   ; Inicjalizuj sumę

        .CODE
        LEA EBX, Num1           ; Wczytaj adres Num1 do EBX
        MOV ECX, Num            ; Ustaw licznik pętli
        MOV EAX, 0              ; Zainicjalizuj sumę
        MOV EDI, 0              ; Zainicjalizuj przesunięcie
Start:  ADD EAX, [EBX+EDI *4]   ; Dodaj EBX-tą liczbę do EAX
        INC EDI                 ; Zwiększ przesunięcie o 1
        DEC ECX                 ; Zmniejsz licznik pętli o 1
        JG  Start               ; Jeżeli licznik jest większy od 0, wróć do etykiety Start
        MOV Sum, EAX            ; Zachowaj wynik w zmiennej Sum
```

Możemy podnieść czytelność tego programu, stosując wyrażenie `loop` (pętla), z tym że będzie on przez to coraz mniej przypominał program napisany w języku asemblera MARIE. Z punktu widzenia składni instrukcja `loop` przypomina instrukcję `jump`, ponieważ ona również wymaga podania etykiety. Powyższą pętlę można przepisać w następujący sposób:

```
        MOV  ECX, Num           ; Ustaw licznik
Start:  ADD  EAX, [EBX + EDI + 4]
        INC  EDI
        LOOP Start
        MOV  Sum, EAX
```

Wyrażenie `loop` w języku asemblera x86 przypomina konstrukcję `do…while`, znaną z języków C, C++ i Java. Różnica polega na tym, że nie podajemy jawnie zmiennej kontrolującej przebieg pętli. Domyślnie przyjmuje się, że jest to wartość przechowywana w rejestrze ECX. W trakcie

wykonywania instrukcji pętli procesor zmniejsza o jeden zawartość rejestru ECX i sprawdza, czy po tej operacji nie jest ona równa zero. Jeśli tak nie jest, sterowanie przekazywane jest do etykiety `Start`. W przeciwnym przypadku wykonywanie pętli jest przerywane. Wyrażenie `loop` jest przykładem instrukcji, której wprowadzenie ułatwia pracę programistom, ale nie jest niezbędne do zaimplementowania określonego algorytmu.

4.8.2. Architektury MIPS

Procesory z rodziny MIPS są prawdziwymi gwiazdami w swojej klasie, a ich projekt jest bardzo elastyczny. MIPS R3000, R4000, R5000, R8000 i R10000 to jedne z wielu nazw handlowych, zarejestrowanych przez firmę MIPS Technologies. Chipy MIPS stosowane są w systemach wbudowanych, komputerach (produkowanych np. przez Silicon Graphics) oraz wielu skomputeryzowanych zabawkach (w tym w urządzeniach firm Nintendo i Sony). Procesory MIPS wykorzystuje też bardzo ceniony producent routerów internetowych, firma Cisco.

Pierwszą architekturą zbioru rozkazów MIPS była MIPS I. Po niej wprowadzono MIPS II, III, IV i V. Współczesne architektury określa się mianem MIPS32 (w przypadku wersji 32-bitowej) i MIPS64 (64 bity). W tym podrozdziale skupimy się na MIPS32. Warto zauważyć, że firma MIPS Technologies podjęła podobną decyzję, jak Intel — mimo iż ISA ewoluowała, przez cały czas zachowano zgodność z poprzednimi wersjami. Ponadto w każdej nowej wersji architektury wprowadzano nowe instrukcje, które miały usprawniać wydajność procesora i obsługiwać wartości zmiennoprzecinkowe. Nowe architektury MIPS32 i MIPS64 charakteryzują się wyraźnymi usprawnieniami w zakresie technologii VLSI i organizacji CPU. Pozwoliło to wyraźnie ograniczyć koszty i zwiększyć wydajność procesorów w porównaniu z tymi o tradycyjnej architekturze.

Podobnie jak IA-32 i IA-64, architektura zbioru rozkazów MIPS składa się z bogatego zbioru rozkazów, wśród których znalazły się instrukcje arytmetyczne, logiczne, porównawcze, a także instrukcje odpowiedzialne za przesył danych, rozgałęzienia, skoki, przesuwanie wartości i obsługę multimediów. MIPS to *architektura typu ładuj-zapisz*, co oznacza, że wszystkie instrukcje (poza tymi realizującymi wczytywanie i zapisywanie danych) muszą operować na zawartości rejestrów (operandy nie mogą być komórkami pamięci). MIPS32 ma 168 32-bitowych rozkazów, ale wiele z nich jest do siebie podobnych. Mamy tu na przykład sześć różnych instrukcji służących do dodawania liczb, z których każda wykorzystuje nieco inne operandy i rejestry. Pomysł stosowania wielu instrukcji realizujących tę samą operację jest dość powszechnie wykorzystywany przy tworzeniu zbiorów rozkazów asemblera. Inną popularną instrukcją jest MIPS NOP (no-op, ang. *brak operacji*), która nie wykonuje zupełnie żadnych działań, a jedynie zajmuje czas procesora (w rozdziale 5. zobaczysz, że instrukcje NOP wykorzystywane są w mechanizmach potokowości).

CPU zbudowany w architekturze MIPS32 ma 32 32-bitowe rejestry ogólnego przeznaczenia, oznaczone od r0 do r31. Dwa z nich pełnią specjalną rolę — r0 ma na stałe (sprzętowo) przypisaną wartość 0, a r31 jest rejestrem domyślnie wykorzystywanym przez wiele instrukcji, co oznacza, że nie trzeba go wymieniać w samych instrukcjach. W asemblerze MIPS rejestry te oznaczamy symbolami $0, $1, …, $31. Rejestr 1 jest zarezerwowany, a rejestry 26 i 27 są wykorzystywane przez jądro systemu operacyjnego. Rejestry 28, 29 i 30 są rejestrami wskaźnikowymi. Do pozostałych rejestrów możemy odwoływać się, podając ich numery, zgodnie z konwencją nazewniczą przedstawioną w tabeli 4.8. Na przykład chcąc skorzystać z 8 rejestru, możemy napisać $8 lub $t0.

TABELA 4.8. Konwencja nazewnicza rejestrów MIPS32

Konwencja nazewnicza	Numer rejestru	Wartości przechowywane w rejestrze
$v0 – $v1	2 – 3	Wyniki, wyrażenia
$a0 – $a3	4 – 7	Argumenty
$t0 – $t7	8 – 15	Wartości tymczasowe
$s0 – $s7	16 – 23	Zapisane wartości
$t8 – $t9	24 – 25	Dodatkowe wartości tymczasowe

Istnieją też dwa rejestry specjalnego przeznaczenia, HI i LO, przechowujące wyniki pewnych operacji na liczbach całkowitych. Oczywiście w procesorze dostępny jest też rejestr PC (licznik rozkazów), co daje w sumie trzy rejestry specjalnego przeznaczenia.

MIPS32 ma też 32 32-bitowe rejestry zmiennoprzecinkowe, które można wykorzystywać w operacjach zmiennoprzecinkowych o pojedynczej precyzji (wartości o podwójnej precyzji są przechowywane w parach składających się z 1 nieparzystego i 1 parzystego rejestru). Istnieją też 4 zmiennoprzecinkowe rejestry sterujące specjalnego przeznaczenia, wykorzystywane przez jednostkę zmiennoprzecinkową.

Kontynuujmy nasze porównanie, przepisując programy z przykładów 4.1 i 4.4 na język asemblera MIPS32.

Przykład 4.5.

```
        . . .
        .data
        # $t0 = sum
        # $t1 = loop counter Ctr
Value: .word 10, 15,20,25,30
        Sum = 0
        Ctr = 5
        .text
        .global main          # deklaracja zmiennej globalnej main
main:   lw $t0, Sum           # Inicjalizuj rejestr zawierający sumę wartością 0
        lw $t1, Ctr           # Skopiuj wartość Ctr do rejestru
        la $t2, value         # $t2 jest wskaźnikiem do bieżącej wartości
while: blez #t1, end_while # Koniec pętli, jeśli licznik <= 0
        lw #t3, 0($t2)        # Wczytaj wartość przesuniętą względem wskaźnika o 0
        add $t0, $t0, $t3     # Dodaj wartość do sumy
        addi $t2, $t2, 4      # Przejdź do następnej wartości
        sub #t1, $t1, 1       # Zmniejsz Ctr
        b while               # Wróć na początek pętli
        la $t4, sum           # Wczytaj do rejestru adres sumy
        sw $t0, 0($t4)        # Zapisz sumę w miejscu w pamięci wskazywanym przez sum
        . . .
```

Przypomina to trochę kod dla procesorów Intela, ponieważ licznik pętli jest kopiowany do rejestru, zmniejszany przy każdej iteracji pętli o 1 i porównywany z wartością 0. Nazwy rejestrów mogą z początku wydawać Ci się dziwne, ale gdy zrozumiesz stosowane konwencje nazewnicze, będziesz mógł z nimi bardzo swobodnie pracować.

Jeżeli jesteś zainteresowany pisaniem programów dla MIPS, a nie masz żadnego komputera z takim procesorem, pamiętaj, że możesz skorzystać z wielu dostępnych emulatorów. Najpopularniejszym z nich jest SPIM, który pozwala uruchamiać programy napisane w języku asemblera procesorów MIPS R2000 i R3000. Zawiera on prosty program do usuwania błędów (debugger) i ma zaimplementowany niemal cały zbiór instrukcji asemblerowych tej architektury. Pakiet SPIM dostarczany jest wraz z kodem źródłowym i pełną dokumentacją. Jest on dostępny w wersjach dla wielu odmian Uniksa (w tym dla Linuksa), Windows (PC i DOS) oraz dla Macintosha. Jeśli chcesz zdobyć więcej informacji na ten temat, zajrzyj do bibliografii na końcu tego rozdziału.

Jeżeli przyjrzysz się uważnie przykładom 4.1, 4.4 i 4.5, zobaczysz, że wykorzystywane w nich instrukcje są do siebie podobne. To prawda, że rejestry mają inne nazwy i odwołujemy się do nich w różny sposób, ale wszystkie wykonywane na nich operacje są praktycznie takie same. Niektóre języki asemblera mają większe zestawy rozkazów, co daje programistom większą swobodę przy kodowaniu algorytmów. Jednak jak widzieliśmy na przykładzie MARIE, duży zbiór instrukcji wcale nie jest niezbędny do tego, aby napisać w pełni działający program.

Podsumowanie

W tym rozdziale przedstawiłyśmy prostą architekturę komputera, MARIE, za pomocą której prezentowałyśmy przebieg cyklu „pobierz, dekoduj, wykonaj" i pokazywałyśmy, jak naprawdę działają komputery. MARIE miała własną architekturę zbioru rozkazów i język asemblera, co pozwalało nam na pisanie programów. Położyłyśmy przy tym szczególny nacisk na zależności między ISA a asemblerem.

Podstawowym elementem każdego komputera jest CPU. Składa się on ze ścieżki danych (rejestrów i jednostki ALU, połączonych ze sobą magistralą) oraz jednostki sterującej, odpowiedzialnej za szeregowanie operacji, przesyłanie danych i wysyłanie impulsów synchronizacyjnych. Te ostatnie są wykorzystywane przez wszystkie podzespoły komputera, które mogą dzięki temu harmonijnie ze sobą współpracować. Podsystem wejścia-wyjścia umożliwia wprowadzanie danych do komputera i zwracanie ich użytkownikowi.

MARIE to bardzo prosta architektura, zaprojektowana z myślą o zilustrowaniu założeń omawianych w tym rozdziale bez konieczności opisywania zbyt wielu szczegółów technicznych. Ma ona 4K pamięci złożonej z 16-bitowych słów i siedem rejestrów. Zawiera też jeden rejestr ogólnego przeznaczenia, AC. Rozkazy dla MARIE składają się z 4-bitowego kodu operacji i 12-bitowego adresu. Wprowadziłyśmy też notację typu rejestr-przesunięcie, która umożliwia symboliczne przedstawienie operacji wykonywanych przez poszczególne instrukcje na poziomie rejestrów.

Na cykl „pobierz, dekoduj, wykonaj" składają się czynności, które komputer musi wykonać, aby uruchomić program. Instrukcja jest pobierana z pamięci, a następnie dekodowana. Pobierane są też wszystkie potrzebne jej operandy, po czym następuje jej wykonanie. Na początku tego cyklu przetwarzane są przerwania. Po zakończeniu ich obsługi cykl „pobierz, dekoduj, wykonaj" wraca do normalnego stanu.

Język maszynowy to ciąg liczb dwójkowych stanowiących reprezentację wykonywalnych instrukcji maszynowych. Z kolei w języku asemblera wykorzystujemy symboliczne nazwy operacji, reprezentujące dane numeryczne, z których budowany *jest* program w języku maszynowym. Język asemblera jest językiem programowania, ale nie oferuje zbyt wielu instrukcji ani typów danych. Nazywamy go językiem niskiego poziomu.

Pewnie zgodzisz się, że programowanie w języku asemblera MARIE jest, najdelikatniej mówiąc, męczące. Widzieliśmy już, że w większości przypadków przekazywanie sterowania do różnych części programu musi być jawnie zdefiniowane przez programistę, za pomocą instrukcji skoku. Nasz język asemblera oddziela też spora przepaść od języków wysokiego poziomu, takich jak C++ czy Ada. Musimy jednak pamiętać, że to na poziomie asemblera instrukcje kodu źródłowego są

przekształcane na postać zrozumiałą dla komputera. Zamieszczając wprowadzenie do asemblera, wcale nie chciałyśmy uczynić z Ciebie programisty piszącego w tym języku. Miałyśmy jedynie nadzieję, że dzięki temu zrozumiesz, jakie zależności zachodzą między językiem programowania a architekturą komputera. Znajomość podstaw asemblera powinna też sprawić, że zdasz sobie sprawę z tego, co dzieje się „od kuchni" w programach pisanych w C++, Javie czy w Adzie. Mimo iż łatwiej jest pisać programy asemblerowe dla x86 i dla MIPS niż dla MARIE, i tak jest to znacznie trudniejsze niż posługiwanie się językami wysokiego poziomu.

Zapoznałyśmy Cię (choć pobieżnie) z architekturami Intela i MIPS — i to przynajmniej z dwóch powodów. Po pierwsze, porównywanie ze sobą różnych architektur, począwszy od najprostszych, przez coraz bardziej rozbudowane, jest naprawdę interesujące. Powinieneś przy tym skupić się zarówno na różnicach, jak i na podobieństwach. Po drugie, mimo iż języki asemblera Intela i MIPS różnią się od asemblera MARIE, są one porównywalne. Poszczególne instrukcje operują na pamięci i na rejestrach. Niektóre z nich służą do przenoszenia danych, inne do wykonywania operacji arytmetycznych i logicznych, a jeszcze inne do implementowania skoków. Zbiór rozkazów MARIE jest bardzo prosty i brakuje w nim wielu instrukcji ułatwiających pracę programistom, które są obecne zarówno w zbiorze rozkazów Intela, jak i MIPS. W tych dwóch architekturach występuje też znacznie więcej rejestrów. Pomijając jednak te różnice, ich języki asemblera są pod względem funkcjonalnym praktycznie takie same.

Dodatkowe źródła informacji

Na stronie domowej tej książki dostępny jest symulator, który pozwala wykonywać programy napisane w asemblerze MARIE.

Bardziej szczegółowe informacje na temat organizacji CPU i architektur ISA znajdziesz w książkach Tanenbauma (1999) i Stallingsa. Podręcznik Mano (1991) zawiera przykłady architektur opartych na mikrokodzie. Z kolei książka Wilkesa (1958) to doskonała pozycja o mikroprogramowaniu.

Więcej informacji na temat programowania w języku asemblera Intela znajdziesz w książkach Abela (2001), Dandamudiego (1998) i Jonesa (1997). Zwłaszcza ta ostatnia opisuje wszystko w prosty i jasny sposób, choć wszystkie trzy są dość wyczerpujące. Jeżeli interesują Cię inne asemblery, zajrzyj do książek Struble'a (1982) (asembler IBM-a), Gilla, Corwina i Logara (1987) (asembler Motoroli) i oraz pozycji wydanej przez Sun Microsystems (1992) (asembler SPARC). Proste wprowadzenie do systemów wbudowanych znajdziesz w pozycji Williamsa (2000).

Osoby zainteresowane tworzeniem oprogramowania dla architektury MIPS powinny zajrzeć do książki Pattersona i Hennessy'ego (1997), w której zagadnienia te zostały przedstawione w znakomity sposób i która zawiera dodatek pełen przydatnych informacji. Środowisko MIPS jest też dobrze opisane w pracy Donovana (1972). Kane i Heinrich (1992) stworzyli doskonały opis zbioru rozkazów MIPS i programowania w języku asemblera tej architektury. Mnóstwo przydatnych informacji znajdziesz też na stronie internetowej firmy MIPS.

Aby dowiedzieć się czegoś więcej na temat architektur Intela, zajrzyj do książek Alperta i Avnona (1993), Breya (2003) i Dulona (1998). Prawdopodobnie jedną z najlepszych książek na temat architektury Pentium jest praca Shanleya (1998). Architektury Motoroli, UltraSparc i Alpha zostały opisane odpowiednio przez Circello (1995), Horela i Lauterbacha (1999) oraz McLellana (1995). Bardziej ogólne omówienie zaawansowanych architektur znajdziesz w pracy Tabaka (1991).

Jeżeli chcesz dowiedzieć się czegoś więcej na temat symulatora MIPS o nazwie SPIM, poczytaj książkę Pattersona i Hennessy'ego (1997) lub zajrzyj na stronę domową SPIM, na której znajdziesz dokumentację, podręczniki i różne inne pliki do ściągnięcia. Doskonałe wprowadzenie do programowania w języku asemblera RISC, a także MIPS znajdziesz w książce Waldrona (1999).

Bibliografia

Peter Abel, *IBM PC Assembly Language and Programming*, wydanie piąte, Upper Saddle River: Prentice Hall, 2001.

D. Alpert, D. Avnon, *Architecture of the Pentium Microprocessor*, „IEEE Micro" 13:3, kwiecień 1993, str. 11 – 21.

B. Brey, *Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor, Pentium II, Pentium III, and Pentium IV: Architecture, Programming, and Interfacing*, wydanie szóste, Englewood Cliffs: Prentice Hall, 2003.

J. Circello, G. Edgington, D. McCarthy, J. Gay, D. Schimke, S. Sullivan, R. Duerden, C. Hinds, D. Marquette, L. Sood, A. Crouch, D. Chow, *The Superscalar Architecture of the MC68060*, „IEEE Micro" 15:2, kwiecień 1995, str. 10 – 21.

S. P. Dandamudi, *Introduction to Assembly Language Programming — From 8086 to Pentium Processors*, New York: Springer Verlag, 1998.

J. J. Donovan, *Systems Programming*, New York: McGraw-Hill, 1972.

C. Dulon, *The IA-64 Architecture at Work*, „Computer" 31:1, lipiec 1998, str. 24 – 32.

A. Gill, E. Corwin, A. Logar, *Assembly Language Programming for the 68000*, Upper Saddle River: Prentice Hall, 1987.

J. Goodman, K.A. Miller, *Programmer's View of Computer Architecture*, Philadelphia: Saunders College Publishing, 1993.

T. Horel, G. Lauterbach, *UltraSPARC III: Designing Third Generation 64-Bit Performance*, „IEEE Micro" 19:3, maj-czerwiec 1999, str. 73 – 85.

W. Jones, *Assembly Language for the IBM PC Family*, wydanie drugie, El Granada: Scott Jones, 1997.

G. Kane, J. Heinrich, *MIPS RISC Architecture*, wydanie drugie, Englewood Cliffs: Prentice Hall, 1992.

Morris M. Mano, *Digital Design*, wydanie drugie, Upper Saddle River: Prentice Hall, 1991.

E. McLellan, *The Alpha AXP Architecture and 21164 Alpha Microprocessor*, „IEEE Micro" 15:2, kwiecień 1995, str. 33 – 43.

Strona domowa MIPS: *http://www.mips.com*

D.A. Patterson, J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, wydanie drugie, San Mateo: Morgan Kaufmann, 1997.

W.A. Samaras, N. Cherukuri, S. Venkataraman, *The IA-64 Itanium Processor Cartridge*, „IEEE Micro" 27:1. styczeń-luty 2001, str. 82 – 89.

T. Shanley, *Pentium Pro and Pentium II System Architecture*, Reading: Addison-Wesley, 1998.

SPARC International, Inc., *The SPARC Architecture Manual: Version 9*, Upper Saddle River: Prentice Hall, 1994.

Strona domowa SPIM: *http://www.cs.wisc.edu/~larus/spim.html*

W. Stallings, *Computer Organization and Architecture*, wydanie piąte, New York: Macmillan Publishing Company, 2000.

G.W. Struble, *Wprowadzenie do programowania w języku ASEMBLERA dla maszyn cyfrowych IBM serii 360 i 370*, Warszawa: Wydawnictwa Naukowo-Techniczne 1982.

D. Tabak, *Advanced Microprocessors*, New York, NY: McGraw-Hill, 1991.

Andrew Tanenbaum, *Structured Computer Organization*, wydanie czwarte, Upper Saddle River: Prentice Hall, 1999.

John Waldron, *Introduction to RISC Assembly Language*, Reading: Addison Wesley, 1999.

M.V. Wilkes, W. Renwick, D.J. Wheeler, *The Design of the Control Unit of an Electronic Digital Computer*, „Proceedings of IEEE", 105, Pan B, No. 20, 1958, str. 121 – 128, 1958.

A. Williams, *Microcontroller Projects with Basic Stamps*, Gilroy: R&D Books, 2000.

Utrwalenie podstawowych terminów i pojęć

(1) Jaka jest funkcja CPU?
(2) Jaką rolę odgrywa ścieżka danych?
(3) Czym zajmuje się jednostka sterująca?
(4) Jakie są rodzaje rejestrów i gdzie są one umieszczone?
(5) Skąd ALU „wie", którą operację ma wykonać?
(6) Dlaczego magistrala jest często „wąskim gardłem" przy przesyłaniu danych?
(7) Jaka jest różnica między magistralą dwupunktową a wielopunktową?
(8) Dlaczego protokół magistrali ma istotne znaczenie?
(9) Wyjaśnij różnicę między szyną danych i szyną adresową a magistralą sterującą.
(10) Co to jest cykl magistrali?
(11) Wymień trzy różne rodzaje magistral i powiedz, gdzie są one stosowane.
(12) Jaka jest różnica między magistralami synchronicznymi a asynchronicznymi?
(13) Wymień cztery rodzaje obsługi konfliktów na magistrali.
(14) Wyjaśnij różnicę między cyklami zegara a częstotliwością zegara.
(15) Czym zegar systemowy różni się od zegara magistrali?
(16) Jaką rolę pełni interfejs I/O?
(17) Wyjaśnij różnicę między wejściem-wyjściem opartym na rozkazach a wejściem-wyjściem opartym na mapowaniu pamięci.
(18) Jaka jest różnica między bajtem a słowem? Co je od siebie odróżnia?
(19) Wyjaśnij różnicę między adresowalnością na poziomie bajtów a adresowalnością na poziomie słów.
(20) Dlaczego wyrównywanie adresów ma duże znaczenie?
(21) Wymień i opisz dwa rodzaje przeplatania pamięci. Wskaż różnice między nimi.
(22) Opisz działanie przerwań i wymień cztery ich rodzaje.
(23) Czym przerwanie maskowalne różni się od niemaskowalnego?
(24) Dlaczego jest tak, że skoro MARIE ma 4K głównej pamięci, to adresy muszą być 12-bitowe?
(25) Wyjaśnij przeznaczenie wszystkich rejestrów MARIE.
(26) Co to jest kod operacji?
(27) Opisz działanie każdego rozkazu MARIE.
(28) Czym język maszynowy różni się od języka asemblera? Czy zachodzi między nimi konwersja typu „jeden do jednego" (tzn. czy każdej instrukcji asemblera odpowiada jedna instrukcja maszynowa)?
(29) Omów znaczenie notacji RTN.
(30) Czy mikrooperacja jest tym samym, co instrukcja maszynowa?
(31) Czym mikrooperacje różnią się od zwykłych instrukcji języka maszynowego?
(32) Opisz poszczególne kroki cyklu „pobierz, dekoduj, wykonaj".
(33) Jak działa wejście-wyjście sterowane przerwaniami?
(34) Wyjaśnij, jak działa asembler. Powiedz, w jaki sposób generuje on tablicę symboli, jakie operacje wykonuje z kodem źródłowym i z kodem obiektowym oraz jak obsługuje etykiety.

(35) Co to jest system wbudowany? Czym różni się on od zwykłego komputera?
(36) Prześledź działanie programu przedstawionego w przykładzie 4.1 (tak, jak to robiliśmy na rysunku 4.13).
(37) Wyjaśnij różnicę między sterowaniem czysto sprzętowym a opartym na mikrokodzie.
(38) Co to jest stos? Dlaczego ma on istotne znaczenie dla programistów?
(39) Porównaj komputery CISC z komputerami RISC.
(40) Czym architektura Intela różni się od architektury MIPS?
(41) Wymień cztery procesory Intela i cztery procesory MIPS.

Ćwiczenia

(1) Jakie są główne zadania CPU?
(2) Wyjaśnij, jakie działanie powinien wykonać CPU po wystąpieniu przerwania. Pamiętaj o opisaniu, w jaki sposób CPU wykrywa przerwanie, jak jest ono obsługiwane i co się dzieje po jego obsłużeniu.
♦(3) Ilu bitów pamięci potrzebowałbyś do zaadresowania 2M × 32 pamięci, jeżeli byłaby ona adresowalna:
  ♦(a) na poziomie bajtów?
  ♦(b) na poziomie słów?
(4) Ilu bitów potrzeba do zaadresowania 4M × 16 pamięci, jeżeli jest ona adresowalna:
  (a) na poziomie bajtów?
  (b) na poziomie słów?
(5) Ilu bitów potrzeba do zaadresowania 1M × 8 pamięci, jeżeli jest ona adresowalna:
  (a) na poziomie bajtów?
  (b) na poziomie słów?
♦(6) Załóżmy, że pamięć 2M × 16 zbudowana jest z chipów RAM 256 KB × 8 i jest adresowalna na poziomie słów:
  ♦(a) ilu potrzeba chipów RAM?
  ♦(b) ile chipów RAM przypada na każde słowo maszynowe?
  ♦(c) ile bitów musi przypadać w adresie na każdy chip?
  ♦(d) ile banków będzie miała ta pamięć?
  ♦(e) ilu bitów adresu potrzeba dla całej pamięci?
  ♦(f) jeżeli stosowany jest przeplot wysokopozycyjny, to gdzie zlokalizowana jest komórka o adresie 14 (E szesnastkowo)?
  ♦(g) wykonaj ćwiczenie 6f dla przeplotu niskopozycyjnego.
(7) Wykonaj ponownie ćwiczenie 6. dla pamięci 16M × 16, zbudowanej z chipów RAM 512 K × 8.
(8) Komputer cyfrowy ma jednostkę pamięci, w której każde słowo złożone jest z 24 bitów. Zbiór rozkazów składa się ze 150 różnych instrukcji. Wszystkie instrukcje składają się z kodu operacji i części adresowej (w której może być umieszczony tylko jeden adres). Każda instrukcja zajmuje jedno słowo w pamięci:
  (a) z ilu bitów musi składać się kod operacji?
  (b) ile bitów pozostaje na część adresową każdej instrukcji?
  (c) jaki jest maksymalny dozwolony rozmiar pamięci?
  (d) jaka jest największa liczba dwójkowa bez znaku, którą można zapisać na jednym słowie pamięci?

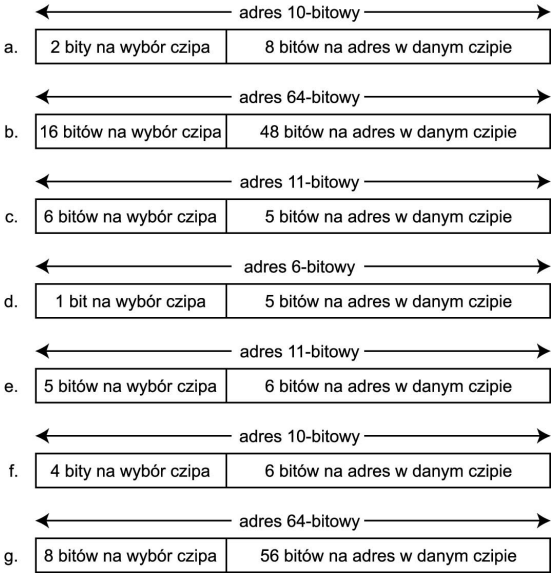
(9) Weźmy pod uwagę pamięć składającą się z 2^{20} bajtów:

♦(a) podaj największy i najmniejszy adres, przy założeniu, że pamięć jest adresowalna na poziomie bajtów;

♦(b) podaj największy i najmniejszy adres, przy założeniu, że pamięć jest adresowalna na poziomie słów, a słowa są 16-bitowe;

(c) podaj największy i najmniejszy adres, przy założeniu, że pamięć jest adresowalna na poziomie słów, a słowa są 32-bitowe.

(10) Jeżeli pamięć ma pojemność 2 048 bajtów i składa się z kilku chipów 64 bajty × 8, a ponadto jest adresowalna na poziomie bajtów, to który z poniższych siedmiu diagramów przedstawia prawidłowy sposób posługiwania się bitami adresu? Umotywuj swoją odpowiedź.



(11) Omów poszczególne kroki cyklu „pobierz, dekoduj, wykonaj”. Nie zapomnij powiedzieć, co dzieje się w poszczególnych rejestrach.

♦(12) Wyjaśnij, dlaczego w architekturze MARIE rejestr MAR ma tylko 12 bitów szerokości, podczas gdy AC jest 16-bitowy.

(13) Podaj kod szesnastkowy następującego programu (zamień go ręcznie na kod maszynowy).

```
Etykieta Adres szesnastkowy Instrukcja
         100                Load A
         101                Add One
         102                Jump S1
S2,      103                Add One
         104                Store A
         105                Halt
S1,      106                Add A
         107                Jump S2
A,       108                HEX 0023
One,     109                HEX 0001
```

♦(14) Jak wyglądałaby zawartość tablicy symboli poprzedniego programu?

(15) Znając zbiór rozkazów MARIE opisany w tym rozdziale:

(a) rozszyfruj poniższe instrukcje zapisane w języku maszynowym MARIE (napisz ich odpowiedniki w języku asemblera):

♦(i) 0010000000000111,
(ii) 1001000000001011,
(iii) 0011000000001001;

(b) zapisz następujący segment kodu w języku asemblera MARIE:

```
if X > 1 then
   Y := X + X;
   X := 0;
endif;
   Y := Y + 1;
```

(c) wyjaśnij, jakie są potencjalne problemy z poniższym fragmentem kodu w języku asemblera MARIE, implementującym procedurę? Procedura ta zakłada, że przekazywany parametr znajduje się w AC i że powinna ona podwoić jego wartość. W głównej części programu (`Main`) występuje przykładowe wywołanie tej procedury. Możesz przyjąć, że jest to część większego programu.

```
Main,  Load  X
       Jump  Sub1
Sret,        Store X
       . . .
Sub1,  Add   X
       Jump  Sret
```

(16) Napisz program dla MARIE, obliczający wartość wyrażenia $A * B + C * D$.

(17) Zapisz poniższy fragment kodu w języku asemblera MARIE:

```
X := 1;
while X < 10 do
   X := X + 1;
endwhile;
```

(18) Zapisz poniższy fragment kodu w języku asemblera MARIE:

```
Sum := 0;
for X := 1 to 10 do
    Sum := Sum + X;
```

(19) Napisz program dla MARIE, w którym wykorzystasz pętlę do przemnożenia dwóch liczb dodatnich metodą wielokrotnego dodawania. Na przykład aby pomnożyć 3 * 6, program powinien dodać do siebie sześć trójek — 3 + 3 + 3 + 3 + 3 + 3.

(20) Napisz podprogram MARIE odejmujący jedną liczbę od drugiej.

(21) Duża liczba rejestrów wydaje się być czymś dobrym, ponieważ dzięki ich zastosowaniu komputer rzadziej sięga po dane zgromadzone w pamięci. Podaj przykład obliczeniowy, który potwierdza tę tezę. Najpierw ustal, do ilu adresów pamięci musi się odwoływać komputer
MARIE, w którym dane przechowywane są w dwóch rejestrach (AC i MBR). Następnie przeprowadź te same obliczenia na procesorze, który może przechowywać dane w więcej niż trzech rejestrach.

(22) MARIE zachowuje w pamięci adres powrotu z procedury, w miejscu określonym przez instrukcję `jump-and-store`. W niektórych architekturach adres ten jest przechowywany w rejestrze, a w niektórych na stosie. Które rozwiązanie bardziej sprzyja rekurencji? Wyjaśnij swoją odpowiedź.

(23) Prześledź wykonywanie programu z przykładu 4.2 (tak, jak to robiliśmy na rysunku 4.13).

(24) Prześledź wykonywanie programu z przykładu 4.4 (tak, jak to robiliśmy na rysunku 4.13).

(25) Powiedzmy, że rozbudowujemy architekturę zbioru rozkazów MARIE o następującą instrukcję:

```
IncSZ Operand
```

Inkrementuje ona wartość przechowywaną pod adresem `Operand` i sprawdza, czy po wykonaniu tej operacji będzie ona równa 0. Jeśli tak, zawartość rejestru rozkazów zostaje zwiększona o 1. Krótko mówiąc, inkrementujemy operand i, jeżeli nowa wartość jest równa 0, pomijamy kolejną instrukcję. Przedstaw działanie tej instrukcji w notacji RTN.

(26) Czy między procesorem a pamięcią lepiej jest stosować magistralę synchroniczną czy asynchroniczną? Wyjaśnij swoją odpowiedź.

*(27) Wybierz sobie dowolną architekturę (inną niż omówione w tym rozdziale) i sprawdź, jak realizowane są w niej operacje, które omawialiśmy w tym rozdziale na przykładzie architektur Intela i MIPS.

Prawda czy fałsz

_____(1) Jeżeli komputer ma czysto sprzętowe sterowanie, jego zbiór rozkazów jest zdeterminowany przez mikroprogram. Nie może on ulec zmianie, chyba że przeprojektujemy całą architekturę.

_____(2) Instrukcja skoku zmienia przepływ informacji, modyfikując rejestr PC.

_____(3) Rejestry są komórkami pamięci umieszczonymi w samym procesorze.

_____(4) Asembler dwuprzebiegowy tworzy za pierwszym razem tablicę symboli, a za drugim kończy tłumaczenie programu z języka asemblera na kod maszynowy.

_____(5) W rejestrach MAR, MBR, PC i IR MARIE można przechowywać dowolne wartości.

_____(6) MARIE ma wspólną magistralę, co oznacza, że współdzieli ją wiele elementów komputera.

_____(7) Asembler to program, który wczytuje program zapisany w języku symbolicznym i tworzy jego odpowiednik w języku maszynowym. Każdej instrukcji języka asemblera odpowiada przy tym dokładnie jedna instrukcja maszynowa.

_____(8) Jeżeli komputer ma sterowanie oparte na mikroprogramie, to jego zbiór rozkazów zależny jest właśnie od mikroprogramu.