ROZDZIAŁ TRZECI: **ORGANIZACJA SYSTEMU**

Napisanie nawet skromnego programu w języku asemblera dla 80x86 wymaga znajomości rodziny 80x86. Napisanie *dobrego* programu w języku asemblera wymaga sporej wiedzy o wykorzystywanym sprzęcie. Niestety wykorzystywany sprzęt nie jest spójny. Techniki które są kluczowe dla 8088 mogą nie być przydatne dla systemów 80486. Podobnie, techniki programistyczne które pozwalają uzyskiwać większą wydajność na chipach 80486 mogą nie być pomocne we wszystkim na 80286.Szczęśliwie,niektóre techniki programistyczne pracują dobrze niezależnie jakiego mikroprocesora używamy. Ten rozdział omawia wpływ jaki sprzęt ma na wydajność programów komputerowych

---

3.0 WSTĘP

Ten rozdział opisuje podstawowe komponenty które stanowią system komputerowy:CPU,pamięć, I/O (wejście/wyjście) i magistrale które je łączą. Chociaż możemy napisać program który nie zna tych pojęć jednak program na wysokim poziomie wymaga kompletnego zrozumienia tego materiału.

Ten rozdział zaczyna się od omówienia organizacji magistral i pamięci. Te dwa komponenty sprzętowe,będą prawdopodobnie miały większy wpływ na wydajność twojego programu niż szybkość CPU. Zrozumienie organizacji systemu magistral pozwoli ci zaprojektować struktury danych działające z maksymalną szybkością. Podobnie wiedza o charakterystycznych właściwościach pamięci, rejonach danych i działaniu na pamięci podręcznej cache pomoże stworzyć ci program działający tak szybko jak to możliwe. Oczywiście, jeśli nie interesuje cię pisanie kodu szybko wykonywalnego możesz opuścić to omówienie jednakże większość ludzi troszczy się o szybkość lub inaczej ta wiedza jest dla nich użyteczna.

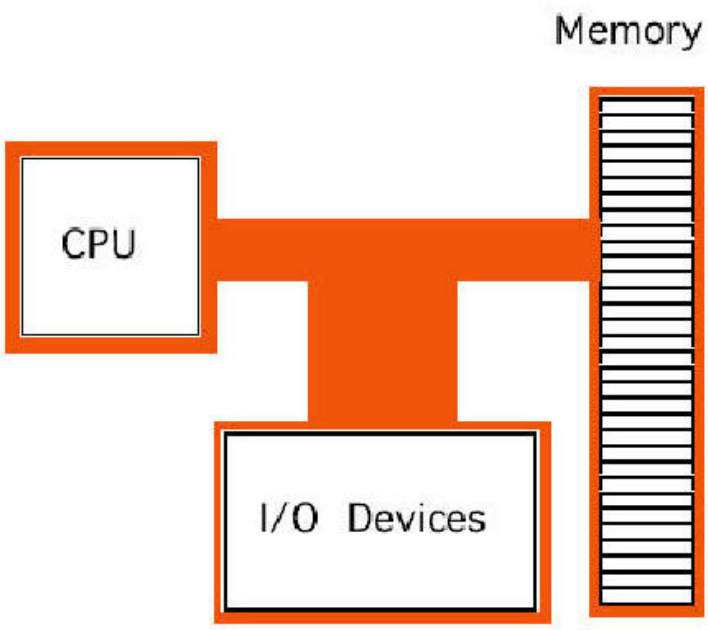
Niestety, rodzina mikroprocesorów 80x86 jest złożoną grupą i często przytłacza początkujących. Dlatego też ten rozdział będzie używał czterech hipotetycznych członków z rodziny 80x86: mikroprocesory 886,8286,8486 i 8686.Przedstawiają one uproszczone wersje chipów 80x86 i pozwalają omówić różne cechy architektury bez grzęźnięcia poprzez ogromy zbiór instrukcji CISC. Ten tekst używa hipotetycznych procesorów x86 do opisu pojęć kodowania instrukcji, trybów adresowania ,wykonywania sekwencyjnego, kolejki rozkazów**,** potokowania i operacji superskalarnych. Rzecz jasna, tych pojęć nie musisz się uczyć jeśli chcesz pisać tylko poprawne programy. Jednakże ,jeśli chcesz pisać dobrze ,szybkie programy zwłaszcza na zawansowanych procesorach takich jak 80486,Pentium i innych, musisz nauczyć się tych pojęć.

Niektórzy mogą argumentować że ten rozdział stanie się zbyt skomplikowany w związku z architekturą komputera. Będą uważać, że taki materiał powinien ukazać się w książce o architekturze, nie zaś w książce o programowaniu w języku asemblera. Nie jest to dalekie od prawdy! Napisanie dobrego programu asemblerowego wymaga solidnej wiedzy o architekturze. W związku z tym taki nacisk na architekturę komputera w tym rozdziale.

3.1 PODSTAWOWYY SYSTEM KOMPONENTÓW

Podstawowy, gotów do działania, projekt systemu komputerowego nazywany jest jego architekturą. John von Neumann,pionier w projektowaniu komputerów, dał podstawy architektury większości komputerów dzisiaj używanych.Na przykład rodzina 80x86 używa Architektury Von Neumanna (VNA). Typowy system Von Neumanna zawiera trzy ważne komponenty: CPU (jednostka centralna),pamięć i wejścia/wyjścia (I/O).Sposób w jaki projektant systemu łączy te trzy komponenty wpływa na wydajność systemu. (zobacz rysunek 3.1).

W maszynach VNA,takich jak rodzina 80x86, CPU bierze udział we wszystkich zachodzących zdarzeniach Wszystkie obliczenia zachodzą wewnątrz CPU. Dane i instrukcje CPU tkwią w pamięci dopóki nie zażyczy ich sobie CPU. Dla CPU ,większość urządzeń I/O, wygląda jak pamięć ponieważ może przechowywać dane



Rysunek 3.1: Typowa maszyna Von Neumanna

z urządzenia wyjściowego i czytać dane z urządzenia wejściowego. Ważną różnicą pomiędzy położeniem w pamięci a położeniem I/O jest fakt że położenie I/O jest generalnie powiązane z zewnętrznymi urządzeniami.

3.1.1 SYSTEM MAGISTRAL

System magistral łączy różne komponenty maszyny VNA.Rodzina 80x86 ma trzy ważne magistrale: magistralę adresową, magistralę danych i magistralę sterującą. Magistrala jest zbiorem przewodów,którymi sygnały elektryczne są przesyłane między komponentami w systemie. Te magistrale różnią się między procesorami. Jednak,każda magistrala przenosi porównywalne informacje we wszystkich procesorach, np. magistrala danych może być inaczej zaimplementowana na 80386 niż 8088,ale obie przenoszą dane pomiędzy procesorem, pamięcią i I/O.

Typowy system komponentów 80x86 używa standardowych poziomów logicznych TTL To znaczy, każdy przewód na magistrali używa standardowego poziomu napięcia dla przedstawiania zera lub jedynki. Zawsze możemy wyszczególnić zero i jeden zamiast poziomów elektrycznych ponieważ te poziomy różnią się dla różnych procesorów (zwłaszcza laptopów).

3.1.1.1 MAGISTRALA DANYCH

Procesory 80x86 używają magistrali danych do przenoszenia danych pomiędzy różnymi komponentami w systemie komputerowym. Rozmiar tej magistrali różni się znacznie w rodzinie 80x86.W rzeczywistości, ta magistrala określa „rozmiar" procesora.

W typowym systemie 80x86,magistrala danych zawiera osiem,16,32 lub 64 linie .Procesory 8088 i 80188 mają ośmiobitową magistralę danych (osiem linii danych). Procesory 8086,80186,80286 i 80386SX mają szesnastobitową magistralę danych. Procesory 80386DX,80486 i Pentium Overdrive mają 32-bitową magistralę danych. Procesory Pentium i Pentium Pro mają 64-bitową magistralę danych Przyszłe wersje chipów (80686/80786?) mogą mieć większe magistrale.

Mając ośmiobitową magistralę danych procesor nie jest ograniczony do ośmiobitowych typów danych. Po prostu, procesor może uzyskać dostęp do jednego bajtu danych na cykl pamięci. (zobacz „Podsystem Pamięci" z opisem cyklu pamięci)

"Rozmiar" procesora
Była spora różnica zdań pomiędzy inżynierami od sprzętu i oprogramowania dotycząca rozmiaru procesorów takich jak 8088. Z perspektywy projektantów sprzętu,8088 jest całkowicie ośmiobitowym procesorem – ma tylko osiem lini danych a jedna magistrala danych jest kompatybilna z pamięcią i urządzeniami I/O zaprojektowanymi pod kątem ośmiobitowych procesorów.Z drugiej strony,inżynierowie oprogramowania sprzeczają się,że 8088 jest 16 bitowym procesorem.Z ich perspektywy nie można rozpoznać między 8088 (z ośmiobitową magistralą danych), a 8086 (który ma 16 bitową magistralę danych).Istotnie,jedyna różnica jest w szybkości przy której te procesory działają;8086 z 16 bitową magisttralą jest szybszy.Ostaczne projektanci sprzętu wygrali.Pomimo faktu,że inżynierowie oprgramowania nie mogą rozróżnić 8088 i 8086 w swoich programach,nazywamy 8088 ośmiobitowym procesorem a 8086 16 bitowym procesorem.Podobnie, 80386SX (który ma 16 bitową magistralę danych) jest procesorem 16 bitowym,podczas gdy 80386DX (który ma pełną 32 bitową magistralę danych) jest 32 bitowym procesorem.

Dlatego też ośmiobitowa magistrala w 8088 może przesyłać tylko połowę informacji na jednostkę czasu (cykl pamięci) podobnie jak 16 bitowa magistrala danych w 8086.Zatem procesory z magistralą 16 bitową są naturalnie szybsze niż procesory ośmiobitowe .Podobnie procesory 32 bitowe są szybsze niż te z 16- lub ośmiobitową magistralą danych.Rozmiar magistrali danych wpływa na wydajność systemu bardziej niż jakakolwiek inna magistrala.

Często słyszymy o procesorach nazywanych procesorami ośmio- ,16 ,32 lub 64 bitowymi.Podczas gdy są umiarkowane kontrowersje dotyczące rozmiaru procesora, większość ludzi zgadza się że liczba linii danych w procesorze określa jego rozmiar.Ponieważ w rodzinie 80x86 magistrale danych są szerokie na oiem,16,32 lub 64 bity większość danych również jest ośmio,16,32 lub 64 bitowych.Chociaż jest możliwe przetwarzanie danych 12 bitowych w 8088,wielu programistów przetwarza 16 bitów ponieważ procesor i tak i tak pobiera i manipuluje 16 bitami.Jest tak ponieważ procesor zawsze pobiera osiem bitów.Pobranie12 bitów wymaga dwóch ośmiobitowych operacji na pamięci.Ponieważ procesor pobiera 16 bitów zamiast 12,większość programistów używa wszystkich 16 bitów.Ogólnie rzecz biorąc,manipulowanie danymi o długości 8,16,32 lub 64 bitów jest najbardziej wydajne.

Chociaż 16,32 lub 64 bitowe członkowie rodziny 80x86 mogą przetwarzać dane do szerokości magistrali mogą jednak uzyskać dostęp do jednostek pamięci mniejszych niż osiem,16 lub 32 bity.Dlatego też, cokolwiek zrobisz na małej magistrali danych,będzie zrobione równie dobrze na większej magistrali danych.;jednak można uzyskać dostęp do pamięci szybciej i można uzyskać dostęp do większych kawałków danych w jednej operacji na pamięci.Możesz przeczytać dokładnie o tym dostępie do pamięci trochę później (zobacz „Podsystem Pamięci").

Processor	Data Bus Size
8088	8
80188	8
8086	16
80186	16
80286	16
80386sx	16
80386dx	32
80486	32
80586 class/ Pentium (Pro)	64

Tablica 17: Rozmiar magistrali danych procesorów 80x86

3.1.1.2 MAGISTRALA ADRESOWA

Magistrala danych w rodzinie procesorów 80x86 przesyła informacje pomiędzy kolejnymi komórkami pamięci lub urządzeniami I/O a CPU.Tylko pozostaje pytanie :"Które komórki pamięci lub urządzenia I/O?" Magistrala adresowa odpowiada na to pytanie.W celu rozróżnienia komórki pamięci i urządzeń I/O projektant systemu przydziela unikalne adresy pamięci dla każdego elementu pamięci i urządzenia I/O.Kiedy program chce uzyskać dostęp do jakiejś szczególnej komórki pamięci lub urządzenia I/O umieszcza odpowiedni adres na magistrali adresowej.Zespół układów skojarzonych z pamięcią lub urządzeniem I/O rozpoznaje ten adres i wydaje polecenie pamięci lub urządzeniu I/O odczytania danych lub umieszczenie danych na magistrali danych.W obu przypadkach,wszystkie inne komórki pamięci ignorują to wywołanie.Tylko urządzenie,którego adres pasuje do wartości na magistrali adresowej,odpowiada.

Z pojedynczą linią adresową,procesor mógł stworzyć dokładnie dwa unikalne adresy:zero i jeden.Z n liniami adresowymi,procesor może stworzyć 2^n unikalnych adresów (ponieważ jest 2^n unikalnych wartości w n-bitowej liczbie binarnej).Dlatego też liczba bitów na magistrali adresowej ustala maksymalną liczbę adresowalnej pamięci i położenia I/O. 8088 i 8086,na przykład, mają 20 bitowe magistrale adresowe.Zatem,mogą one uzyskać dostęp góra do 1,048,576 (lub 2^{20}) komórek pamięci.Większa magistrala adresowa może uzyskać dostęp do większej liczby komórek pamięci.8088 i 8086,na przykład,cierpią z powodu małej przestrzeni adresowej - ich magistrala adresowa jest zbyt mała.Późniejsze procesory mają większe magistrale adresowe:

Processor	Address Bus Size	Max Addressable Memory	In English!
8088	20	1,048,576	One Megabyte
8086	20	1,048,576	One Megabyte
80188	20	1,048,576	One Megabyte
80186	20	1,048,576	One Megabyte
80286	24	16,777,216	Sixteen Megabytes
80386sx	24	16,777,216	Sixteen Megabytes
80386dx	32	4,294,976,296	Four Gigabytes
80486	32	4,294,976,296	Four Gigabytes
80586 / Pentium (Pro)	32	4,294,976,296	Four Gigabytes

Tablica 18: Rozmiary magistrali adresowych rodziny 80x86

Przyszłe procesory 80x86 prawdopodobnie,będą wsparte 48 bitową magistralą adresową.Nadchodzi czas kiedy większość programistów będzie uważać cztery gigabajty pamięci za zbyt małą.,chociaż dzisiaj uważają jeden megabajt za niewystarczający. Na szczęście architektura 80386,80486 i późniejsze chipy uwzględniają łatwość rozszerzenia do 48 bitowej magistrali adresowej przez segmentację.

3.1.1.3 MAGISTRALA STERUJĄCA

Magistrala sterująca jest zbiorem sygnałów które sprawdzają jak procesor komunikuje się z resztą systemu.Rozważmy na chwilę magistralę danych. CPU wysyła dane do pamięci i przyjmuje dane z pamięci na magistralę danych.Tu rodzi się pytanie,"Czy on wysyła czy przyjmuje?" Są dwie linie na magistrali sterującej,odczyt i zapis,które wyszczególniają kierunek przepływu danych.inne sygnały zwierają system taktowania,linie przerwań,linie stanu itd.Magistrale sterujące zmieniają się wraz z procesorami rodziny 80x86.Jednak niektóre linie sterujące są powszechne we wszystkich procesorach i są warte krótkiej wzmianki.

Linie sterujące „odczyt' i „zapis" sterują kierunkiem danych na magistrali danych.Kiedy oba mają logiczną jedynkę,CPU i pamięć -I/O nie mogą się skomunikować między sobą. Jeśli linia odczytu jest w stanie niskim (logiczne zero) CPU jest w stanie odczytu danych z pamięci (to znaczy,system przesyła dane z pamięci do CPU) Jeśli linia zapisu jest w stanie niskim,system przenosi dane z CPU do pamięci.

„Linie aktywujące bajt" są innym zbiorem ważnych linii sterujących.Te linie sterujące pozwalają 16,32 i 64 bitowym procesorom radzić sobie z mniejszymi kawałkami danych.Dodatkowe szczegóły pojawią się w następnej sekcji.

Rodzina 80x86,w odróżnieniu od innych procesorów,dostarcza dwóch odrębnych przestrzeni adresowych: jedną dla pamięci i jedną dla I/O.Podczas gdy magistrale adresowe pamięci na różnych procesorach 80x86 różnią się rozmiarami,magistrala adresowa I/O na wszystkich CPU 80x86 ma szerokość 16 bitów.To pozwala procesorowi adresować do 65,536 różnych lokacji I/O.Większość urządzeń(takich jak klawiatura,drukarka,dyski,itp.) wymagają więcej niż jedną lokację I/O. Pomimo to, 65,536 lokacji I/O jest wystarczające dla większości aplikacji. Oryginalna konstrukcja IBM PC pozwala tylko na używanie 1,024 z nich.

Chociaż rodzina 80x86 utrzymuje dwie przestrzenie adresowe,nie ma dwóch magistral adresowych (dla I/O i pamięci). Zamiast tego,system dzieli magistralę adresową na I/O i pamięć.Dodatkowe linie sterujące decydują czy adres jest przeznaczony dla pamięci czy I/O..Kiedy takie sygnały są aktywne, urządzenia I/O używają adresów z najmniej znaczących 16 bitów magistrali adresowej.Kiedy są nieaktywne,urządzenia I/O ignorują sygnały na magistrali adresowej (przejmuje je podsystem pamięci).

3.1.2 PODSYSTEM PAMIĘCI

Typowy procesor 80x86 adresuje maksymalnie 2n różnych komórek pamięci, gdzie n jest liczbą bitów na magistrali adresowej już widzieliśmy procesory 80x86 mają 20,24 i 32 bitową magistralę adresową (z 48 bitową „w drodze").

Oczywiście pierwszym pytaniem jakie możemy zadać jest, „Czym dokładnie jest (lokacja) komórka pamięci?".80x86 wspiera „pamięć adresowalną bajtem".Zatem,podstawową jednostką pamięci jest bajt.Więc z 20,24 i 32 liniami adresowymi procesor 80x86 może zaadresować ,odpowiednio,jeden megabajt,16 megabajtów i cztery gigabajty pamięci.

Myślimy o pamięci jako o liniowej tablicy bajtów.Adres pierwszego bajtu to zero a adres ostatniego bajtu to 2^{n-1}.Dla 8088 z 20 bitową magistralą adresową, następująca pseudo-Pascalowa deklaracja tablicy jest dobrym przybliżeniem pamięci:

Memory:array[0..1048575] of byte;

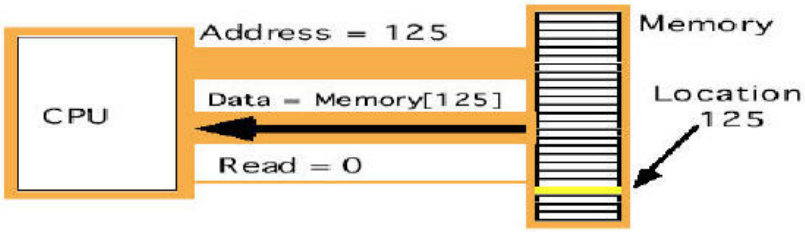
Wykonując odpowiednik Pascalowego wyrażenia"Memory[125]:=0" CPU umiejscawia wartość zero na magistrali danych, adres 125 na magistrali adresowej,i inicjuje linię zapisu (ponieważ CPU zapisuje daną do pamięci,zobacz rysunek 3.2)

Wykonując odpowiednik wyrażenia „CPU:=Memory[125};" CPU umiejscawia adres 125 na magistrali adresowej,inicjuje linię odczytu (ponieważ CPU odczytuje dane z pamięci) a potem odczytuje dane wynikowe z magistrali danych (zobacz rysunek 3.3)

Powyższe rozważania mają zastosowanie tylko kiedy uzyskujemy dostęp do pojedynczego bajtu w pamięci.Więc co się wydarzy kiedy procesor uzyska dostęp do słowa lub podwójnego słowa? Ponieważ pamięć składa się tablicy bajtów,jak możemy sobie poradzić z wartościami dłuższymi niż osiem bitów?

Różne systemy komputerowe mają różne rozwiązania tego problemu.Rodzina 80x86 radzi sobie z tym problemem przez przechowanie mniej znaczącego bajtu słowa pod wyspecyfikowanym adresem a najbardziej znaczący bajt w następnej komórce.Dlatego też,słowo pochłania dwa kolejne adresy pamięci

Rysunek 3.2: Operacja zapisu do pamięci



Rysunek 3.3 : Operacja odczytu z pamięci

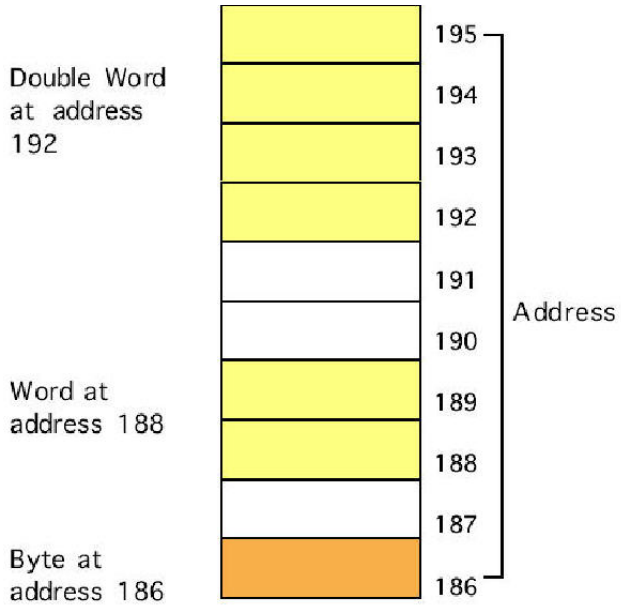
(jak można się było spodziewać,ponieważ słowo składa się z dwóch bajtów). Podobnie podwójne słowo zużywa cztery kolejne komórki pamięci.Adres podwójnego słowa jest adresem jego najmniej znaczącego bajtu.Pozostałe trzy bajty następujące po najmniej znaczącym bajcie aż do najbardziej znaczącego, pojawiają się przy adresie podwójnego słowa „ plus trzy" (zobacz rysunek 3.4).Bajty,słowa i podwójne słowa mogą zaczynać się od każdego poprawnego adresu w pamięci.Wkrótce zobaczymy,jednak,że rozpoczynanie dużych obiektów od dowolnych adresów nie jest dobrym pomysłem.

Zauważ,że jest całkiem możliwe nakładanie się na siebie wartości bajtu,słowa czy podwójnego słowa.Na przykład,na rysunku 3.4 możemy mieć słowo zaczynające się przy adresie 193,bajt przy adresie 194 i podwójne słowo zaczynające się przy adresie 192.Te wartości wszystkie nakładają się na siebie.

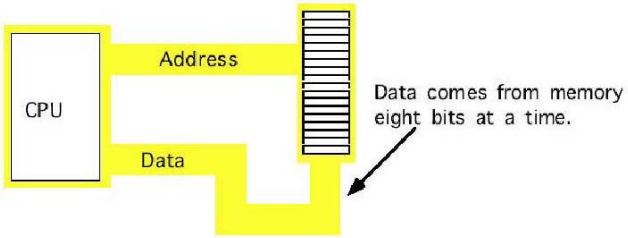
Mikroprocesor 8088 i 80188 mają ośmiobitową magistralę danych.To znaczy,że CPU może przesyłać osiem bitów na raz.Ponieważ każdy adres pamięci odpowiada ośmiobitowemu bajtowi,oznacza to wiele dogodnych ustawień (z perspektywy sprzętowej).,zobacz rysunek 3.6.

Termin „tablica pamięci adresowana bajtem” ,oznacza ,że CPU może adresować pamięć w kawałkach tak małych jak pojedynczy bajt.Znaczy to również,że jest to najmniejsza jednostka pamięci,do której możemy uzyskać dostęp od razu przez procesor.To znaczy,jeśli procesor chce uzyskać dostęp do wartości czterech bitów ,musi odczytać osiem bitów i potem zignorować pozostałe cztery bity.Również uświadomić sobie trzeba,że bajt adresujący nie sugeruje ,że CPU może uzyskać dostęp do ośmiu bitów dla każdego przypadkowego bitu granicznego.Kiedy wyspecyfikujemy adres 125 w pamięci, otrzymamy całe osiem bitów z tego adresu,ni mniej ni więcej.Adresy są całkowite ;nie możemy, na przykład, wyspecyfikować adresu 125,5 dla przeniesienia mniej niż ośmiu bitów.

8088 i 80188 mogą manipulować wartościami słowa i podwójnego słowa,nawet z ich ośmiobitową magistralą danych.Jednak,to wymaga wielokrotnych operacji na pamięci,ponieważ te procesory mogą tylko przenosić osiem bitów danych jednorazowo.Ładowanie słowa wymaga dwóch działań na pamięci;ładowanie podwójnego słowa wymaga czterech działań na pamięci.



Rysunek 3.4: Bajt,słowo i podwójne słowo przechowywane w pamięci



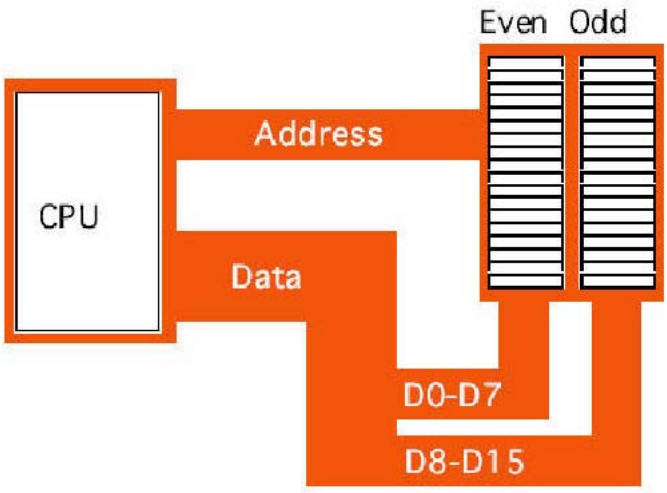
Rysunek 3.5: Wzajemne oddziaływanie ośmiobitowy CPU - Pamięć

Procesory 8086,80186,80286 i 80386SX mają szesnastobitowe magistrale danych.Pozwala to tym procesorom uzyskiwać dostęp do dwa razy takiej ilości pamięci w takiej samej ilości czasu niż ich ośmiobitowi bracia. Te procesory organizują pamięć w dwa banki:"parzysty" bank i „nieparzysty" bank (zobacz rysunek 3.6).Rysunek 3.7 ilustruje połączenie do CPU (D0-D7 oznacza mniej znaczący bajt z magistrali danych,D8-D15 oznacza bardziej znaczący bajt magistrali danych):

16 bitowi członkowie rodziny 80x86 mogą ładować słowo z każdego dowolnego adresu.Jak wspominaliśmy wcześniej,procesor pobiera mniej znaczący bajt wartości spod wyspecyfikowanego adresu a bardziej znaczący bajt z następnego, kolejnego adresu. To stwarza subtelny problem,jeśli spojrzysz dokładnie na diagram powyżej.Co się stanie,kiedy uzyskujesz dostęp do słowa spod nieparzystego adresu?Przypuśćmy,że chcesz odczytać słowo z komórki 125.Okay,najmniej znaczący bajt słowa przychodzi z komórki 125 a bardziej znaczący bajt pochodzi z komórki 126..W czym rzecz?Okazuje się,że są dwa problemy z tym związane.

Even	Odd
6	7
4	5
2	3
0	1

Rysunek 3.6:Adresy bajtów w pamięci słowa

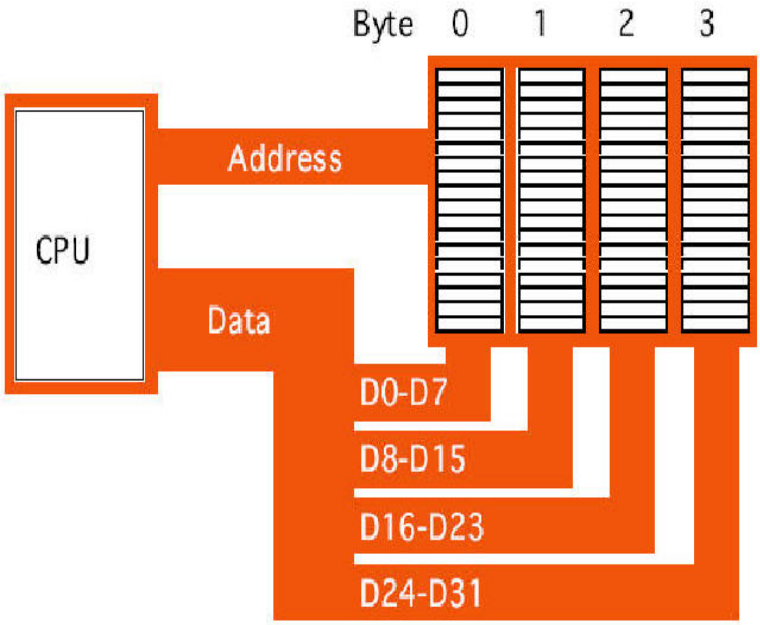


Rysunek 3.7: Organizacja pamięci 16 bitowego procesora (8086,80186,80286,80386SX)

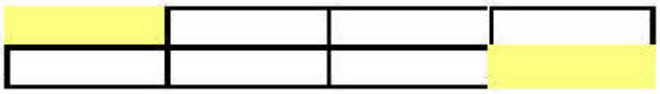
Po pierwsze, spójrz na rysunek 3.7. Linie 8-15 (bardziej znaczący bajt) magistrali danych są połączone z bankiem nieparzystym, a linie 0 -7 magistrali danych (mniej znaczący bajt) połączone są z bankiem „parzystym".Uzyskujemy dostęp do komórki pamięci 125 przesyłając dane do CPU na bardziej znaczący bajt magistrali danych.; ale my chcemy te dane na mniej znaczącym bajcie! Na szczęście , CPU 80x86 rozpoznają tą sytuację i automatycznie przesyłają dane z D8-D15 do mniej znaczącego bajta.

Drugi problem jest nawet bardziej niejasny .Kiedy uzyskujemy dostęp do słów,w rzeczywistości uzyskujemy dostęp do dwóch oddzielnych bajtów,z których każdy ma swój własny adres bajtowy.Więc powstaje pytanie „Jaki adres pojawia się na magistrali danych?" 16 bitowe CPU 80x86 zawsze umiejscawiają parzyste adresy na magistrali.Parzyste bajty zawsze pojawiają się na liniach danych D0-D7 a bajty nieparzyste zawsze pojawiają się na liniach danych D8-D15. Jeśli chcemy uzyskać dostęp do słowa przy adresie parzystym,CPU możemy pobrać całkowicie 16 bitowy kawałek ,w jednym działaniu na pamięci.Podobnie jeśli chcemy uzyskać dostęp do pojedynczego bajtu,CPU uruchamia odpowiedni bank (używając „aktywującego bajtu" linii sterującej) Jeśli bajt pojawia się pod nieparzystym adresem,CPU automatycznie przeniesie go z bardziej znaczącego bajtu na magistrali do mniej znaczącego bajtu.

Więc co się zdarzy kiedy CPU uzyska dostęp do słowa przy nieparzystym adresie,jak w przykładzie podanym wcześniej?Cóż, CPU nie może umieścić adresu 125 na magistrali adresowej i odczytać 16 bitów z pamięci.Nie ma nieparzystych adresów wychodzących z 16 bitowego CPU 80x86.Adresy są zawsze parzyste.Więc jeśli próbujesz położyć 125 na magistralę adresową,wtedy położysz 124 na magistralę adresową.Przy odczytywaniu 16 bitów z tego adresu,otrzymasz słowo od adresu 124 (mniej znaczący bajt) i 125 (bardziej znaczący bajt) - nie to czego oczekiwałeś.Uzyskanie dostępu do słowa przy adresie nieparzystym wymaga dwóch działań na pamięci. Najpierw,CPU musi odczytać bajt z pod adresu 125,potem musi odczytać bajt spod adresu 126.Ostatecznie trzeba pozamieniać pozycjami te bajty,ponieważ oba są wprowadzone na złych połówkach magistrali danych



Rysunek 3.8:Organizacja pamięci 32 bitowego procesora (80386,80486,Pentium Overdrive)



Rysunek 3.9: Uzyskanie dostępu do słowa przy (adres **mod** 4) = 3

Na szczęście, 16 bitowy CPU 80x86 ukrywa takie szczegóły przed nami.Nasze programy mogą uzyskiwać dostęp do słowa przy każdym adresie a CPU stosownie uzyska dostęp i wymieni (jeśli będzie konieczne) dane w pamięci.Jednakże,uzyskanie dostępu do słowa przy adresie nieparzystym wymaga dwóch operacji na pamięci (podobnie jak 8088/80188).Dlatego,uzyskanie dostępu do słów przy adresach nieparzystych na 16 bitowych procesorach jest wolniejsze niż przy adresach parzystych.Bądź ostrożny ustalając jak używasz pamięci możesz poprawić szybkość swojego programu.

Uzyskanie dostępu do 32 bitowej wartości zawsze zabiera,co najmniej,dwie operacje na pamięci na 16 bitowym procesorze..Jeśli chcesz uzyskać dostęp do wielkości 32 bitowej od adresu niepatrzystego,procesor będzie potrzebował trzech operacji na pamięci dla dostępu do danych.

32 bitowe procesory (80386,80486 i Pentium Overdrive) używają czterech banków pamięci połączonych do 32 bitowej magistrali danych (zobacz rysunek 3.8).Adres umiejscowiony na magistrali adresowej jest zawsze mnożony przez cztery.Używając kilku linii „bajtu aktywującego" CPU może wybierać,do którego z czterech bajtów tego adresu,program chce uzyskać dostęp.Podobnie jak przy procesorze 16 bitowym,CPU automatycznie przestawia bajty jeśli jest to konieczne.

Przy 32 bitowej pamięci,CPU może uzyskać dostęp do każdego bajtu przy jednej operacji na pamięci.Jeśli (Adres **MOD** 4) nie uzyskamy trzy,wtedy 32 bitowy CPU może uzyskać dostęp do słowa przy tym adresie używając pojedynczej operacji na pamięci.Jednak, jeśli reszta wynosi trzy, wtedy zabierze to dwie operacje na pamięci w uzyskaniu dostępu do słowa (zobacz rysunek 3.9) .Jest to ten sam problem z jakim zetknęliśmy się przy procesorach 16 bitowych, z tym,że zdarza się on o połowę częściej.

32 bitowy CPU może uzyskać dostęp do podwójnego słowa w pojedynczej operacji na pamięci jeśli adres tej wartości jest równo dzielony przez cztery.Jeśli nie,CPU wymaga dwóch operacji na pamięci.

I znowu,CPU radzi sobie ze wszystkim automatycznie.Przy ładowaniu prawidłowych danych CPU radzi sobie ze wszystkim za ciebie.Jako generalna zasadę zawsze umiejscawiaj wartości słowa pod parzystymi adresami a wartości podwójnego słowa pod adresami które są zawsze podzielne przez cztery.To przyspieszy działanie twoich programów.

3.1.3 PODSYSTEM I/O

Poza 20,24 i 32 liniami adresowymi,dzięki którym uzyskujemy dostęp do pamięci,rodzina 80x86 posiada 16 bitową magistralę adresową I/O.Daje to CPU 80x86 dwie oddzielne przestrzenie adresowe,jedną dla pamięci i jedną dla operacji I/O.Linie na magistrali sterującej rozróżniają pomiędzy adresami pamięci a I/O.Za wyjątkiem oddzielnych linii sterujących i mniejszej magistrali,adresowanie I/O następuje dokładnie tak jak adresowanie pamięci.Pamięć i urządzenia I/O obie dzielą tą samą magistralę danych i mniej znaczące 16 linii na magistrali adresowej

Są trzy ograniczenia dotyczące podsystemu I/O na IBM PC: po pierwsze, CPU 80x86 wymagają specjalnych instrukcji dla uzyskania dostępu do urządzeń I/O; po drugie, projektanci z IBM PC użyli „najlepszych" lokacji I/O dla swoich własnych celów,zmuszając innych do używania mniej osiągalnych lokacji;po trzecie, system 80x86 może adresować nie więcej niż 65,536 (2^{16}) adresów I/O. Kiedy popatrzymy, że typowa karta graficzna VGA wymaga ponad 128 000 różnych loakacji,widać,że będziemy mieli problem z rozmiarem magistrali I/O.

Na szczęście,projektanci sprzętu mogą odwzorowywać swoje urządzenia I/O wewnątrz przestrzeni adresowej pamięci,tak łatwo jak w przestrzeni adresowej I/O.Tak więc przez użycie odpowiednich zespołów układów,mogą uczynić urządzenia I/O wyglądające tak jak pamięć.

Dostęp do urządzeń I/O jest tematem,który powróci w późniejszych rozdziałach.Na razie założymy,że dostęp do I/O i pamięci następuje w ten sam sposób.

3.2 SYSTEM SYNCHRONIZACJI

Chociaż nowoczesne komputery są całkiem szybkie i stają się szybsze, cały czas,wymagają jeszcze skończonej ilości czasu do osiągnięcia nawet najmniejszego zadania.Na maszynach Von Neumanna,takich jak 80x86,większość operacji jest szeregowanych.To znaczy,że komputer wykonuje polecenia w określonym porządku.. Nie wykona ,na przykład,wyrażenia I:=I*5+2 przed I:=J; w następującej sekwencji:

```
I:=J;
I:=I*5+2;
```

Najwyraźniej potrzebujemy sposobu do sterowania które wyrażenie wykonać pierwsze a które drugie .

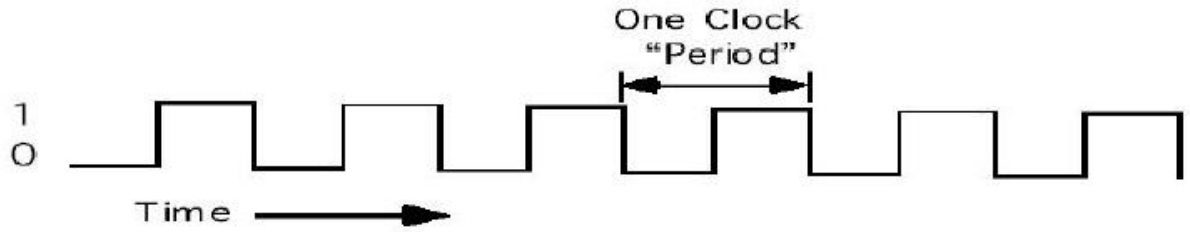
Oczywiście,w rzeczywistym systemie komputerowym, operacje nie występują natychmiastowo. Przesunięcie kopii J do I zabiera określoną ilość czasu. Podobnie mnożenie I przez 5 a potem dodanie dwa i

zachowanie wyniku w I zabiera czas.Jak można się było spodziewać,drugie wyrażenie Pascalowskie zabiera wykonuje się troszkę dłużej niż pierwsze.Dla zainteresowanych pisaniem szybkich programów,naturalnym pytanie jest „Jak procesor wykonuje wyrażenia, i jak mierzymy,jak długo się one wykonują?"

CPU jest bardzo złożonym elementem zespołu układów.Bez zagłębienia się w zbyt wiele szczegółów,możemy powiedzieć,że operacje wewnątrz CPU muszą być bardzo ostrożnie koordynowane ,lub CPU będzie tworzył błędne rezultaty.W celu zapewnienia,że wszystkie operacje odbędą się we właściwym momencie,CPU 80x86 używa alternatywnych sygnałów nazywanych systemem zegarowym.

3.2.1 ZEGAR SYSTEMOWY

Przy większości podstawowych poziomów zegar systemowy obsługuje całą synchronizację wewnątrz systemu komputerowego.Zegar systemowy ,jest to sygnał elektryczny na magistrali sterującej,który naprzemiennie przechodzi od zero do jeden,w okresowym tempie (zobacz rysunek 3.10). CPU jest dobrym przykładem złożonego synchronicznego systemu logicznego (zobacz poprzedni rozdział).Zegar systemowy zawiera dużo logicznych bramek które przygotowują CPU pozwalając mu działać w sposób zsynchronizowany.



Rysunek 3.10: Zegar systemowy

Częstotliwość z jaką zegar systemowy przechodzi od zera do jeden,jest to „częstotliwość zegara systemowego". Czas jaki jest potrzebny zegarowi systemowemu do przełączenia między zero i jeden i ponownie do zera nazywa się „taktem zegarowym".Jeden pełny okres nazywany jest również „cyklem zegarowym". W większości nowoczesnych systemów komputerowych,zegar systemowy przełącza między zero i jeden w tempie przekraczającym kilka milionów razy na sekundę.Częstotliwość zegara jest po prostu liczbą cykli zegarowych które wykonują się w ciągu sekundy.Typowy chip 80486 pracuje z szybkością 66 milionów cykli na sekundę."Herc" (Hz) jest technicznym terminem oznaczającym jeden cykl na sekundę.Dlatego też,wyżej wymieniony chip 80486 pracuje przy 66 milionach herców,lub inaczej 66 Megahercach (MHz).Typowa częstotliwość dla części 80x86 to zakres od 5 MHz do 200 MHz i wyższa.Zauważ,że jeden takt zegarowy (ilość czasu dla jednego kompletnego cyklu zegarowego) jest wartością odwrotną do częstotliwości zegara.Na przykład, zegar 1MHz będzie miał takt zegarowy jedną mikrosekundę (1/1,000,000 sekundy).Podobnie zegar 10 MHz,ma takt zegarowy 100 nanosekund (100 miliardów na sekundę).CPU pracujący przy 50 MHZ ma takt zegarowy 20 nanosekund.Zauważ,że zazwyczaj wyrażamy takt zegarowy w milionach lub miliardach na sekundę.

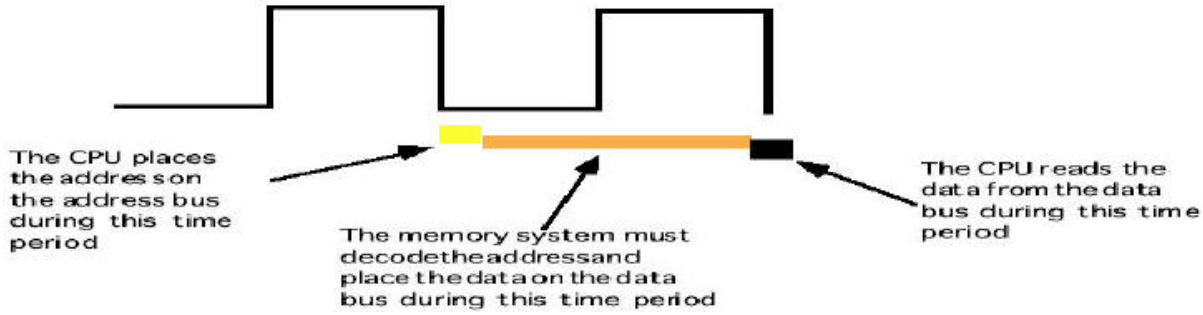
Dla zapewnienia synchronizacji większość CPU zaczyna operacje albo przy zboczu opadającym (kiedy zegar opada z jeden na zero) albo przy zboczu wznoszącym (kiedy zegar rośnie od zera do jeden) Zegar systemowy poświęca większość swojego czasu albo zeru albo jedynce a bardzo mało czasu przełączając między nimi dwoma.Dlatego też zbocze zegarowe jest doskonałym punktem synchronizującym.

Ponieważ wszystkie operacje CPU są synchronizowane poprzez zegar,CPU nie może wykonywać zadań szybciej niż zegar.Jednak,ponieważ CPU pracuje przy jakiejś częstotliwości zegara,nie znaczy to,że jest wykonywane tak dużo operacji w każdej sekundzie.Wiele operacji używa wielokrotności taktu zegarowego dla zakończenia więc CPU często wykonuje operacje przy znacznie niższym tempie.

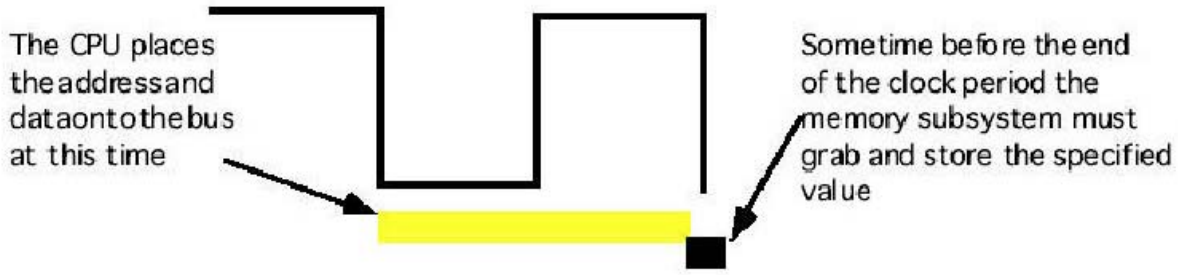
3.2.2 DOSTĘP DO PAMIĘCI A ZEGAR SYSTEMOWY

Dostęp do pamięci jest prawdopodobnie najpowszechniejszym zajęciem CPU.Dostęp do pamięci jest zdecydowanie operacją synchronizowaną przez zegar systemowy.To znaczy,odczytywanie wartości z pamięci lub zapisywanie wartości zdarza się nie częściej niż raz na każdy cykl zegarowy.Istotnie,na wielu procesorach 80x86,jest potrzebnych kilka cykli zegarowych przy dostępie do komórki pamięci."Czas dostępu do pamięci" jest liczbą cykli zegarowych,których system wymaga przy dostępie do komórki pamięci;jest to ważna wartość ponieważ dłuższy czas dostępu do pamięci prowadzi do niższej wydajności.

Różne procesory 80x86 mają różne czasy dostępu do pamięci,w zakresie od jednego do czterech cykli zegarowych.Na przykład,8088 i 8086 wymagają czterech cykli zegarowych przy dostępie do pamięci;80486 wymaga tylko jednego.Zatem,80486 będzie wykonywał programy z dostępem do pamięci szybciej niż 8086,nawet kiedy pracują przy tej samej częstotliwości zegara.



Rysunek 3.11: Cykl odczytu z pamięci dla 80486



Rysunek 3.12: Cykl zapisu do pamięci dla 80486

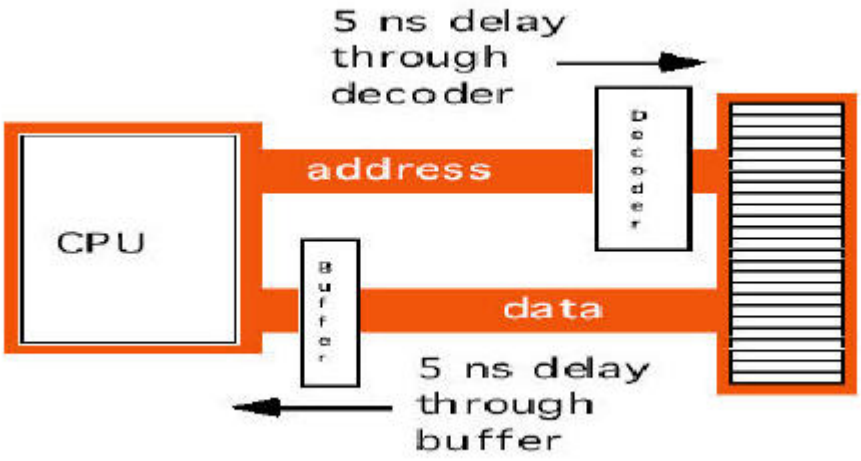
Czas dostępu do pamięci jest to różnica czasu między operacją na pamięci (odczyt lub zapis) a czasem zakończenia tej operacji.Na 5 MHz CPU 8088/8086 czas dostępu do pamięci wynosi mniej więcej 800 nanosekund.Na 50 MHz 80486,ten czas jest mniejszy niż 20 ns.Zauważ,że czas dostępu do pamięci dla 80486 jest 40 razy szybszy niż dla 8088/8086.Jest tak ponieważ częstotliwość zegara 80486 jest 10 krotnie większa i wykorzystuje on jedną czwartą cyklu zegarowego przy dostępie do pamięci.

Kiedy odczytujemy z pamięci,czas dostępu do pamięci jest to ilość czasu od punktu w którym CPU umieszcza adres na magistrali adresowej a CPU zbiera dane z magistrali danych.Na CPU 80486 z jednym cyklem czasu dostępu do pamięci,odczyt wygląda podobnie jak przedstawiony na rysunku 3.11.Zapisywanie danych do pamięci jest podobne (zobacz rysunek 3.11).

Zauważ,że CPU nie czeka na pamięć.Czas dostępu jest wyspecyfikowany przez częstotliwość zegara.Jeśli podsystem pamięci nie pracuje dostatecznie szybko,CPU będzie odczytywał dane dziwne w czasie operacji odczytu z pamięci i nie będzie odpowiednio przechowywał danych dla operacji zapisu do pamięci.Będzie to z pewnością przyczyna nieprawidłowej pracy systemu.

Pamięć ma różne parametry ale dwoma najważniejszymi są pojemność i szybkość (czas dostępu). Typowy dynamiczny RAM (pamięć o dostępie swobodnym) ma pojemność od czterech (lub więcej) megabajtów i szybkość 50-100 ns.Można kupić większe i szybsze urządzenia,ale są zbyt drogie.Typowy system 33 MHz 80486 używa pamięci 70 ns .

Poczekaj chwilę!Przy 33 MHz takt zegarowy wynosi mniej więcej 33 ns. Jak twórca komputera może wykorzystać 70ns pamieci?Odpowiedzią jest stan oczekiwania (WAIT)



Rysunek 3.13:Dekodowanie i buforowanie opóźnień

3.2.3 STAN OCZEKIWANIA

Stan oczekiwania jest niczym innym jak dodatkowym cyklem zegarowym dający jakiemuś urządzeniu czas na zakończenie operacji. Na przykład, 50 Megahercowy 80486 ma takt zegarowy 20 ns.To sugeruje,że potrzebujemy pamięci 20 ns .W rzeczywistości sytuacja jest gorsza.W większości systemów komputerowych jest dodatkowy zespół układów między CPU a pamięcią: dekodowania i buforowania logicznego. Te dodatkowe układy wprowadzają dodatkowe opóźnienie do systemu. (zobacz rysunek 3.13).Na tym diagramie system traci 10 ns na dekodowanie i buforowanie.Więc jeśli CPU potrzebuje danych w 20 ns,pamięć musi odpowiedzieć w mniej niż 10 ns.

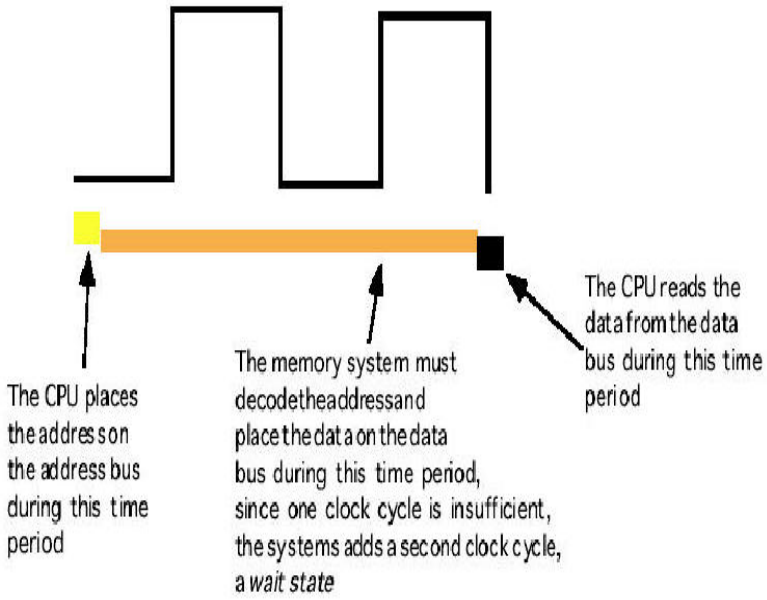
Możemy właściwie kupić pamięć 10 ns.Jednak jest to bardzo kosztowne,nieporęczne,pochłaniające dużo mocy i generujące dużo ciepła. To są złe cechy.Superkomputery używają pamięci tego typu.Jednak superkomputery kosztują miliony doalrów,zajmują całe pokoje,wymagają specjalnego chłodzenia i mają gigantyczne zasilanie.Żadna z tych rzeczy raczej nie zmieści się na Twoim biurku.

Jeśli kosztowna pamięć nie chce pracować z szybkim procesorem,jak poradzić sobie bez kupowania szybszego PC? Jedną z odpowiedzi jest stan oczekiwania.Na przykład,jeśli mamy procesor 20 MHz z czasem cyklu pamięci 50 ns i tracimy 10 ns na buforowanie i dekodowanie,będziemy potrzebować pamięci 40 ns.A co jeśli możemy pozwolić sobie na pamięć 80ns w 20 MHZ systemie? Dodatkowy stan oczekiwania rozszerza cykl pamięci do 100 ns (dwa cykle zegarowe) co rozwiąże ten problem.Odjęcie 10 ns na dekodowanie i buforowanie pozostawi 90 ns.Zatem,pamięć 80 ns odpowie zanim CPU zażyczy sobie danych.

Prawie każdy istniejący CPU dostarcza sygnał na magistralę sterującą pozwalając na wprowadzenie stanu oczekwiania.Generalnie,zestaw układów dekodujących zapewnia opóźnienie tej linii o jeden dodatkowy takt zegarowy,jeśli to konieczne.Daje to pamięci wystarczający czas dostępu a system pracuje właściwie.(zobacz rysunek 3.14).

Czasami pojedynczy stan oczekiwania jest niewystarczający.Rozpatrzmy 80486 pracujący przy 50 MHz.Normalnie czas cyklu pamięci jest mniejszy niż 20 ns.Dlatego też,mniej niż 10 ns jest dostępnych po odjęciu dekodowania i buforowania.Jeśli używamy pamięci 60 ns w systemie,dodanie pojedynczego stanu oczekiwania nie zrobi tej sztuczki. Każdy stan oczekiwania daje nam 20 ns,więc pojedynczy stan oczekiwania potrzebowałby pamięci 30 ns.Pracując z pamięcią 60 ns będziemy musieli dodać trzy stany oczekiwania (zerowy stan oczekiwania = 10 ns, jeden stan oczekiwania = 30 ns,dwa stany oczekiwania =50 ns,trzy stany oczekwiania=70 ns).

Rzecz jasna ,z punktu widzenia wydajności systemu,stany oczekiwania nie są dobrą rzeczą.Podczas gdy CPU czeka na dane z pamięci,nie może operować na danych.



Rysunek 3.14 : Wprowadzanie stanu oczekiwania do operacji odczytu z pamięci

Dodanie pojedynczego stanu oczekiwania do cyklu pamięci w CPU 80486 podwaja ilość czasu wymaganą dla uzyskania dostępu do danych.To z kolei, zmniejsza szybkość dostępu do pamięci.Wykonywanie ze stanem oczekiwania przy każdym dostępie do pamięci jest prawie jak przecięcie częstotliwości zegara procesora na połowę.Możemy wtedy wykonać dużo mniej pracy w tej samej ilości czasu.

Prawdopodobnie widziałeś coś takiego :80386DX,33MHz,8megabajtów 0 stanów oczekiwania RAM ..... tylko $1000!"Jeśli przyjrzałeś się dokładnie tej specyfikacji,zauważyłeś,że producent używa pamięci 80 ns..Jak można zbudować system który pracuje przy 33 MHz i ma zero stanów oczekiwania? Prosto.Kłamiąc.

Nie ma mowy,żeby 80386 mógł pracować przy 33MHz,wykonując dowolny program,bez wprowadzania stanu oczekiwania.To jest niemożliwe.Jednak możliwe jest zaprojektowanie podsystemu pamięci który pod pewnymi,specjalnymi warunkami dałby sobie radę przy działaniu bez stanu oczekiwania jakiś czas.

Jednak,nie jesteśmy skazani na wolne wykonywanie przez dodanie stanu oczekiwania.Jest kilka sztuczek projektantów sprzętu dzięki którym możemy osiągać zerowy stan oczekiwania przez większość czasu.Najbardziej powszechną z nich jest użycie pamięci podręcznej cache**.**

3.2.4 PAMIĘĆ PODRĘCZNA CACHE

Jeśli patrzymy na typowy program odkryjemy,że ma zwyczaj uzyskiwać dostęp do tej samej komórki pamięci wielokrotnie.Co więcej odkryjemy,że program często uzyskuje dostęp do sąsiednich komórek pamięci.Techniczne nazwy tegoż zjawiska to czasowa lokalność odniesienia i przestrzenna lokalność odniesienia.Jeśli wskazujemy lokalność przestrzenną,program uzyskuje dostęp do sąsiednich komórek pamięci.jeśli wskazujemy czasową lokalność odniesienia program wielokrotnie uzyskuje dostęp do tej samej komórki pamięci podczas krótkiego odcinku czasu.Obie formy lokalności występują w następującym fragmencie kodu pascalowskiego:

```
For i:= 0 to 10 do
        A[i] := 0;
```

Występują wewnątrz tej pętli zarówno przestrzenna i czasowa lokalność odniesienia

W powyższym pascalowym kodzie,program odnosi się do zmiennej i kilka razy.Pętla for, przypisuje zmiennej i wartości od 0 do 10,do czasu aż pętla się skończy.Zwiększa również i o jeden po wykonaniu każdego przypisania.To wyrażenie używa również i jako indeksu tablicy.To pokazuje czasową lokalność odniesienia w akcji, ponieważ CPU uzyskuje dostęp do i w trzech punktach,w krótkim okresie czasu.Ten program pokazuje również przestrzenną lokalność odniesienia.Pętla wypełnia zerami elementy tablicy A poprzez zapis zera do pierwszego

elementu w tablicy A,potem do drugiego elementu i tak dalej .Zakładając, że Pascal przechowuje elementy z A w kolejnych komórkach pamięci,każda iteracja pętli uzyskuje dostęp do sąsiedniej komórki pamięci.

Jest to dodatkowy przykład czasowej i przestrzennej lokalności odniesienia w powyższym przykładzie w Pascalu,chociaż nie jest on tak oczywisty.Instrukcje komputerowe które mówią systemowi co robić z wybranym zadaniem,również występują w pamięci.Te instrukcje pojawiają się sekwencyjnie w pamięci - część lokalności przestrzennej.Komputer również wykonuje te instrukcje wielokrotnie, raz dla każdej iteracji pętli - część lokalności czasowej.

Jeśli popatrzymy na charakterystykę wykonania typowego programu,odkryjemy,że typowy program wykonuje mniej niż połowę wyrażeń.Generalnie,typowy program może używać tylko 10 - 20% przydzielonej mu pamięci.W danym czasie, program jednomegabajtowy może uzyskać dostęp od czterech do ośmiu kilobajtów danych i kodu.Więc jeśli płacisz oburzające sumy pieniędzy za drogie RAMy z zerowym stanem oczekiwania,nie będziesz mógł używać większości z nich**.**Czyż nie byłoby milej,jeśli mógłbyś kupić mniejszą ilość szybszego RAMu z dynamiczniejszym przydzielaniem adresów przy wykonywaniu programu?

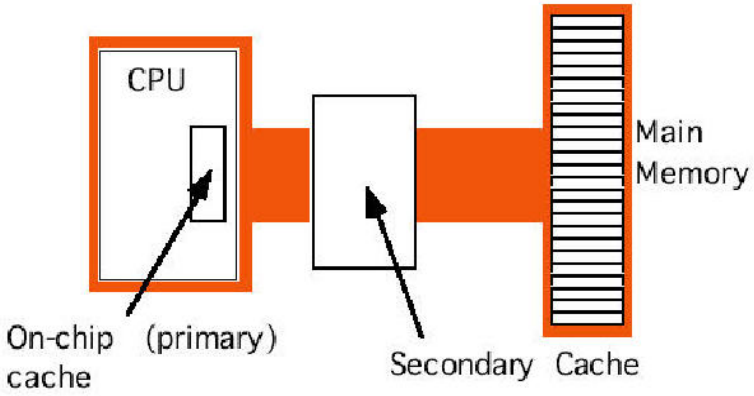
To jest właśnie to co pamięć podręczna cache robi dla nas.Pamięć cache „siedzi" między CPU i pamięcią główną.Jest to mała ilość bardzo szybkiej (zero stanu oczekiwania) pamięci.W odróżnieniu od normalnej pamięci,bajty pojawiające się w cache'u nie mają stałych adresów.Zamiast tego,pamięć cache może zmienić przydział adresom danych.Pozwala to systemowi zachować ostatnio udostępnioną wartość danej w cache'u.Adresy,do których CPU nie uzyskał dostępu w pozostają w głównej (wolnej) pamięci.Ponieważ większość dostępów do pamięci to ostatnie uzyskane dostępy do zmiennych (lub blisko położonej komórki ostatnio uzyskanego dostępu do innej komórki),dane generalnie pojawiają się w pamięci cache.

Pamięć cache,nie jest doskonała.Chociaż program może spędzić znaczną ilość czasu na wykonywaniu kodu w jednym miejscu,ostatecznie może wezwać procedurę lub przejść do jakiejś sekcji kodu na zewnątrz pamięci cache.W takim przypadku CPU musi przejść do pamięci głównej aby pobrać dane.Ponieważ pamięć główna jest wolna będzie to wymagało wprowadzenia stanu oczekiwania.

Trafienie z pamięci podręcznej występuje ,gdy CPU uzyskuje dostęp do pamięci i znajduje dane w cache'u.W takim przypadku CPU może zazwyczaj uzyskać dane z zerowym stanem oczekiwania.Brak trafienia z pamięci podręcznej,występuje jeśli CPU uzyskuje dostęp do pamięci a dane nie są przechowywane w pamięci podręcznej.Wtedy CPU może odczytać dane z pamięci głównej, ponosząc utratę wydajności.Wykorzystując korzyści z lokalności odniesienia,CPU kopiuje dane do pamięci podręcznej zawsze kiedy przystępuje do adresu nie zawartego w pamięci podręcznej cache.Ponieważ jest prawdopodobne,że system będzie chciał wkrótce uzyskać dostęp to tej samej lokacji,system zaoszczędzi na stanie oczekiwania poprzez posiadanie danych w pamięci podręcznej

Jak opisano powyżej,pamięć podręczna posługuje się czasowymi aspektami dostępu do pamięci,ale nie aspektami przestrzennych.Buforowanie podręczne komórek pamięci kiedy uzyskujemy dostęp do nich nie przyspiesza programu jeśli mamy stały dostęp do kolejnych komórek.(przestrzenna lokalność odniesienia)Rozwiązanie tego problemu jest większe buforowanie systemu przy odczytywaniu kilku kolejnych bajtów z pamięci kiedy występuje brak trafienia z pamięci podręcznej.Na przykład 80486 odczytuje 16 bajtów przy strzale do spudłowanego (nie trafionego) cache'a..Jeśli odczytujemy 16 bajtów,dlaczego odczytujemy je w blokach zamiast tak jak potrzebujemy?Jak się okazuje,większość chipów pamięci, dostępnych dzisiaj,ma specjalny tryb który pozwala szybko uzyskać dostęp do kilku kolejnych komórek pamięci **.**Pamięć podręczna cache wykorzystuje tą zdolność do redukowania liczby stanów oczekiwania potrzebnych przy dostępie do pamięci.

Jeśli piszemy program który losowo uzyskuje dostęp do pamięci,używajac pamięci podręcznej cache można w rzeczywistości go spowolnić.Odczytywanie 16 bajtów przy każdym braku trafienia w pamięci podręcznej cache jest kosztowne jeśli uzyskujemy dostęp tylko do kilku bajtów w odpowiedniej linii cache.



Rysunek 3.15: Dwupoziomowy system cache

Nie powinno to być niespodzianką że stosunek trafień w pamięci podręcznej do braku trafień w pamięci podręcznej wzrasta wraz z rozmiarem (w bajtach) podsystemu pamięci cache.Chip 80486,na przykład, ma 8,192 zintegrowanej z układem pamięci podręcznej cache.Intel twierdzi,że uzyskuje współczynnik trafień 80-95% trafień na taką pamięcią podręczną (w znaczeniu 80-95% czasu w jakim CPU znajduje dane w pamięci podręcznej). Wydaje się to bardzo imponujące.Jednakże,jeśli pobawimy się troszkę liczbami odkryjemy,że nie jest to wszystko imponujące.Przypuśćmy,że wybraliśmy 80% cyfr.Wtedy, średnio ,jeden z każdych pięciu dostępów do pamięci,nie będzie w pamięci podręcznej.Jeśli mamy procesor 50 MHz i pamięć o czasie dostępu 90 ns cztery z pięciu dostępów do pamięci potrzebują tylko jednego cyklu zegarowego (ponieważ są w pamięci podręcznej cache) a piąty będzie potrzebował około 10 stanów oczekiwania.Generalnie system wymaga 15 cykli zegarowych przy dostępie do pięciu komórek pamięci,lub trzech cyklów zegarowych na dostęp.Jest to odpowiednik dwóch stanów oczekiwania dodanych do każdego dostępu do pamięci.Teraz już wierzysz,że Twoja maszyna, pracuje przy zerowym stanie oczekiwania?

Jest parę sposobów na poprawienie tej sytuacji.Po pierwsze,możemy dodać więcej pamięci podręcznej.To poprawi współczynnik trafień w pamięci podręcznej,zredukuje liczbę stanów oczekiwania.Na przykład, zwiększenie współczynnika trafień z 80% do 90% pozwoli nam uzyskać dostęp do 10 komórek pamięci w ciągu 20 cykli.To zredukuje średnią liczbę stanów oczekiwania na dostęp do pamięci do jednego stanu oczekiwania - solidna poprawa. Niestety nie można wyciągnąć chipu 80486,rozebrać na części i przylutować więcej pamięci podręcznej na chipe.Jednak,CPU 80586/Pentium ma znacznie większą pamięć podręczną niż 80486 i operuje zmniejszą ilością stanów oczekiwania.

Innym sposobem poprawienia wydajności jest zbudowanie dwupoziomowego systemu pamięci podręcznej. Wiele systemów 80486 pracuje w ten sposób.Pierwszym poziomem jest zintegrowana z układem 8,192 bajtowa pamięć podręczna.Następnym poziomem ,pomiędzy zintegrowaną pamięcią podręczną a pamięcią główną jest pomocnicza pamięć podręczna wbudowany na płycie głównej komputera.(zobacz rysunek 3.15).

Typowa pomocnicza pamięć podręczna zawiera gdzieś od 32,786 do jednego megabajta pamięci.Powszechnymi rozmiarami na PC są 65,536 i 262,114 bajty pamięci podręcznej.

Możemy zapytać ”Dlaczego zawracać sobie głowę dwupoziomowym cache’m.?Dlaczego nie użyć 262,144 bajtów pamięci podręcznej cache od razu” Cóż,pomocnicza pamięć podrzędna generalnie nie operuje przy zerowym stanie oczekiwania.Układ wspierający 262,144 bajty z 10 ns pamięcią (20 ns całkowitego czasu dostępu) byłby bardzo kosztowny.Większość projektantów systemu używa wolniejszych pamięci,które wymagają jednego lub dwóch stanów ozekiwania.To jest i tak dużo szybsze niż pamięć główna.W połączeniu ze zintegrowaną z układem pamięcią podręczną **,**możemy uzyskać lepszą wydajność systemu.

Rozważmy poprzedzi przykład z 80% wskaźnikiem trafień.Jeśli pomocnicza pamięć podręczna wymaga dwóch cykli dla każdego dostępu do pamięci i trzech cykli dla pierwszego dostępu ,wtedy brak trafień na zintegrowanej z układem pamięci podręcznej będzie wymagał całkowicie sześciu cykli zegarowych.Średnią wydajnością systemu będą dwa cykle na dostęp do pamięci.Trochę szybciej niż trzy wymagane przez system bez pomocniczej pamięci podręcznej.Co więcej,pomocnicza pamięć podręczna może uaktualniać swoje wartości równolegle z CPU.Więc liczba braku trafień w pamięci podręcznej (które wpływają na osiągi CPU) idzie w dół.

Prawdopodobnie myślisz ”Dotychczas,to wszystko wydaje się interesujące,ale co to ma wspólnego programowania?”Całkiem sporo,w rzeczywistości.Pisząc swoje programy starannie,wykorzystując sposób z systemem pamięci podręcznej,możemy poprawić wydajność swojego programu.Przez alokację zmiennych których powszechnie używamy razem w tej samej linii cache,możemy wymusić na pamięci podręcznej załadowanie tych zmiennych jako grupy,oszczędzając extra stany oczekiwania na każdy dostęp.

Jeśli organizujemy, nasz program tak,że będzie wykonywał tą samą sekwencję instrukcji wielokrotnie będzie miał wysoki stopień czasowej lokalności odniesienia i zarazem,szybsze wykonywanie.

3.3 „HIPOTETYCZNE” PROCESORY 886,8286,8486 I 8686

Po zrozumienie jak można poprawić wydajność systemu,czas zgłębić wewnętrzne operacje CPU.Niestety,procesory z rodziny 80x86 są złożonymi bestiami. Omówienie ich wewnętrznych operacji mogłoby być przyczyną większego zamieszania niż rozjaśnienia sprawy.Więc użyjemy procesorów 886,8286,8486 i 8686 (procesorów „x86”) Te „papierowe procesory” są ekstremalnym uproszczeniem różnych członków rodziny 80x86.Określają one ważne cechy architektury 80x86.

Procesory 886,8286,8486 i 8686 są prawie identyczne z wyjątkiem sposobu wykonywania instrukcji.One mają ten sam zbiór rejestrów, i „wykonują” ten sam zbiór instrukcji.Zdanie to zawiera kilka nowych pomysłów;zaatakujmy je od razu.

3.3.1 REJESTRY CPU

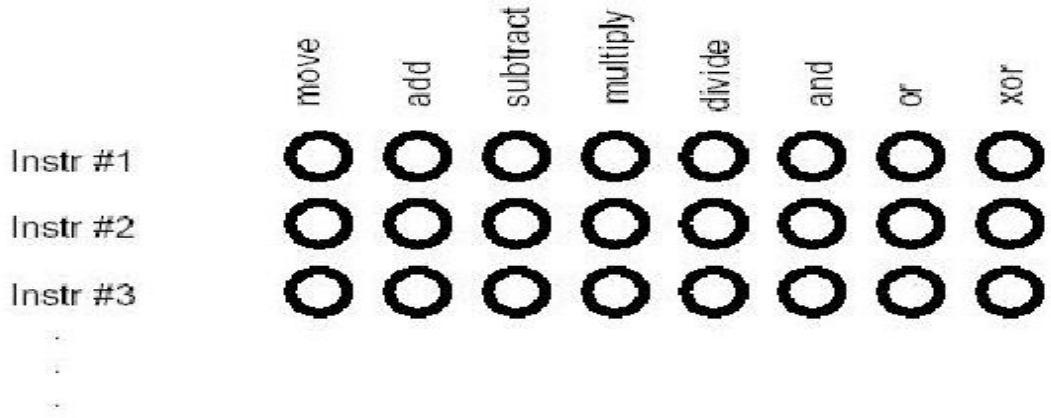
Rejestry CPU są to bardzo specjalne komórki pamięci zbudowane z przerzutników.Nie są one częścią pamięci głównej,CPU implementuje je jako zintegrowane z układem. Różni członkowie rodziny 80x86 mają różne rozmiary rejestrów,CPU 886,8286,8486 i 8686 maja dokładnie cztery rejestry,wszystkie o szerokości 16 bitów.Wszystkie operacje arytmetyczne i na komórkach zdarzają się w rejestrach.

Ponieważ procesor x86 ma tylko kilka rejestrów,nadamy każdemu rejestrowi jego własną nazwę i będziemy się odnosić do nich poprzez nazwę zamiast przez adres.Nazwy dla rejestrów x86 to:

AX - akumulator
BX - rejestr bazowy
CX - licznik
DX - rejestr danych

Poza tymi rejestrami wymienionymi u góry,które są widoczne dla programisty ,procesor x86 ma również rejestr wskaźnika rozkazów (IP),który zawiera adres następnej instrukcji do wykonania.Jest również rejestr znaczników (flag),który przechowuje wyniki porównań.Restr flag zapamiętuje czy jedna wartość była mniejsza niż,równa czy tez większa niż inna wartość.

Ponieważ rejestry są zintegrowane z układem i obsługiwane specjalnie przez CPU,muszą być dużo szybsze niż pamięć.Dostęp do komórki pamięci wymaga jednego lub więcej cyklu zegarowego.Dostęp do danych w rejestrach zabiera zero cykli zegarowych.Dlatego możemy spróbować trzymać zmienne w rejestrach. Zbiór rejestrów jest bardzo mały a większość rejestrów ma specjalne przeznaczenie które ogranicza ich używanie jako zmiennych,ale są one doskonałym miejscem na przechowywanie danych tymczasowych.



Rysunek 3.16 Tablica Połączeń programowych

3.3.2 JENDOSTKA ARYTMETYCZNO -LOGICZNA

Jednostka arytmetyczno-logiczna (ALU) występuje tam gdzie ma miejsce większość zdarzeń wewnątrz CPU.Na przykład,jeśli chcemy dodać wartość pięć do rejestru AX,CPU:

Kopiuje wartość z AX do ALU
Wysyła wartość pięć do ALU
Informuje ALU by dodało razem te wartości
Przenosi z powrotem wynik do rejestru AX

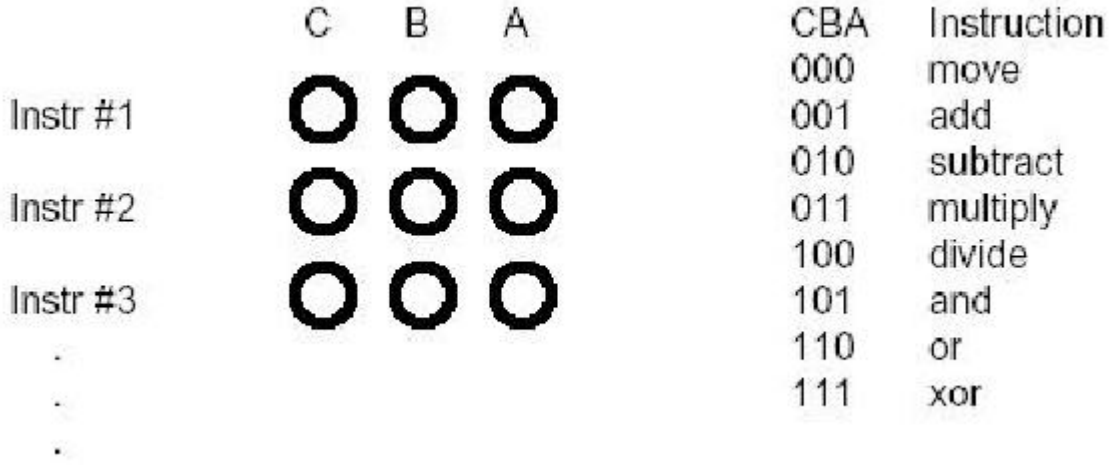
3.3.3 JEDNOSTKA SPRZĘGAJĄCA Z MAGISTRALĄ

Jednostka sprzęgająca z magistralą (BIU) jest odpowiedzialna za sterowanie magistral adresowej i danych kiedy uzyskujemy dostęp do pamięci głównej.Jeśli jest obecna pamięć podręczna,wtedy BIU jest również odpowiedzialna za uzyskiwanie dostępu do danych w tej pamięci.

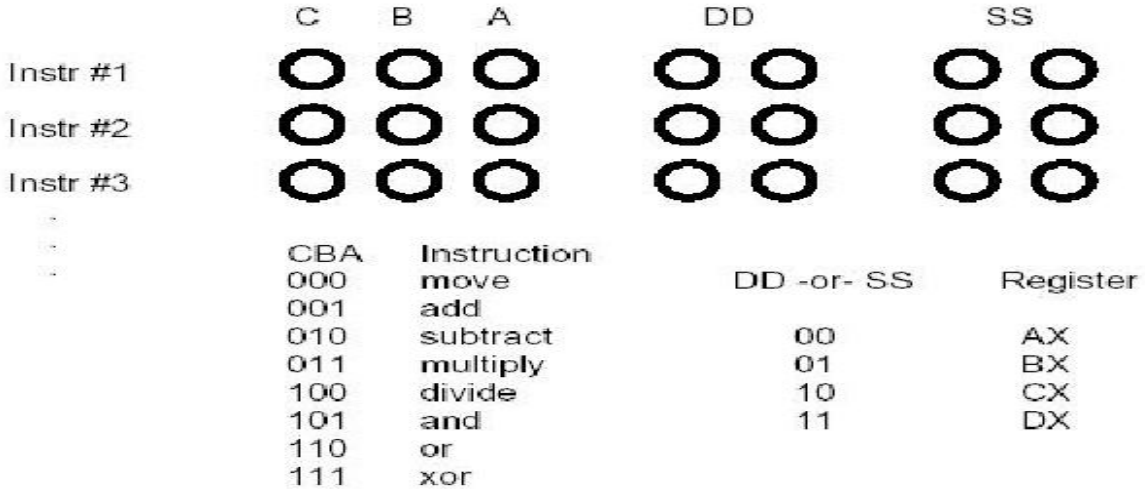
3.3.4 JEDNOSTKA STERUJĄCA I ZBIÓR INSTRUKCJI

W tym punkcie rodzi się pytanie:”Jak dokładnie CPU wykonuje przydzielone zadania?”Jest to znakomite pytanie.zważywszy,że CPU pracuje na stałym zbiorze poleceń lub instrukcji.Zapamiętajmy,że projektanci CPU,skonstruowali te procesory używając bramek logicznych do wykonywania tych instrukcji..Utrzymują liczbę bramek logicznych w rozsądnym małym zbiorze (dziesięć lub sto tysięcy).Projektanci CPU muszą z konieczności ograniczać liczbę i złożoność poleceń rozpoznawanych przez CPU.Ten mały zbiór poleceń to zbiór instrukcji CPU.

Wczesne programy (przed Von Neumannem) były często „zaszyte” wewnątrz układów połączeń.To znaczy,komputerowe połączenia decydowały jaki problem komputer mógłby rozwiązać.Musiano wymieniać układ połączeń żeby zmienić program.Bardzo trudne zadanie.Następnym posunięciem w projektowaniu komputerów były programowalne systemy komputerowe które pozwalały programiście łatwo „wymieniać” kolejność gniazdek i podłączać przewody.Program komputerowy składał się ze zbioru rzędu dziur(gniazdek),gdzie każdy rząd przedstawiał jedną operację w czasie wykonywania programu.Programista mógł wybrać jedną z kilku instrukcji,poprzez wetknięcie przewodu do odpowiedniego gniazdka dla żądanej instrukcji. (zobacz rysunek 3.16).Oczywiście,główną trudnością na tym schemacie jest to,że liczba możliwych instrukcji jest poważnie ograniczona przez liczbę gniazdek ,jaką fizycznie możemy umieścić w jednym rzędzie.Jednak projektanci CPU szybko odkryli,że mała ilość dodatkowych układów logicznych,mogłaby zredukować liczbę gniazdek wymaganych z n-dziur dla n-instrukcji do $\log_2(n)$ dziur dla n-instrukcji.Zrobili to poprzez przydzielenie kodu liczbowego do każdej instrukcji a potem kodowaniu tych instrukcji jako liczby



Rysunek 3.17: Kodowanie instrukcji



Rysunek 3.18:Kodowanie instrukcji polami źródła i przeznaczenia

binarnej używając $\log_2(n)$ dziur (zobacz rysunek 3.17),Wymagało to dodatkowych ośmiu funkcji logicznych do dekodowania bitów A,BiC z tablicy połączeń**,** ale ten extra układ jest wart tego kosztu ponieważ redukuje liczbę gniazdek które muszą być powtarzane dla każdej instrukcji.

Oczywiście wiele instrukcji CPU nie jest autonomicznych.Na przykład, instrukcja przesyłania jest poleceniem które przesyła dane z jednej lokacji w komputerze do innej (np. z jednego rejestru do innego).Dlatego instrukcja ta wymaga dwóch argumentów(operandów):argumentu źródłowego i argumentu przeznaczenia.Projektanci CPU zazwyczaj kodują te operandy źródłowy i przeznaczenia jako część instrukcji maszynowych,pewnych gniazdek odpowiadających operandowi źródłowemu i pewnych gniazdek odpowiadających operandowi przeznaczenia..Rysunek 3.17 pokazuje jedną z możliwych kombinacji gniazdek wyjaśniających to.Instrukcja move przesunęłaby dane z rejestru źródłowego do rejestru przeznaczenia,instrukacja add dodałaby wartość z rejestru źródłowego do rejestru przeznaczenia,itd.

Jednym z podstawowych postępów w projektowaniu komputerów jakie wprowadziła VNA jest koncepcja programu w pamięci .Jeden wielki problem z metodą programowania tablicą połączeń jest to,że liczba kroków programu (instrukcji maszynowych) jest ograniczona przez liczbę rzędów gniazdek dostępnych w maszynie.John Von Neumann i inni rozpoznali związki między gniazdkami na tablicy połączeń a bitami w pamięci:doszli do wniosku,że można przechowywać binarne odpowiedniki programu maszynowego w pamięci głównej i przekazywać każdy program z pamięci,ładować go do specjalnego rejestru dekodującego który byłby podłączony bezpośrednio do układu dekodującego instrukcje w CPU.Nie było sztuką dodanie jeszcze jednego układu do CPU.Ten układ,jednostka sterująca (CU),przekazywał kody instrukcji(znane również jako kody operacji lub opcody) z pamięci i przesuwał je do rejestru dekodującego instrukcje. Jednostka sterująca zawiera specjalny rejestr,wskaźnik rozkazów(IP),który zawiera adres wykonywanej instrukcji.CU pobiera kod instrukcji z pamięci i umieszcza ją w rejestrze dekodującym do wykonania.Po wykonaniu instrukcji ,CU zwiększa wskaźnik rozkazów i pobiera następna instrukcję z pamięci do wykonania i tak dalej.

Kiedy projektowano zbiór instrukcji,projektanci CPU,generalnie wybrali opcody,które są wielokrotnością długości ośmiu bitów, więc CPU może łatwo pobierać komplet instrukcji z pamięci. Zadaniem projektanta CPU jest przydzielić odpowiednią liczbę bitów do pola klasy instrukcji (move,add,subtract,itd.) i pola operandów. Wybranie większej ilości bitów dla pola instrukcji pozwala mieć więcej instrukcji,wybranie dodatkowych bitów dla pola operandów pozwala wybrać większą liczbę operandów(np. komórki pamięci lub rejestry) Jest to dodatkowa komplikacja. Niektóre instrukcje mają tylko jeden operand lub, nie mają żadnego operandu wcale. Zamiast marnować bity skojarzone z tymi polami,projektanci CPU często ponownie używają tych pól do kodowania dodatkowych opcodów. Rodzina CPU Intela 80x86 używa maksymalnie instrukcji z zakresu od jednego do dziesięciu bajtów długości.Ponieważ jest to trochę zbyt trudne abyśmy poradzili sobie na tak wczesnym stadium,CPU x86 będą używały różnych, prostszych schematów kodowania.

3.3.5 ZBIÓR INSTRUKCJI x86

CPU x86 dostarcza 20 podstawowych klas instrukcji.Siedem z tych instrukcji ma dwa operandy,osiem z tych instrukcji ma pojedynczy operand,a pięć instrukcji nie ma wcale operandów.Te instrukcje to mov (dwie formy),add,sub,cmp,and,or,not,je,jne,jb,jbe,ja,jae,jmp,brk,iret,halt,get i put.Poniższy paragraf opisuje jak każda znich pracuje**.**

Instrukcja mov jest właściwie dwoma klasami instrukcjami połączonymi wewnątrz tej samej instrukcji.Dwie formy instrukcji mov mają następujący kształt:

```
mov     reg,  reg/pamięć/stała
mov     pamięć/reg
```

gdzie reg jest jednym z rejestrów ax,bx,cx lub dx:;stała jest stałą liczbową (używając notacji heksadecymalnej),a pamięć wyszczególnioną komórką pamięci.Następna sekcja opisuje możliwe formy operandu pamięci jakich można używać. Operand ”reg/pamięć/stała” mówi nam,że tym szczególnym operandem może być rejestr,komórka pamięci lub stała.

Arytmetyczne i logiczne instrukcje przyjmują następujące formy:

```
add     reg,    reg/pamięć/stała
sub     reg,    reg/pamięć/stała
cmp     reg,    reg/pamięć/stała
and     reg,    reg/pamięć/stała
```

```
or      reg,  reg/pamięć/stała
not     reg/pamięć
```

Instrukcja **add** dodaje wartość drugiego operandu do pierwszego (rejestr) operandu,wynik umieszczając w pierwszym operandzie.Instrukcja **sub** odejmuje wartość drugiego operandu od pierwszego,różnicę umieszczając wynik w pierwszym operandzie.Instrukcja **cmp** porównuje pierwszy operand z drugim a wynik zachowuje do użycia w jednej z warunkowych instrukcji skoku (omówionych za chwilę) Instrukcje **and** i **or** obliczają odpowiadające sobie na poziomie bitowym,operacje logiczne na dwóch operandach i przechowują wynik w pierwszym operandzie.Instrukcja **not** odwraca bity w pojedynczym operandzie pamięci lub rejestru.

Instrukcje skoków przerywają sekwencyjne wykonywanie instrukcji w pamięci i przenoszą sterowanie do innego punktu w pamięci albo bezwarunkowo, albo po sprawdzeniu wyniku poprzedzającej instrukcji cmp instrukcje są następujące:

```
ja      dest    -- skok,jeśli powyżej
jae     dest    -- skok, jeśli powyżej lub równe
jb      dest    -- skok, jeśli poniżej
jbe     dest    --skok,jeśli poniżej lub równe
je      dest    -- skok,jeśli równe
jne     dest    --skok,jeśli nie równe
jmp     dest    --skok bezwarunkowy
iret            --powrót z przerwania
```

Pierwsze sześć instrukcji pozwala nam sprawdzić czy wynik poprzedzającej instrukcji cmp jest większy niż większy lub równy mniejszy niż mniejszy lub równy,równy lub nierówny.Na przykład,jeśli porównujemy rejestry ax i bx instrukcją cmp i wykonujemy instrukcję ja,CPU x86 skoczy do wyszczególnionego miejsca przeznaczenia jeśli ax będzie większe niż bx.Jeśli ax nie będzie większe niż bx,sterowanie przejdzie do następnej instrukcji w programie.Instrukcja skoku bezwarunkowego jmp przenosi sterowanie do instrukcji o adresie przeznaczenia.Instrukcja iret zwraca sterowanie z podprogramu obsługi przerwań który omówimy później.

Instrukcje **get** i **put** pozwalają odczytać i zapisać wartości całkowite.Get zatrzyma i zachęci użytkownika do podania wartości heksadecymalnej a potem przechowa tą wartość w rejestrze ax.Instrukcja put,wyświetla (heksadecymalnie) wartość z rejestru ax.

Pozostałe instrukcje nie wymagają żadnych operandów,są to instrukcje **halt** i **brk**.Halt przerywa wykonywanie programu a brk zatrzymuje program, w stanie w którym może być zrestartowany.

Procesor x86 wymaga unikalnych opcodów dla każdej instrukcji,nie tak jak klasa instrukcji.Chociaż „mov ax,bx” i „mov ax,cx” ,obie są tej samej klasy muszą mieć inne opcody,żeby CPU mógł je odróżnić.Jednakże,zanim przejrzymy wszystkie możliwe opcody,być może będzie dobrym pomysłem nauczyć się o wszystkich możliwych operandach dla tych instrukcji.

3.3.6 TRYBY ADRESOWANIA W x86

Instrukcje x86 używają pięciu różnych typów operandów: rejestry,stałe i trzy schematy adresowania pamięci.Każda z tych form nazywana jest trybem adresowania.Procesory x86 obsługują tryb adresowania rejestrowy,tryb adresowania natychmiasowego,tryb adresowania pośredniego,tryb adresowania z indeksowaniem i bezpośredni tryb adresowania.Ten paragraf wyjaśni każdy z tych trybów.

Operandy rejestrów jest najłatwiejszy do zrozumienia

Rozpatrzmy następujące formy instrukcji mov:

```
mov     ax, ax
mov     ax, bx
mov     ax, cx
mov     ax , dx
```

Pierwsza instrukcja nie realizuje absolutnie niczego.Kopiuje wartość z rejestru ax z powrotem do rejestru ax.Pozostałe trzy instrukcje kopiują wartości z bx,cx i dx do ax. Zauważ,że oryginalne wartości z bx,cx i dx pozostają bez zmian.Pierwszy operand (przeznaczenia) nie jest ograniczony tylko do ax;możemy przenosić wartości do każdego z tych rejestrów.

Stałe są równie łatwe do opanowania Rozpatrzmy następujące instrukcje:

```
mov     ax, 25
```

```
mov     bx, 195
mov     cx, 2056
mov     dx, 1000
```

Te wszystkie instrukcje są równie proste;ładują do rejestrów stałe heksadecymalne.

Są trzy tryby adresowania które radzą sobie z uzyskaniem dostępu do danych w pamięci.Te tryby adresownia mają następujące formy:

```
mov     ax, [1000]
mov     ax, [bx]
mov     ax, [1000+bx]
```

Pierwsza instrukcja powyżej,używa bezpośredniego trybu adresowania do ładowania ax, szesnastobitową wartością przechowywaną w pamięci zaczynającą się od komórki 1000h

Instrukcja mov ax, [bx] ładuje ax z komórki pamięci wyszczególnionej przez zawartość rejestru bx.Jest to pośredni tryb adresowania.Zamiast używać wartości z bx,ta instrukcja uzyskuje dostęp do komórki pamięci której adres znajduje się w bx.Zauważ,że dwie następujące instrukcje:

```
mov     bx, 1000        mov     ax, [bx]
```

są równoważne jednej instrukcji :

```
mov     ax, [1000]
```

Oczywiście,druga sekwencja jest bardziej pożądana.Jednak,jest wiele przypadków gdzie użycie trybu pośredniego jest szybsze,krótsze i lepsze. Zobaczymy tego kilka przykładów kiedy będziemy przyglądać się pojedynczym procesorom z rodziny x86 trochę później.

Ostatnim trybem adresowania jest adresowanie z indeksowaniem.Przykładem takiego adresowania pamięci jest:

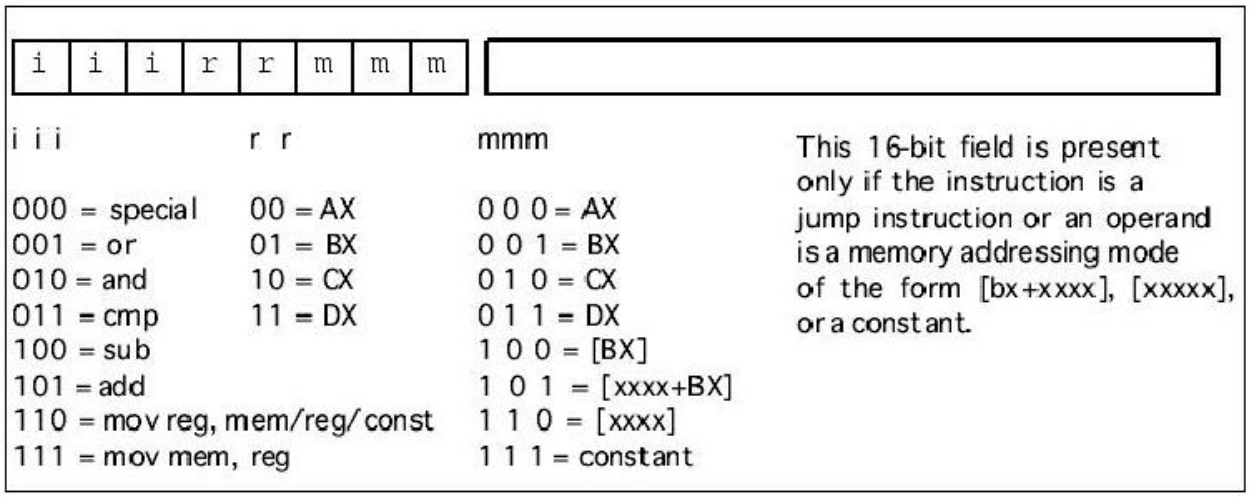
```
mov     ax, {1000+bx]
```

Ta instrukcja dodaje zawartość rejestru bx i 1000,tworząc wartość adresu pamięci do przekazania.Ta instrukcja jest przydatna przy dostępie do elementów tablic,rekordów i innych struktur danych.

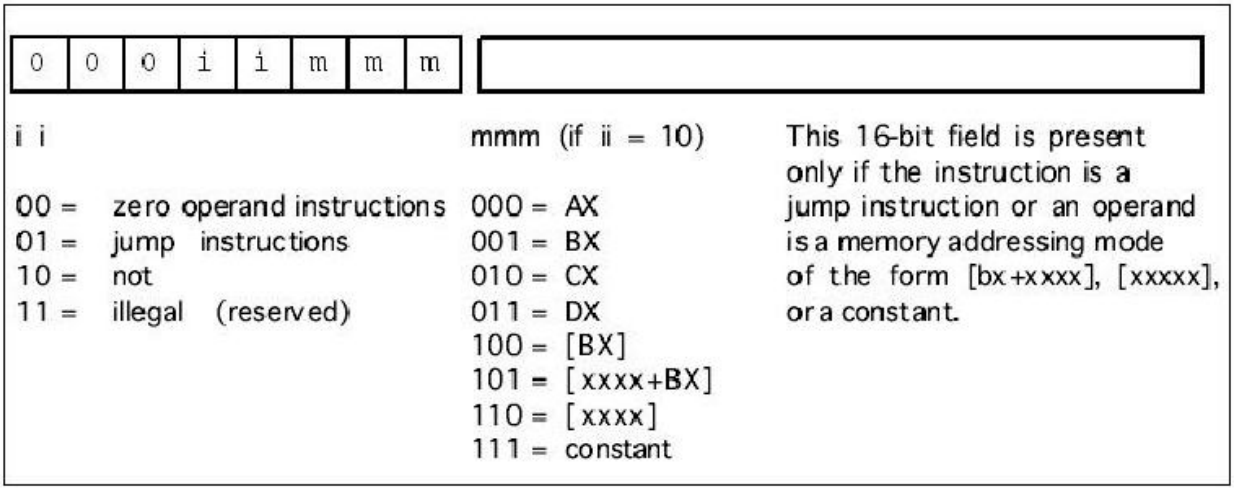
3.3.7 KODOWANIE INSTRUKCJI x86

Chociaż możemy przypadkowo przydzielać opcody do każdej instrukcji x86,zapamiętaj,że w rzeczywistości CPU używa układów logicznych do dekodowania opcodów i właściwego na nich działania.Typowo opcody CPU używają pewnej liczby bitów w opcodzie do oznaczania klasy instrukcji (np. mov,add,sub) i pewnej liczby bitów do kodowania każdego z operandów.Niektóre systemy(np. CISC lub Komputer z pełną listą rozkazów) koduje te pola w bardzo złożony sposób,tworząc niewielkich rozmiarów instrukcje.Inne systemy (np. RISC lub Komputer o uproszczonej liście rozkazów) koduje opcody w bardzo prosty sposób nawet jeśli oznacza to marnowanie niektórych bitów w opcodzie lub ograniczenie liczby instrukcji.Rodzina Intela 80c86 jest zdecydowanie CISCowska i ma jeden z najbardziej złożonych schematów dekodowania jaki wymyślono.Celem hipotetycznych procesorów x86 jest przedstawienie koncepcji kodowania instrukcji bez całej złożoności rodziny 80x86, przez zademonstrowanie kodowania CISC.

Typowa instrukcja x86 przyjmuje postać taką jak pokazana na rysunku 3.19.Podstawowa instrukcja ma długość albo jednego albo trzech bajtów. Opcod instrukcji składa się z pojedynczego bajtu który zawiera trzy pola.Pierwsze pole,najbardziej znaczące,trzy bitowe,definiuje klasę instrukcji. Daje to osiem kombinacji.Jeśli sobie przypominasz,mamy 20 klas instrukcji;nie możemy zakodować 20 klas instrukcji w trzech bitach, więc będziemy musieli zastosować sztuczkę aby uzyskać inne klasy.. Jak widać na rysunku 3.19,podstawowy opcod koduje instrukcję mov (dwie klasy,jedna gdzie pole rr określa miejsce przeznaczenia,jedna gdzie pole mmm określa miejsce źródłowe),instrukcje add,sub,cmp,and i or.Jest jedna dodatkowa klasa:specjalna.



Rysunek 3.19: Kodowanie podstawowych instrukcji x86



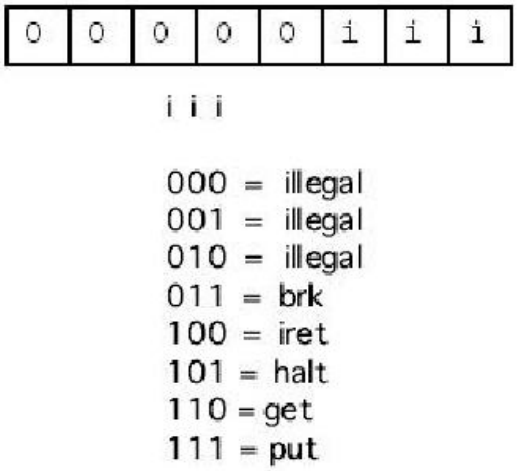
Rysunek 3.20:Kodowanie pojedynczego operandu instrukcji

Ta specjalna klasa instrukcji dostarcza mechanizmów,które pozwalają nam powiększać liczbę dostępnych klas instrukcji,wrócimy do tych klas wkrótce..

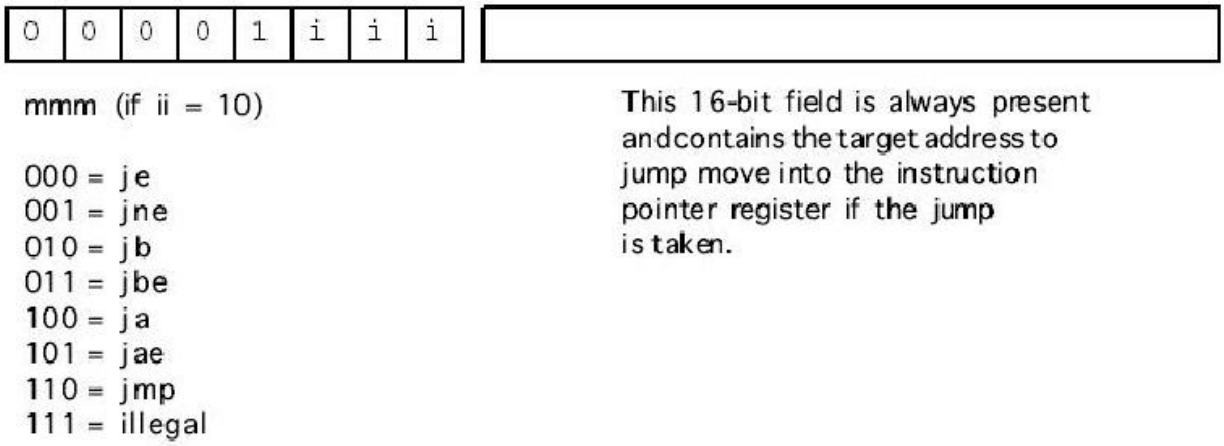
W celu ustalenia poszczególnych opcodów instrukcji,musimy tylko wybrać właściwe bity dla pól iii,rr i mmm.Na przykład, dla kodowania instrukcji mov ax,bx wybierzemy iii=100 (mov reg,reg),rr=00(ax) i mmm= 001 (bx).Da to nam instrukcję jednobajtową 11000001 lub 0C0h.

Niektóre instrukcje x86 wymagają więcej niż jednego bajtu.Na przykład, instrukcja mov ax,[1000] ładuje do rejestru ax zawartość spod adresu komórki 1000.Kodowanie tego opcodu to 11000110 lub 0C6h.jednakże,kodowanie dla opcodu mov ax,[2000] to również 0C6h.Wyraźnie jednak widać,że te dwie instrukcje robią różne rzeczy.,jedna ładuje do rejestru ax zawartość z komórki pamięci o adresie 1000h podczas gdy druga ładuje do rejestru ax wartość z komórki pamięci o adresie 2000h.Kodując adres w trybie adresowania [xxxx] lub [xxxx+bx],lub kodowania stałych w trybie natychmiastowym,musisz zastosować opcod adresowany 16 bitowo lub stałą,z najmniej znaczącym bajtem bezpośrednio następującym po opcodzie w pamięci i bardziej znaczącym bajtem po nim.Tak więc trzy bajty kodowane dla instrukcji mov ax,[2000] będą wynosić 0C6h,00h,20h.

Specjalny opcod pozwala CPU x86 do rozszerzenia zbioru dostępnych instrukcji.Ten opcod obsługuje instrukcjami z jednym lub zerowym operandem jak pokazano na rysunku 3.20 i 3.21



Rysunek 3.21:Kodowanie instrukcji bez operandu



Rysunek 3.22: Kodowanie instrukcji skoku

Mamy cztery klasy instrukcji jednooperandowych.Po pierwsze, odkodowanie (00) bardzo rozszerza ilość instrukcji za pomocą zero operandowych instrukcji(zobacz rysunek 3.21).Po drugie opcod jest również opcodem rozszerzonym który dostarcza wszystkich instrukcji skoku x86 (zobacz rysunek 3.22)
Po trzeci opcod jest instrukcją not.Jest to logiczna operacja **not** na poziomie bitowym która odwraca wszystkie bity w rejestrze przeznaczenia lub operandzie pamięci.Po czwarte, opcod pojedynczego operandu nie jest obecnie używany.Każda próba wykonania tego opcodu zatrzyma procesor z instrukcją błędu. Projektanci CPU często rezerwują nieużywane opcody ,jako te do rozszerzenia zbioru instrukcji na przyszłe dane (jak zrobił Intel przechodząc z procesorów 80286 na procesory 80386).

Jest siedem instrukcji skoku w zbiorze instrukcji x86.Wszysytkie mają następującą postać:

jxx adres

Instrukcja jmp kopiuje 16 bitową wartość bezpośrednią (adres) następnego opcodu do rejestru IP.Dlatego też,CPU będzie pobierał następna instrukcję z tego adresu docelowego;faktycznie program „skacze” od punktu instrukcji skoku do instrukcji pod adresem docelowym.

Instrukcja jmp jest przykładem instrukcji skoku bezwarunkowego.Zawsze przekazuje sterowanie pod adres docelowy.Pozostałe sześć insrtukcji,jest instrukcjami skoków warunkowych.Sprawdzają one pewne warunki, i skaczą jeśli warunek jest spełniony,przechodzą do następnej instrukcji jeśli warunek nie jest spełniony

Te sześć instrukcji to: ja,jae,jb.jbe,je i jne pozwala sprawdzić czy większe niż,większe niż lub równe,mniejsze niż,mniejsze niż lub równe,równe nie równe.Normalnie będziemy wykonywać te instrukcje natychmiast po instrukcji cmp ponieważ ustawia ona flagi,które są sprawdzane przez instrukcje skoków..Zauważ,że jest osiem możliwych opcodów skoku,ale x86 używa tylko siedmiu z nich. Ósmy opcod jest innym niedopuszczalnym opcodem.

Ostatnią grupą instrukcji,są instrukcje bezoperandowe, pokazane na rysunku 3.21.Trzy z tych instrukcji są niedopuszczalnymi opcodami instrukcji .Instrukcja brk (break) pauzuje CPU do momentu,aż użytkownik nie zrestartuje go ręcznie. Pauzowaniu programu jest użyteczne podczas prowadzenia obserwacji wyników działania. Instrukcja iret (interrupt return) zwraca sterowanie z podprogramu obsługi przerwań.Omówimy to później.Halt przerywa wykonywanie programu.Instrukcja get odczytuje wartość heeksadecymalną od użytkownika i zwraca tą wartość do rejestru ax; instrukcja put odczytuje tą wartość z rejestru ax.

3.3.8 WYKONYWANIE INSTRUKCJI KROK PO KROKU

CPU x86 nie kończy wykonywania instrukcji w jednym cyklu zegarowym. CPU wykonuje kilka kroków dla każdej instrukcji.Na przykład, CU wydaje następujące polecenia przy wykonywaniu instrukcji mov reg,reg/pamieć/stała:

- Pobiera rozkaz bajtowy z pamięci
- Uaktualnia rejestr IP wskazujący następny bajt
- Dekoduje instrukcję aby zobaczyć co ona robi
- W razie potrzeby pobiera 16 bitowy operand instrukcji z pamięci
- W razie potrzeby,uaktualnia IP do punktu za operandem
- Oblicza adres operandu,w razie potrzeby (n.p,bx+xxxx)
- Pobiera operand
- Przechowuje pobraną wartość w rejestrze przeznaczenia

Opis krok po kroku,może pomóc w wyjaśnieniu co CPU robi.W pierwszym kroku,CPU pobiera rozkaz bajtowy z pamięci.Robi to kopiując wartość z rejestru IP na magistralę adresową i odczytuje bajt spod tego adresu.To zajmuje jeden cykl zegarowy.

Po pobraniu rozkazu bajtowego,CPU uaktualnia IP żeby wskazywał następny bajt w ciągu instrukcji.Jeśli bieżąca instrukcja jest instrukcją wielobajtową,IP wskazuje operand dla tej instrukcji.Jeśli instrukcja jest jednobajtowa,IP będzie wskazywał następną instrukcję.To zajmuje jeden cykl zegarowy.

Następnym krokiem jest dekodowanie instrukcji aby zobaczyć co ona robi.Powie ona CPU,między innymi,czy musi pobrać dodatkowy bajt operandu z pamięci.To zabiera jeden cykl zegarowy.

Podczas dekodowania, CPU określa wymagane typy operandów instrukcji .Jeśli instrukcja wymaga 16 bitowego stałego operandu (np. jeśli pole mmm wynosi 101,110 lub 111) wtedy CPU pobiera tą stałą z pamięci. Ten krok może wymagać zero, jednego lub dwóch cykli zegarowych. Nie wymaga żadnego cyklu jeśli nie jest 16 bitowym operandem; potrzebuje jednego cyklu jeśli 16 bitowy operand jest word-aligned (słowem wyrównanym) (to znaczy, zaczyna się przy parzystym adresie);potrzebuje dwóch cykli zegarowych jeśli operand nie jest słowem wyrównanym (to znaczy, zaczyna się przy nieparzystym adresie).

Jeśli CPU pobiera 16 bitowy operand pamięci musi zwiększyć IP o dwa, żeby wskazać następny bajt następujący po operandzie. Ta operacja zajmuje zero lub jeden cykl zegarowy. Zero cykli zegarowych jeśli nie ma operandu; jeden cykl zegarowy jeśli operand jest obecny.

Następnie CPU oblicza adres operandu pamięci. Ten krok jest wymagany tylko wtedy kiedy pole mmm rozkazu bajtowego wynosi 101 lub 100.jeśli pole mmm zawiera 101 wtedy CPU oblicza sumę rejestru bx i stałej szesnastobitowej; to wymaga dwóch cykli, jednego dla pobrania wartości bx, drugiej do obliczenia sumy z bx i xxxx. Jeśli pole mmm zawiera 100,wtedy CPU pobiera wartość w bx dla adresu pamięci ,wymaga to jednego cyklu. Jeśli pole mmm nie zawiera 100 lub 101,ten krok zajmuje zero cykli.

Pobieranie operandu zabiera zero, jeden, dwa lub trzy cykle w zależności od rodzaju operandu.Jeśli operand jest stałą (mmm=111) wtedy ten krok wymaga zero cykli ponieważ.mamy już pobraną tą stałą z pamięci w poprzednim kroku.Jeśli operand jest rejestrem (mmm=000,001,010 lub 011) wtedy ten krok wymaga jednego cyklu.Jeśli operandem pamięci jest wyrównane słowo (mmm=100,101 lub 110) wtedy ten krok wymaga dwóch cykli zegarowych.Jeśli jest to nie ustawiony operand pamięci, zajmuje to trzy cykle zegarowe do pobrania jego wartości.

Ostatnim krokiem instrukcji mov jest przechowanie wartości wewnątrz miejsca przeznaczenia.Ponieważ miejscem przeznaczenia ładowania instrukcji jest zawsze rejestr ta operacja zawiera pojedynczy cykl.

Generalnie instrukcja mov zabiera od pięciu do jedenastu cykli, w zależności od jej operandów i ich ustawienia (adresów startowych) w pamięci..

CPU robi następujące rzeczy dla instrukcji mov pamięć,reg:

- Pobiera rozkaz bajtowy z pamięci (jeden cykl zegarowy)
- Uaktualnia rejestr IP aby wskazywał następny bajt (jeden cykl zegarowy)
- Dekoduje instrukcję,aby wiedzieć co robi (jeden cykl zegarowy)
- W razie potrzeby,pobiera operand z pamięci (zero cykli jeśli w trybie adresowania [bx],jeden cykl jeśli w trybie adresowania [xxxx],[xxxx+bx] lub xxxx a opcod wartości natychmiastowej xxxx zaczyna się od parzystego adresu.,lub dwa cykle zegarowe jeśli wartość xxxx zaczyna się od adresu nieparzystego).

- W razie potrzeby uaktualnia IP aby wskazywał poza operand (zero cykli jeśli brak operandu jeden cykl jeśli operand jest)
- Oblicza adres operandu (zero cykli jeśli tryb adresowania to nie [bx] lub [xxxx+bx],jeden cykl jeśli tryb adresowania to [bx] lub dwa cykle jeśli tryb adresowania to [xxxx+bx].
- Dostarcza wartość z rejestru do przechowania (jeden cykl zegarowy)
- Przechowuje pobrana wartość w miejscu przeznaczenia (jeden cykl jeśli w rejestrze dwa cykle jeśli operand pamięci jest wyrównanym słowem lub trzy cykle jeśli operand pamięci o nieparzystym adresie).

Synchronizacja dla tych dwóch ostatnich pozycji jest różne dla różnych mov,ponieważ ta instrukcja może czytać dane z pamięci ta „wersja" instrukcji mov „ładuje" jej dane z rejestru.Instrukcji tej, wykonanie, zajmuje pięć do jedenastu cykli.

Instrukcje add,sub,cmp,and i or robią co następuje:

- Pobierają rozkaz bajtowy z pamięci (jeden cykl)
- Uaktualniają IP aby wskazywał następny bajt (jeden cykl)
- Dekodują instrukcję (jeden cykl)
- W razie potrzeby pobierają stały operand z pamięci (zero cykli jeśli tryb adresowania [bx],jeden cykl jeśli tryb adresowania [xxxx],[xxxx+bx] lub xxxx a opcod wartości natychmiastowej zaczyna się od parzystego adresu lub dwa cykle zegarowe jeśli wartość xxxx zaczyna się od adresu nieparzystego).
- W razie potrzeby uaktualniają IP aby wskazywał stały operand (zero lub jeden cykli zegarowych)
- Obliczają adres operandu(zero cykli jeśli trybem adresowania nie jest [bx] lib [xxxx+bx],jeden
  cykl jeśli tryb adresowania to [bx],lub dwa cykle jeśli trybem adresowania jest [xxxx+bx]
- Pobierają wartość operandu i wysyłają go do ALU (zero cykli jeśli stała jeden cykl jeśli rejestr dwa cykle jeśli operand jest słowem wyrównanym lub trzy cykle jeśli nieparzysto ustawiony operand pamięci.
- Pobierają wartość z pierwszego operandu (rejestr) i wysyłają go do ALU (jeden cykl zegarowy)
- Instruują ALU o wartościach dodawania,odejmowania,porównania,logicznego AND
  lub logicznego OR (jeden cykl)

- Przechowują wynik w pierwszym operandzie rejestru (jeden cykl)

Te instrukcje wymagają od ośmiu do siedemnastu cykli zegarowych do wykonania.

Instrukcja not jest podobna do powyższych,ale może być trochę szybsza ponieważ ma tylko pojedynczy operand:

- Pobiera rozkaz bajtowy z pamięci (jeden cykl zegarowy)
- Uaktualnia IP aby wskazywał następny bajt (jeden cykl zegarowy)

- Dekoduje instrukcję (jeden cykl)
- W razie potrzeby pobiera stały operand z pamięci (zero cykli zegarowych jeśli tryb adresowania to]bx],jeden cykl jeśli tryb adresowania [xxxx]lub [xxxx+bx],a opcod wartości natychmiastowej xxxx zaczyna się od parzystego adresu lub dwa cykle zegarowe jeśli wartość xxxx zaczyna się od adresu nieparzystego)
- W razie potrzeby uaktualnia IP aby wskazywał poza stały operand (zero lub jeden cykl zegarowy)
- Oblicza adres operandu (zero cykli zegarowych jeśli tryb adresowania nie jest [bx] lub[xxxx+bx],jeden cykl jeśli tryb adresowania to [bx] lub dwa cykle jeśli tryb adresowania to [xxxx+bx]
- Pobiera wartość operandu i wysyła ją do ALU (jeden cykl jeśli rejestr,dwa cykle jeśli operator pamięci jest słowem wyrównanym lub trzy cykle zegarowe jeśli operand pamięci o nieparzystym adresie)
- Instruuje ALU o przeprowadzeniu logicznego not (jeden cykl zegarowy)
- Przechowuje wynik w operandzie (jeden cykl jeśli to rejestr,dwa cykle jeśli komórka pamięci o parzystym adresie,trzy cykle jeśli komórka pamięci o nieparzystym adresie.

Wykonanie instrukcji not zajmuje od sześciu do piętnastu cykli

Instrukcja skoku warunkowego pracuje jak następuje:

- Pobiera rozkaz bajtowy z pamięci (jeden cykl zegarowy)
- Uaktualnia IP aby wskazywał następny bajt (jeden cykl)
- Dekoduje instrukcję (jeden cykl)
- Pobiera adres docelowy operandu z pamięci (jeden cykl jeśli xxxx jest parzystym adresem dwa cykle jeśli jest nieparzystym adresem)
- Uaktualnia IP aby wskazywał poza ten adres (jeden cykl)
- Testuje „mniejsze niż" lub „równe" flagi CPU (jeden cykl)
- Jeśli wartości flag są właściwe dla poszczególnych warunków skoku,CPU kopiuje 16 bitową stałą do rejestru IP (zero cykli jeśli się nie rozgałęzia,jeden cykl jeśli występuje rozgałęzienie)

Instrukcja skoku bezwarunkowego jest identyczna jak w instrukcją mov reg,xxxx z wyjątkiem rejestru przeznaczenia którym jest rejestr x86 IP zamiast ax,bx,cx lub dx.

Instrukcje brk,iret,halt,put i get nie są tu dla nas interesujące.. Pojawiły się w zbiorze instrukcji głównie dla programów i eksperymentów.Nie możemy policzyć im „cykli" ponieważ mogą zabierać nieograniczoną ilość czasu dla wykonania tego zadania.

3.3.9 RÓŻNICE MIĘDZY PROCESORAMI x86

Wszystkie procesory x86 dzielą ten sam zbiór istrukcji,te same tryby adresowania i wykonują te instrukcje używając tej samej sekwencji kroków.Więc jakie są różnice?Dlaczego nie wymyślono jednego procesora zamiast czterech?

Głównym powodem dla wykonania tego ćwiczenia jest wyjaśnienie przedstawionych różnic ,powiązanych czterech cech hardware: kolejka rozkazów**,**pamięć podręczna cache**,**przetwarzanie potokowe i projektowanie superskalarne**.**Procesor 886 jest niedrogim „urządzeniem" ,które nie implementuje każdej z tych wymyślnych cech. Procesor 8286 implementuje kolejkę rozkazów.8486 ma kolejkę rozkazów,pamięć podręczną cache i przetwarzanie potokowe.8686 ma wszystkie z powyższych cech z operacjami superskalarnymi.Poprzez studiowanie tych procesorów możemy zobaczyć korzyści tych cech.

3.3.10 PROCESOR 886

Procesor 886 jest najwolniejszym członkiem rodziny x86.Czasy wykonania dla każdej instrukcji były omawiane w poprzedniej sekcji.Instrukcja mov,na przykład, zabiera od pięciu do dwunastu cykli zegarowych zależnie od operandów.Następująca tablica pokazuje czasy wykonania dla różnych form instrukcji na procesorach 886.

Instruction ⇒ Addressing Mode ⇓	mov (both forms)	add, sub, cmp, and, or,	not	jmp	jxx
reg, reg	5	7			
reg, xxxx	6-7	8-9			
reg, [bx]	7-8	9-10			
reg, [xxxx]	8-10	10-12			
reg, [xxxx+bx]	10-12	12-14			
[bx], reg	7-8				
[xxxx], reg	8-10				
[xxxx+bx], reg	10-12				
reg			6		
[bx]			9-11		
[xxxx]			10-13		
[xxxx+bx]			12-15		
xxxx				6-7	6-8

Tablica 19: Czasy wykonania dla instrukcji 886

Są trzy ważne rzeczy do zapamiętania z tego.Po pierwsze dłuższa instrukcja zabiera więcej czasu na wykonanie.Po drugie,instrukcje nie mające odniesienia do pamięci generalnie wykonują się szybciej,jest to prawda zwłaszcza gdy są stany oczekiwania powiązane z dostępem do pamięci (powyższa tabela zakłada zero stanów oczekiwania)W końcu,instrukcje używające złożonych trybów adresowania wykonują się wolniej.Instrukcje,używające rejestru jako operandu są krótsze,nie mają dostępu do pamięci i nie używają złożonych trybów adresowania.Jest tak,dlatego,że trzymamy swoje zmienne w rejestrach.

3.3.11 PROCESOR 8286

Kluczem do poprawienia szybkości procesora jest równoległe przeprowadzanie operacji.Jeśli w czasach wykonania danych dla 886,moglibyśmy robić dwie operacje na każdy cykl zegarowy,CPU wykonywałby instrukcje dwa razy szybciej pracując z tą samą szybkością zegara.Jednakże, proste decydowanie o wykonaniu dwóch operacji na cykl zegarowy nie jest tak łatwe.Wiele kroków w wykonywaniu instrukcji dzieli jednostki funkcjonalne w CPU (jednostki funkcjonalne są logicznymi grupami które wykonują wspólne operacje np. ALU i CU).Jednostka funkcjonalna jest zdolna wykonać tylko jedna operację w czasie.Dlatego też,nie możemy zrobić dwóch operacji które używają tych samych jednostek funkcjonalnych jednocześnie.(np. zwiększanie rejestru IP i dodawanie dwóch wartości razem).Inną trudnością z robieniem pewnych operacji jednocześnie jest to,że jedna operacja może zależeć od wyniku innej.Na przykład,ostatnie dwa kroki instrukcji add wymagają dodania wartości i potem przechowania ich sumy.Nie możemy przechować sumy w rejestrze zanim wyliczymy sumę.Również kilka innych zasobów CPU nie może dzielić kroków w instrukcji.Na przykład jest tylko jedna magistrala danych;CPU nie może pobierać opcodów instrukcji w tym samym czasie kiedy próbuje przechować jakieś dane w pamięci.Sztuczka w projektowaniu CPU jest taka,że wykonywanie kilku kroków równolegle polega na ułożeniu tych kroków,tak aby zredukować konflikty lub dodać dodatkową logikę aby dwie (lub więcej) operacji mogło wystąpić równolegle,poprzez wykonanie w różnych jednostkach funkcjonalnych.

Rozważmy znów kroki instrukcji mov reg , pamięć/reg/stała:

- Pobranie rozkazu bajtowego z pamięci
- Uaktualnienie rejestru IP wskazując następny bajt

- Dekodowanie instrukcji żeby wiedzieć co robi
- W razie potrzeby pobranie 16 bitowego operandu instrukcji z pamięci
- W razie potrzeby,uaktualnienie IP do punktu za operandem
- Obliczanie adresu operandu,w razie potrzeby (n.p,bx+xxxx)
- Pobranie operandu
- Przechowanie pobranej wartości w rejestrze przeznaczenia

Pierwsza operacja używa wartości z rejestru IP (więc nie możemy powiększać IP) a używa magistrali do pobrania opcodu instrukcji z pamięci.Każdy krok ,który następuje po tym zależy od opcodu pobranego z pamięci,więc jest to niepodobne abyśmy mogli założyć wykonanie tego kroku z innymi.

Druga i trzecia operacja nie dzielą żadnej jednostki funkcjonalnej, nie dekodują opcodu w zależności od wartości rejestru IP**.**Dlatego też,możemy łatwo modyfikować jednostkę sterującą żeby zwiększyć rejestr IP w tym samym czasie dekodując instrukcje.To ujmie **j**eden cykl z wykonywania instrukcji mov.

Trzecia i czwarta operacja powyżej (dekodowanie i opcjonalne pobieranie 16 bitowego operandu) nie wyglądają aby mogły być zrobione równolegle ponieważ musimy dekodować instrukcje aby zdecydować czy CPU musi pobrać 16 bitowy operand z pamięci.Jednak moglibyśmy zaprojektować CPU tak aby i pobrał operand tak czy owak,aby był dostępny jeśli go będziemy potrzebować.Jest jeden problem z tym pomysłem ,musimy mieć adres operandu do pobrania (wartość w rejestrze IP) i musimy czekać dopóki nie zwiększymy rejestru IP przed pobraniem tego operandu.Jeśli zwiększamy IP w tym samym czasie kiedy dekodujemy instrukcję,musimy czekać do następnego cyklu który pobierze ten operand.

Ponieważ następne trzy kroki są opcjonalne jest kila możliwych sekwencji instrukcji w tym punkcie:

#1 (krok4, krok5, krok6 i krok7)- np. mov ax,[1000+bx]
#2 (krok4, krok5 i krok7) - np. mov ax,[1000]
#3 (krok6 i krok7) - np. mov ax,[bx]
#4 (krok7) - np. mov ax,bx

W tej sekwencji krok siedem zawsze jest zależny od poprzedniej instrukcji w sekwencji.
Dlatego też krok siedem nie może wykonywać się równolegle z innym z kroków,Krok sześć również zależy od kroku czwartego.Krok pięć nie może wykonywać się równolegle z krokiem czwartym ponieważ krok czwarty używa wartości z rejestru IP,jednak,krok pięć może wykonywać się równolegle z każdym innym krokiem.
Dlatego też,możemy ująć 2 cykle z pierwszych dwóch sekwencji jak następuje:

#1 (krok4, krok5/6 i krok7)
#2 (krok4, krok5/7)
#3 (krok6 i krok7)
#4 (krok7)

Oczywiście,nie ma mowy o wykonaniu kroku siedem i osiem w instrukcji mov,ponieważ ona musi na pewno pobrać wartość przed zachowaniem jej.Przez kombinację tych kroków, uzyskamy następujące kroki dla instrukcji mov:

- Pobranie rozkazu bajtowego z pamięci
- Dekodowanie instrukcji i uaktualnienie IP
- W razie potrzeby,pobranie 16 bitowego operandu instrukcji z pamięci
- Obliczenie adresu operandu,w razie potrzeby (np bx+xxxx)
- Pobranie operandu,w razie potrzeby uaktualnienie IP aby wskazywał poza xxxx
- Przechowanie przyniesionej wartości do rejestru przeznaczenia

Poprzez dodanie małej ilości logiki do CPU,możemy odjąć jeden lub dwa cykle wykonania instrukcji mov.Ta prosta optymalizacja pracy z większością instrukcji jest dobra.

Inny problem z wykonywaniem instrukcji mov dotyczy ustawienia opcodu.Rozważmy instrukcję mov ax,[1000] która pojawia się w komórce 100 pamięci.
CPU daje jeden cykl pobierając opcod i ,po zdekodowaniu instrukcji i określeniu ,że ma 16 bitowy operand, bierze dwa dodatkowe cykle dla pobrania tego operandu z pamięci (ponieważ ten operand pojawia się pod nieparzystym

adresem -101).Prawdziwą parodią tu jest to,że ten dodatkowy cykl zegarowy pobierający te dwa bajty jest zbyteczny,mimo wszystko,CPU pobiera najmniej znaczący bajt operandu kiedy przejmuje opcod (pamiętaj,że CPU x86 są procesorami 16 bitowymi i zawsze pobierają 16 bitów z pamięci),dlaczego nie zachowa tego bajtu i nie użyje tylko dodatkowego cyklu zegarowego dla pobrania bardziej znaczącego bajtu.? To ujmie czas wykonywania kiedy instrukcja zaczyna się od parzystego adresu (więc operand wpada pod adres nieparzysty)Wymagałoby to tylko jednobajtowego rejestru i małej ilości dodatkowej logiki do osiągnięcia tego.

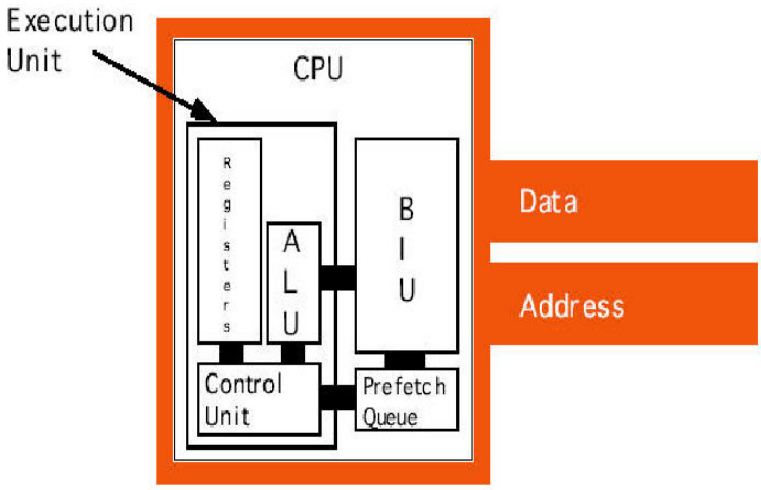
Podczas gdy dodamy bajt operandu rejestru do bufora,ropatrzymy kilka dodatkowych otymalizacji,które mogą używać tej samej logiki.Na przykład rozpatrzmy co się stanie z instrukcją mov podczas wykonywania.Jeśli pobierzemy opcod i najmniej znaczący bajt operandu w pierwszym cyklu i bardziej znaczący bajt operandu w drugim cyklu,w rzeczywistości odczytamy cztery bajty ,nie trzy.Tym czwartym bajtem jest opcod następnej instrukcji.Gdybyśmy mogli zachować ten opcod aż do przeprowadzenia następnej instukcji,moglibyśmy odjąć cykl czasu wykonania ponieważ nie musi pobierać bajtu opcodu .Co więcej ponieważ dekoder instrukcji jest nieczynny podczas gdy CPU wykonuje instrukcję mov,możemy w rzeczywistości dekodować następną instrukcję podczas wykonywania bieżącej instrukcji,tym samym ujmowaniem następnego cyklu z wykonywania w następnej instrukcji Średnio możemy pobrać ten extra bajt na każdą inną instrukcję.Dlatego też implementacja tego prostego schematu pozwoli nam odjąć dwa cykle z około 50% instrukcji które wykonujemy..

Czy możemy zrobić cokolwiek z drugim 50% instrukcji? Odpowiedź brzmi tak. Zauważ że wykonanie instrukcji mov nie uzyskuje dostępu do pamięci w każdym cyklu zegarowym.Na przykład gdy przechowujemy dane w rejestrze przeznaczenia magistrala jest nieczynna.Podczas okresu czasu kiedy magistrala jest nieczynna możemy pobrać wstępnie opcody instrukcji i operandów i zachować te wartości dla wykonywania następnej instrukcji.

Ważnym ulepszeniem procesora 8286 w stosunku do procesora 886 jest kolejka rozkazów.Zawsze kiedy CPU nie używa jednostki sprzęgającej z magistralą (BIU),BIU może pobrać dodatkowe bajty ze strumienia instrukcji Zawsze kiedy CPU potrzebuje bajtu instrukcji lub operandu,korzysta z następnego dostępnego bajtu z kolejki rozkazów**.**

Jednak nie gwarantuje to,że wszystkie instrukcje i operandy będą osadzone w kolejce rozkazów kiedy ich potrzebujemy.Na przykład instrukcja jmp 1000 unieważnia zawartość kolejki rozkazów.Jeśli ta instrukcja występuje w komórkach 400,401 i 402 w pamięci, kolejka rozkazów będzie zwierała bajty spod adresów 403,404,405,406,407 itd.Po załadowaniu do IP 1000 spod adresu 403 itp.,niemożemy zrobić dużo.System więc musi spauzować na chwilę aby pobrać podwójne słowo spod adresu 1000 zanim może iść dalej.

Inną poprawą jaką możemy zrobić jest zachodzenie na siebie dekodowania instrukcji w ostatnim kroku poprzedniej instrukcji.Po przetworzeniu przez CPU tego operandu następny dostępny bajt w kolejce rozkazów jest opcodem,a CPU może zdekodować go przewidując jego wykonanie..Oczywiście,jeśli bieżąca instrukcja modyfikuje rejestr IP, czas zużyty na dekodowanie następnej instrukcji marnuje się ale ponieważ zdarza się to równolegle z innymi operacjami,nie spowalnia to systemu.



Rysunek 3.23: CPU z Kolejką Rozkazów

Ta sekwencja optymalizacji systemu wymaga sporo zmian w sprzęcie.Diagram systemu pokazano na rysunku 3.23.Sekwencja wykonywania instrukcji teraz przybiera następujące wydarzenia występujące w tle:

CPU Zdarzenie Pobierania Wstępnego:

- Jeśli kolejka rozkazów nie jest pełna (ogólnie rzecz biorąc można trzymać miedzy ośmioma a trzydziestoma dwoma bajtami,w zależności od procesora) a BIU jest nieczynne w bieżącym cyklu zegarowym pobiera następne słowo z pamięci spod adresu w IP przy rozpoczynaniu cyklu zegarowego
- Jeśli dekoder instrukcji jest nieczynny,a bieżąca instrukcja nie wymaga operandu instrukcji zaczyna dekodowanie opcodu przed kolejką rozkazów (jeśli obecny), w przeciwnym razie zaczyna dekodowanie trzeciego bajtu w kolejce rozkazów (jeśli obecny) Jeśli nie ma pożądanego bajtu w kolejce rozkazów to zdarzenie nie wykonuje się.

Czasy wykonywania instrukcji spełniają kilka optymistycznych założeń,mianowicie każdy niezbędny opcod i operand instrukcji są zawsze obecne w kolejce rozkazów i,że już mają zdekodowane opcody bieżących instrukcji.Jeśli obojętnie który przypadek nie jest prawdą wykonywanie instrukcji 80286 będzie opóźniało chwilowo system przy pobieraniu danych z pamięci lub dekodowaniu instrukcji. Dla każdej instrukcji 8286 są następujące kroki:

mov reg, pamięć/reg/stała

- W razie potrzeby oblicza sumę z [xxxx+bx] (1 cykl)
- Pobiera operand źródłowy.Zero cykli jeśli stała (znajdująca się już w kolejce rozkazów),jeden cykl jeśli rejestr,dwa cykle jeśli wyrównywana parzyście wartość pamięci,trrzy cykle jeśli wyrównywana nieparzyście wartość pamięci
- Przechowuje wynik w rejestrze przeznaczenia jeden cykl

mov pamięć, reg

- Jeśli jest to wymagane oblicza sumę [xxxx+bx](jeden cykl)
- Pobiera operand źródłowy (rejestr),jeden cykl
- Przechowuje wewnątrz operandu przeznaczenia.Dwa cykle jeśli wartość pamięci wyrównywana parzyście **i** trzy cykle jeśli wyrównywana nieparzyście wartość pamięci

Instr reg, pamięć/reg/stała (instr = add,sub,cmp,and,or)

- W razie potrzeby oblicza sumę z [xxxx+bx] (jeden cykl)
- Pobiera operand źródłowy.Zero cykli jeśli stałą(znajdujący się już w kolejce rozkazów) ,jeden cykl jeśli rejestr,dwa cykle jeśli wyrównywana parzyście wartość pamięci,trzy cykle jeśli wyrównywana nieparzyście wartość pamięci
- Pobiera zawartość z pierwszego operandu (rejestr),jeden cykl
- Oblicza sumę,różnicę itp.,odpowiednio,jeden cykl
- Przechowuje wynik w rejestrze przeznaczenia,jeden cykl

Not pamięć/reg

- W razie potrzeby oblicza sumę z [xxxx+bx] (jeden cykl)
- Pobiera operand źródłowy.Jeden cykl jeśli rejestr,dwa cykle jeśli wyrównywana parzyście wartość pamięci,trzy cykle jeśli wyrównywana nieparzyście wartość pamięci
- Wykonuje logiczne NOT wartości jeden cykl
- Oblicza sumę,różnicę itp.,odpowiednio,jeden cykl
- Przechowuje wynik jeden cykl jeśli rejestr,dwa cykle jeśli wyrównywana parzyście wartość pamięci,trzy cykle jeśli wyrównywana nieparzyście wartość pamięci

jcc xxxx (skok warunkowy cc=a,ae,b,be,e,ne)

- Testuje bieżący warunek wskaźnika (mniejszy niż lub równy) flag,jeden cykl

- Jeśli wartość flag jest właściwa dla poszczególnego odgałęzienia warunku,CPU kopiuje 16 bitowy operand instrukcji do rejestru IP,jeden cykl

jmp xxxx

- CPU kopiuje 16 bitowy operand instrukcji do rejestru IP,jeden cykl

Jeśli chodzi o 886 nie możemy rozpatrywać czasów wykonania innych instrukcji x86 ponieważ większość z nich jest nieokreślona.

Wygląda na to,że instrukcje skoku wykonują się bardzo szybko na 8286.Faktycznie,mogą wykonywać się bardzo wolno.Nie zapomnijmy że skakanie z jednej lokacji do innej zmienia kolejność w kolejce rozkazów**.**Więc mimo że instrukcja skoku wygląda na możliwą do wykonania w jednym cyklu zmusza CPU do opróżnienia kolejki rozkazów,a zatem poświęca kilka cykli pobierając następną instrukcję dodatkowy operand i dekodując tą instrukcję.Istotnie, wykonują się dwie lub trzy instrukcje po instrukcji skoku zanim CPU wróci do punktu gdzie kolejka rozkazów operuje płynnie i CPU dekoduje opcody równolegle z wykonywaniem poprzedniej instrukcji.Ma to jedną,ważną implikację dla naszych programów: jeśli chcemy pisać szybkie kody upewnijmy się,że uniknęliśmy skakania w kółko w naszych programach tak bardzo jak to możliwe.

Zauważ że instrukcje skoków warunkowych tylko unieważniają kolejkę rozkazów ,jeśli w rzeczywistości wykonują skok.Jeśli warunek jest fałszywy, przechodzą do następnej instrukcji i kontynuują używanie wartości w kolejce rozkazów jak i predekodują opcody instrukcji.Zatem,możesz określić,podczas pisania programu, który warunek programu jest bardziej prawdopodobny (np. mniejszy niż czy nie mniejszy niż) powinieneś zorganizować swoje programy tak,że najbardziej powszechne nie dochodzą do skutku więc raczej stawiaj na skoki zależne od warunków każdej instrukcji pokolei.Rozmiar instrukcji (w bajtach) może również wpływać na wydajność kolejki rozkazów.Nigdy nie wymaga to więcej niż jednego cyklu zegarowego pobranie pojedynczego bajtu instrukcji ale zawsze wymaga dwóch cykli do pobrania trzech bajtów instrukcji.Zatem,jeśli celem instrukcji skoku są dwie jednobajtowe instrukcje,BIU może pobrać obie instrukcje w jednym cyklu zegarowym i zacząć dekodowanie drugiej podczas wykonywania pierwszej.Jeśli te instrukcje są instrukcjami trzy bajtowymi,CPU może nie mieć dosyć czasu do pobrania i zdekodowania drugiej lub trzeciej zanim skończy pierwszą.Dlatego też powinniśmy próbować używać krótszych instrukcji kiedy to możliwe ponieważ one poprawiają wydajność kolejki rozkazów**.**

Poniższa tabela uwzględnia (optymistyczne) czasowe wykonanie instrukcji 8286:

Instruction ⇒ Addressing Mode ⇓	mov (both forms)	add, sub, cmp, and, or,	not	jmp	jxx
reg, reg	2	4			
reg, xxxx	1	3			
reg, [bx]	3-4	5-6			
reg, [xxxx]	3-4	5-6			
reg, [xxxx+bx]	4-5	6-7			
[bx], reg	3-4	5-6			
[xxxx], reg	3-4	5-6			
[xxxx+bx], reg	4-5	6-7			
reg			3		
[bx]			5-7		
[xxxx]			5-7		
[xxxx+bx]			6-8		
xxxx				1+pfd[a]	2[b] 2+pfd

a. Cost of prefetch and decode on the next instruction.
b. If not taken.

Tabela 20: Czasy wykonania instrukcji dla 8286

Zauważ jak dużo szybciej instrukcja mov chodzi na 8286 w porównaniu z 886.Jest tak dlatego,że kolejka rozkazów pozwala procesorowi nakładać na siebie wykonywanie sąsiadujących instrukcji.Jednakże,ta tablica rysuje bardzo różowy obraz.Zauważ to zaprzeczenie „zakładając że opcod jest obecny w kolejce rozkazów zostanie zdekodowany".Rozpatrzmy następującą sekwencje trzech instrukcji:

```
????:       jmp 1000
1000:       jmp 2000
2000:       mov cx, 3000[bx]
```

Druga i trzecia instrukcja nie wykonają się tak szybko jak sugerują czasy wykonań w powyższej tabeli. Kiedy tylko zmodyfikujemy wartość rejestru IP,CPU opróżni kolejkę rozkazów.Więc CPU nie może pobrać i zdekodować następnej instrukcji Zamiast tego musi pobrać opcod, zdekodować go itp, zwiększając czasy wykonania tych instrukcji .W tym momencie tylko co możemy zrobić to wykonać operację „uaktualnienia IP" równolegle z innym krokiem.

Zazwyczaj, zastosowanie kolejki rozkazów poprawia wydajność. Dlatego Intel wprowadził kolejkę rozkazów w każdym modelu 80x86 ,z 8088 włącznie. Na tych procesorach ,BIU stale pobiera dane dla kolejki rozkazów, gdy tylko program nie jest aktywny czytaniem lub zapisaniem danych

Kolejka rozkazów pracuje najlepiej kiedy mamy szeroką magistralę danych. Procesor 8286 pracuje dużo szybciej niż 886 ponieważ może trzymać pełną kolejkę rozkazów. Zatem rozpatrzmy następujące instrukcje:

```
100:        mov  ax,[1000]
105:        mov  bx,[2000]
10A:        mov cx,[3000]
```

Ponieważ rejestry ax,bx i cx są szesnastobitowe, oto co się wydarzy (zakładając ,że pierwsza instrukcja jest w kolejce rozkazów i zdekodowana):

- Pobranie bajtu opcodu z kolejki rozkazów (zero cykli)
- Dekodowanie instrukcji (zero cykli)
- Jest operand do tej instrukcji, więc otrzymujemy go z kolejki rozkazów (zero cykli)
- Dostajemy wartość z drugiego operandu (jeden cykl)Uaktualnienie IP
- Przechowanie przyniesionej wartości w rejestrze przeznaczenia (jeden cykl)
- Pobranie dwóch bajtów ze strumienia kodu .Dekodowanie następnej instrukcji.

Koniec pierwszej instrukcji. Dwa bajty obecnie w kolejce rozkazów**.**

- Pobranie bajtu opcodu z kolejki rozkazów (zero cykli)
- Dekodowanie instrukcji aby zobaczyć co robi (zero cykli)
- Jeśli jest operand do tej instrukcji ,dostajemy ten operand z kolejki rozkazów (jeden cykl zegarowy ponieważ mamy brakujący jeden bajt)
- Dostajemy wartość z drugiego operandu (jeden cykl).Uaktualnienie IP.
- Przechowanie przyniesionej wartości w rejestrze przeznaczenia (jeden cykl)Pobranie dwóch bajtów z strumienia kodu. Dekodowanie następnej instrukcji.

Koniec drugiej instrukcji. Trzy bajty obecnie w kolejce rozkazów.

- Pobranie bajtu opcodu z kolejki rozkazów (zero cykli)
- Dekodowanie instrukcji aby zobaczyć co robi (zero cykli)
- Jeśli jest operand do tej instrukcji dostajemy ten operand z kolejki rozkazów (zero cykli)
- Dostajemy wartość z drugiego operandu(jeden cykl).Uaktualnienie IP.
- Przechowanie przyniesionej wartości w rejestrze przeznaczenia(jeden cykl)Pobranie dwóch bajtów ze strumienia kodu.Dekodowanie następnej instrukcji.

Jak możemy zobaczyć ,druga instrukcja wymaga jeden więcej cykl niż pozostałe dwie instrukcje .Jest tak ponieważ BIU nie może wypełnić kolejki rozkazów tak szybko jak CPU wykonuje instrukcje. Ten problem doprowadza do rozpaczy, kiedy ograniczymy rozmiar kolejkę rozkazów do kilku bajtów**.** Ten problem nie istnieje na procesorach 8286,ale z dużą pewnością istnieje w procesorach 8086.

Wkrótce zobaczymy, że procesory 80x86 mają w zwyczaju wyczerpywać kolejkę rozkazów bardzo łatwo .Oczywiście, jeśli kolejka rozkazów jest pusta, CPU musi czekać na BIU .który pobierze nowe opcody z pamięci, spowalniając program. Wykonywanie krótszych instrukcji pomoże utrzymywać pełną kolejkę rozkazów. Na

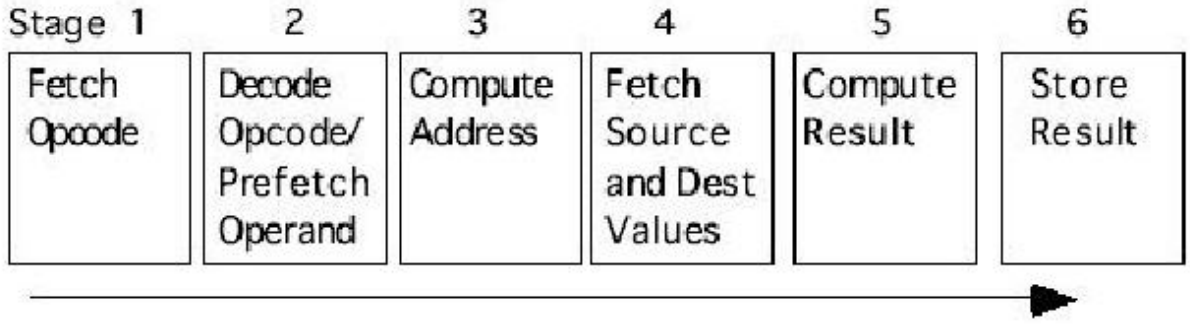
przykład,8286 może ładować dwie jednobajtowe instrukcje w pojedynczym cyklu pamięci, ale zabiera 1,5 cyklu zegarowego pobranie pojedynczej trzybajtowej instrukcji. Zazwyczaj, dłużej zabiera wykonanie tych czterech jednobajtowych instrukcji niż robi to wykonanie pojedynczej trzy bajtowej instrukcji. Daje to kolejce rozkazów czas do wypełnienia i zdekodowania nowych instrukcji które operują szybciej niż odpowiedni zbiór czterech czterobajtowych instrukcji. Powodem jest to,że kolejka rozkazów ma czas do powtórnego wypełnienia się krótszymi instrukcjami.

Morał z tej historii: kiedy programujesz procesor z kolejką rozkazów, zawsze używaj możliwie jak najkrótszych instrukcji realizujących dane zadanie.

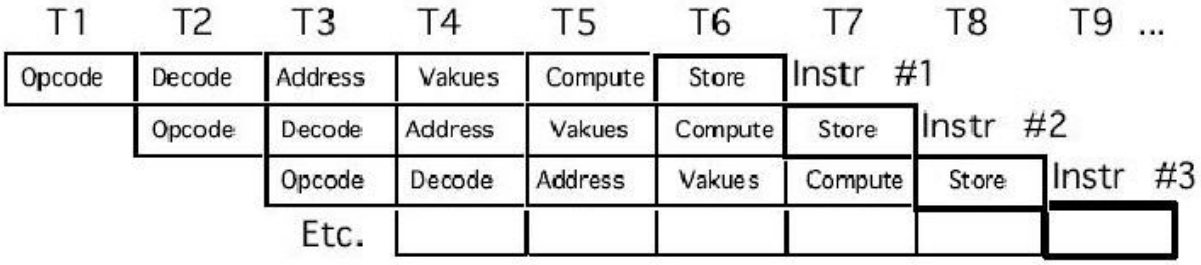
3.3.12 PROCESOR 8486

Wykonywanie instrukcji równolegle używając jednostki sprzęgającej z magistralą i jednostki wykonawczej jest specjalnym przypadkiem przetwarzania potokowego.8486 zawiera w sobie przetwarzanie potokowe do poprawy wydajności**.** Z kilkoma wyjątkami ,zobaczymy ,że przetwarzanie potokowe pozwala nam wykonywać jedną instrukcję na cykl zegarowy.

Przewagą kolejki rozkazów było to,że pozwalała CPU nakładać na siebie pobieranie instrukcji i dekodowanie z wykonaniem instrukcji. To znaczy, podczas gdy jedna instrukcja jest wykonywana, BIU pobiera i dekoduje następna instrukcję .Zakładając ,że chętnie dodamy sprzętu



Rysunek 3.24 Implementacja potoku wykonywania instrukcji



Rysunek 3.25:Wykonywanie instrukcji w potoku

możemy wykonywać prawie wszystkie operacje równolegle. To jest cała idea przetwarzania potokowego

---

3.3.12.1 PRZETWARZANIE POTOKOWE 8486

Rozważmy kroki do zrobienia ogólnej operacji:

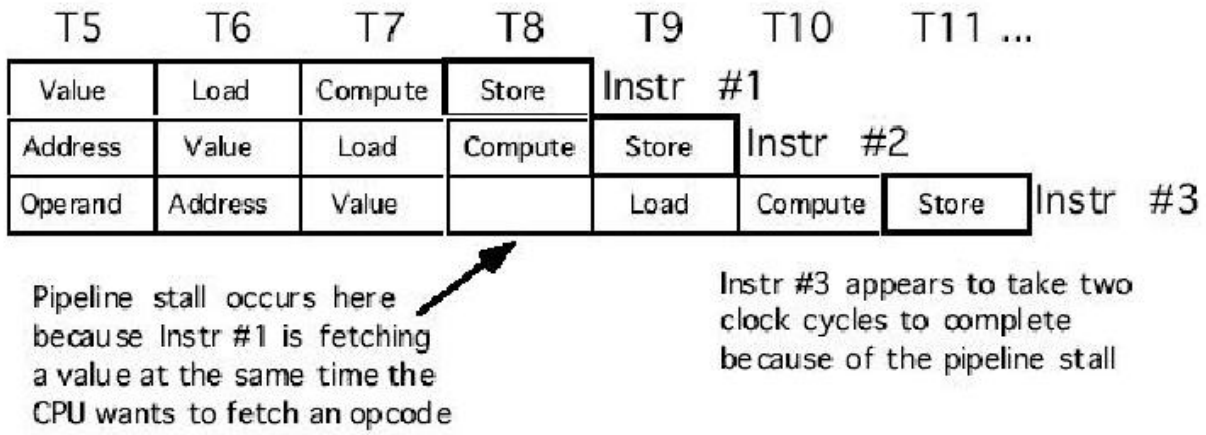
- Pobranie opcodu
- Dekodowanie opcodu i (równolegle) wstępne pobranie możliwego 16 bitowego operandu
- Obliczanie złożonego trybu adresowania (np [xxxx+bx], w stosownych przypadkach
- Pobranie wartości źródłowej z pamięci (jeśli operand pamięci) i wartość rejestru przeznaczenia, w stosownych przypadkach
- Obliczanie wyniku
- Przechowanie wyniku w rejestrze przeznaczenia

Zakładając, że chętnie zapłacimy za jakiś extra „krzem", możemy zbudować mały „mini-procesor" obsługujący każdy z powyższych kroków. Organizacja mogłaby wyglądać tak jak na rysunku 3.24

Jeśli zaprojektujemy oddzielną część sprzętową dla każdego etapu w przetwarzaniu potokowym ,prawie wszystkie te kroki mogą mieć miejsce równolegle. Oczywiście,nie można pobrać i zdekodować opcodu dla każdej z instrukcji w tym samym czasie, ale możemy pobrać jeden opcod podczas dekodowania poprzedniej instrukcji.Jeśli mamy n-etapowe przetwarzanie potokowe, zazwyczaj będziemy mieli n wykonywanych instrukcji jednocześnie. Procesor 8486 ma sześć etapów przetwarzania potokowego ,więc nakłada się wykonanie sześciu oddzielnych instrukcji.

Rysunek 3.25 Wykonywanie Instrukcji w przetwarzaniu potokowym ,pokazuje przetwarzanie potokowe.T1,T2,T3 itd. przedstawiają kolejne „odliczania" zegara systemowego. Przy T=T1 CPU pobiera bajt opcodu dla pierwszej instrukcji.

Przy T=T2,CPU zaczyna dekodować opcod dla pierwszej instrukcji. Równocześnie, pobiera16 bitów z kolejki rozkazów w przypadku gdy instrukcja ma operand. Ponieważ pierwsza instrukcja nie potrzebuje dłużej układu pobierającego opcod, CPU instruuje go do pobrania opcodu drugiej instrukcji równolegle z dekodowaniem pierwszej instrukcji .Zauważ ,że jest tu mały konflikt .CPU próbuje pobrać następny bajt z kolejki rozkazów do użycia jako operandu,w tym samym czasie jest pobierane 16 bitów z kolejki rozkazów do użycia jako opcod. Jak możemy zrobić obie od razu? Zobaczymy to rozwiązanie za kilka chwil.



Rysunek 3.26:Kolejka przetwarzania potokowego

Przy T=T3 CPU oblicza adres operandu dla pierwszej instrukcji. CPU nie robi nic na pierwszej instrukcji jeśli nie używa ona trybu adresowania [xxxx+bx].Podczas T3,CPU również dekoduje opcod drugiej instrukcji i pobiera potrzebny operand. Koniec końców ,CPU również pobiera opcod dla trzeciej instrukcji .Z każdym taktem zegara, inny krok w wykonywaniu każdej instrukcji w przetwarzaniu jest zakończony a CPU pobiera inną instrukcję z pamięci.

Przy T=T6 CPU kończy wykonywanie pierwszej instrukcji, oblicza rezultat dla drugiej, itd. w końcu pobiera opcod dla szóstej instrukcji w potoku. Ważną rzeczą jaką widzimy jest to,że po T=T5 CPU kończy instrukcje na każdym cyklu zegarowym. Zaraz po tym jak CPU wypełni potok ,kończy jedną instrukcję w jednym cyklu .Zauważ, że jest to prawda nawet jeśli jest złożony tryb adresowania obliczający ,operandy pamięci do pobrania lub inne operacje które używają cykli na procesorach niepotokowych. Wszystko co musisz zrobić to dodać więcej etapów do potoku ,i możesz jeszcze bardziej skutecznie przetwarzać instrukcje w jednym cyklu.

3.3.12.2 KOLEJKA W PRZETWARZANIU POTOKOWYM

Niestety ,przedstawiony scenariusz w poprzedniej sekcji jest trochę zbyt uproszczony. Są dwie wady tego uproszczonego przetwarzania potokowego :spór magistral między instrukcjami i niesekwencyjne wykonywanie programu. Oba problemy mogą zwiększyć średni czas wykonania instrukcji w potoku.

Spór magistral występuje gdy tylko instrukcja musi uzyskać dostęp do jakiegoś punktu w pamięci. Na przykład ,jeśli instrukcja mov pamięć , reg musi przechować dane w pamięci a instrukcja mov pamieć, reg czyta dane z pamięci, spór magistrali adresowej i danych musi nastąpić ponieważ CPU będzie próbował równocześnie pobierać dane i wpisać dane w pamięci

Jedyny sposób obejścia tego sporu magistral jest przez kolejka potoku. CPU, kiedy zaistnieje spór magistral, daje priorytet dalszym instrukcjom w potoku .CPU zawiesza pobieranie opcodów aż do pobrania opcodu

bieżącej instrukcji (lub przechowania) .To powoduje, że nowa instrukcja w potoku zabiera dwa cykle wykonania zamiast jednej (zobacz rysunek 3.26).

Ten przykład jest tylko jednym przykładem sporu magistral .Jest ich dużo więcej. Na przykład ,jak zauważyliśmy wcześniej, pobieranie operandów instrukcji wymaga dostępu do kolejki rozkazów w tym samym czasie kiedy CPU musi pobrać opcod. Co więcej ,na procesorach bardziej zaawansowanych niż 8486 (np. 80486) są inne źródła sporu magistral .Podany powyżej prosty schemat ,jest mało prawdopodobny, dlatego,że większość instrukcji wykonywałoby w jednym cyklu jedną instrukcję (CPI)

Na szczęście, inteligentne użycie systemu pamięci podręcznej może wyeliminować wiele kolejek potoku jak omówione powyżej. Następna sekcja o buforowaniu podręcznym ,omówi jak jest to robione. Jednakże, nie jest to zawsze możliwe ,nawet z pamięcią podręczną, unikania kolejek w potoku. **C**zego nie możemy naprawić sprzętowo ,możemy dopilnować programowo .Jeśli unikamy używania pamięci ,możemy zredukować spory magistral i nasz program będzie wykonywał się szybciej .Podobnie, używając krótszych instrukcji również zredukujemy spory magistral i możliwość kolejki potoku

Co się stanie jeśli instrukcja zmodyfikuje rejestr IP? Zanim instrukcja

```
jmp  1000
```

zakończy wykonywanie, mamy już rozpoczętych pięć innych instrukcji i jest tylko jeden cykl zegarowy po zakończeniu pierwszej instrukcji**.** Oczywiście, CPU nie może wykonać tych instrukcji lub obliczyć niewłaściwych wyników.

Prawdopodobnym rozwiązaniem jest opróżnienie całego potoku i rozpoczęcie pobierania na nowo. Jednakże ,takie postępowanie jest ukarane dłuższym czasem wykonania**.** Zabiera to sześć cykli zegarowych (długość potoku 8486) zanim następna instrukcja zakończy wykonywania. Wyraźnie możemy uniknąć używania instrukcji ,które przerywają sekwencję wykonywania programu. Pokazuje to również, inny problem - długość potoku. Dłuższy potok oznacza ,że możemy więcej osiągnąć na cykl w systemie. Jednak, wydłużanie potoku może spowolnić program jeśli będzie skakał w kółko. Niestety, nie możemy sterować liczbą etapów w potoku. Możemy ,jednak, sterować liczbą przenoszonych instrukcji które pojawiają się w naszym programie. Oczywiście powinieneś trzymać ich minimalna w potoku.

3.3.12.3 PAMIĘĆ PODRĘCZNA,KOLEJKA ROZKAZÓW I 8486

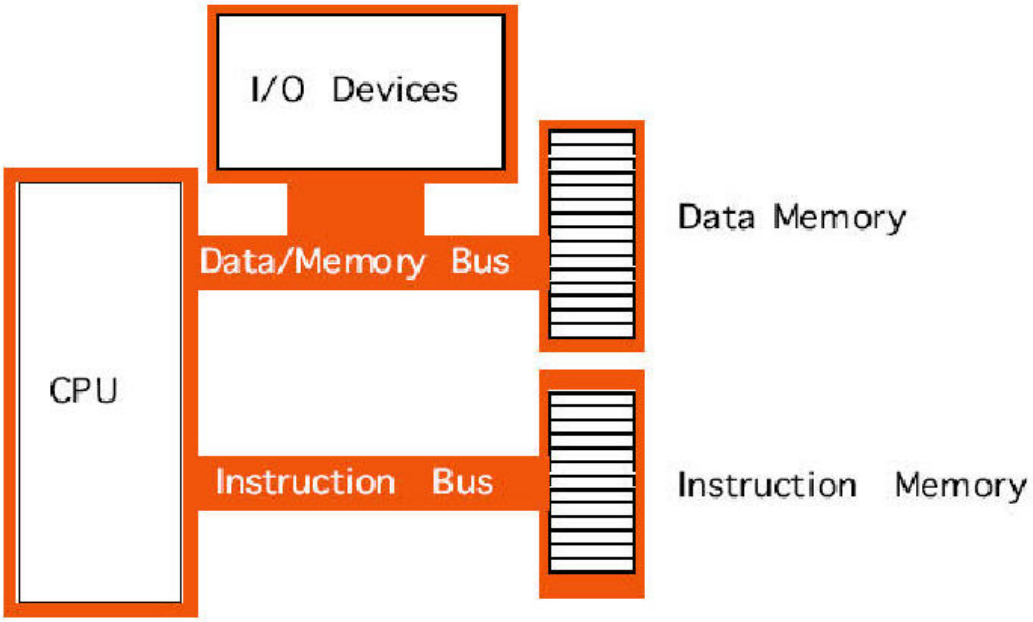
Projektanci systemu mogą rozwiązać wiele problemów ze sporem magistral poprzez inteligentne używanie kolejki rozkazów i podsystemu pamięci podręcznej. Mogą zaprojektować kolejkę rozkazów do buforowania danych ze strumienia instrukcji i mogą zaprojektować pamięć podręczną z oddzielnymi polami danych i kodu. Obie techniki mogą poprawić wydajność systemu przez wyeliminowanie kilku konfliktów magistral.

Kolejka rozkazów po prostu działa jako bufor między strumieniem instrukcji w pamięci a układem pobierającym opcody .Niestety**,** kolejka rozkazów w 8486 nie czerpie korzyści jakie ma 8286**.**Kolejka rozkazów pracuje dobrze dla 8286 ponieważ CPU nie ma stałego dostępu do pamięci. Kiedy CPU nie ma stałego dostępu do pamięci, BIU może pobrać dodatkowe opcody instrukcji do kolejki rozkazów. Niestety CPU 8486 ma stały dostęp do pamięci ponieważ pobiera bajt opcodu na każdy cykl zegarowy. Dlatego kolejka rozkazów nie może wykorzystywać każdego „martwego" cyklu magistrali do pobierania dodatkowych bajtów opcodów - nie ma żadnego „martwego" cyklu magistrali. Jednakże, kolejka rozkazów jest wartościowa dla 8486 z bardzo prostej przyczyny: BIU pobiera dwa bajty na każdy dostęp do pamięci ,ale mimo to instrukcje są tylko jednobajtowe. Bez kolejki rozkazów, system musiałby wyraźnie pobrać każdy opcod ,nawet jeśli BIU ma już „przypadkowo" pobrany opcod wraz z poprzednią instrukcją. Z kolejką rozkazów, jednak ,system nie może pobierać żadnych opcodów .Pobiera je raz i zachowuje do użycia przez jednostkę pobierania opcodów**.**

Na przykład ,jeśli wykonujemy dwie jednobajtowe instrukcje z rzędu, BIU może pobrać oba opcody w jednym cyklu pamięci, zwalniając magistrale dla innych operacji. CPU może użyć tych wolnych cykli magistral do pobrania dodatkowych opcodów lub poradzenia sobie z innym dostępem do pamięci.

Oczywiście nie wszystkie instrukcje są długie na jeden bajt.8486 ma dwa rozmiary instrukcji: jeden bajt i trzy bajty .Jeśli wykonujemy ładowanie kilku trzybajtowych instrukcji z rzędu, ruszymy wolniej, np.

```
mov ax,1000
mov bx,2000
mov cx,3000
add ax,5000
```



Rysunek 3.27 Typowa maszyna Harwardzka

Każda z tych instrukcji odczytuje bajt opcodu i 16 bitowy operand (stała). Dlatego też ,zabiera to średnio 1,5 cyklu zegarowego do odczytu każdej instrukcji powyższej. W wyniku ,instrukcje wymagają sześciu cykli zegarowych to wykonania zamiast czterech.

Raz jeszcze wrócimy do tej samej zasady: najszybsze programy to programy, które używają najkrótszych instrukcji.Jeśli możemy używać krótszych instrukcji do osiągnięcia zadania, zróbmy to. Następująca sekwencja instrukcji dostarcza dobrego przykładu:

```
mov  ax,1000
mov  bx,1000
mov  cx,1000
mov  dx,1000
```

Możemy zredukować rozmiar tego programu i zwiększyć szybkość wykonania przez zmianę:

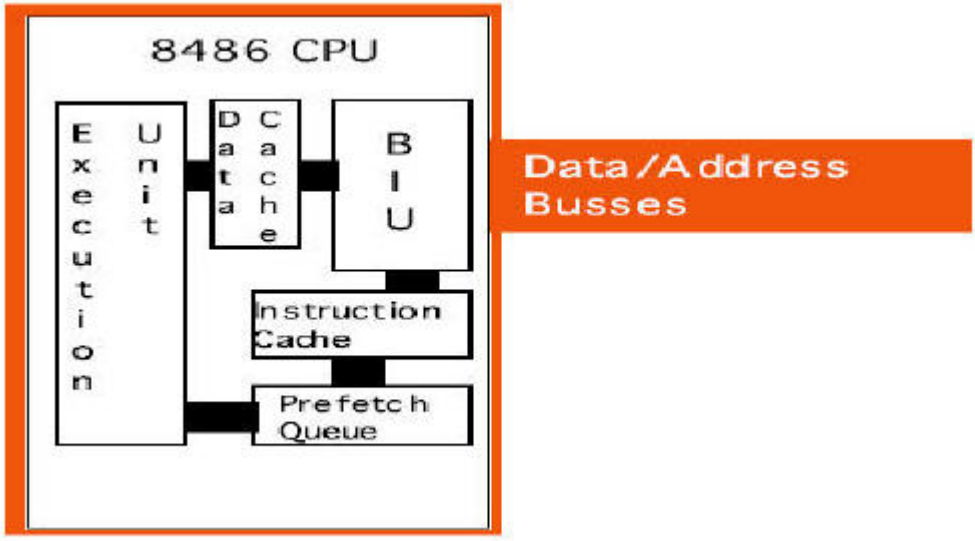
```
mov  ax,1000
mov   bx ,ax
mov  cx, ax
mov  ax ,ax
```

Ten kod ma tylko pięć bajtów długości w porównaniu do 12 bajtów z poprzedniego przykładu. Poprzedni kod zabierałby minimum pięć cykli zegarowych do wykonania, więcej jeśli są inne problemy ze sporami magistral. Ostatni przykład zabiera tylko cztery .Co więcej, drugi przykład zostawia wolne magistrale dla trzech z tych czterech okresów zegarowych, więc BIU może załadować dodatkowe opcod. Pamiętaj, krótsze często znaczy szybsze.

Podczas gdy kolejka rozkazów może uwolnić cykle magistral i wyeliminować spory magistral, niektóre problemy jeszcze istnieją .Przypuśćmy, że średnia długość instrukcji dla sekwencji instrukcji jest 2.5 bajtowa (osiąga się przez posiadanie trzech trzy bajtowych instrukcji i jednej jednobajtowej razem).W takim przypadku magistrala będzie zajęta pobieraniem opcodów i operandów instrukcji. Nie będzie żadnego wolnego czasu na odniesienie się do pamięci. Zakładając że kilka z tych instrukcji odnosi się do pamięci , potok wchodząc w kolejkę ,spowalnia wykonanie.

Przypuśćmy ,na chwilę ,że CPU ma dwie oddzielne przestrzenie pamięci, jedną dla instrukcji i jedną dla danych, każda ze swoją własną magistralą .Jest to nazywane Architekturą Harwardzką ponieważ pierwsza taka maszyna została zbudowana na Harvardzie maszynie harwardzkiej nie byłoby sporów na magistralach. BUI może

kontynuować pobieranie opcodów na magistrali instrukcji podczas uzyskiwania dostępu do pamięci na magistrali dane/pamięć (zobacz rysunek 3.27)



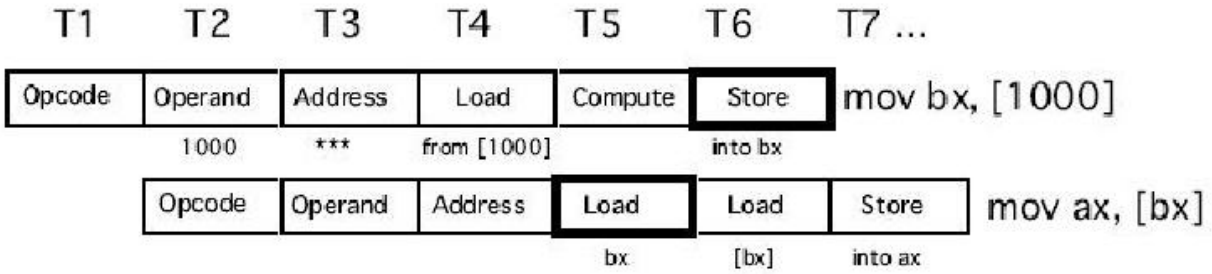
Rysunek 3.28: Wewnętrzna struktura CPU 8486

W prawdziwym świecie, jest bardzo mało prawdziwych maszyn harvardzkich .Ekstra wyprowadzenia potrzebne procesorowi dla wsparcia dwóch fizycznie oddzielnych magistral powiększa koszt procesora i przedstawia wiele innych problemów inżynierskich .Jednakże, projektanci mikroprocesorów odkryli ,że mogą uzyskać większe profity z architektury harvardzkiej z mniejszymi wadami przez użycie oddzielnych wbudowanych pamięci podręcznych dla danych i instrukcji. Zaawansowane CPU używają wewnętrzną architekturę harvardzką i zewnętrzną architekturę Von Neumanna. Rysunek 3.28 pokazuje strukturę 8486 z oddzielnymi pamięciami podręcznymi danych i instrukcji.

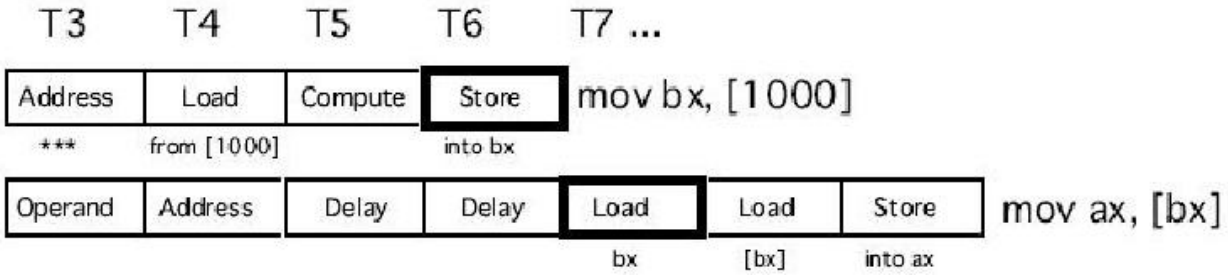
Każda ścieżka wewnątrz CPU przedstawia niezależne magistrale. Dane mogą przepływać na wszystkich ścieżkach jednocześnie. To znaczy ,że kolejka rozkazów może ściągać opcody instrukcji z pamięci podręcznej instrukcji podczas gdy jednostka wykonawcza zapisuje dane do pamięci podręcznej danych. Teraz BIU tylko pobiera opcody z pamięci zawsze kiedy nie może zlokalizować ich w pamięci podręcznej instrukcji .Podobnie, pamięć podręczna danych buforuje pamięć. CPU używa magistral danych/adresowej tylko kiedy odczytuje wartość która nie jest w pamięci podręcznej lub kiedy opróżnia bufor danych do pamięci głównej.

W ten sposób,8486 radzi sobie z pobieraniem operandów/opcodów instrukcji załatwiając problem w ten podstępny sposób .Poprzez dodanie ekstra układu dekodera, dekoduje instrukcje zaczynając z kolejki rozkazów i trzy bajty do kolejki rozkazów równolegle. Wtedy, jeśli poprzednia instrukcja nie miała 16 bitowego operandu, CPU użyje wyniku z pierwszego dekodera, jeśli poprzednia instrukcja używa operandu ,CPU używa wyniku z drugiego dekodera.

Chociaż nie można sterować obecnością, rozmiarem lub typem pamięci podręcznej w CPU ,jako programiści asemblerowi musimy być świadomi jak działa pamięć podręczna, przy pisaniu najlepszych programów. Instrukcje wbudowanej pamięci podręcznej są generalnie całkiem małe (8,192 bajty dla 80486,na przykład).Dlatego też ,skracajmy nasze instrukcje, większość z nich wypełni pamięć podręczną. Przy większości instrukcji które mamy w pamięci podręcznej mniej będzie występowało sporów magistral .Podobnie używając rejestrów do przechowania wyników tymczasowych ,umieszczamy mniejsze obciążenie w pamięci podręcznej danych, więc nie musimy często opróżniać danych do pamięci lub odzyskiwać danych z pamięci Używajmy rejestrów gdzie tylko możliwe!



Rysunek 3.29: Przypadek na 8486



Rysunek 3.30: Przypadek na 8486

3.3.12.4 PRZYPADKI NA 8486

Jest inny problem z używaniem przetwarzania potokowego : przypadkowe dane. Spójrzmy na profil wykonania następującej sekwencji instrukcji:

```
mov   bx,[1000]
mov ax,[bx]
```

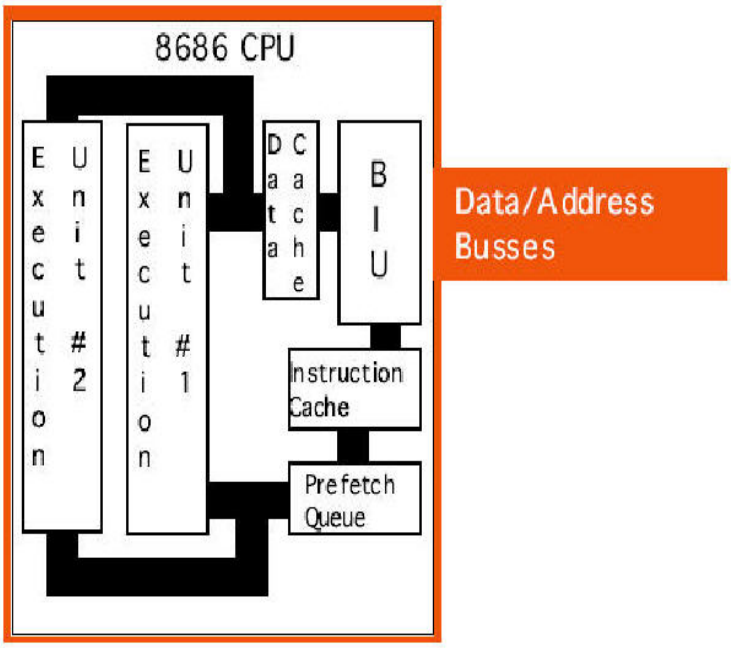
Kiedy te dwie instrukcje są wykonywane, potok szuka wygląda jak na rysunku 3.29.Zauważ główny problem. Te dwie instrukcje pobierają 16 bitowe wartości których adres pojawia się w lokacji 1000 w pamięci. Ale ta sekwencja instrukcji nie pracuje poprawnie! Niestety ,druga instrukcja już użyła wartości w bx zanim pierwsza instrukcja załaduje zawartość lokacji pamięci 1000 (T4 i T6 w powyższym diagramie).

Procesory CISC, podobnie jak 80x86, radzą sobie z przypadkiem automatycznie. Jednak, zakolejkują (zatrzymają) potok aż do synchronizacji dwóch instrukcji. Wykonanie w rzeczywistości na 8486 będzie wyglądało jak na pokazano rysunku 3.30

Poprzez opóźnienie drugiej instrukcji o dwa cykle zegarowe,8486 gwarantuje ,że ładowana instrukcja załaduje ax z właściwego adresu. Niestety, druga ładowana instrukcja wykonuje się w trzech cyklach zamiast w jednym. Jednak ,zastosowanie dwóch extra cykli zegarowych jest lepsze niż tworzenie niewłaściwych wyników. Na szczęście, możemy zredukować wpływ przypadku na szybkość wykonywania naszego programu.

Zauważ ,że dane przypadkowe występują wtedy kiedy operand źródłowy jednej instrukcji był operandem przeznaczenia poprzedniej instrukcji .Nie ma nic złego w ładowaniu bx z [1000] a potem ładowania ax z [bx],chyba, że występują one jedna po drugiej .Przypuśćmy ,że mamy taką sekwencję kodu:

```
mov   cx,2000
mov    bx,[1000]
mov    ax,[bx]
```



Rysunek 3.31: Wewnętrzna struktura CPU 8686

Możemy zredukować efekt przypadku które istnieje w tej sekwencji kodu poprzez proste przestawienie instrukcji. Zróbmy to i uzyskamy co następuje:

```
mov bx,[1000]
mov  cx,2000
mov ax,[bx]
```

Teraz instrukcja mov ax wymaga tylko jednego dodatkowego cyklu zamiast dwóch .Przez wprowadzenie dodatkowej instrukcji między instrukcje mov bx a mov ax możemy wyeliminować dane przypadkowe całkowicie.

Na procesorze potokowym, porządek instrukcji w programie może dramatycznie wpłynąć na wydajność tego programu. Zawsze szukajmy możliwego przypadku w naszych sekwencjach instrukcji .Eliminujmy je gdziekolwiek to możliwe przez przestawianie instrukcji.

3.3.13 PROCESOR 8686

Z architekturą potokową z 8486 możemy osiągnąć w najlepszym razie, czas wykonania instrukcji jeden CPI (clock per instruction).Czy jest możliwe wykonywać instrukcje szybciej? Na pierwszy rzut oka możesz pomyśleć ”Oczywiście, że nie ,możemy zrobić najwyżej jedną operację na cykl .Więc nie ma sposobu abyśmy wykonali więcej niż jedną instrukcję na cykl ”Zapamiętaj jednak ,że pojedyncza instrukcja nie jest pojedynczą operacją. W przykładach wcześniejszych każda instrukcja zabierała między sześć a osiem zakończonych operacji .Przez dodanie siódmej lub ósmej oddzielnej jednostki do CPU,możemy skutecznie wykonać te osiem operacji w jednym cyklu dając jeden CPI. Jeśli dodamy więcej sprzętu i wykonamy, powiedzmy 16 operacji od razu, czy możemy osiągnąć 0,5 CPI? Połowiczna odpowiedź brzmi „tak” CPU zawierający ten dodatkowy sprzęt jest to CPU superskalarny i może wykonać więcej niż jedną instrukcję podczas pojedynczego cyklu. Taką możliwość dodaną ma procesor 8686.

CPU superskalarny ma ,zasadniczo ,kilka jednostek wykonawczych (zobacz rysunek 3.31).Jeśli spotka dwie lub więcej instrukcji w strumieniu instrukcji (np. kolejka rozkazów) ,które mogą wykonywać się niezależnie ,robi to.

Są dwie zalety stosowania superskalarów. Przypuśćmy ,że mamy następujące instrukcje w strumieniu instrukcji:

```
mov   ax,1000
mov bx,2000
```

Jeśli nie ma problemów lub przypadku w kodzie otaczającym, i wszystkie sześć bajtów dla tych dwóch instrukcji jest obecnych w kolejce rozkazów ,nie ma powodów dlaczego CPU nie może pobrać i wykonać obu instrukcji równolegle. Wszystko to zależy od ekstra „krzemu" w chipie CPU implementującego dwie jednostki wykonawcze.

Poza tym przyspieszając niezależne instrukcje, CPU superskalarny może również przyspieszyć sekwencje programu zawierające przypadek. Jedynym ograniczeniem 8486 jest to,że przypadek się zdarza, naruszając całkowicie instrukcje w kolejce potoku .Każda instrukcja która następuje również będzie musiała czekać aż CPU zsynchronizuje wykonywanie instrukcji .Z superskalarnym CPU ,jednak, instrukcja następująca po przypadku może kontynuować wykonywanie poprzez potok tak długo dopóki nie mają swoich własnych przypadków.

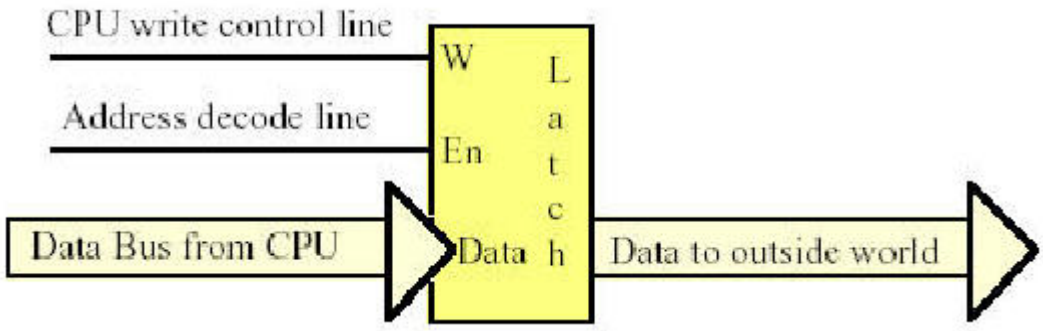
Programista asemblerowy, chcący pisać programy dla CPU superskalarnego, może radykalnie wpłynąć na jego wydajność. Pierwszą i główną zasadą która prawdopodobnie już znasz jest :używaj krótkich instrukcji .Skracaj swoje instrukcje, większość instrukcji CPU może pobrać w pojedynczej operacji ,dlatego też, jest bardziej prawdopodobne, że CPU będzie wykonywał szybciej niż jeden CPI .Większość superskalarnych CPU nie całkiem powiela jednostki wykonawcze. Mogą to być wielokrotne ALU, jednostki zmiennoprzecinkowe, itp. To znaczy, że pewna sekwencja instrukcji może wykonywać się bardzo szybko, podczas gdy inne nie. Musisz przestudiować dokładnie budowę swojego CPU, aby zadecydować która sekwencja instrukcji stworzy najlepsze wydajność

3.4 I/O (WEJŚCIE/WYJŚCIE)

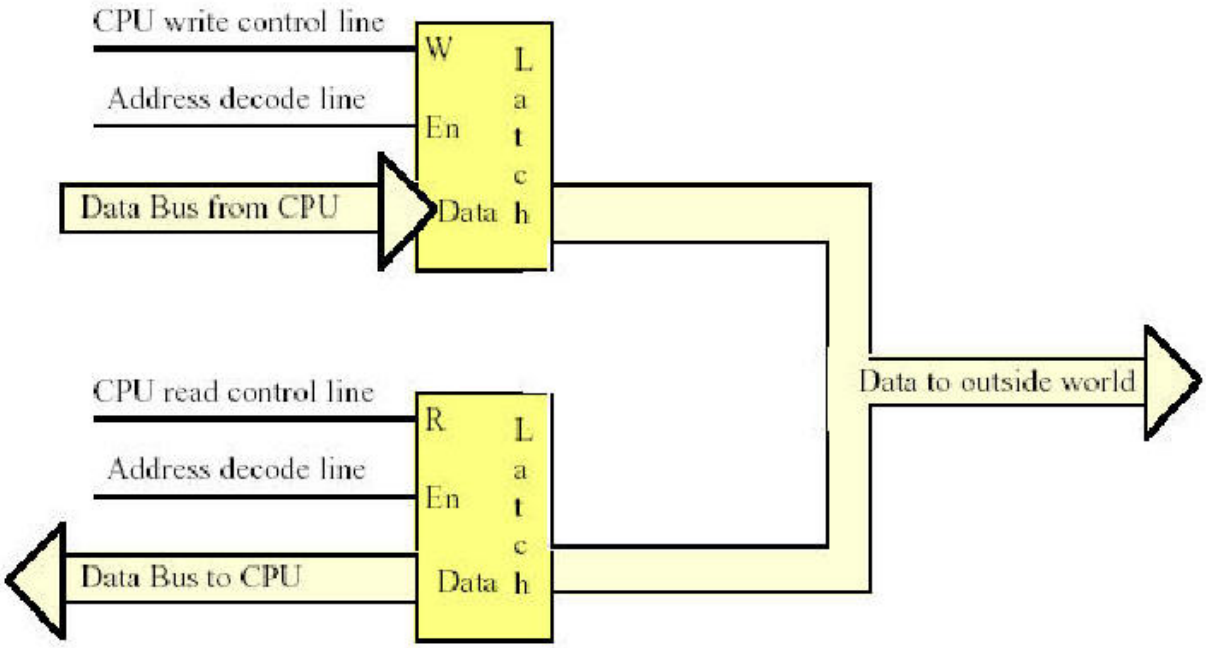
Są trzy podstawowe postacie wejścia i wyjścia ,których używa typowy system komputerowy :I/O-mapped I/O(odwzorowywanie I/O),memory mapped input/outpu (odwzorowywanie w pamięci I/O) i direct memory access (DMA) (bezpośredni dostęp do pamięci). I/O mapped I/O używa specjalnych instrukcji do przenoszenia danych między systemem komputerowym a światem zewnętrznym; memory mapped I/O używa specjalnej lokacji w pamięci w normalnej przestrzeni adresowej CPU do porozumiewania się z urządzeniami świata realnego; DMA jest specjalna postacią memory-mapped I/O gdzie urządzenia peryferyjne odczytują i zapisują pamięć bez przechodzenia przez CPU. Każdy mechanizm I/O ma swój własny zbiór zalet i wad, które omówimy w tej sekcji.

Pierwszą rzeczą do nauczenia się o subsystemie I/O jest to,że I/O w typowym komputerze jest radykalnie różny niż I/O w typowym języku programowania wysokiego poziomu. W prawdziwym systemie komputerowym rzadko znajdziemy instrukcje maszynowe które zachowują się jak writeln, printf lub nawet instrukcje x86 get i put. Faktycznie, większość instrukcji I/O zachowuje się dokładnie jak instrukcja mov x86.Wysyłając dane do urządzenia wyjściowego, CPU po prostu przesuwa te dane do specjalnej lokacji pamięci (w przestrzeni adresowej I/O, jeśli I/O-mapped I/O [zobacz Podsystem I/O] lub pod adres w przestrzeni adresowej pamięci jeśli używamy memory-mapped I/O).Odczytując dane z urządzenia wejściowego, CPU po prostu przesuwa dane spod adresu (I/O lub pamieć) tego urządzenia wewnątrz CPU. Zazwyczaj jest więcej stanów oczekiwania powiązanych z typowym urządzeniem peryferyjnym niż rzeczywista pamięć ,operacje wejścia lub wyjścia wyglądają bardzo podobnie jak operacje odczytu lub zapisu pamięci. (zobacz Dostęp do Pamięci a System zegarowy)

Port I/O jest urządzeniem które wygląda jak komórka pamięci komputera ale zawiera połączenia do świata zewnętrznego. Port I/O typowo używa zatrzasków zamiast przerzutników do implementacji komórki pamięci. Kiedy CPU zapisuje pod adres powiązany z zatrzaskiem, urządzenie zatrzasku przechwytuje dane umożliwiając podstawienie przewodów zewnętrznych do CPU (Zobacz rysunek 3.32) Zauważ ,że port I/O może być tylko do odczytu, tylko do zapisu lub odczytu/zapisu. Port na rysunku 3.32, na przykład, jest portem tylko do zapisu. Ponieważ



Rysunek 3.32:Port wyjściowy stworzony z pojedynczego zatrzasku



Rysunek 3.33Port I/O wymagający dwóch zatrzasków

wyjścia na zatrzasku nie zwracają na magistralę danych CPU,CPU nie może odczytać danych zawartych na zatrzasku .Oba ,dekodowanie adresu i zapis linii sterujących muszą być aktywne przy operacji na zatrzasku.; kiedy odczytuje z adresu zatrzasku linia dekodująca jest aktywna, ale linia sterująca zapisem nie.

Rysunek 3.33 pokazuje jak stworzyć port odczyt/zapis I/O. Dane zapisane na wyjście portu zwracają przeźroczysty zatrzask .Obojętnie kiedy CPU odczytuje adres dekodujący linie odczytu i dekodujące są wzbudzone a to wzbudzenie obniża zatrzask. Miejsca gdzie poprzednio zapisano dane do portu wyjściowego na magistrali danych CPU, pozwalają CPU odczytać te dane. Port tylko do odczytu (wejściowy) jest po prostu obniżony do połowy z rysunku 3.33;system ignoruje dane zapisane do portu wejściowego.

Doskonałym przykładem portu wyjściowego jest równoległy port drukarki. CPU typowo zapisuje znaki ASCII szerokości bajtu do portu wyjściowego który łączy łączem DB-25F z tyłu komputera. Kabel transmituje tą daną do drukarki gdzie port wejściowy (drukarki) odbiera daną. Procesor wewnątrz drukarki konwertuje te znaki ASCII na sekwencję punktów drukowanych na papierze.

Generalnie dane urządzenie peryferyjne używa więcej niż pojedynczego portu I/O.W PC typowe równoległe łącze drukarki, na przykład, używa trzech portów :port odczyt/zapis ,port wejściowy, port wyjściowy .Port odczyt/zapis jest portem danych (pozwala CPU odczytać ostatni znak ASCII zapisany do portu drukarki).Port wejściowy zwraca sygnały sterujące z drukarki ,sygnały te wskazują czy drukarka jest gotowa do zaakceptowania kolejnego znaku ,czy jest wyłączona, czy jest papier itp. Port wyjściowy przekazuje informacje sterujące do drukarki takie jak czy dane są dostępne do druku.

Dla programisty, różnicami między operacjami I/O mapped i memory-mapped I/O są używane instrukcje .Dla memory-mapped I/O każda instrukcja która uzyskuje dostęp do pamięci, może uzyskać dostęp do portu memory-mapped I/O W x86 instrukcje mov, add,sub,cmp,and ,or i not mogą czytać pamięć ;instrukcje mov i not mogą zapisać dane do pamięci. I/O mapped I/O używa specjalnych instrukcji przy dostępie do portów. Na przykład, CPU x86 używa instrukcji get i put .Rodzina Intela 80x86 używa instrukcji in i out. Instrukcje in i out 80x86 pracują tak jak instrukcja mov z wyjątkiem umiejscowienia swoich adresów na magistrali adresowej I/O zamiast magistrali adresowej pamięci (zobacz Podsystem I/O).

Podsystem memory-mapped I/O i podsystem I/O mapped obie wymagają CPU do przesunięcia danych między urządzeniem peryferyjnym a pamięcią główna .Na przykład, dane wejściowe 10 bajtowe są podawane z portu wejściowego i przechowywane w pamięci, CPU musi odczytać każdą wartość i przechować ją w pamięci. Dla bardzo szybkich urządzeń I/O CPU może być zbyt wolny kiedy przetwarza te dane po bajcie. Takie urządzenia generalnie zawierają interfejs do magistrali CPU więc odczytuje i zapisuje bezpośrednio pamięć. Jest to znane jako bezpośredni dostęp do pamięci ponieważ urządzenia peryferyjne uzyskują dostęp do pamięci bezpośrednio ,bez użycia CPU jako pośrednika. To często pozwala kontynuować operacje I/O równolegle z innymi operacjami CPU

,tym samym rośnie całkowita szybkość systemu .Zauważ ,jednak ,że CPU i urządzenia DMA nie mogą oba używać magistral adresowych i danych w tym samym czasie. Zatem ,bezpośrednia obróbka zdarza się jeśli CPU ma pamięć podręczną a wykonywany kod i dostępne dane znajdują się w pamięci podręcznej (więc magistrala jest wolna).Niemniej jednak, nawet jeśli CPU musi się zatrzymać i czekać na zakończenie operacji DMA ,I/O jest dużo szybsze ponieważ wiele operacji na magistralach I/O lub memory mapped I/O składa się z instrukcji pobrania lub dostępu do portu I/O które są nieobecne podczas operacji DMA.

3.5 PRZERWANIA I ZAPYTANIE I/O

Wiele urządzeń I/O nie może przyjmować danych w przypadkowy sposób .Na przykład, Pentium jest zdolny wysyłać kilka milionów znaków na sekundę do drukarki, ale ta drukarka (prawdopodobnie) nie jest w stanie wydrukować tak dużo znaków na sekundę. Podobnie ,urządzenia wejściowe takie jak klawiatura jest niezdolna dostarczyć kilka milionów uderzeń w klawisze na sekundę. (ponieważ to zależy od szybkości człowieka nie komputera) CPU potrzebuje jakiegoś mechanizmu do koordynowania przekazywania danych między komputerem a jego urządzeniami peryferyjnymi.

Jednym z powszechnych sposobów koordynowania transferu danych jest dostarczenie kilku bitów stanu w pomocniczym porcie wejściowym. Na przykład, jeden w pojedynczym bicie w porcie I/O może powiedzieć CPU, że drukarka jest gotowa do zaakceptowania więcej danych, zero wskazywałoby, że drukarka jest zajęta i CPU nie powinno wysyłać nowych danych do drukarki. Podobnie bit jeden w różnych portach może powiedzieć CPU, że uderzenie w klawisz z klawiatury jest dostępne na porcie danych klawiatury ,zero na tym samym bicie może wskazywać, że uderzenie w klawisze jest niedostępne. CPU może testować te bity przed odczytaniem klawisza z klawiatury lub zapisu znaku do drukarki.

Załóżmy, że port danych drukarki jest odwzorowany w pamięci pod adresem 0FE0h a stan portu drukarki jest bitem zero z portu odwzorowanego w pamięci pod 0FFE2h.Następujący kod czeka aż do chwili kiedy drukarka jest gotowa do zaakceptowania bajtu danych potem zapisania bajtu w najmniej znaczącym bajcie ax do portu drukarki:

```
0000:   mov     bx,[FFE2]
0003:   and     bx,1
0006:   cmp     bx,0
0009:   je      0000
000C:   mov     [FFE0],ax
```

- -
- -
- -
- -

Pierwsza instrukcja pobiera dane do portu wejścia .Druga instrukcja logicznie dodaje tą wartość z jedynką zerując bity od jeden przez piętnaście i ustawia bit zero na bieżący stan portu drukarki. Zauważ ,że to tworzy wartość zero w bx, jeśli drukarka jest zajęta. tworzy wartość jeden w bx jeśli drukarka jest gotowa do akceptacji dodatkowej danej .Trzecia instrukcja sprawdza bx aby zobaczyć czy zawiera zero (np. drukarka jest zajęta)Jeśli drukarka jest zajęta ,ten program skacze do lokacji zero i powtarza ten proces tak długo jak długo bit stanu drukarki wynosi jeden.

Następujący kod dostarcza przykładu czytania z klawiatury. Przyjmiemy, że bit stanu klawiatury jest zero spod adresu 0FFE6h (zero znaczy, że nie wciśnięto klawisza) a kod ASCII klawisza pojawia się pod adresem 0FFE4h kiedy bit zero z lokacji 0FFE6h zawiera jeden:

```
0000:   mov     bx,[FFE6]
0003:   and     bx,1
0006:   cmp     bx,0
0009:   je      0000
000C:   mov     ax,[FFE4]
```

Ten typ operacji I/O, gdzie CPU stale testuje port aby zobaczyć ,czy dana jest dostępna, to - technika odpytywania ,to znaczy ,CPU pyta port czy ma dostępną daną lub czy jest zdolny do przyjęcia danej. Zapytanie I/O jest z natury systemem niewydolnym. Rozważmy co zdarzy się w poprzednim segmencie kodu jeśli użytkownikowi zajmie 10 sekund naciśnięcie klawisza na klawiaturze - CPU wykona nic nie robiącą pętle dla tych dziesięciu sekund.

W pierwszych komputerach osobistych (np. .Apple II),jest dokładnie opisane jak program mógł odczytać dane z klawiatury :Kiedy musiał odczytać klawisz z klawiatury, musiał odpytywać stan portu klawiatury aż do momentu kiedy klawisz był dostępny. Takie komputery nie mogły robić innych operacji podczas oczekiwania na wciśnięcie klawisza .Co ważniejsze, jeśli zbyt dużo czasu mija między sprawdzeniem stanu portu klawiatury, użytkownik mógł nacisnąć drugi klawisz a pierwsze naciśnięcie było gubione.

Rozwiązaniem tego problemu jest wprowadzenie mechanizmu przerwań .Przerwanie jest to zewnętrzne zdarzenie sprzętowe (jak naciśnięcie klawisza),które powoduje ,że CPU przerywa bieżącą sekwencję instrukcji i wywołuje specjalny podprogram obsługi przerwań (ISR).ISR zachowuje wartość wszystkich rejestrów i flag (żeby nie przeszkadzały w obliczaniu przerwań**),** robi jakieś operacje konieczne do poradzenia sobie z przerwaniem, przywraca rejestry i flagi, a potem rozpoczyna się na nowo wykonywanie kodu sprzed przerwania .W wielu systemach komputerowych (np. PC) wiele urządzeń I/O generuje przerwania, kiedy tylko mają dostępne dane lub mogą zaakceptować dane z CPU.ISR szybko przetwarza prośbę w tle ,pozwalając kilku innym obliczeniom kontynuowanie pierwszoplanowe.

CPU ,które wspierają przerwania muszą dostarczyć jakiś mechanizm, który pozwoli programiście wyspecyfikować adres ISRa do wykonania kiedy wystąpi przerwanie. Typowym wektorem przerwania jest specjalna komórka pamięci która zawiera adres ISRa do wykonania kiedy wywołane jest przerwanie. CPU x86,na przykład zawierają dwa wektory przerwań: jeden dla przerwań ogólnego zastosowania i jeden dla przerwania reset (przerwanie reset odpowiada naciśnięciu przycisku reset na większości PC).Rodzina Intela 80x86 wspiera do 256 różnych wektorów przerwań.

Po zakończeniu operacji ISR, ogólnie rzecz biorąc, zwraca sterowanie do zadania pierwszoplanowego specjalną instrukcją „powrót z przerwania" W x86,to zadanie spełnia instrukcja iret (interrupt returns).ISR zawsze powinno kończyć się tą instrukcją ponieważ ISR może zwrócić sterowanie do programu który przerwał.

Typowy wejściowy system sterowany przerwaniami używa ISR do odczytu danych z portu wejściowego i buforowania go gdy tylko dane staną się dostępne. Program pierwszoplanowy może czytać te dane z bufora bez pośpiechu bez zagubienia żadnej danej z portu. Podobnie, wyjściowy system sterowany przerwaniami (przerwanie występuje gdy tylko urządzenie wyjściowe jest gotowe zaakceptować więcej danych) może usunąć dane z bufora gdy tylko urządzenie peryferyjne jest gotowe zaakceptować nowe dane.

3.8 PODSUMOWANIE

Napisanie dobrego programu w asemblerze wymaga sporej wiedzy o wykorzystywanym sprzęcie. Prosta znajomość zbioru instrukcji jest nie wystarczająca .Chcąc tworzyć najlepsze programy musimy zrozumieć jak sprzęt wykonuje nasz program i uzyskuje dostęp do danych.

Większość nowoczesnych systemów komputerowych przechowuje programy i dane w tej samej przestrzeni pamięci (architektura Von Neumanna).Jak większość maszyn vonNeumana, system 80x86 ma trzy główne komponenty: CPU ,I/O i pamięć .Zobacz:

- „Podstawowy System Komponentów"

Dane podróżują między CPU, urządzeniami I/O i pamięcią w systemie magistral. Są trzy główne magistrale zastosowane w rodzinie 80x86,magistrala adresowa, magistrala danych i magistrala sterująca. Magistrala adresowa przenosi liczby binarne które są wyspecyfikowane w lokacji pamięci lub porcie I/O do którego CPU życzy sobie uzyskać dostęp; magistrala danych przenosi dane między CPU a pamięcią lub I/O, magistrala sterująca przenosi ważne sygnały które określają czy CPU odczytuje czy zapisuje dane z pamięci lub uzyskuje dostęp do portu I/O. Zobacz:

- 'System Magistral"
- 'Magistrala Danych"
- „Magistrala Adresowa"
- „Magistrala Sterująca"

Liczba linii danych na magistrali danych określa rozmiar procesora. kiedy mówimy, że procesor jest ośmiobitowy, mamy na myśli osiem linii danych na jego magistrali danych. Rozmiar danych z którymi procesor może sobie radzić CPU nie wpływa na rozmiar CPU. Zobacz:

- „Magistrala Danych"
- „Rozmiar Procesora"

Magistrala adresowa przenosi liczbę binarną z CPU do pamięci i I/O wybierając poszczególne elementy pamięci lub portu I/O. Liczba linii na magistrali adresowej ustawia maksymalną liczbę lokacji do których CPU może mieć dostęp. Typowy rozmiar magistrali adresowej w 80x86 to 20.24 lub 32 bity. Zobacz:

- „Magistrala Adresowa”

CPU 80x86 również mają magistralę sterującą która zawiera kilka sygnałów koniecznych dla właściwych operacji systemu Zegar systemowy, odczyt/zapis sygnałów sterujących, sygnały sterujące I/O i pamięci są kilkoma próbkami wielu linii które pojawiają się na magistrali sterującej .Zobacz:

- :Magistrala sterująca”

Podsystem pamięci to miejsce gdzie CPU przechowuje instrukcje programu i danych. W systemach opartych na 80x86, pamięć jawi się jako tablica bajtów ,każdy z własnym unikalnym adresem. Adres pierwszego bajtu w pamięci to zero, a adres ostatniego wolnego bajtu w pamięci to 2^{n-1},gdzie n to liczba linii na magistrali adresowej.80x86 przechowuje słowa w dwóch kolejnych lokacjach w pamięci. najmniej znaczący bajt słowa jest pod niższym adresem z tych dwóch bajtów; bardziej znaczący bajt bezpośrednio następuje w następnym, wyższym adresem. Chociaż słowo zużywa dwa adresy pamięci, kiedy pracujemy ze słowem ,po prostu używamy adresu jego najmniej znaczącego bajtu jako adresu słowa. Podwójne słowo zużywa cztery kolejne bajty w pamięci. Najmniej znaczący bajt pojawia się pod najniższym adresem z tych czterech, najbardziej znaczący pod najwyższym. ”Adresem” podwójnego słowa jest adres bajtu najmniej znaczącego .Zobacz:

- „Podsystem Pamięci”

CPU z 16,32 lub 64 bitowymi magistralami danych generalnie organizują pamięć w banki.16 bitowy podsystem pamięci używa dwóch banków po osiem bitów każdy,32 bitowy podsystem pamięci używa czterech banków po osiem bitów każdy a 64 bitowy podsystem pamięci używa ośmiu banków po osiem bitów każdy. Uzyskanie dostępu do słowa lub podwójnego słowa pod tym samym adresem wewnątrz wszystkich banków jest szybsze niż uzyskanie dostępu do obiektu, który jest podzielony między dwa adresy w różnych bankach Dlatego, powinniśmy próbować ustawiać dane słowa tak,żeby zaczynały się pod parzystym adresem a podwójne słowa danych tak,żeby zaczynały się pod adresem który równo dzieli się przez cztery .Możemy umiejscowić bajt danych pod każdym adresem .Zobacz

- „Podsystem Pamięci”

CPU 80x86 dostarcza oddzielnej 16 bitowej przestrzeni adresowej I/O która pozwala CPU uzyskać dostęp do każdego z 65,536 różnych portów I/O Typowe urządzenie I/O połączone z IBM PC używa tylko 10 z tych linii adresowych, ograniczając system do 1,024 różnych portów .Głównym zyskiem z używania przestrzeni adresowej I/O zamiast odwzorowywania wszystkich urządzeń I/O w przestrzeni pamięci jest to,że urządzenia I/O nie muszą naruszać przestrzeni adresowej pamięci. I/O i dostęp do pamięci, rozróżniają specjalne linie sterujące w systemie magistral. Zobacz:

- „Magistrala Sterująca
- „Podsystem I/O”

Zegar systemowy steruje szybkością przy której procesor wykonuje podstawowe operacje. Większość CPU działa opierając się na rosnących lub opadających zboczach zegarowych. Przykłady obejmują wykonywanie instrukcji, dostęp do pamięci i sprawdzanie stanu oczekiwania. Im szybciej chodzi zegar tym szybciej wykonuje się program; jednakże pamięć musi być tak szybka jak zegar systemowy lub musimy wprowadzić stany oczekiwania ,które spowalniają system. Zobacz:

- „System Synchronizacji”
- „Zegar Systemowy”
- „Dostęp Do Pamięci i Zegar systemowy”
- „Stan Oczekiwania”

Większość programów wykazuje lokalność odniesienia. Uzyskują dostęp do tej samej lokalizacji pamięci wielokrotnie na przestrzeni małego okresu czasu (czasowa lokalność) lub uzyskują dostęp do sąsiadujących lokacji pamięci podczas krótkiego okresu czasu (przestrzenna lokalność) Podsystem pamięci podręcznej wykorzystuje ten fenomen do zredukowania stanów oczekiwania w systemie. Mała pamięć podręczna może osiągnąć poziom 80-95% trafień. Dwupoziomowa pamięć podręczna używa dwóch różnych pamięci podręcznych (jedna zintegrowana z CPU, jedna poza CPU) dla osiągnięcia lepszej wydajności systemu.. Zobacz:

- „Pamięć Podręczna Cache”

CPU ,takie jak z rodziny 80x86, przerywają wykonywanie instrukcji maszynowych wewnątrz kilku odrębnych kroków ,każdy wymagający jednego cyklu zegarowego. Te kroki zawierają pobieranie opcodów instrukcji ,dekodowania tych opcodów, pobierania operandów dla instrukcji, obliczania adresów pamięci ,uzyskiwania dostępu do pamięci, wykonywania podstawowych operacji i przechowywania wyników dalej .Na bardzo uproszczonym CPU, proste instrukcje mogą zabierać kilka cykli zegarowych. Najlepszym sposobem poprawy wydajności CPU jest wykonanie kilku wewnętrznych operacji równolegle z innymi. Prosty schemat to umieszczenie instrukcji w kolejce rozkazów w CPU. Pozwala to zachodzić na siebie opcodom pobieranym i dekodowanie wykonywanych instrukcji ,często obcinając czas wykonania o połowę .Inną alternatywą jest użycie instrukcji potokowych ,gdzie możemy wykonać kilka instrukcji równolegle .W końcu, możemy zaprojektować superskalarny CPU który wykonuje dwie lub więcej instrukcji w tym samym czasie. Te techniki pozwalają przyspieszyć działanie programów. Zobacz:

- „Procesor 886”
- „Procesor 8286”
- :Procesor 8486”
- „Procesor 8686”

Chociaż CPU potokowy i superskalarny poprawiają całkowicie wydajność systemu, uzyskanie najlepszej wydajności z tak złożonego CPU wymaga ostrożnego planowania przez programistę. Kolejki potokowe i przypadki mogą spowodować poważną stratę wydajności w kiepsko zorganizowanym programie. Poprzez ostrożne organizowanie sekwencji instrukcji w programie możemy uczynić program dwu lub trzy razy szybszym. Zobacz:

- „Przetwarzanie Potokowe w 8486”
- „Kolejka potoku”
- „Pamieć Podręczna, Kolejka Rozkazów i 8486”
- „Przypadek w 8486”
- „Procesor 8686”

Podsystem I/O jest trzecim głównym komponentem z maszyny VonNeumanna. Są trzy podstawowe sposoby przemieszczania danych między systemem komputerowym a światem zewnętrznym: I/O mapped I/O, memory mapped I/O i bezpośredni dostęp do pamięci (DMA).Po więcej informacji zajrzyj:

- „I/O (Wejście/Wyjście)”

Dla poprawienia wydajności systemu, większość nowoczesnych komputerów używa przerwań dla zawiadomienia CPU kiedy operacja I/O jest zakończona. Pozwala to CPU kontynuować inne przetwarzanie zamiast czekać na zakończenie operacji I/O (odpytywanie portów I/O)Po więcej informacji na ten temat zajrzyj

- „Przerwania i Zapytanie I/O”

3.9 PYTANIA

1. Jakie są trzy komponenty stanowiące maszynę Von Neumanna
2. Jaki jest cel:
   a) magistrali systemowej
   b) magistrali adresowej
   c) magistrali danych
   d) magistrali sterującej

3 3. Jak magistrala definiuje „rozmiar” procesora?
4 4. Z której magistrali sterującej możemy mieć dużo pamięci?
5. Czy rozmiar magistrali danych steruje maksymalną wartością jaką CPU może przetworzyć? Wyjaśnij
6. jakie są rozmiary magistrali danych:
   a) 8088 b)8086 c)80286 d)80386sx e)80386 f)80486 g)80586?Petium
7. jaki jest rozmiar magistral adresowych powyższych procesorów?
8. Jak dużo „banków” pamięci posiada każdy z powyższych procesorów?

9. Wyjaśnij jak przechowuje się słowo w pamięci adresowanej bajtem (to znaczy pod jakim adresem).Wyjaśnij jak przechowuje się podwójne słowo
10. Jak dużo operacji na pamięci zabiera odczyt słowa z następujących adresów na tych procesorach?

11.Powtórz powyższe dla podwójnego słowa
12. Wyjaśnij który adres jest najlepszy dla zmiennych bajtowych, słowa i podwójnego słowa w procesorach 8088,80286 i 80386.
13. Jak dużo różnych lokacji I/O można zaadresować w chipie 80x86?Jak dużo jest dostępnych na PC?
14. Jaki jest cel zegara systemowego?
15. Co to jest cykl zegarowy?
16. .Jakie są związki między częstotliwością zegara a okresem zegarowym?
17. Jak dużo cykli zegarowych jest wymaganych dla każdego następnego odczytu z pamięci?
    a) 8088 b)8086 c)080486
18. Co oznacza termin „ czas dostępu do pamięci""
19. Co to jest stan oczekiwania
20. Jeśli jest uruchomiony 80486 przy następujących szybkościach zegara, jak dużo stanów oczekiwania jest wymaganych jeśli używamy 80 ns RAM (nie zakładając innych opóźnień)
    a) 20 MHz b)25 MHz c) 33 MHz d)50 MHz e) 100 MHz
21. Jeśli CPU pracuje przy 50 MHz,20 ns RAM prawdopodobnie nie będzie dość22. szybki aby operować23. przy zerowym stanie oczekiwania Wyjaśnij dlaczego
24. Ponieważ pod-10ns RAM jest dostępny, dlaczego niema wszystkich zerowych stanów oczekiwania w systemie?
25. Wyjaśnij jak pamięć podręczna obsługuje zachowanie stanów oczekiwania?
26. Jaka jest różnica między czasową lokalnością odniesienia przestrzenną lokalnością odniesienia?
27. Wyjaśnij gdzie czasowa i przestrzenna lokalność odniesienia wydarzy się w następującym kodzie pascalowskim:

```
while 1<10 do begin
                x:=x*1;
                i:=1+l;
end;
```

28. Jak pamięć29. podręczna poprawia wydajność sekcji kodu wystawionej w przestrzennej lokalności odniesienia
30. W jakich okolicznościach pamięć podręczna nie zachowa żadnego stanu oczekiwania
31. Jaka jest skuteczna (średnia) liczba stanów oczekiwania w następujących systemach posługujących się pod:
    a) 80% trafień,10 stanów oczekiwania(WS) dla pamięci,0 WS dla pamięci podręcznej
    b) 90% trafień,7 WS dla pamięci,0 WS dla pamięci podręcznej
    c) 95 trafień ,10 WS pamięć,1 WS pamięć podręczna
    d) 50 % trafień,2 WS pamięć,0 WS pamięć podręczna
32. Jaki jest cel dwupoziomowego systemu pamięci podręcznej? Co on przechowuje?
33. Jaka jest skuteczna liczba stanów oczekiwania dla następujących systemów:
    a) 80 % trafień głównej pamięci podręcznej (HR) zero WS;95% pomocnicza pamięć podręczna HR z 2 WS;10 WS dla dostępu do głównej pamięci
    b) 50 % główna pamięć podręczna HR, zero WS,98% pomocnicza pamięć podręczna HR, jeden WS; pięć WS dla dostępu do pamięci głównej
    c) 95% główna pamięć podręczna HR, jeden WS;98% pomocnicza pamięć podręczna HR,4 WS;10 WS dla pamięci głównej
34. Wyjaśnij cel jednostki sprzęgającej z magistralą (BIU),jednostki wykonawczej i jednostki sterującej
35. Dlaczego zabiera więcej niż jeden cykl zegarowy wykonanie instrukcji. Podaj kilka przykładów x86
36. Jak kolejka rozkazów przechowuje czas? Podaj kilka przykładów
37. Jak przetwarzanie potokowe pozwala nam (pozornie) wykonywać jedną instrukcję na cykl zegarowy? Podaj przykład
38. Co to jest przypadek**?**
39. Co zdarzy się w 8486 kiedy pojawi się przypadek?
40. Jak można wyeliminować efekt przypadku?

41. Jak skok (Jmp/jcc) wpływa na:
    a) kolejkę rozkazów
    b) przetwarzanie potokowe
42. Co to jest kolejka potokowa?
43. Poza tym oczywistym pożytkiem redukowania stanów oczekiwania ,jak pamięć podręczna może poprawić wydajność systemu potokowego?
44. Co to jest Harvardzka Architektura Maszyny
45. Jak superskalarny CPU przyspiesza wykonanie?
46. Jakie są dwie główne techniki które powinniśmy użyć47. w superskalarnym CPU aby zapewnić48. wykonywanie kodu tak szybko jak to możliwe?(zauważ, są to szczegóły mechaniczne, ”lepsze algorytmy” nie liczą się tutaj)
49. Co to jest przerwanie? Jak poprawia osiągi systemu
50. Co to jest odpytywanie I/O?
51. Jaka jest różnica między memory mapped i mapped I/O
52. DMA jest specjalnym przypadkiem memory-mapped I/O. Wyjaśnij.