Stanisław Ambroszkiewicz

ARCHITEKTURA KOMPUTERÓW

dla i z punktu widzenia programistów

Na podstawie kilkunastoletnich wykładów i laboratoriów z Architektury systemów komputerowych na kierunku informatyka Uniwersytetu w Siedlcach.

Spis treści

Rozdział 1. Wstęp

Istnieje bardzo obszerna literatura o architekturze komputerów – patrz ostatni rozdział tej książki. Większość z nich wprowadza mnóstwo technicznych szczegółów (ważnych punktu widzenia sprzętu – *hardware*), ale zaciemniających istotę ważną dla informatyków, a szczególnie dla programistów. Ta istota to architektura komputera według von Neumanna.

Ta książka (oparta na wieloletnich wykładach i laboratoriach programistycznych prowadzonych przez autora) jest wprowadzeniem do współczesnej architektury komputerów, przeznaczonym głównie dla programistów.

1. Co to jest komputer?

Podstawowe pojęcia w informatyce to: liczby i liczenie, dane (w postaci strumieni bajtów) i ich interpretacja (znaczenie) oraz przetwarzanie.

https://en.wikipedia.org/wiki/Computer
Komputer to maszyna, którą można zaprogramować do automatycznego wykonywania sekwencji operacji arytmetycznych lub logicznych (obliczeń).

https://www.britannica.com/science/computer-science/Architecture-and-organization
Computer architecture deals with the design of computers, data storage devices, and networking components that store and run programs, transmit data, and drive interactions between computers, across networks, and with users.

1.1. Z perspektywy raczej informatyka niż elektronika

Poniższe cytaty powinny być inspiracją dla przyszłych młodych adeptów informatyki, że należy raczej „mierzyć siły na zamiary, a nie zamiary podług sił”, cytując naszego wieszcza.

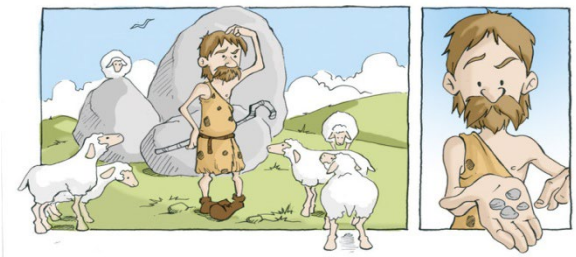
- *DOS addresses only 1 MB of RAM because we cannot imagine any applications needing more.* Microsoft, 1980.
- *640KB ought to be enough for anybody.* Bill Gates, 1981.
- *I think there is a world market for maybe five computers*. Thomas Watson, IBM Chairman, 1943.
- *There is no reason anyone would want a computer in their home*. Ken Olsen, DEC founder, 1977.

Komputer – słowo pochodzące od angielskiego słowa „computer”, czyli „ten, który liczy”. Cel, jaki przyświeca tej książce, to zrozumieć, co to znaczy liczyć i jak można zautomatyzować liczenie. Człowiek potrafi liczyć. Można to liczenie zautomatyzować i wówczas maszyna może liczyć. Komputer to maszyna, która potrafi liczyć. Czy tak jak człowiek?

Kluczowe pojęcia to:

- informacja i jej przetwarzanie,
- algorytmy i programy.

Liczby – jak dawniej liczono (owce, barany i kozły, monety, …).



Liczby i operacje na liczbach:
Liczby to abstrakcja z liczenia konkretnych przedmiotów.
W historii powstawały różnorodne reprezentacje liczb i różnorodne systemy liczenia.

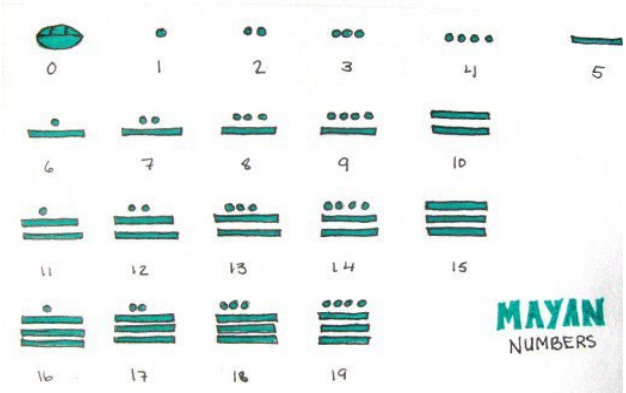
- Najprostszy i najbardziej naturalny system pozycyjny to:
  - | jeden,
  - || dwa,
  - ||| trzy itd.

Najbardziej popularne i potrzebne operacje na liczbach to:

- dodawanie,
- odejmowanie,
- mnożenie,
- dzielenie.

System Majów jest szczególnie ciekawy; jest tutaj zero.

We współczesnych komputerach używany jest system dwójkowy (więcej będzie później).



1.2 Algorytmy

Algorytm to sposób obliczania (na liczbach) oraz przetwarzania danych (strumieni bajtów).
Przykłady: pisemne dodawanie, odejmowanie, mnożenie, dzielenie (na podstawie dodawania cyfr i prostego odejmowana) i tabliczki mnożenia. Również przetwarzanie wyrazów i zdań.

Definicja algorytmu:

Jednoznaczna metoda obliczenia wyniku (*output*) w skończonym czasie na podstawie danych wejściowych (*input*). Te dane to np. liczby naturalne, bajty, (złożone) struktury danych jako ciągi bajtów, odpowiednio interpretowane.
Funkcje na liczbach naturalnych: 1, 2, 3, 4, ... n, ... , jeśli są efektywne (dają wynik w skończonej liczbie operacji), utożsamiane są często z algorytmami.

Rekurencja jest kluczowa dla algorytmów. Na podstawie funkcji h definiowana jest nowa funkcja f:

- f(1) := c; stała c,
- f(n+1) := h(f(n), n+1)).

Np. suma kolejnych liczb naturalnych do n, oznaczona przez S(n)

- S(1) = 1,
- S(n+1) = S(n) + n+1.

Fryderyk Gauss (jako uczeń) podał prostszy wzór: S(n) := n(n+1)/2. Jak na to wpadł?

Ciąg Fibonacciego Fib(n) definiowany rekurencyjnie:

- Fib(1) := 1
- Fib(2) := 1
- dla n > 2, Fib(n) := Fib(n − 1) + Fib(n − 2)
- początkowe elementy tego ciągu: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Innym przykładem jest wyliczanie największego wspólnego dzielnika za pomocą algorytmu Euklidesa:

1. $\gcd(0, n) = n$
2. $\gcd(k, n) = \gcd(n \bmod k, k)$ dla $k > 0$ ($n \bmod k$ oznacza tu resztę z dzielenia n przez k)

lub inaczej:

$$\gcd(k, n) = \begin{cases} n & \text{dla } k = 0; \\ \gcd(n \bmod k, k) & \text{dla } k > 0. \end{cases}$$

Algorytm może być rozumiany jako funkcja f: T^{in} -> T^{out}.
Przy czym:

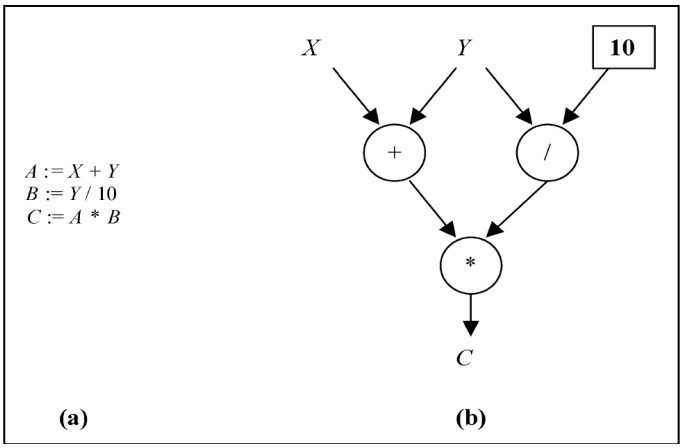
- T^{in} to typ danych wejściowych – input;
- T^{out} to typ danych wyjściowych – output.

Można również interpretować relację jako funkcję R: T_n -> Boolean = {true; false}.
Np. dla typu liczb naturalnych są to relacje:

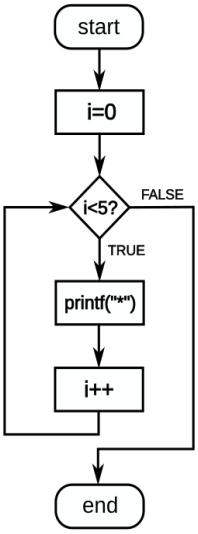
- równości =,
- większości > ,
- oraz mniejszości < .

Algorytm to funkcja f: T^{in} -> T^{out} skonstruowana z takich pierwotnych funkcji i relacji.
A jak jest w językach programowania i w komputerze? Podobnie, ale (w architekturze von Neumanna) bardziej skomplikowanie.
Wykonanie programu może się zapętlić – konieczna jest zewnętrzna interwencja, żeby zakończyć wykonanie.

Prosty graf *dataflow* (graf przepływu danych) jako algorytm.



Bardziej skomplikowany diagram przepływu (ang. *flow diagram*) z użyciem pętli i warunku wyjścia z tej pętli:



Schemat blokowy (diagram przepływu) dla pętli `for` w języku C:

- `for(i=0;i<5;i++) printf("*");`

Wykonanie tej pętli spowoduje wydrukowanie pięciu gwiazdek.
By Giacomo Alessandroni - File:For-loop-diagram.svg, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=128811834.

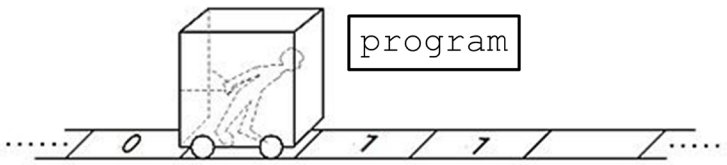
1.3 Teoretyczne modele komputera (automatycznych obliczeń)

Na marginesie: będziemy często używać oryginalnej (powszechnie używanej w informatyce) terminologii angielskiej, nie siląc się na spolszczenia tam, gdzie jest to zbędne.

Takich teoretycznych modeli powstało wiele, ale najważniejsze to maszyna Turinga-Posta, maszyna Turinga oraz maszyna rejestrowa, przedstawione niżej.

1.3.1 Maszyna Turinga-Posta (1936)

Wyjaśnienie działania jest w poniższym rysunku.



Program

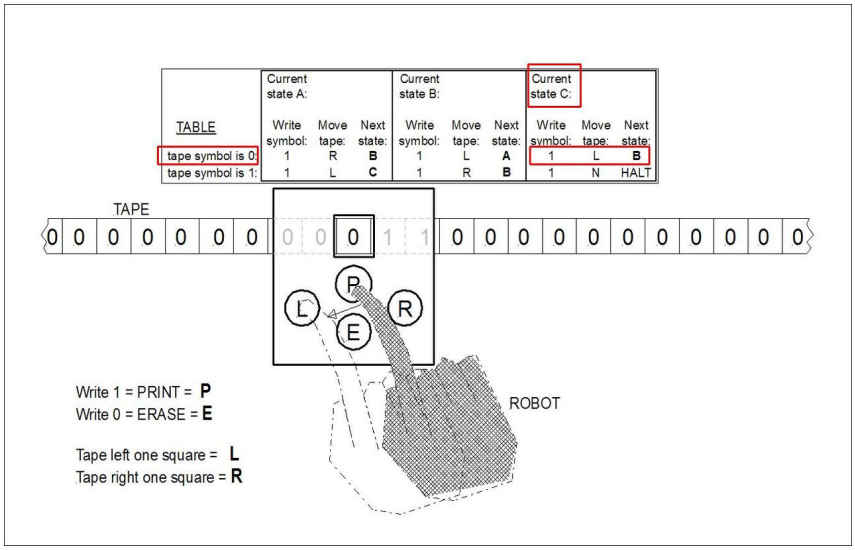
```
1.  ins_1
2.  ins_2
...
n.  ins_n
```

Instrukcje:

```
1.  write 0
2.  write 1
3.  move left
4.  move right
5.  if scanning 0 then goto instruction i
6.  if scanning 1 then goto instruction j
```

1.3.2 Maszyna Turinga (1936)

Wyjaśnienie działania jest również w tym rysunku, ale też dalej.



1.3.3 Opis i porównanie tych dwóch modeli

Powstały w latach 30. XX wieku.

Alan Turing (1912-1954)

Maszyna Turinga składa się z nieskończonej (!) taśmy podzielonej na komórki oraz głowicy, która czyta z pojedynczej komórki symbol tam zawarty (tj. 0, 1 lub blank) oraz zapisuje w komórce symbole 0 lub 1. Głowicą steruje program (funkcja przejścia), który na pod-

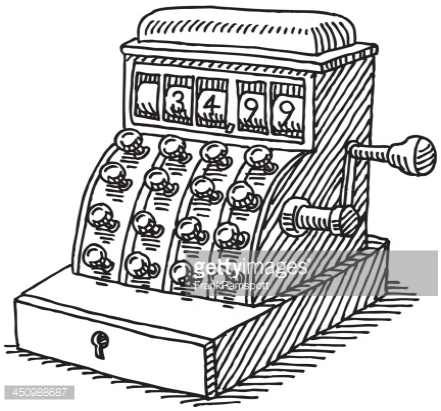
stawie stanu maszyny (zbiór stanów jest skończony) oraz bieżącego odczytu głowicy wyznacza, jaki symbol ma być zapisany na taśmie, a także czy głowica ma zostać przesunięta w prawo, lewo, czy pozostać na miejscu. Osiągnięcie specjalnego stanu Halt oznacza zatrzymanie działania i tym samym zakończenie obliczenia.

Początkowo na taśmie jest skończony ciąg zero-jedynkowy jako *input*, po zakończeniu obliczeń wynikiem (*output*) jest ciąg zero-jedynkowy na taśmie.

Jeśli ktoś uważa, że maszyny Turinga to dobry model obliczalności, to niech spróbuje skonstruować taką maszynę dla prostej funkcji dodającej kolejne liczby naturalne od 1 do n.

Maszyna Turinga-Posta to jakby połączenie oryginalnej maszyny Turinga z maszyną rejestrową przedstawioną niżej.

1.3.4 Maszyna rejestrowa (*register machine*) 1950/1960



Składa się z nieograniczonej liczby rejestrów **R_n**, przy czym n jest liczbą naturalną. Każdy rejestr **R_n** zawiera dowolną liczbę naturalną **r_n** (a więc jest nieograniczony pamięciowo).

Program **P** zawiera skończoną liczbę instrukcji na bazie czterech podstawowych instrukcji:

- zero **Z(n)** (zamień **r_n** na 0),
- następnik **S(n)** // zamień **r_n** na **r_n + 1**,
- transfer **T(m; n)** // zamień **r_n** na **r_m**,
- skok **J(m; n; q)** // jeśli **r_m = r_n**, to przejdź do instrukcji o numerze **q** w **P**, w przeciwnym przypadku przejdź do następnej instrukcji w **P**.

Program realizujący funkcję **f** (zliczającą kolejne liczby naturalne) jest prosty.
Załóżmy, że skonstruowaliśmy już dodatkową instrukcję **plus(m,n,k)** do dodawania zawartości dwóch rejestrów (**r_m**, **r_n**) a wynik wpisywany jest do rejestru **r_k**.
Początkowo w rejestrze **r_1** jest dana wejściowa, czyli **n**.
W rejestrach **r_2**, **r_3**, oraz **r_4** jest **0**.
Program **P**:

1. S(2)
2. S(4)
3. plus(3,2,3)
4. J(1;2;6)
5. **J(4;2;1)** // warunek skoku zawsze spełniony
6. STOP // wynik jest w **r_3**

1.3.5 Podejście formalne: arytmetyka Peano

Obliczenia to operacje wykonywane na liczbach, a dokładniej sekwencje (warunkowe) takich operacji. Algorytmy (programy) składają się z takich operacji.

Co to są operacje na liczbach? Czym są liczby?

Liczby naturalne to abstrakcja liczenia. Typ liczb naturalnych ma swój operator, jest to pierwotna operacja następnika, obiekt pierwotny, czyli 1, oraz operację pierwotną poprzednika i pierwotne relacje równości, większości i mniejszości.
Jeśli dodamy jeszcze parę aksjomatów, w tym schemat pierwotnej rekursji, to (plus logika predykatów) mamy arytmetykę Peano pierwszego rzędu jako formalną teorię.

Prosty przykład
Funkcja **f: N → N**, symbol **N** oznacza liczby naturalne.
f(n) = 1+2+ ... +(n-1)+n // to nie jest ani definicja, ani konstrukcja.
f(n) = f(n-1)+n oraz **f(1) =1** // to jest definicja rekurencyjna.
Obliczenie **f(5) =**

f(4) +5 =
f(3) + 4 +5 =
f(2) +3 + 4 +5 =
f(1) +2 +3 + 4 +5 =
1 + 2 + 3 + 4 +5 = 15

dla uproszczenia – dodawanie jest pierwotne.
Czyli dla dowolnego **n**: **f(n) = n(n+1)/2**.
Ale to już jest twierdzenie w formalnej teorii, jaką jest arytmetyka Peano. Dowód tego twierdzenia można przeprowadzić poprzez tzw. indukcję matematyczną, będącą jednym z aksjomatów tej teorii.

Funkcje pierwotnie rekurencyjne

W arytmetyce Peano występuje schemat rekursji do definicji nowych funkcji (z liczb naturalnych w liczby naturalne) na podstawie poprzednio zdefiniowanych oraz postulowanych (jako pierwotne): funkcji następnika, funkcji stałych, projekcji. Jest jeszcze kompozycja funkcji.

Schemat pierwotnej rekursji. Jeśli funkcje: **h** (o k argumentach) oraz **g** (o k+2 argumentach) są już zdefiniowane, to nowa funkcja **f** (o k+1 argumentach) jest definiowana poprzez równości:

$x = (x_1, \dots , x_k)$
$f(1, x) = h(x)$
$f(n+1, x) = g(n, f(n, x), x)$

Obliczanie wartości funkcji **f** dla zadanego argumentu **n** sprowadza się do rozpisania według równań rekurencyjnych, schodząc z **n** co **1** aż do **1**.

Funkcje rekurencyjne

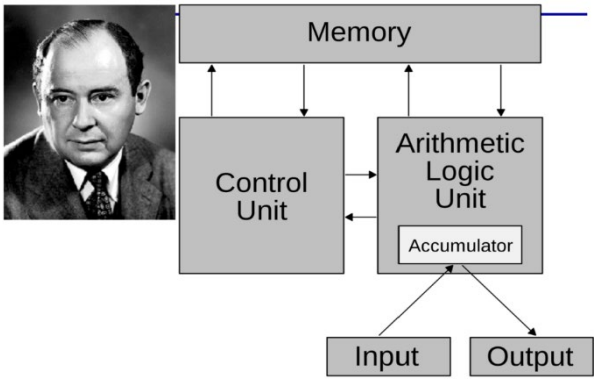
Funkcje obliczalne, według tezy Church-Turing, mogą być częściowe, tj. nie dla wszystkich argumentów są one określone.

Funkcje rekurencyjne i ich definicje za pomocą równości

Pierwotna rekursja jest przykładem takiej definiującej równości.

Funkcje rekurencyjne (wg Herbrand-Gödel) są definiowane poprzez (niesprzeczne) równości i dowód (w arytmetyce Peano), że jest to funkcja globalna, tj. określona dla wszystkich argumentów ze swojej dziedziny.

1.4 Komputery



Jedyna (jak do tej pory) architektura rzeczywistego komputera pochodzi od Johna von Neumanna. Była inspirowana przez maszynę Turinga.

Przedstawiony obok schemat tej architektury będzie dokładniej wyjaśniony w następnych rozdziałach. Z grubsza polega ona na tym, że zarówno program, jak i dane są w pamięci (Memory). Kolejne instrukcje i dane do tych instrukcji są pobierane z pamięci do odpowiednich rejestrów w CPU (Control Unit i Arithmetic Logic Unit – ALU). Tam te operacje są wykonywane, zaś wynik jest zapisywany w jednym z rejestrów, a następnie wysyłany do pamięci. Dane mogą być również pobierane z zewnątrz (*input*) do rejestru (Accumulator) w ALU, oraz wyprowadzane na zewnątrz (*output*) z tegoż rejestru.

Programowanie w komputerze von Neumnna. Prosty przykład.

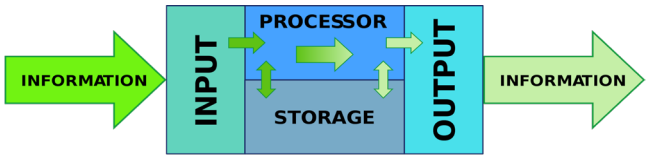
```
begin
        int n, i, x;
        input(n);
        i:=1;
        x:=0;
        while (n > i) do ( i++; x:= x + i);
        output (x);
end
```

W asemblerze (kodzie maszynowym procesora) pętla "**while**" jest trochę bardziej złożona. Ale typ **int** (w realizacji, tj. kompilacji) to skończona i ograniczona reprezentacja liczb w postaci ciągu bitów.

Zmienna to miejsca w pamięci adresowane jej nazwą. W asemblerze nazwa zmiennej to jej adres; są tam instrukcje warunkowego skoku (w zależności od stanu flag). W zasadzie kod jest bardzo podobny (powiedzmy, że izomorficzny), tylko dłuższy. Bo przecież każdy program jest kompilowany do kodu maszynowego.

1.5 Czy możliwe są inne architektury?

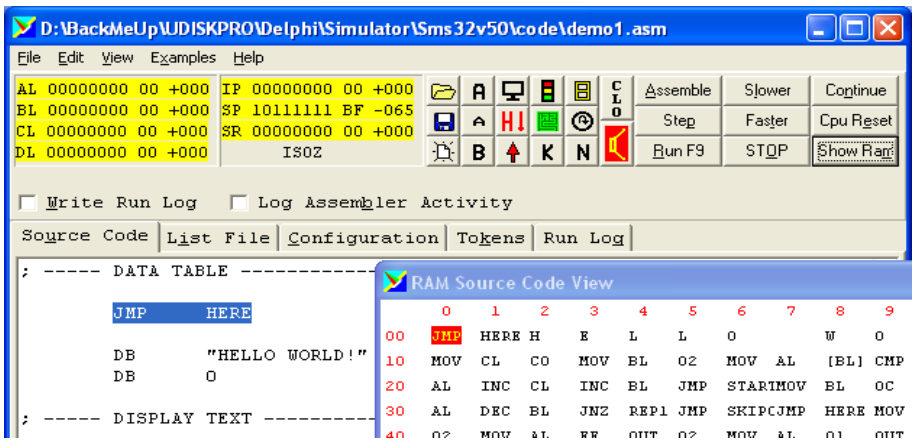
Abstrakcyjny model komputera opiera się na przetwarzaniu informacji. Informacja to sygnał analogowy albo cyfrowy.



Umysł człowieka nie jest komputerem von Neumanna.
Czy zostały skonstruowane do tej pory istotnie różne (non-von Neumann) komputery? Odpowiedź to definitywne **NIE!**

1.6 Podsumowanie

Główny cel tej książki to przybliżyć tę architekturę komputera. Osobny cel – to pokazać na przykładach, jak można programować na prostym modelu komputera opartego na tej architekturze, pochodzącej od von Neumanna. Czyli nie tylko poznać, ale i programować w prostym asemblerze na prostej maszynie, tj. symulatorze **Microprocessor Simulator V5.0**, dostępnym pod **URL-em** nbest.co.uk/Softwareforeducation/sms32v50/index.php.



Temu jest poświęcony 5 i przedostatni, 13 rozdział tej książki, w którym zamieszczono obszerne materiały do prowadzenia laboratoriów wraz z ciekawymi zadaniami programistycznymi.

Rozdział 2. Jak działa komputer?

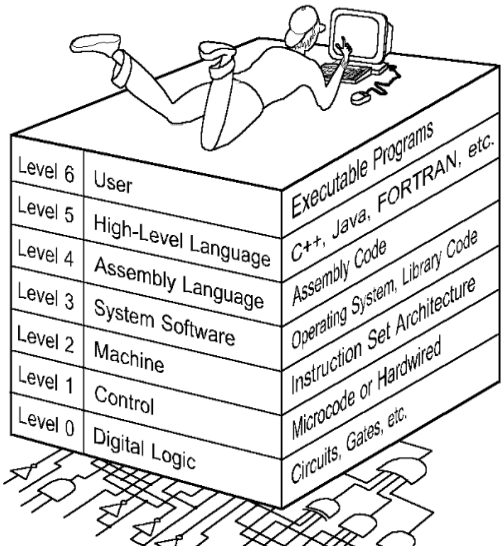
Bardzo prosto, ale tylko w swojej podstawowej, pierwotnej wersji, opartej na oryginalnej architekturze von Neumanna. Natomiast współczesne komputery są bardzo skomplikowane, chociaż podstawowa zasada, na jakiej działają, jest bardzo prosta.

2.1 Pierwsze proste podejście

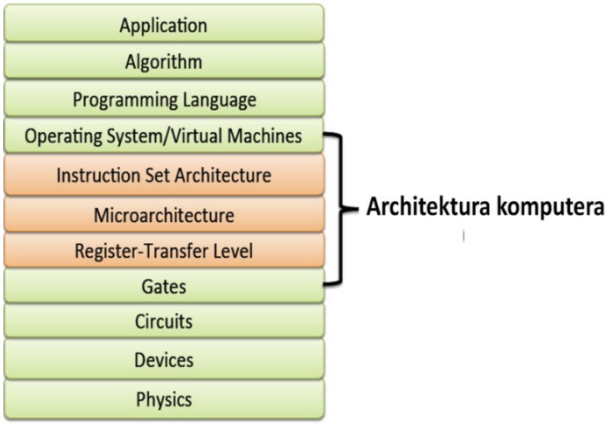
Częściowo zostały wykorzystane publicznie dostępne slajdy z wykładu:
http://www.cs.princeton.edu/courses/archive/spr05/cos111/index.php .

2.1.1 Z czego składa się komputer? Podejście abstrakcyjne

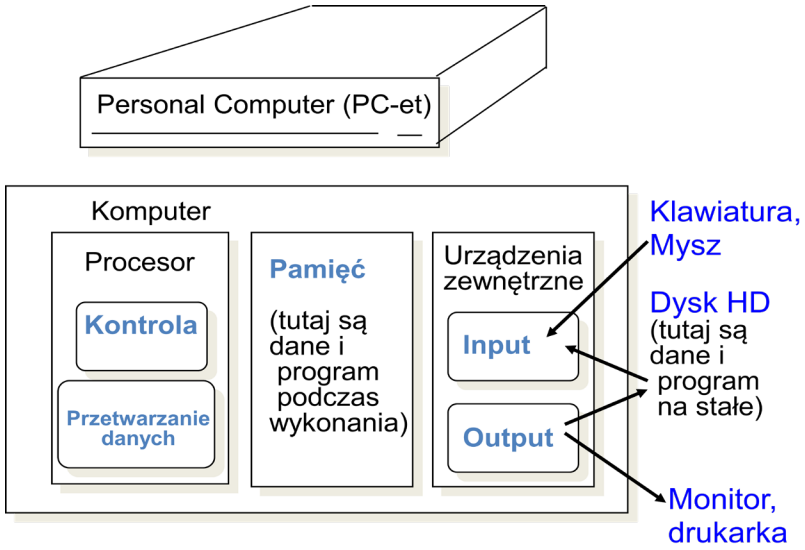
Obraz wart jest 1000 słów (przysłowie chińskie). Poniższe obrazki są tego przykładem.



Architektura to interfejs pomiędzy hardwarem (sprzętem) a softwarem (systemem operacyjnym i oprogramowaniem).



2.1.2 Pięć podstawowych komponentów komputera

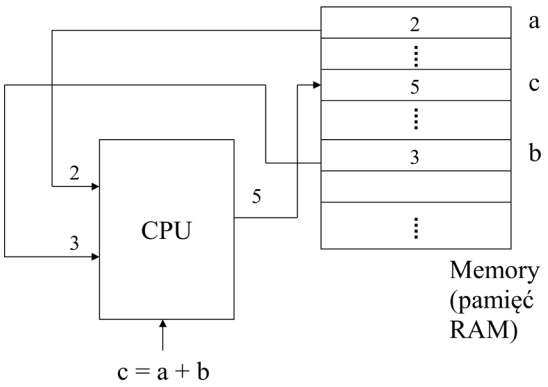


2.1.3 Przykład prostego programu

```
read (a)      ; wczytaj pierwsza wartość i zapisz na zmienną a
read (b)      ; wczytaj drugą wartość i zapisz na zmienną b
c = a + b     ; dodaj a do b a wynik zapisz na zmiennej c
read (d)      ; wczytaj trzecią wartość i zapisz na zmienną d
e = d * c     ; pomnóż c przez d; rezultat zapisz w zmiennej e
print (e)     ; wyświetl wartość zmiennej e na ekran
```

- Zmienna to konkretny obszar w pamięci komputera
- Komputer wykonuje kolejno te *instrukcje*

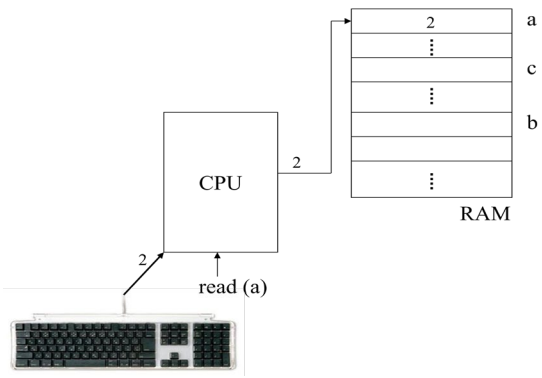
Wykonanie instrukcji `c = a + b;` przez CPU



c = a + b

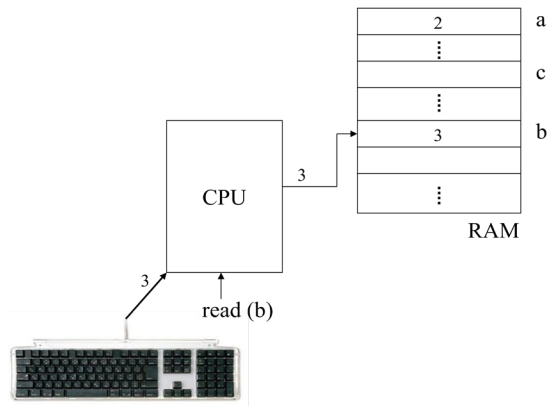
Zawartości komórek pamięci RAM o adresach `a` i `b` są kopiowane do rejestrów CPU. Wykonywana jest operacja dodawania przez CPU, a wynik jest zapisany w RAM pod adresem `c`. Jest to całkiem złożona instrukcja, a jej szczegółowe wykonanie poznamy później.

A teraz trochę bardziej złożony program.



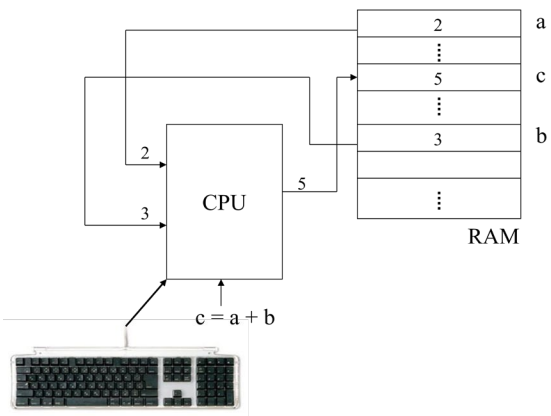

```
read (a)
read (b)
c = a + b
print (c)
```

Wczytywana jest liczba z klawiatury do komórki pamięci RAM o adresie `a`.



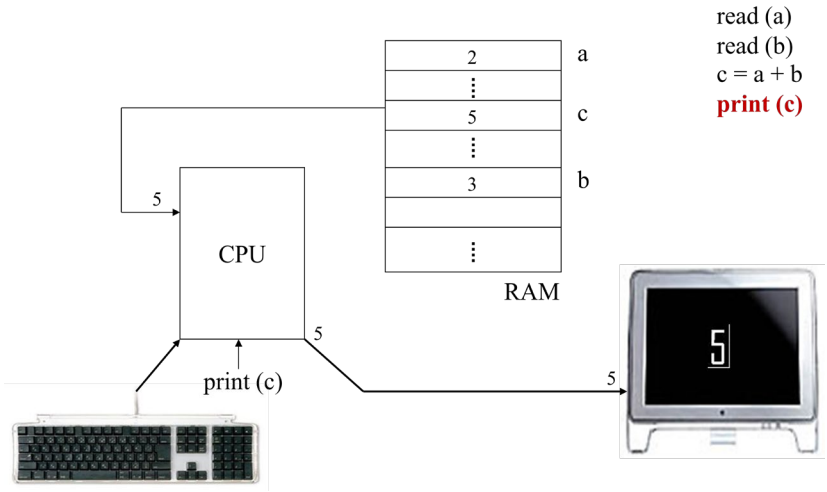

```
read (a)
read (b)
c = a + b
print (c)
```

Wczytywana jest następna liczba z klawiatury do pamięci RAM o adresie `b`.




```
read (a)
read (b)
c = a + b
print (c)
```

Wykonywane jest dodawanie opisane poprzednio. Wynik jest zapisany w komórce o adresie `c` w RAM.



Pobierana jest wartość komórki RAM o adresie `c` i wyświetlana poprzez port wyjściowy (*output*) na monitorze.

Co dla komputera znaczą "`a`", "`b`", "`4`" etc.?

"`a`" oraz "`b`" to adresy komórek w pamięci komputera,

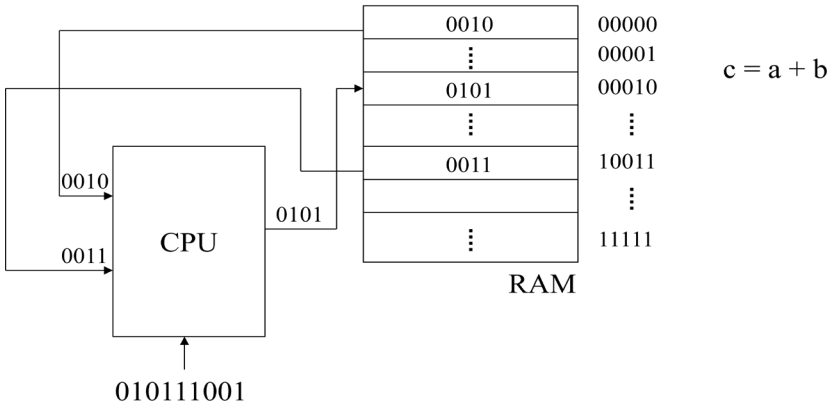
"`4`" to liczba cztery.

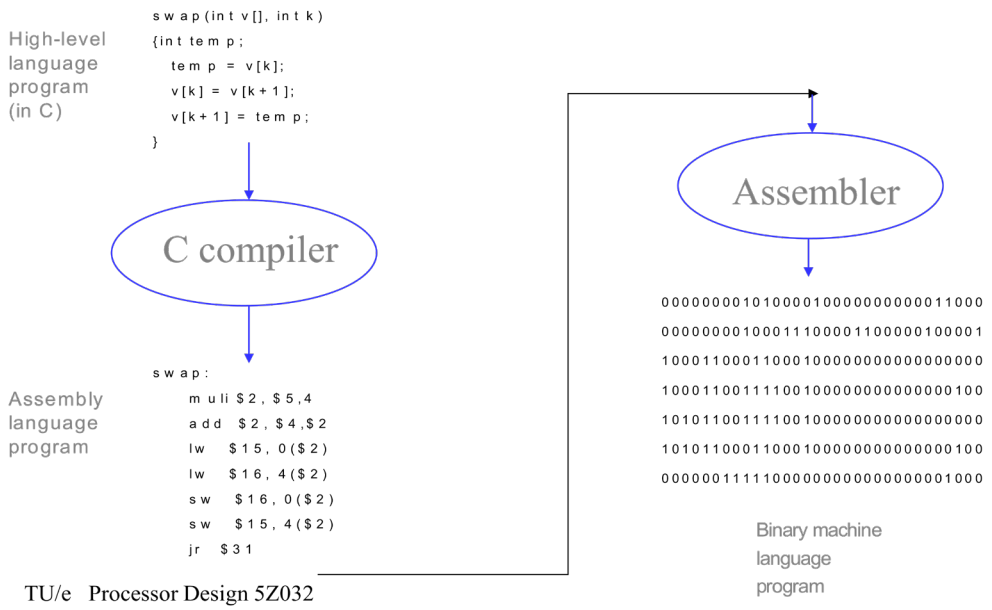
Jak komputer rozumie instrukcje (rozkazy), np. "`read(a)`"?

Jak i gdzie jest zapisany program, tj. sekwencja rozkazów?

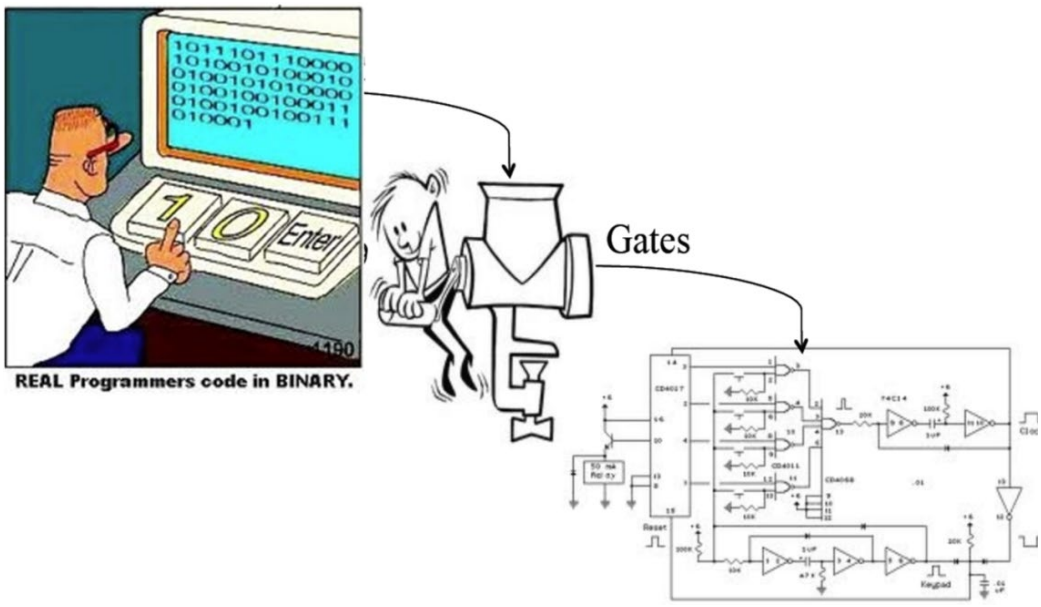
Jak komputer pobiera i wykonuje kolejne instrukcje z programu?

ODPOWIEDŹ: wszystko to jest zapisane jako bity: zero – **0** oraz jeden – **1** .
Dokładniej: jako ciągi bitów. Komputer „żyje" w świecie bitów.

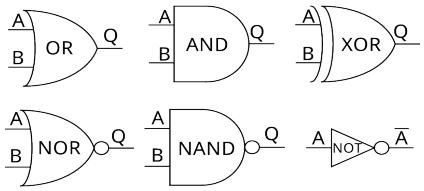




Program z binarnego języka maszynowego należy skompilować na bramki (*gates*). Współczesny procesor zawiera > 100 mln bramek. Każda bramka to kilka tranzystorów.



2.1.4 Bramki



OR	A=0	A=1
B=0	0	1
B=1	1	1

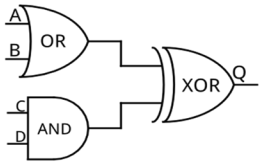
AND	A=0	A=1
B=0	0	0
B=1	0	1

XOR	A=0	A=1
B=0	0	1
B=1	1	0

NOR	A=0	A=1
B=0	1	0
B=1	0	0

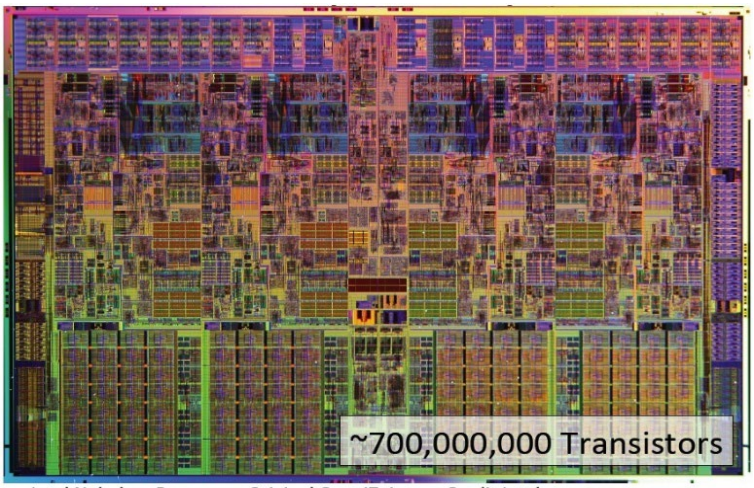
NAND	A=0	A=1
B=0	1	1
B=1	1	0

NOT A	$\overline{A}$
0	1
1	0



Źródłao https://learn.sparkfun.com/tutorials/digital-logic/combinational-logic

CD \ AB	00	01	10	11
00	0	1	1	1
01	0	1	1	1
10	0	1	1	1
11	1	0	0	0



Intel Nehalem Processor, Original Core i7, Image Credit Intel:
http://download.intel.com/pressroom/kits/corei7/images/Nehalem_Die_Shot_3.jpg

Każda bramka to od 2 do 4 tranzystorów. Tak więc współczesne procesory składają się z setek milionów tranzystorów.

2.1.5 Praca komputera: taktowanie

Jeden **takt**, to jeden rozkaz wykonany przez CPU. **Takty** są wyznaczone przez **zegar** (ang. *clock*). Stan CPU jest zapisywany w specjalnych rejestrach.

Procesor (tj. CPU) ma wiele wejść różnych typów:

- do pobierania / zapisywania danych (np. 2, 3 etc.),
- pobierania kolejnych instrukcji (np. read a; c=a+b etc.),
- dla zegara do wyznaczania kolejnych taktów.

2.1.6 Komputer: kompilacja i wykonywanie programów

Programy są pisane (przez np. studentów) w języku (np. prosty asembler) dla nich zrozumiałym. Muszą być przetłumaczone (skompilowane) do języka maszynowego, do rozkazów w postaci bajtów, tzw. *binary code.*

System operacyjny wykonuje binarny kod, zarządzając hardwarem. W tym samym czasie może być wykonywanych wiele programów (aplikacji, np. przeglądarka www, email, MS Word, MP3 Player) chociaż CPU (procesor) wykonuje rozkazy sekwencyjnie, tj. jeden rozkaz w jednym takcie.

2.1.7 Podsumowanie tego intuicyjnego wstępu

Jeśli porównamy komputer do mózgu człowieka, to:

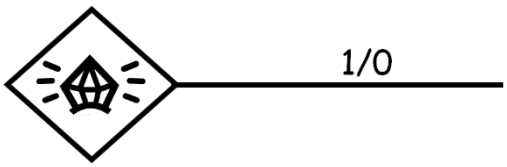
- Processor to mózg.
- Memory (RAM) to pamięć w mózgu (nikt nie wie, jak to działa w mózgu!).
- Disk HD to trwała zewnętrzna pamięć, np. notatki.
- I/O to komunikacja (poprzez zmysły) i wykonywanie czynności, np. głos, dotyk, chodzenie.
- Software to: sposoby, metody, algorytmy, planowanie, czyli ogólnie jest to myślenie.

Rozdział 3. Konkretnie

Poszczególne części komputera zostaną omówione bardziej szczegółowo.

3.1 Zegar

Komputer (CPU) pracuje krok po kroku, według taktowania zegara. Pokażemy to na przykładzie. Zegar jest zbudowany na krysztale kwarcu, który pobudzony, wysyła impulsy: interpretowane jako zera i jedynki na przemian. Cykl pracy zegara: z 0 do 1, a następnie z 1 do 0 itd.



3.2 Pamięć

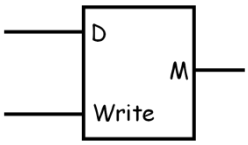
Wielkie tablice komórek (słowa kilkubajtowe) to RAM (skrót od Random Access Memory). W RAM jest przechowywana informacja (dane) podczas działania komputera.
Dane traktowane są jako wartości zmiennych (komórek w pamięci RAM). Zmienne to adresy tych komórek.
Także program (sekwencja instrukcji z instrukcjami kontrolnymi, tj., skokami), który jest wykonywany, jest przechowywany w RAM.

3.3 Program to dane w pamięci

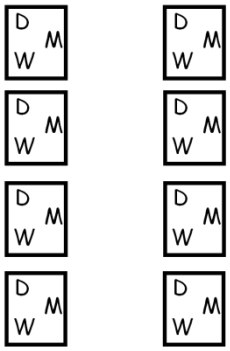
Jest to jedna z najbardziej istotnych cech architektury von Neumanna. Instrukcje oraz dane, które są przetwarzane przez te instrukcje, są w tej samej pamięci RAM. Proste a genialne w swojej prostocie: tak jest we współczesnych komputerach. Jest to podstawa dla współczesnych metod obliczeniowych.

Programy mogą być zmieniane podczas ich wykonywania. Mogą być dodawane lub/i usuwane instrukcje. Wielkim odkryciom/wynalazkom towarzyszy zawsze wielkie ryzyko. Tutaj tym ryzykiem są Computer Viruses! Więcej będzie później.

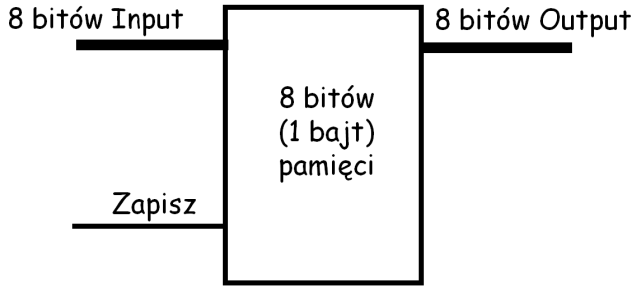
3.4 Co to jest RAM i jak to działa?



pojedynczy bit w pamięci RAM

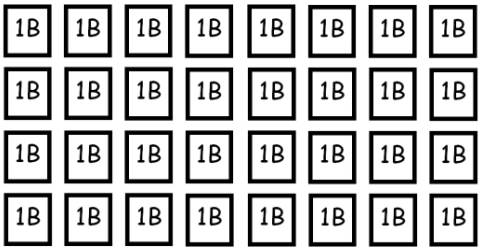


8 bitów (b) = 1 bajt (B)



Połącz 8 bitów razem w bajt

RAM składa się z bajtów. Bajt pamięci (8-bitowy rejestr).



Potrzebujemy ogromnych pamięci (obecnie gigabajtowych – GB) składających się z 1-bajtowych komórek.

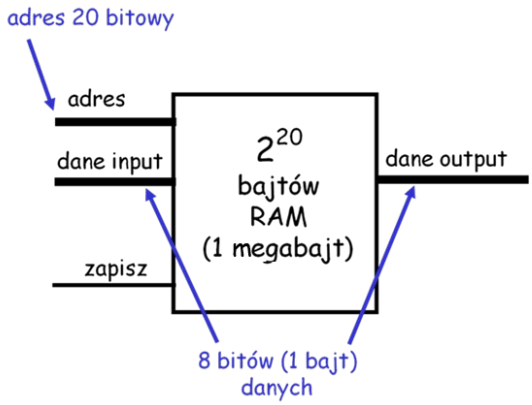
Problem: Jak wyznaczać komórkę pamięci do czytania / zapisywania ?
Rozwiązanie: przypisz każdej komórce unikalny adres (liczbę binarną o ustalonej długości). W ten sposób ponumerujemy wszystkie komórki.

Wtedy możemy zapytać RAM:

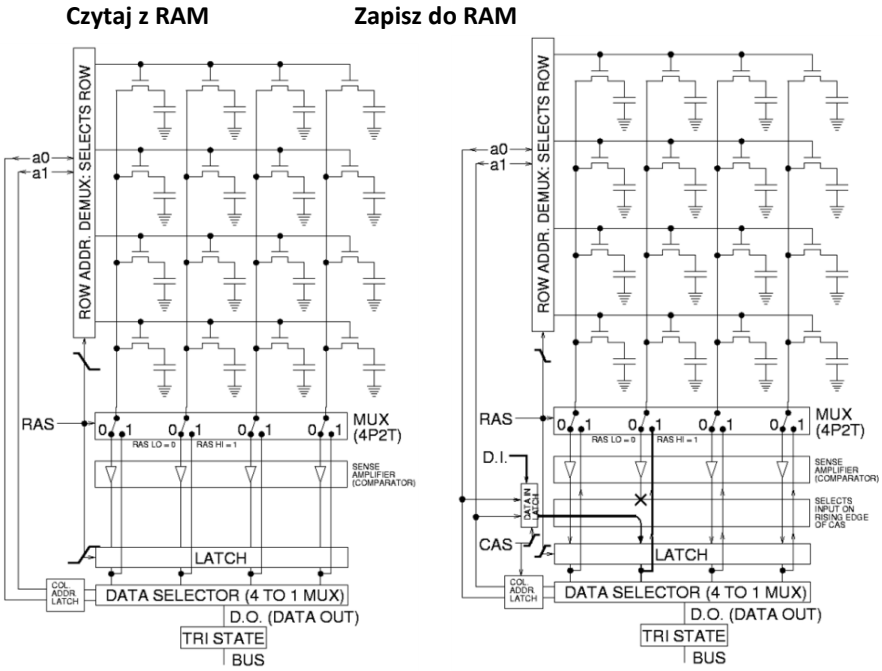
- Jaka jest zawartość komórki pamięci o adresie 011010101010?

lub kazać RAM:

- Wpisz “11101101” do komórki pamięci o adresie 011010101010.



A tak to jest realizowane na bramkach w tranzystorach:

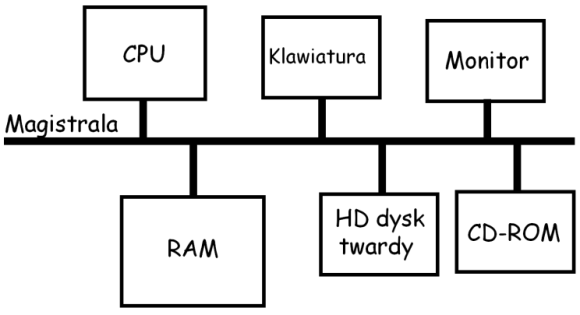


Zazwyczaj jak mówimy o pamięci komputera, to mamy na myśli RAM; ale jest jeszcze HD – Hard Drive. Po wyłączeniu zasilania dane w RAM giną, ale pozostają, jeśli są zapisane w HD.

Program jest zapisany w pamięci RAM, co znaczy, że sekwencja instrukcji jest kodowana jako bajty w kolejnych komórkach pamięci RAM, zaczynając od ustalonego adresu. Dane (wartości zmiennych) są zapisywane również w RAM, ale niekoniecznie w kolejnych komórkach (zależy to od systemu operacyjnego).
Instrukcje oraz dane są dostępne w RAM poprzez swoje adresy.

3.5 Architektura procesora

Przypominamy: CPU to skrót od Central Processing Unit.



Magistrala lub szyna (ang. *bus*), to prosty i szybki sposób komunikacji, tj. przesyłania bajtów pomiędzy urządzeniami. Taka „autostrada" do przesyłania informacji, a w zasadzie „pojemnik" dostępny dla wielu urządzeń. Działa to w następujący sposób:

Załóżmy, że CPU chce sprawdzić, czy przypadkiem użytkownik nie wcisnął jakiegoś klawisza na klawiaturze.

CPU wstawia na magistralę zapytanie "Klawiaturo, czy użytkownik coś nacisnął?".

Każde urządzenie bez przerwy sprawdza, co jest na magistrali i czy to jego dotyczy.

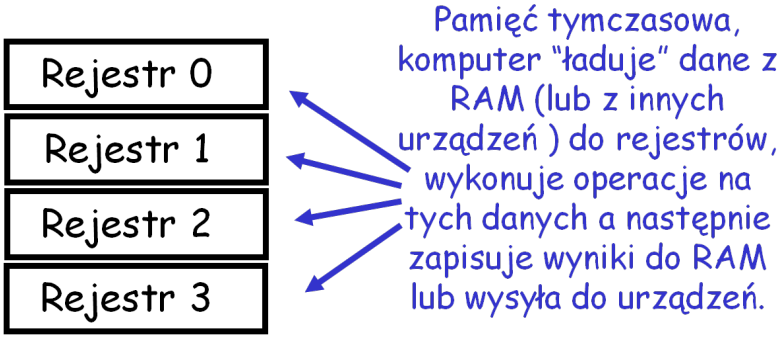
Klawiatura czyta, co jest aktualnie na magistrali, i zauważa, że to dotyczy jej. Pozostałe urządzenia ignorują tę wiadomość, bo ich nie dotyczy.

Klawiatura wpisuje wiadomość na magistralę "CPU: Tak, użytkownik nacisnął klawisz z literą 'a'".

CPU czyta, co jest na magistrali, i w ten sposób dostaje odpowiedź od klawiatury.

Takie rozwiązanie komunikacji (poprzez współdzieloną magistralę) pomiędzy CPU a pozostałymi urządzeniami jest efektywne. Czy można inaczej? Co to są przerwania sprzętowe i ich numery? O tym będzie później.

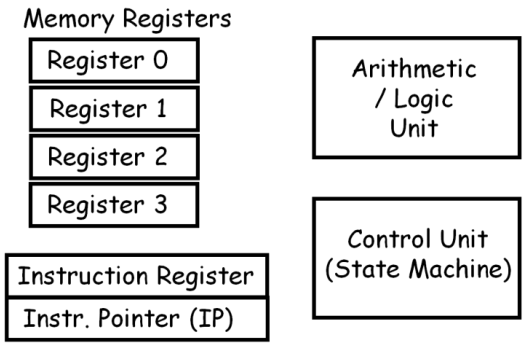
CPU jest „mózgiem" komputera. To tutaj wykonywane są kolejne instrukcje programu. Zobaczmy, co jest w środku.



Z poprzedniego przykładu:

- „czytaj (tj. kopiuj) wartość zmiennej `a` z pamięci do odpowiedniego rejestru;
- czytaj wartość zmiennej `b` do innego rejestru;
- wykonaj `c = a+b`;
- wynik zapisz w zmiennej `c`".

Czytanie odbywa się do rejestrów. Operacja dodawania jest wykonana na rejestrach; wynik jest też zapisany w rejestrze.



Arithmetic / Logic Unit (ALU) wykonuje podstawowe operacje (arytmetyczno-logiczne?) na bajtach (z rejestrów) i zapisuje wynik również w rejestrach.

W **Instruction Registe**r jest aktualna instrukcja programu do wykonania.

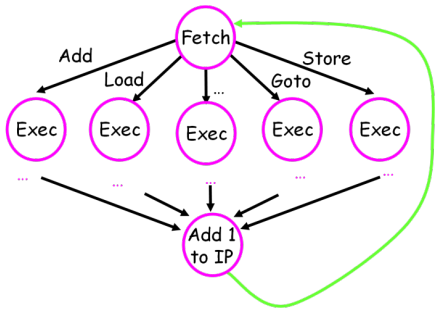
W **Instruction Pointer (IP)** jest przechowywany adres (w RAM) aktualnej instrukcji wykonywanego programu.

Control Unit (jednostka kontrolna) steruje działaniem procesora (CPU).
Działanie Control Unit jest proste i składa się z następujących 3 kroków:

1) **Fetch:** pobierz z RAM instrukcje, której adres jest zapisany w rejestrze IP, i zapisz ją w Instruction Register (IR).

2) **Execute:** wykonaj aktualną instrukcję z IR; rodzajów instrukcji jest niewiele i są one proste.

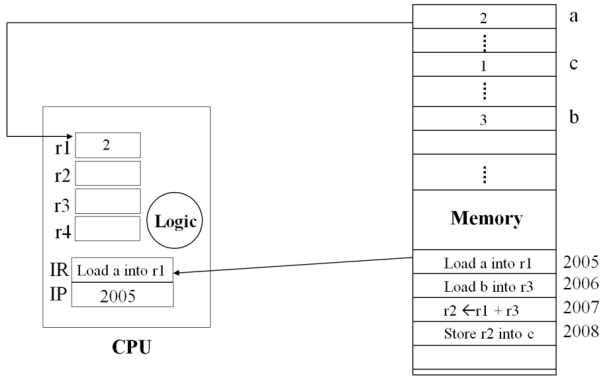
3) **Powtarzaj:** dodaj 1 do adresu w rejestrze IP (ale jeśli jest to instrukcja skoku, to może być inne IP) i przejdź do kroku 1.

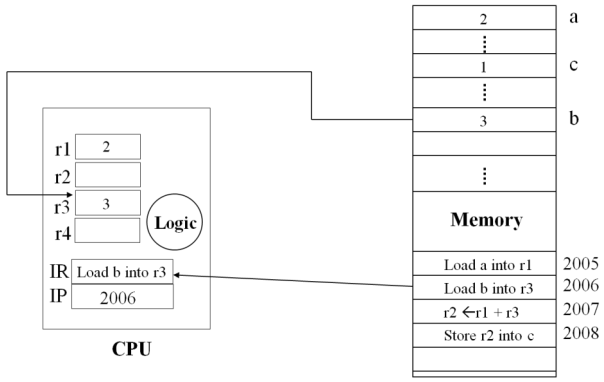


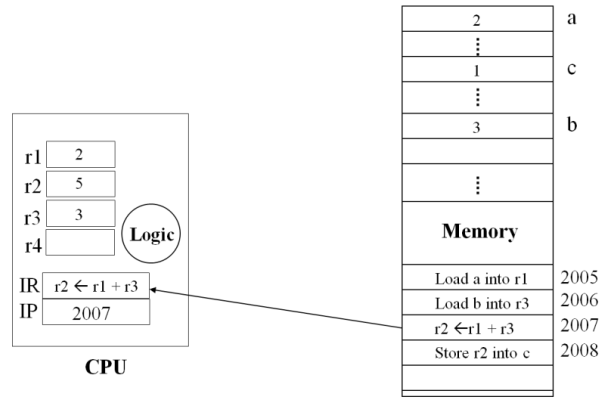
Przykład prostego programu.

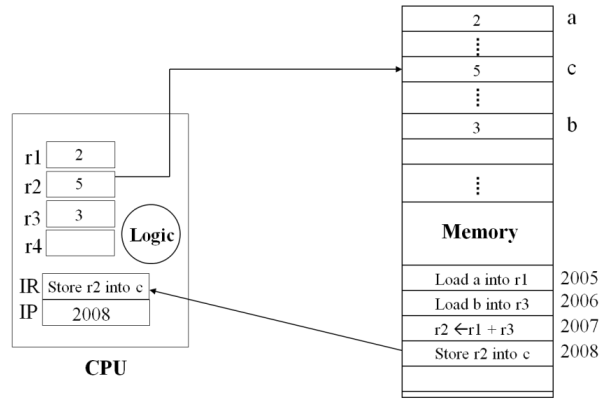
Dodaj wartości zmiennych `a` oraz `b`. Wynik zapisz na zmiennej `c`, tj. `c ← a+b`.
Sekwencja instrukcji (program):

- zapisz wartość zmiennej `a` do rejestru `r1`,
- zapisz wartość zmiennej `b` do rejestru `r3`,
- wykonaj `r2 ← r1 + r3`,
- zapisz wartość z rejestru `r2` do zmiennej `c`.


2
a
1
c
3
b
Memory
Load a into r1
Load b into r3
r2 ←r1 + r3
Store r2 into c
2005
2006
2007
2008
r1
r2
r3
r4
2
Logic
IR
Load a into r1
IP
2005
CPU

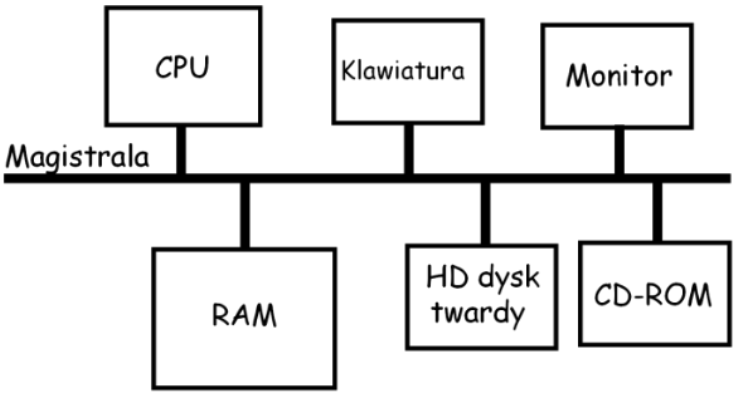

2
a
1
c
3
b
Memory
Load a into r1
Load b into r3
r2 ←r1 + r3
Store r2 into c
2005
2006
2007
2008
r1
r2
r3
r4
2
3
Logic
IR
Load b into r3
IP
2006
CPU


2
a
1
c
3
b
Memory
Load a into r1
Load b into r3
r2 ←r1 + r3
Store r2 into c
2005
2006
2007
2008
r1
r2
r3
r4
2
5
3
Logic
IR
r2 ← r1 + r3
IP
2007
CPU


2
a
5
c
3
b
Memory
Load a into r1
Load b into r3
r2 ←r1 + r3
Store r2 into c
2005
2006
2007
2008
r1
r2
r3
r4
2
5
3
Logic
IR
Store r2 into c
IP
2008
CPU

3.6 Podsumujmy

Komputer składa się z wielu elementów połączonych magistralą (szyną):



Pamięć RAM jest główną pamięcią komputera, nie licząc rejestrów, które są podręczną pamięcią procesora (CPU). W RAM jest program i są dane.

CPU pracuje w cyklu, czytając instrukcje z RAM i je wykonując, aż napotka instrukcję HALT kończącą wykonanie programu.

Ten cykl pracy procesora (CPU) jest zarządzany (dyrygowany) przez jednostkę kontrolną Control Unit.

Control Unit patrzy do komórki RAM o adresie wskazanym przez rejestr IP, czyta instrukcję z tej komórki i wstawia do Instruction Register, a następnie ją wykonuje.

Aby wykonać instrukcję, Control Unit używa ALU, RAM oraz/lub rejestrów (Registers).

Rozdział 4. Programowanie

Dokładniej o instrukcjach i programie.

4.1 Programy i instrukcje

Program składa się z sekwencji instrukcji. CPU wykonuje jedną instrukcję w każdym cyklu zegarowym. Współczesne procesory wielordzeniowe wykonują trochę więcej, ale na razie nie jest to istotne.
Poziomy programowania:

- najniższy poziom: machine language,
- pośredni poziom: assembly language,
- do tworzenia aplikacji : high-level programming language.

W językach programowania wysokiego poziomu każda instrukcja jest dekodowana na wiele instrukcji niskiego poziomu.
Assembly language specyfikuje instrukcje niskiego poziomu do postaci mnemonicznej, np. `Load r1, a.`
Prosty przykład: `c ← a + b` jest kompilowana do:

- `Load a into r1`
- `Load b into r3`
- `r2 ← r1 + r3`
- `Store r2 into c`

Takie instrukcje są następnie kompilowane do instrukcji w machine language. Tutaj instrukcje są ciągami bitów, np. 1101101000001110011.
Załóżmy, że mamy maszynę, która wykonuje takie instrukcje. Zasadnicze pytanie: jakie to są instrukcje? Jak te instrukcje mają się do sprzętu (*computer hardware*)?

4.2 Complex vs Simple Instructions

Są zasadniczo dwa zestawy instrukcji:

- Złożony z zestawu instrukcji CISC (Complex Instruction Set Computer). Jeszcze 30 lat temu prawie wszystkie procesory były oparte na CISC.
- W latach 80. XX wieku wprowadzony został zestaw RISC (Reduced Instruction Set Computer). Jest tam mniej prostszych instrukcji, łatwiejszych do zaprojektowania CPU, ale też wystarczających, żeby z nich złożyć wszystko, co potrzeba, tj. niezbędne złożone instrukcje.

Okazało się, że dla wielu ważnych aplikacji procesory oparte na RISC są bardziej wydajne niż te oparte na CISC. Tym niemniej, np. Pentium był jeszcze oparty na CISC! Powód był prosty: musiała być utrzymana kompatybilność (zgodność) ze starym oprogramowaniem, w tym również z systemami operacyjnymi.

Nowe rodzaje aplikacji (multimedia) są bardziej wydajne, jeśli są wykonywane na specjalnych pół-autonomicznych układach, np. karty graficzne.

Obecnie technologie informacyjne są zbyt złożone, żeby ograniczać się tylko do RISC *versus* CISC.

4.3 Typowe instrukcje w asemblerze

1) "Load" – skopiuj zawartość komórki RAM i wstaw do jednego z rejestrów.
2) "Load Direct" – wstaw słowo (np. bajt) do jednego z rejestrów.
3) "Store" – zapisz słowo z rejestru do pamięci RAM.
4) "Add" – dodaj zawartości dwóch rejestrów, a wynik wstaw do trzeciego rejestru.
5) "Compare" – porównaj, czy wartość w jednym rejestrze jest większa niż w drugim rejestrze. Jeśli tak, to wstaw do rejestru r0 wartość "0", jeśli nie, to wstaw wartość "1".
6) "Jump" – jeśli wartość w rejestrze r0 jest "0", to zmień wartość w rejestrze IP (Instruction Pointer) na wartość, która jest aktualnie w ustalonym rejestrze.
7) "Branch" – jeśli wartość w rejestrze r1 jest większa niż w rejestrze r2, to wstaw do rejestru IP ustaloną wartość.

Różne rodzaje procesorów posiadają różne zestawy instrukcji, np.

- Pentium family / Celeron / Xeon / AMD K6 / Cyrix … (Intel x86 family),
- PowerPC (Mac),
- DragonBall (Palm Pilot),
- StrongARM/MIPS (WinCE),
- i wiele innych (specjalizowanych i uniwersalnych)

Instrukcje takie są różnie kodowane w assembly/machine languages.

4.4 Prosty przykład CPU

Przedstawimy tutaj uproszczony CPU i odpowiadający mu machine language.

CPU posiada:

- 8 rejestrów – każdy po 16 bitów (2 bajty).

RAM:

- Można czytać i wpisywać (loads oraz stores) w blokach po 16 bitów (2 bajtów).
- Potrzebne jest $2^8 = 256$ adresów.
- Wielkość tej pamięci to 256*2 = 512 bajtów.

4.4.1 Zapisywanie w pamięci

Będziemy operować 16-bitowymi ciągami, np. 0110110010100101.

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

8	1 0 0 0
9	1 0 0 1
A	1 0 1 0
B	1 0 1 1
C	1 1 0 0
D	1 1 0 1
E	1 1 1 0
F	1 1 1 1

Trudno zapamiętać takie ciągi, więc używać będziemy heksadecymalnego (szesnastkowego) systemu do kodowania danych oraz instrukcji, w którym jest 16 cyfr.

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

np.

0110 1100 1010 0101

6 C A 5

8	1 0 0 0
9	1 0 0 1
A	1 0 1 0
B	1 0 1 1
C	1 1 0 0
D	1 1 0 1
E	1 1 1 0
F	1 1 1 1

Wszystko jest binarne, ale w notacji heksadecymalnej.

Rejestry

R0:	0000
R1:	0CA8
R2:	A9DB
R3:	0705
R4:	1011
R5:	90A0
R6:	0807
R7:	00A0

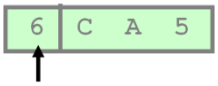
Pamięć

00:	0CA9	ABCD	0000	0000
04:	0000	0000	0000	0000
08:	0000	0000	FFFF	0000
0C:	0000	0000	0000	0000
10:	B106	B200	B001	1221
…	…			
F8:	0000	0000	0000	0000
FC:	0000	0000	FFEE	0000

W pierwszej tabeli powyżej, w kolumnie po lewej stronie są nazwy rejestrów.
W drugiej tabeli, w kolumnie po lewej stronie są adresy poszczególnych wierszy pamięci.
Zerowy adres na początku w pierwszym wierszu jest dla komórki o zawartości 0CA9.
Komórka o adresie 01 ma zawartość ABCD.
Adres 04 w drugim wierszu jest dla komórki o zawartości 0000.
Adres 10 w piątym wierszu jest dla komórki o zawartości B106 itd.

4.4.2 Kodowanie instrukcji

Maszynowe instrukcje reprezentowane są również jako ciągi 16-bitowe, czyli czterocyfrowe liczby heksadecymalne. Dzielimy te bity na grupy:

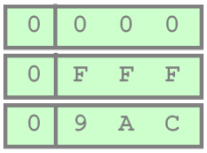


Pierwsze 4 bity są nazwane Op-Code (kody operacji) i wskazują na typ instrukcji.
Na 4 bitach można zakodować 16 typów operacji.

```
0: halt
1: add
2: subtract
3: multiply
4: bus output
5: jump
6: jump if positive
7: jump & count
8: bus input
9: load
A: store
B: load direct/addr.
C: NAND
D: AND
E: Shift Right
F: Shift Left
```

Zapoznamy się tylko z niektórymi.

Opcode 0: Halt
Instrukcja stopu, tj. zatrzymanie działania; pozostałe bity są ignorowane, np.



Dają taki sam efekt

Opcode B: Load Direct / Address
Wstawia do rejestru ustaloną wartość.

kod: 

Efekt: w rejestrze A jest ta wartość.

Opcode 1: Add
Dodaje zawartość dwóch rejestrów i wstawia wynik do trzeciego rejestru

kod:

Efekt: wartość w rejestrze A wynosi rejB + rejC.

Prosty program: dodawanie kolejnych liczb

```
10: Load R0 ← 01          (zawsze 1)
11: Load R2 ← 00          (wynik)
12: Load R1 ← 01          (licznik)

13: Add R2 ← R2 + R1      (R2=1)
14: Add R1 ← R1 + R0      (R1=2)

15: Add R2 ← R2 + R1      (R2=3)
16: Add R1 ← R1 + R0      (R1=3)

17: Add R2 ← R2 + R1      (R2=6)
18: Add R1 ← R1 + R0      (R1=4)

19: Add R2 ← R2 + R1      (R2=A)
1A: Add R1 ← R1 + R0      (R1=5)

1B: Add R2 ← R2 + R1      (R2=F)
1C: Add R1 ← R1 + R0      (R1=6)

1D: Add R2 ← R2 + R1      (R2=15)
1E: halt
```

1 + 2 + 3 + 4 + 5 + 6 = 21 = 15 (hex)

Algorytm:
Na początku ‘licznik’ R1 ustawiamy na 1.
Dodajemy ‘licznik’ R1 do ‘wynik’ R2 i zwiększamy za każdym razem ‘licznik’ R1 o 1
Sprawdzamy: Jeśli R1=6, to halt (kończymy).

Opcode 2: Subtract (odejmij)
Podobne do Add.

kod:

Efekt: rejestr A ma wartość = rejestr B - rejestr C

Instrukcja kontrolna:

Opcode 6: Jump if Positive

Jump (skocz) do komórki RAM pod adres, jeśli rejestr A > 0

kod:

Efekt: Jeśli rejestr A > 0, Go To (przejdź) do instrukcji pod adresem w RAM, tj. zmień wartość rejestru IP (Instruction Pointer) na ten adres.

Co zrobić, żeby uprościć program?
Te same instrukcje występują sześciokrotnie w naszym prostym programie dodającym liczby od 1 do 15 (hex).
A jeśli w programie jest dodawanie liczb od 1 do 15000? Jak sobie wtedy poradzić?
Rozwiązaniem są pętle (Loops). Za pomocą instrukcji jump (lub branch).
Wtedy warunkowo jest zmieniana wartość rejestru IP do wcześniejszej instrukcji w programie.

Dodawanie liczb jeszcze raz
Użyj pętli Loop z pomocą instrukcji Jump po to, żeby policzyć 1 + 2 + 3 + 4 + 5 + … + N.
Uwaga: zmniejszamy zamiast zwiększać dodawane liczby.

```
10: Load R1 ← 0006          (N = 6)
11: Load R2 ← 0000          (wynik)
12: Load R0 ← 0001          (zawsze 1)

13: Add R2 ← R2 + R1        (dodaj N do wyniku R2)
14: Sub R1 ← R1 - R0        (N = N-1)

15: Jump to 13 if (R1>0)    (Jeśli N ≠ 0, skocz do lini 13)

16: halt                    (N = 0, oraz R2 = 1+2+…+N)
```

Instrukcje do operowania na RAM:

Opcode A: Store (do pamięci)
Zapisz wartość z rejestru do RAM

kod:

Efekt: zapisz wartość z rejestru A do komórki RAM pod adres.

Opcode 9: Load (z pamięci)
skopiuj z pamięci RAM i zapisz do rejestru

kod:

Efekt: skopiuj komórkę z RAM o adresie [B+C] i wpisz do rejestru A.

4.4.3 Uwaga: wirusy!

Program, który modyfikuje (replikuje) sam siebie!
Nadpisywane (patrz niżej) są kolejno linie 15, 14, 13, 12, 11, 10 na to, co jest pod adresami 0F, 0E, 0D, 0C, 0B, 0A.

```
0A  Load R1 ← 0005            (długość kodu do replikacji -1)
0B  Load R2 ← 0010            (koniec tego kodu +1)
0C  Load R3 ← 000A            (początek tego kodu)

0D  Load R0 ← Address[R3+R1]   (wstaw instrukcję o adresie 0F do R0)
                               (w następnych krokach będą to poprzednie)
0E  Store Address[R2+R1] ← R0 (zapisz tę instrukcję pod adres 15)
                               (następne instrukcje zapisuj o 1 wyżej)
0F  If (R1>0), Jump to 0D      (jeśli w R1 nie ma zera, to skocz do 0D
              and decrease R1          i zmniejsz R1 o 1)
10  ....
11  ....
12  ....
13  ....
14  ....
15  ....
...  ____________________________________________________________________
..:  . . . .                  (dalsza część kodu)
..:  . . . .
```

```
0A  Load R1 ← 0005            (długość kodu do replikacji -1)
0B  Load R2 ← 0010            (koniec tego kodu +1)
0C  Load R3 ← 000A            (początek tego kodu)

0D  Load R0 ← Address[R3+R1]   (wstaw instrukcję o adresie 0F do R0)
                               (w następnych krokach będą to poprzednie)
0E  Store Address[R2+R1] ← R0 (zapisz tę instrukcję pod adres 15)
                               (następne instrukcje zapisuj o 1 wyżej)
0F  If (R1>0), Jump to 0D     (jeśli w R1 nie ma zera, to skocz do 0D
              and decrease R1          i zmniejsz R1 o 1)
10  ....
11  ....
12  ....
13  ....
14  ....
15  If (R1>0), Jump to 0D and decrease R1
...  ____________________________________________________________________
..:  . . . .                  (dalsza część kodu)
..:  . . . .
```

R1 = 5

```
0A  Load R1 ← 0005            (długość kodu do replikacji -1)
0B  Load R2 ← 0010            (koniec tego kodu +1)
0C  Load R3 ← 000A            (początek tego kodu)

0D  Load R0 ← Address[R3+R1]   (wstaw instrukcję o adresie 0F do R0)
                               (w następnych krokach będą to poprzednie)
0E  Store Address[R2+R1] ← R0 (zapisz tę instrukcję pod adres 15)
                               (następne instrukcje zapisuj o 1 wyżej)
0F  If (R1>0), Jump to 0D     (jeśli w R1 nie ma zera, to skocz do 0D
              and decrease R1          i zmniejsz R1 o 1)
10  ....
11  ....
12  ....
13  ....
14  Store Address[R2+R1] ← R0
15  If (R1>0), Jump to 0D and decrease R1
...  ____________________________________________________________________
..:  . . . .                  (dalsza część kodu)
..:  . . . .
```

R1 = 4

```
0A  Load R1 ← 0005            (długość kodu do replikacji -1)
0B  Load R2 ← 0010            (koniec tego kodu +1)
0C  Load R3 ← 000A            (początek tego kodu)

0D  Load R0 ← Address[R3+R1]   (wstaw instrukcję o adresie 0F do R0)
                               (w następnych krokach będą to poprzednie)
0E  Store Address[R2+R1] ← R0 (zapisz tę instrukcję pod adres 15)
                               (następne instrukcje zapisuj o 1 wyżej)
0F  If (R1>0), Jump to 0D     (jeśli w R1 nie ma zera, to skocz do 0D
              and decrease R1          i zmniejsz R1 o 1)
10  ……
11  ……
12  ……
13  Load R0 ← Address[R3+R1]
14  Store Address[R2+R1] ← R0
15  If (R1>0), Jump to 0D and decrease R1
...
..: . . . .                   (dalsza część kodu)
..: . . . .
```

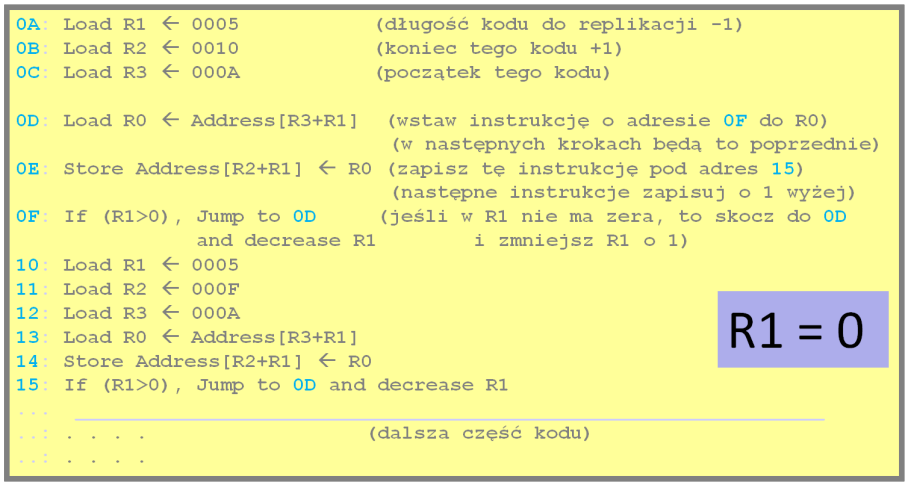
R1 = 3

```
0A  Load R1 ← 0005            (długość kodu do replikacji -1)
0B  Load R2 ← 0010            (koniec tego kodu +1)
0C  Load R3 ← 000A            (początek tego kodu)

0D  Load R0 ← Address[R3+R1]   (wstaw instrukcję o adresie 0F do R0)
                               (w następnych krokach będą to poprzednie)
0E  Store Address[R2+R1] ← R0 (zapisz tę instrukcję pod adres 15)
                               (następne instrukcje zapisuj o 1 wyżej)
0F  If (R1>0), Jump to 0D     (jeśli w R1 nie ma zera, to skocz do 0D
              and decrease R1          i zmniejsz R1 o 1)
10  ……
11  ……
12  Load R3 ← 000A
13  Load R0 ← Address[R3+R1]
14  Store Address[R2+R1] ← R0
15  If (R1>0), Jump to 0D and decrease R1
...
..: . . . .                   (dalsza część kodu)
..: . . . .
```

R1 = 2

```
0A  Load R1 ← 0005            (długość kodu do replikacji -1)
0B  Load R2 ← 0010            (koniec tego kodu +1)
0C  Load R3 ← 000A            (początek tego kodu)

0D  Load R0 ← Address[R3+R1]   (wstaw instrukcję o adresie 0F do R0)
                               (w następnych krokach będą to poprzednie)
0E  Store Address[R2+R1] ← R0 (zapisz tę instrukcję pod adres 15)
                               (następne instrukcje zapisuj o 1 wyżej)
0F  If (R1>0), Jump to 0D     (jeśli w R1 nie ma zera, to skocz do 0D
              and decrease R1          i zmniejsz R1 o 1)
10  ……
11  Load R2 ← 000F
12  Load R3 ← 000A
13  Load R0 ← Address[R3+R1]
14  Store Address[R2+R1] ← R0
15  If (R1>0), Jump to 0D and decrease R1
...
..: . . . .                   (dalsza część kodu)
..: . . . .
```

R1 = 1



4.4.4 Podsumowanie

Był to opis (bardzo zgrubny) typowego języka maszynowego i jego instrukcji:

- Halt Instruction
- Data Instructions
- Arithmetic/Logic Instructions
- Control Instructions
- Memory Instructions

Było kilka prostych przykładów programów w asemblerze. Czy teraz rozumiesz, jak działają komputery!? Współczesne programowanie zazwyczaj jest w językach wysokiego poziomu, ale w systemach wbudowanych kodowanie na niskim poziomie jest nadal ważne ze względu na efektywność kodu.

4.5 Krótka historia języków programowania i tragicznych skutków ich używania

Fortran (1954) został opracowany przez Johna Backusa w IBM, do obliczeń naukowych (głównie w fizyce), ale też przy konstrukcjach bomb atomowych i termojądrowych.

Cobol (1959) (akronim od ang. *Common Business-Oriented Language*) – wysokopoziomowy język programowania stworzony i używany do tworzenia aplikacji biznesowych. Cobol jest językiem imperatywnym, proceduralnym oraz – od 2002 roku – obiektowym. Nadal w bankowości są używane aplikacje napisane w Cobolu.

Algol (1958) był bardziej uniwersalny niż Fortran, ale nadal są tam skoki w formie „go to”.

Lisp (1958) to programowanie na listach.

A Programming Language (APL) – język programowania wysokiego poziomu, znany ze swojej zwięzłości i możliwości generowania macierzy. Opracował go Kenneth E. Iverson w połowie lat 60.

Algol + Fortran → PL/1 (1964).

Basic (1964) – w zamierzeniu dla każdego.

Simula (1967) + Algol → Smalltalk (1969) – pierwszy obiektowy język.

C – imperatywny, proceduralny język programowania stworzony na początku lat 70. przez Dennisa Ritchiego. System operacyjny Unix powstał w C.

Algol → Pascal (1971) → Modula 1,2,3.

C++ (1983) to C z obiektami. C jest nadal używany w systemach wbudowanych i kompilatorach.

Awk (1978) → Perl (1987) – Web programming language

Java (1991) – Web applets i nie tylko.

Visual Basic (1991) macros and programs. Core of Microsoft systems.

Jaki powinien być dobry język programowania?

- Łatwy do kodowania.
- Chronić przed popełnianiem błędów.
- Obsługiwać debugowanie, gdy jest to potrzebne.
- Mieć wszechstronny zestaw narzędzi.

Big number bug (błąd dużej liczby)

4 czerwca 1996 r. bezzałogowa rakieta Ariane 5 wystrzelona przez Europejską Agencję Kosmiczną (ESA) wybuchła zaledwie czterdzieści sekund po starcie z Kourou w Gujanie Francuskiej. To był pierwszy start rakiet tego typu, po 10 latach konstruowania, kosztującego 7 miliardów dolarów. Zniszczoną rakietę i jej ładunek wyceniono na 500 milionów dolarów. Komisja śledcza zbadała przyczyny wybuchu i po dwóch tygodniach opublikowała raport. Okazało się, że przyczyną awarii był błąd oprogramowania w bezwładnościowym układzie odniesienia. W szczególności 64-bitowa liczba zmiennoprzecinkowa, odnosząca się do prędkości poziomej rakiety względem platformy, została przekonwertowana na 16-bitową liczbę całkowitą ze znakiem. Liczba była większa niż 32 768, największa liczba całkowita, którą można zapisać w 16-bitowej liczbie całkowitej, więc konwersja nie powiodła się.

Pentium II bug

Błąd oprogramowania procesora zakodowany sprzętowo. Algorytm podziału wykorzystuje tabelę wyszukiwania zawierającą 1066 pozycji. Tylko 1061 wpisów jest pobieranych do PLA (zaprogramowana tablica logiczna, z której wykorzystywane są dane).
Intel musiał wycofać wszystkie sprzedane procesory. Koszt ogromny.

Kropka zamiast przecinka

NASA Mariner 1, Venus probe (1962).
Miał to być pierwszy amerykański statek kosmiczny, który by odwiedził inną planetę. Podczas startu, 22 lipca 1962 r., po czterech minutach zachowywał się nieregularnie.

Został zniszczony poprzez wewnętrzną procedurę. Przyczyna: kropka zamiast przecinka w pętli DO w kodzie FORTRAN-owym.

Błąd kontroli przepływu
AT&T usługa sieciowa dalekiego zasięgu nie działała przez dziewięć godzin (zła instrukcja BREAK w kodzie C). 15 stycznia 1990 r. 70 milionów ze 138 milionów klientów w USA straciło usługi. Koszt AT&T wynosił od 75 do 100 milionów USD (plus utrata reputacji).

Błąd zarządzania strukturą danych
Przepełnienie bufora poczty e-mail (1998). Kilka serwerów e-mail (SMTP) było wrażliwych na „błąd przepełnienia bufora", tj. gdy odbierane są wyjątkowo długie adresy e-mail, to te serwery pozwalały na przepełnienie buforów, powodując awarię aplikacji. Hakerzy mogli wykorzystać ten błąd, aby uruchomić złośliwą aplikację.

Meltdown (podatność na zagrożenia)


Meltdown to luka sprzętowa wpływająca na mikroprocesory Intel x86, procesory IBM POWER i niektóre mikroprocesory oparte na ARM. Pozwala to nieuczciwemu procesowi na odczyt całej pamięci (w tym loginy i hasła), nawet jeśli nie jest do tego upoważniony.
W momencie ujawnienia dotyczyło to wszystkich urządzeń z dowolną wersją systemu iOS, Linux, macOS oraz Windows, w tym serwerów i usług w chmurze, większości inteligentnych urządzeń i urządzeń wbudowanych wykorzystujących procesory oparte na ARM (urządzenia mobilne, inteligentne telewizory i inne).

4.6 Podsumowanie

Programowanie jest trudne. Programiści muszą:

- dokładnie zrozumieć zadanie;
- przewidzieć wszystkie możliwości;
- pisać dobry (?) kod;
- przewidzieć, co mogą zrobić użytkownicy (bardzo trudne).

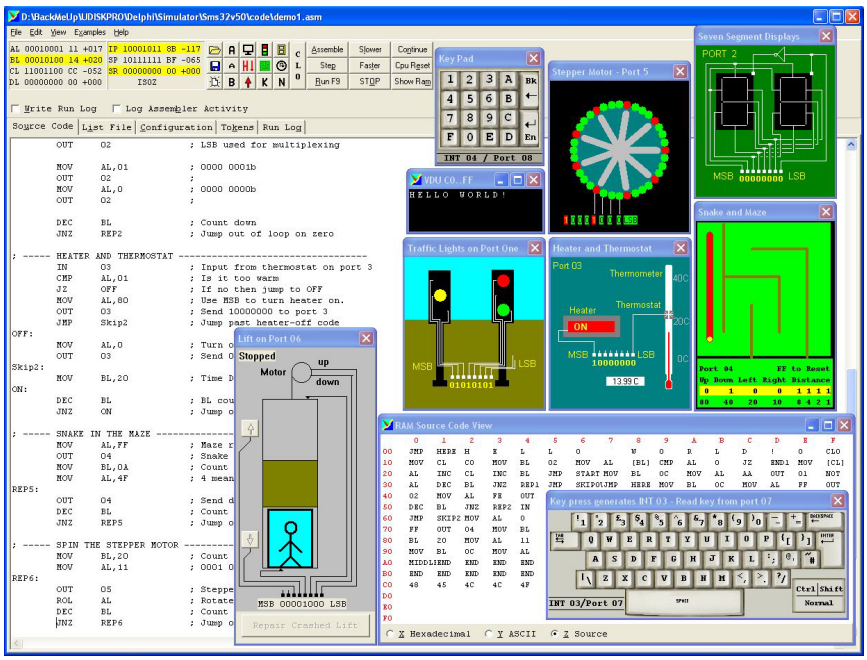
Języki (i frameworki) programowania pozwalają używać narzędzi do budowania kodu. Tam też mogą być błędy. Koszt błędu może być bardzo duży.

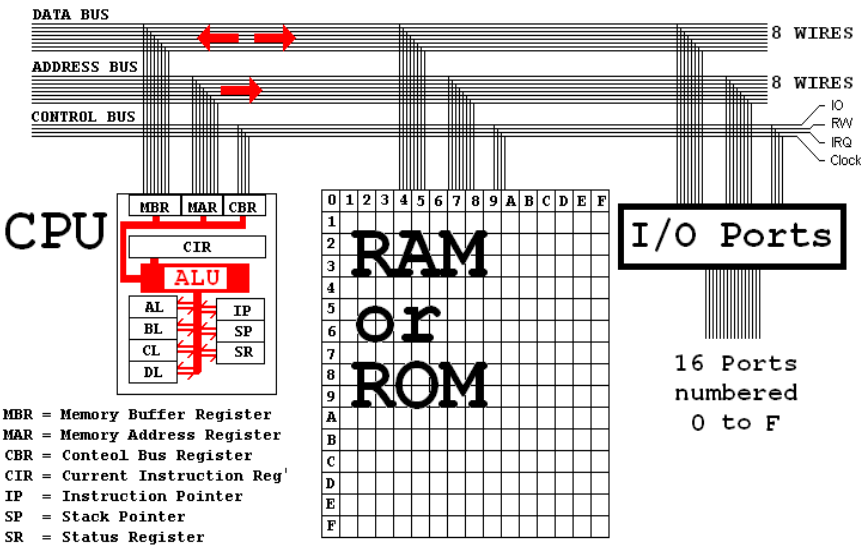
Nie istnieje prawo Moore'a dotyczące oprogramowania. Tzw. prawo Moore'a stwierdza, że moc obliczeniowa komputerów i urządzeń elektronicznych będzie rosła w tempie wykładniczym.

Oprogramowanie staje się coraz bardziej skomplikowane i co za tym idzie również bardziej zawodne. Programiści korzystają z gotowych i rozbudowanych bibliotek (API), nie wiedząc dokładnie jak działają, a znając tylko niektóre ich zewnętrzne skutki.

Rozdział 5. Programowanie w prostym asemblerze na symulatorze Microprocessor Simulator V5.0

https://nbest.co.uk/Softwareforeducation/sms32v50/index.php





Procesor – (CPU) *Central Processing Unit*
256 bajtów pamięci (RAM). 16 portów input oraz output (IO); każdy adresowany od 0 do F. Tylko 6 jest używanych. Sprzętowy zegar uruchamiany przerwaniem numer 02. O przerwaniach będzie osobny podrozdział 5.2. Można oprogramować kolejne takty zegara. Długość taktów można konfigurować. Klawiatura symulowana jest obsłużona przerwaniem sprzętowym o numerze 03. Klawiatura rzeczywista jest obsłużona instrukcją IN 00. Urządzenia zewnętrzne (symulowane) podłączone za pomocą portów input-output.

Jest to też typowa architektura mikrokontrolerów, gdzie zamiast RAM jest zazwyczaj ROM – *read only memory*.

CPU – *Central Processing Unit*. Jest to „mózg" komputera. Tam są wykonywane wszystkie obliczenia (przetwarzanie bajtów), pobieranie z pamięci i z portów wejściowych (*input*) oraz zapisywanie do pamięci lub wysyłanie danych do portów wyjściowych (*output*). CPU ma lokalną pamięć w postaci rejestrów.

ALU (*Arithmetic and Logic Unit*) służy do przetwarzania danych w postaci bajtów. Dane są brane z rejestrów, przetwarzane i umieszczane w rejestrach. Instrukcja MOV służy do przesyłania danych pomiędzy rejestrami a komórkami pamięci RAM.

Rejestry ogólnego przeznaczenia – *General Purpose Registers*
Cztery rejestry AL, BL, CL oraz DL. Każdy ma po 8 bitów, czyli 1 bajt. Można w nim umieścić liczbę bez znaku od 0 do 255 oraz liczbę ze znakiem w przedziale od –128 do +127. Jest to pamięć szybka, podręczna, tymczasowa. Szybkie procesory rzeczywiste posiadają znacznie więcej takich rejestrów. Te cztery rejestry w symulatorze mają swoje odpowiedniki w rzeczywistych 16-bitowych procesorach. Są to rejestry A, B, C oraz D. Przy czym AL to dolna (ang. *low*) część rejestru A, zaś AH to jego górna (ang. *high*) część. Podobnie dla pozostałych rejestrów B, C oraz D.

Rejestry specjalnego przeznaczenia – *Special Purpose Registers*
IP, SR oraz SP.

IP – *Instruction pointer*
Zawiera adres komórki w RAM, gdzie jest instrukcja aktualnie wykonywana. Po wykonaniu instrukcji zazwyczaj ten adres wzrasta o jedną „instrukcję", chyba że jest wykonywany skok; wywoływana jest wtedy procedura (CALL) lub następuje przerwanie programowe (INT) lub sprzętowe. W pamięci RAM ta instrukcja jest podświetlana na czerwono z żółtym tekstem.

SR – *Status Register*
Ten rejestr zawiera następujące flagi (pojedyncze bity) wskazujące na aktualny stan CPU:

- Flaga '**Z**' *zero* jest ustawiona (bit 1), jeśli wynik ostatniego obliczenia jest 0.
- Flaga '**S**' *sign* – znaku – jest ustawiona (bit 1), jeśli wynik ostatniego obliczenia jest ujemny.
- Flaga '**O**' *overflow* – przepełnienia – jest ustawiona (bit 1), jeśli wynik ostatniego obliczenia wykracza poza rejestr jednobajtowy.
- Flaga '**I**' *interrupt* – przerwania – jest ustawiona, jeśli umożliwione są przerwania sprzętowe.

SP – Stack Pointer
Stos to specjalnie wydzielony obszar pamięci RAM, który dostępny jest poprzez kolejkę LIFO (*last in first out*). Wartość w rejestrze SP wskazuje na następną wolną komórkę stosu. Pierwsza komórka stosu ma adres BF; następne są na lewo od niej. Można wpisywać wartość na stos oraz zdejmować to, co zostało ostatnio zapisane. W RAM aktualny wskaźnik stosu (komórka o tym adresie) jest zaznaczony na niebiesko z żółtym tekstem.

CIR – *Current Instruction Register*
To rejestr, w którym znajduje się aktualnie wykonywana instrukcja programu.

MBR to rejestr (jako bufor) do przechowywania bajta danych do wpisania do RAM lub portu IO albo czytanego z RAM lub portów IO.

MAR to rejestr (jako bufor) do przechowywania adresu do RAM lub portów IO.

CBR to rejestr dla magistrali kontrolnej, przechowujący jej aktualny tryb działania.

RAM – *Random Access Memory*
Symulator posiada 256 jednobajtowych komórek pamięci RAM. Komórki te są adresowane od 0 do 255, ale zapisywane w Symulatorze jako dwucyfrowe liczby heksadecymalne w nawiasach kwadratowych od [00] do [FF].
Program jest umieszczany na początku pamięci RAM.

Magistrale – *Busses*
Magistrale (inaczej szyny) to wiązki przewodów do przekazywania impulsów elektrycznych kodujących bity. W chipach (układach scalonych) są to równoległe ścieżki miedziane. W 8-bitowych procesorach magistrale składają się z 8 przewodów. Ale 64-bitowe procesory maja magistrale złożone z 64 przewodów.

Magistrala danych – *Data Bus*
Służy do przekazywania danych (w postaci pojedynczych bajtów) pomiędzy CPU, RAM oraz portami IO. Symulator ma 8-bitową magistralę danych.

Magistrala adresowa – *Address Bus*
Służy do lokalizacji komórki w RAM (lub portu IO) na podstawie adresu (lub numeru portu). Symulator ma 8-bitową magistralę adresową.

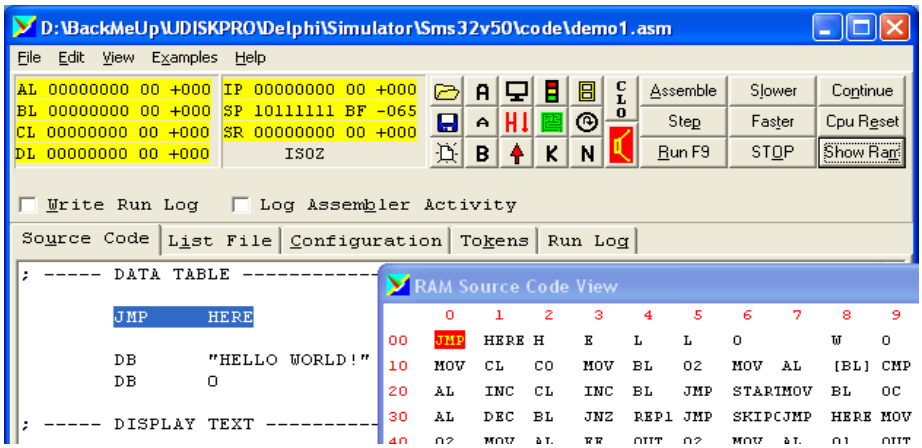
Magistrala kontrolna – *Control Bus*
Służy do wyznaczania dostępu albo do RAM, albo do portów IO oraz do wyznaczania trybu:

- Wpisywanie do RAM (albo portów IO) albo czytanie z RAM, albo z portów IO.
- IRQ do przerwań.
- RW do read-write.
- Zegar (System Clock) poprzez odpowiedni przewód przekazuje impulsy służące do taktowania pracy CPU.

Przerwania sprzętowe – Hardware Interrupts
Wymagają co najmniej jednego przewodu do powiadamiania CPU, że nastąpiło przerwanie, tj. urządzenie zewnętrzne podłączone do portu wejściowego (*input*) żąda dostępu do CPU, poprzez wykonanie odpowiedniego kodu (procedury) związanej z tym urządzeniem. Adres procedury (dokładniej – adres pierwszego bajta tej procedury w RAM) jest wyznaczany poprzez numer przerwania przypisany temu urządzeniu. To w komórce o adresie równym temu numerowi przerwania znajduje się adres tej procedury. W tym symulatorze są tylko trzy numery przerwań: 2, 3 i 4.

Numer 2 odpowiada za przerwanie dla zegara; 3 za przerwanie dla symulowanej klawiatury pełnej, zaś 4 za przerwanie dla symulowanej klawiatury numerycznej. Rzeczywiste procesory maja znacznie więcej przewodów, a więc również więcej numerów przerwań.

5.1 Programowanie



Sam język programowania jest prosty, tak że również bez symulatora można kompilować i wykonywać program w „pamięci”, znając jedynie ograniczenia na RAM (256 bajtów) i sama strukturę RAM-u, w której zapisany jest kod programu, stos oraz pamięć VDU, czyli naszego standardowego wyjścia w postaci monitora z 4 ostatnimi wierszami po 16 bajtów każdy. Ale sam symulator uławia zarówno pisanie, kompilacje, jak i wykonanie, zwłaszcza dla początkujących. Ciekawe, ale też i proste zadania programistyczne są w rozdziale 13, wraz ze wprowadzeniem oraz przykładowymi rozwiązaniami w postaci gotowego kodu.

Pomoc – Help naciśnij klawisz F1.

Piszemy Program
Naciśnij Alt+U. Możesz teraz pisać swój program. Najlepiej w małych fragmentach, kompilując i wykonując, patrząc jednocześnie, co się dzieje. Możesz też metodą copy-paste wklejać program lub fragmenty programu. ALE ZDECYDOWANIE NAJLEPIEJ PISAĆ KOD Z KLAWIATURY. Należy wstawiać komentarze – pomagają zapamiętać, o co w ko-

dzie chodzi. Komentarze są pomijane podczas kompilacji. Zaczynają się od średnika „;” i są do końca linii.

Asembler

Assemble	Programy są pisane w asemblerze. Następnie są kompilowane do kodu maszynowego poprzez naciśnięcie tego klawisza lub poprzez **Alt+A**.
☐ Log Assembler Activity	Możesz zobaczyć, jak to się dokonuje, zaznaczając ten box.
Run F9	Jeśli naciskasz ten klawisz po raz pierwszy (od zmiany w kodzie), to program jest również kompilowany do kodu maszynowego

Asembler - kompilacja

1. **zapisuje kod**
2. **parsuje** i ew. zwraca listę błędów
3. **oblicza skoki**

Prosty przykład kodu:

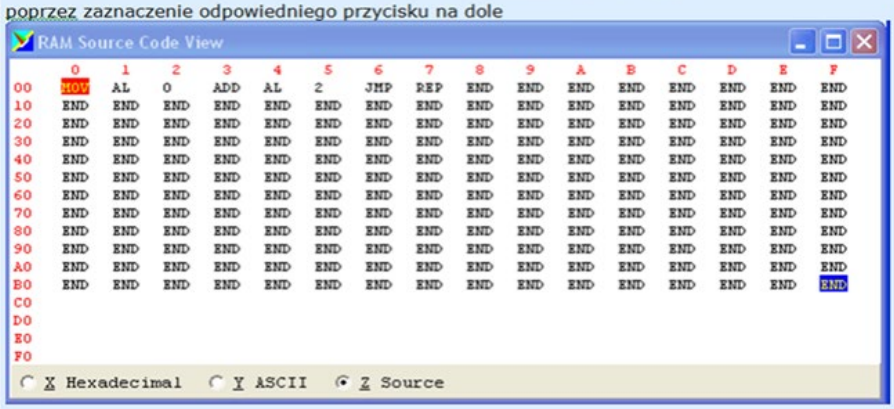
```
; ===== dodaj 2 ================================================
        MOV     AL,0    ; wstaw 0 do rejestru AL
REP:                    ; ta etykieta jest dla instrukcji skoku JMP
        ADD     AL,2    ; dodaj 2 do AL
        JMP     REP     ; skocz do etykiety REP

        END             ; koniec programu
; ==============================================================
```

Wykonanie programu

Step	Naciskając ten klawisz lub **Alt+P**, wykonujesz pojedynczy krok programu.
Run F9	Naciskając ten klawisz lub **F9,** lub **Alt+R**, uruchamiasz cały program.
Slower Faster	Żeby zwolnić lub odpowiednio przyspieszyć wykonywanie programu. Można również przez **Alt+L** lub **Alt+T**.
STOP	Zatrzymaj wykonywanie programu. Można również poprzez **Alt+O** lub **Escape**.
Continue	Kontynuacja (po zatrzymaniu) wykonywania programu. Można również poprzez **Alt+N**.
Cpu Reset	Restart od początku. Można również poprzez **Alt+E**.
Show Ram	Otwórz okno z RAM. Można również poprzez **Alt+M**.

Można zobaczyć kod maszynowy zapisany w RAM

poprzez zaznaczenie odpowiedniego przycisku na dole



Hexadecimal – odpowiada kodowi binarnemu, na jakim pracuje CPU.

ASCII – wyświetla dane tekstowe, jeśli są w RAM.

Source Code – kod asemblera. Można zobaczyć, gdzie zapisany jest program.

List File

Przycisk „List File" służy do listowania oryginalnego kodu.
Liczby w nawiasach kwadratowych, np. [1C], oznaczają adresy w RAM gdzie poszczególne instrukcje są zapisywane.
Po prawej stronie jest pokazany kod maszynowy.

Typowa linia.

```
  MOV CL,C0     ; [10]  D0 02 C0 ; bazowy adres Video w RAM
```

Instrukcja powoduje wstawienie liczby C0 do rejestru AL.
Kod maszynowy tej instrukcji jest zapisany w RAM pod adresem [10].
Ten kod to D0 00 C0.
Komentarze są po prawej stronie.

Poniższy kod jest przykładem, jak można uczynić kod kompletnie nieczytelnym.

Komentarze są ważne.

Przykład - 99nasty.asm

```
; ----- jak nie należy kodować -----
_: Mov BL,C0 Mov AL,3C Q: Mov [BL],AL CMP AL,7B
JZ Z INC AL INC BL JMP Q Z: MOV CL,40 MOV AL,20
MOV BL,C0 Y: MOV [BL],AL INC BL DEC CL JNZ Y JMP
_ END ; naciśnij List File, będzie trochę lepiej!
; ---------------------------------------------
```

Niepoprawny komentarz:

```
INC BL ; dodaj jeden do BL
```

Komentarz, który pomoże zrozumieć kod:

```
INC BL ; zwiększ adres w BL, żeby wskazywał na następną komórkę
       ; w VDU - video RAM
```

A to jest kod dobrze napisany i z odpowiednimi komentarzami

```
; ----- Program pokazujący kody ASCII dla niektórych znaków --
; ----- Dobrze napisany, z komentarzami ------
; ----- Można zrozumieć kod ----------------
; ----- Etykiety maja odpowiednie nazwy --

Start:
        Mov BL,C0       ; zapisz w BL adres do początku video RAM
        Mov AL,3C       ; 3C jest kodem ASCII dla symbolu 'większy'
Here:
        Mov [BL],AL     ; zapisz ten kod ASCII (z AL) do RAM pod adres wskazany przez BL
        CMP AL,7B       ; porównaj AL z '{'
        JZ Clear        ; jeśli AL zawiera '{' to skocz do etykiety Clear:
        INC AL          ; wstaw następny kod ASCII do AL
        INC BL          ; zwiększ adres w BL, aby wskazywał następną komórkę w video RAM
        JMP Here        ; wróć do Here
Clear:
        MOV CL,40       ; ma być 40 (hex) powtórzeń
        MOV AL,20       ; kod ASCII dla spacji wpisz do AL
        MOV BL,C0       ; w BL jest adres początku video RAM
Loop:
        MOV [BL],AL     ; wstaw kod ASCII dla spacji (z AL) do video RAM pod adres z BL
        INC BL          ; zwiększ adres w BL, aby wskazywał na następną komórkę w video RAM
        DEC CL          ; zmniejsz ilość powtórzeń w CL o jeden
        JNZ Loop        ; jeśli CL nie jest zerem, to skocz do etykiety Loop
        JMP Start       ; CL jest zero, a więc cofnij się do etykiety Start

        END
; -------------------------------------------------------------
```

...

Instrukcja MOV. Flagi nie są ustawiane.

Asembler		Kod maszynowy		Opis
MOV	AL,15	**D0 00 15**	AL = 15	**Wstaw 15 do AL**
MOV	BL,[15]	**D1 01 15**	BL = [15]	**Skopiuj i wstaw RAM[15] do BL**
MOV	[15],CL	**D2 15 02**	[15] = CL	**Skopiuj i wstaw CL do RAM[15]**
MOV	DL,[AL]	**D3 03 00**	DL = [AL]	**Skopiuj i wstaw RAM[AL] do DL**
MOV	[CL],AL	**D4 02 00**	[CL] = AL	**Skopiuj i wstaw AL do RAM[CL]**

Bezpośrednie instrukcje arytmetyczno-logiczne.
Flagi mogą być ustawiane.

Asembler		Kod maszynowy	
ADD	AL,BL	**A0 00 01**	AL = AL + BL
SUB	BL,CL	**A1 01 02**	BL = BL - CL
MUL	CL,DL	**A2 02 03**	CL = CL * DL
DIV	DL,AL	**A3 03 00**	DL = DL / AL
INC	DL	**A4 03**	DL = DL + 1
DEC	AL	**A5 00**	AL = AL - 1
AND	AL,BL	**AA 00 01**	AL = AL AND BL
OR	CL,BL	**AB 03 02**	CL = CL OR BL
XOR	AL,BL	**AC 00 01**	AL = AL XOR BL
NOT	BL	**AD 01**	BL = NOT BL
ROL	AL	**9A 00**	Rotuj bity w lewo.
ROR	BL	**9B 01**	Rotuj bity w prawo.
SHL	CL	**9C 02**	Przesuń bity w lewo.
SHR	DL	**9D 03**	Przesuń bity w prawo.

...

Pośrednie instrukcje arytmetyczno-logiczne.
Flagi mogą być ustawiane.

Asembler		Kod maszynowy	
ADD	AL,12	**B0 00 12**	AL = AL + 12
SUB	BL,15	**B1 01 15**	BL = BL - 15
MUL	CL,03	**B2 02 03**	CL = CL * 3
DIV	DL,02	**B6 03 02**	DL = DL / 2
AND	AL,10	**BA 00 10**	AL = AL AND 10
OR	CL,F0	**BB 02 F0**	CL = CL OR F0
XOR	AL,AA	**BC 00 AA**	AL = AL XOR AA

Instrukcje porównania.
Flagi mogą być ustawiane.

Asembler		Kod maszynowy	Opis
CMP	AL,BL	**DA 00 01**	flaga 'Z' ustawiona, jeśli AL = BL flaga 'S' ustawiona, jeśli AL < BL
CMP	BL,13	**DB 01 13**	flaga 'Z' ustawiona, jeśli BL = 13. flaga 'S' ustawiona, jeśli BL < 13.
CMP	CL,[20]	**DC 02 20**	flaga 'Z' ustawiona, jeśli CL = [20]. flaga 'S' ustawiona, jeśli CL < [20].

...

Uzupełnienie dwójkowe

Bit pierwszy od lewej to bit znaku

```
1 0 1 0 1 0 1 0
^
^ ten bit ma wartość -128 dziesiętnie oraz -80 hexadecymalnie
```

Pozostałych siedem bitów jest traktowanych jako liczby od 0 do 127.

Np.

```
1 1 1 1 1 1 1 1     - 128d + 127d = -1d
^
^ -128d
```

Liczba dziesiętna 127

```
0 1 1 1 1 1 1 1     0 + 127d = 127d
^
^ bit zerowy wskazuje na liczbę 0.
```

...

Liczby ujemne															
Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
-128	80	-127	81	-126	82	-125	83	-124	84	-123	85	-122	86	-121	87
-120	88	-119	89	-118	8A	-117	8B	-116	8C	-115	8D	-114	8E	-113	8F
-112	90	-111	91	-110	92	-109	93	-108	94	-107	95	-106	96	-105	97
-104	98	-103	99	-102	9A	-101	9B	-100	9C	-099	9D	-098	9E	-097	9F
-096	A0	-095	A1	-094	A2	-093	A3	-092	A4	-091	A5	-090	A6	-089	A7
-088	A8	-087	A9	-086	AA	-085	AB	-084	AC	-083	AD	-082	AE	-081	AF
-080	B0	-079	B1	-078	B2	-077	B3	-076	B4	-075	B5	-074	B6	-073	B7
-072	B8	-071	B9	-070	BA	-069	BB	-068	BC	-067	BD	-066	BE	-065	BF
-064	C0	-063	C1	-062	C2	-061	C3	-060	C4	-059	C5	-058	C6	-057	C7
-056	C8	-055	C9	-054	CA	-053	CB	-052	CC	-051	CD	-050	CE	-049	CF
-048	D0	-047	D1	-046	D2	-045	D3	-044	D4	-043	D5	-042	D6	-041	D7
-040	D8	-039	D9	-038	DA	-037	DB	-036	DC	-035	DD	-034	DE	-033	DF
-032	E0	-031	E1	-030	E2	-029	E3	-028	E4	-027	E5	-026	E6	-025	E7
-024	E8	-023	E9	-022	EA	-021	EB	-020	EC	-019	ED	-018	EE	-017	EF
-016	F0	-015	F1	-014	F2	-013	F3	-012	F4	-011	F5	-010	F6	-009	F7
-008	F8	-007	F9	-006	FA	-005	FB	-004	FC	-003	FD	-002	FE	-001	FF

...

Liczby dodatnie															
Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex
+000	00	+001	01	+002	02	+003	03	+004	04	+005	05	+006	06	+007	07
+008	08	+009	09	+010	0A	+011	0B	+012	0C	+013	0D	+014	0E	+015	0F
+016	10	+017	11	+018	12	+019	13	+020	14	+021	15	+022	16	+023	17
+024	18	+025	18	+026	1A	+027	1B	+028	1C	+029	1D	+030	1E	+031	1F
+032	20	+033	21	+034	22	+035	23	+036	24	+037	25	+038	26	+039	27
+040	28	+041	29	+042	2A	+043	2B	+044	2C	+045	2D	+046	2E	+047	2F
+048	30	+049	31	+050	32	+051	33	+052	34	+053	35	+054	36	+055	37
+056	38	+057	39	+058	3A	+059	3B	+060	3C	+061	3D	+062	3E	+063	3F
+064	40	+065	41	+066	42	+067	43	+068	44	+069	45	+070	46	+071	47
+072	48	+073	49	+074	4A	+075	4B	+076	4C	+077	4D	+078	4E	+079	4F
+080	50	+081	51	+082	52	+083	53	+084	54	+085	55	+086	56	+087	57
+088	58	+089	59	+090	5A	+091	5B	+092	5C	+093	5D	+094	5E	+095	5F
+096	60	+097	61	+098	63	+099	63	+100	64	+101	65	+102	66	+103	67
+104	68	+105	69	+106	6A	+107	6B	+108	6C	+109	6D	+110	6E	+111	6F
+112	70	+113	71	+114	72	+115	73	+116	74	+117	75	+118	76	+119	77
+120	78	+121	79	+122	7A	+123	7B	+124	7C	+125	7D	+126	7E	+127	7F

Flagi i instrukcje warunkowego skoku

Warunkowe instrukcje skoku korzystają z rejestru flag SR (ang. Status Register). Rejestr ten jest modyfikowany przez operacje arytmetyczne i logiczne. Zawiera on następujące flagi:

- Flaga "Z" (zera) jest ustawiana na jeden, jeśli obliczenia dały wynik zerowy.
- Flaga "S" (znaku) jest ustawiana na jeden, jeśli obliczenia dały wynik ujemny.

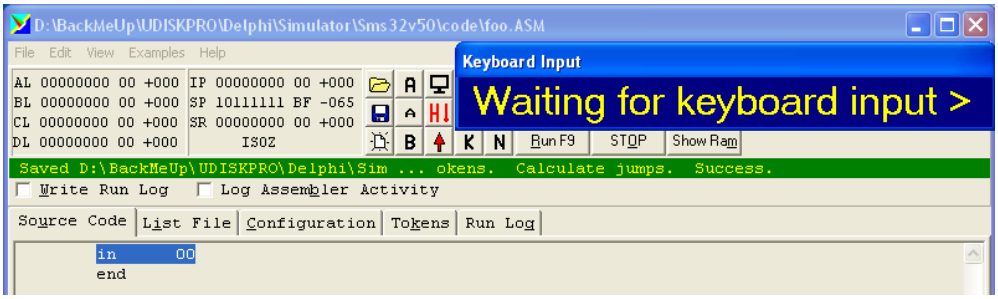
- Flaga "O" (przepełnienia) jest ustawiana, jeśli wynik był zbyt duży, aby zmieścić się w rejestrze. Gdy wartość rejestru osiągnie $7F_{(16)}$ lub $127_{(10)}$, to następna powinna być liczba 128, ale ze względu na sposób, w jaki liczby są przechowywane w systemie binarnym, następną liczbą jest minus 128. Efekt ten nazywany jest przepełnieniem.
- Flaga "I" (przerwania) jest ustawiana, jeśli przerwania są włączone.

Skoki:

- JMP to skok bezwarunkowy; wykonanie zawsze skacze do wskazanej etykiety.
- JZ – skacze do wskazanej etykiety, jeśli flaga "Z" jest ustawiona.
- JNZ – skacze do wskazanej etykiety, jeśli flaga "Z" nie jest ustawiona.
- JS – skacze do wskazanej etykiety, jeśli flaga "S" jest ustawiona.
- JNS – skacze do wskazanej etykiety, jeżeli flaga "S" nie jest ustawiona.
- JO – skacze do wskazanej etykiety, jeżeli flaga "O" jest ustawiona.
- JNO – skacze do wskazanej etykiety, jeśli flaga "O" nie jest ustawiona.

Przykładowe instrukcje w wyjaśnieniami:

```
CALL 30          ; Zapisz adres IP na stosie i przejdź do procedury pod adresem 30.
RET              ; Przywróć adres IP ze stosu i skocz do niego.
INT 02           ; Zapisz adres IP na stosie i przejdź do adresu (wektora przerwań)
                 ; pobranego z pamięci RAM[02].
IRET             ; Przywróć adres IP ze stosu i skocz do niego.
PUSH BL          ; BL jest zapisany na stosie.
POP CL           ; CL jest pobrany ze stosu.
PUSHF            ; Flagi z SR są zapisane na stosie.
POPF             ; SR flagi ze stosu są ponownie zapisane do rejestru SR.
IN 07            ; Dane (bajt) z I/O port 07 są zapisane w AL.
OUT 01           ; Dane z AL są wysłane na I/O port 01.
CLI              ; Zamknij flagę przerwania I; uniemożliwia przerwanie sprzętowe.
STI              ; Postaw flagę przerwania I; umożliwia przerwanie sprzętowe.
CLO              ; Zamknij widoczne okna peryferyjne.
HALT             ; Zatrzymaj procesor.
NOP              ; Nie rób nic przez jeden cykl zegara.
ORG     40       ; Wprowadź do RAM kod procedury zaczynając od adresu 40.
DB "Hello"       ; Zapisz napis 'Hello' (w kodzie ASCII) w RAM.
DB      84       ; Zapisz 84 (w notacji HEX) w RAM.
IN 00            ; Patrz na obrazek niżej (wielkość liter nie ma znaczenia).
```



Prosty program

```
        ; ---------------------------------------------------------------
        ; wpisywanie znaków z klawiatury do momentu naciśnięcia klawisza Enter.
        ; ---------------------------------------------------------------
CLO                         ; Zamknij niepotrzebne okna.
        Rep:    IN 00       ; Czekaj na znak z klawiatury fizycznej;
                            ; zapisz kod ASCII znaku do AL.
        CMP AL, 0D          ; Sprawdź, czy Enter? (kod ASCII to 0D).
        JNZ Rep             ; Nie – skocz do etykiety Rep:.
                            ; Tak – wykonaj następną instrukcję.
END
        ; patrz co się dzieje z rejestrem AL !
```

Jeszcze jeden prosty program

```
        ; ---------------------------------------------------------------
        ; wpisywanie znaków z klawiatury do VDU (cztery ostatnie wiersze w RAM)
        ; do momentu naciśnięcia klawisza Enter.
        ; ---------------------------------------------------------------
CLO
                MOV BL, C0  ; Wstaw do rejestru BL liczbę C0;
                            ; C0 to adres pierwszej komórki VDU.
Rep:            IN 00
                MOV [BL], AL  ; Kopiuj z AL do komórki RAM.
                INC BL        ; Zwiększ BL o jeden.
                CMP AL, 0D    ; Sprawdź, czy Enter? (kod ASCII to 0D).
                JNZ Rep       ; Jeśli nie, to skocz do Rep; jeśli tak, to przejdź dalej.
END
```

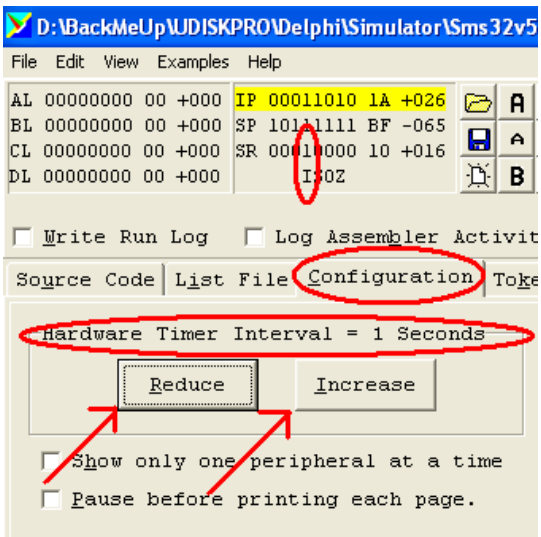
I jeszcze jeden prosty program

```
        ; ---------------------------------------------------------------
        ; wpisywanie znaków z klawiatury do VDU (poprzez stos)
        ; do momentu naciśnięcia klawisza Enter.
        ; ---------------------------------------------------------------

CLO
                MOV DL, 00      ; Pierwszy licznik dla stosu (wiersz B w RAM).
                MOV BL, C0      ; Wstaw do rejestru BL liczbę C0.
Rep_1:
                IN 00
                PUSH AL.        ; Skopiuj z AL i wstaw na stos.
                INC DL          ; Zwiększ licznik o 1;
                                ; UWAGA: stos ma tylko 16 bajtów!
                CMP AL, 0D      ; Enter? (kod ASCII to 0D)
                JNZ Rep_1
Rep_2:
                POP AL          ; Zdejmij ze stosu na AL.
                MOV [BL], AL
                INC BL          ; Zwiększ BL o jeden.
                DEC DL          ; Zmniejsz DL o jeden.
                CMP DL, 00      ; Porównaj, czy DL = 0.
                JNZ Rep_2
END
```

5.2 Przerwania

Przerwanie to sygnał powodujący zmianę w aktualnie wykonywanym programie. Polega na zatrzymaniu aktualnie wykonywanego programu i wykonanie przez procesor procedury obsługi przerwania. Po jej wykonaniu następuje powrót do kolejnej instrukcji w programie.

Przerwania dzielą się na sprzętowe (pochodzące od urządzeń, również zewnętrznych, takich jak klawiatura czy zegar) i programowe, wywoływane bezpośrednie z kodu programu (niewiele różnią się od zwykłych procedur, ale są zestandaryzowane w systemie operacyjnym). Każde przerwanie ma swój numer, którego znaczenie będzie wyjaśnione niżej na przykładach.



5.2.1 Przerwania w symulatorze

Przerwanie zegara:

Timer (zegar) sprzętowy generuje przerwanie INT 02 w regularnych odstępach czasu. Przedział czasu można zmienić za pomocą karty **Configuration**, jak pokazano na obrazku. CPU zignoruje INT 02,

chyba że ustawiona jest flaga (I) w rejestrze statusu (SR). Użyj STI, aby ustawić flagę (I). Użyj CLI, aby wyczyścić flagę (I).

Przykładowy kod poniżej przetwarza INT 02, ale nic nie robi. Jeśli zegar procesora jest zbyt wolny, nowe INT 02 może wystąpić zanim poprzednie zostanie obsłużone. Niekoniecznie jest to problem, jeśli procesor zdąży wykonać przerwanie. Ale jeśli kod obsługi przerwań będzie zapisywał i odtwarzał wszystkie używane rejestry, to może ten problem wystąpić. Użyj PUSH i PUSHF, aby zapisać wartości rejestrów na stosie. Użyj POPF i POP do przywrócenia rejestrów. Pamiętaj, żeby zdejmować ze stosu w odwrotnej kolejności, w jakiej zostały tam wstawione.

Jeśli procesor jest zbyt wolny a będą coraz to nowe przerwania, to stos będzie się stopniowo powiększał i pochłaniał całą dostępną pamięć RAM. Ostatecznie stos nadpisze program, powodując awarię. Jest to przydatne ćwiczenie do spowolnienia zegara procesora i obserwowania tego, co się wówczas stanie.
Przykład użycia takiego przerwania (wielkość liter nie ma znaczenia w kodzie):

```
        jmp     start
        db      10      ; Hardware Timer Interrupt Vector

; ===== Hardware Timer =======
        org     10
        CLI
        nop             ; Do something useful here
        nop
        nop
        nop
        nop
        STI
        iret
; ============================

; ===== Idle Loop ============
start:
        STI             ; Set (I) flag
idle:
        nop             ; Do something useful here
        nop
        nop
        nop
        nop
        jmp     idle
; ============================
        end
; ============================
```

Następne przerwanie dotyczy pełnej symulowanej klawiatury

To jedno z bardziej złożonych urządzeń. Aby klawiatura była widoczna, użyj OUT 07. Za każdym naciśnięciem klawisza generowane jest przerwanie sprzętowe INT 03. Domyślnie CPU ignoruje to przerwanie. Aby przetworzyć przerwanie, na początku programu użyj polecenia STI, żeby ustawić flagę przerwania (I) w rejestrze statusu CPU (SR). Umieść wektor przerwań w komórce [03] w RAM. Zawartość tej komórki (bajt) powinna wskazywać na adres w RAM, gdzie znajduje się kod obsługi tego przerwania. Program obsługi przerwań powinien użyć IN 07, aby odczytać kod ASCII znaku naciśniętego klawisza; ten kod będzie w rejestrze AL.

Gdy STI ustawi flagę (I) w rejestrze statusu (SR), zostaną również wygenerowane przerwania z timera sprzętowego. Te też muszą być przetwarzane. Timer sprzętowy generuje INT 02. Aby przetworzyć to przerwanie, umieść wektor przerwania w komórce [2] w RAM. Powinno to wskazywać na kod obsługi przerwań timera. Kod timera może być tak prosty, jak IRET. Spowoduje to przerwanie bez wykonywania żadnego innego przetwarzania i powrót do wykonania głównego programu.

Przykład użycia takiego przerwania:

```
        jmp     start
        db      10      ; Hardware Timer Interrupt Vector
        db      20      ; Keyboard Interrupt Vector

; ===== Hardware Timer =======
        org     10
        nop
        iret
; ============================

; ===== Keyboard Handler =====
        org     20
        CLI             ; Prevent re-entrant use
        push    al
        pushf

        in      07
        nop             ; Process the key press here
        nop
        nop
```

```
        nop
        nop

        popf
        pop     al
        STI
        iret
; ============================

; ===== Idle Loop ============
start:
        STI             ; Set (I) flag
        out     07      ; Make keyboard visible
idle:
        nop             ; Do something useful here
        nop
        nop
        nop
        nop
        jmp     idle
; ============================
```

Następne przerwanie dotyczy symulowanej klawiatury numerycznej

To również jedno z bardziej złożonych urządzeń. Aby uczynić klawiaturę numeryczną widoczną, użyj OUT 08. Przy każdym naciśnięciu klawisza generowane jest przerwanie sprzętowe INT 04. Domyślnie CPU ignoruje to przerwanie. Aby przetworzyć przerwanie, na początku programu użyj polecenia STI, żeby ustawić flagę przerwania (I) w rejestrze statusu CPU (SR). Umieść wektor przerwań w adresie RAM 04. Powinno to wskazywać na kod obsługi przerwań. Program obsługi przerwań powinien użyć IN 08, aby odczytać kod ASCII znaku po naciśnięciu klawisza; ten kod będzie w rejestrze AL.

Gdy STI ustawi flagę (I) w rejestrze statusu (SR), zostaną również wygenerowane przerwania z timera sprzętowego. Te też muszą być przetwarzane. Timer sprzętowy generuje INT 02. Aby przetworzyć to przerwanie, umieść wektor przerwania w lokalizacji RAM 02. Powinno to wskazywać na kod obsługi przerwań timera. Kod timera może być tak prosty, jak IRET. Spowoduje to przerwanie bez wykonywania instrukcji i powrót do wykonywania głównego programu.

Przykład użycia takiego przerwania:

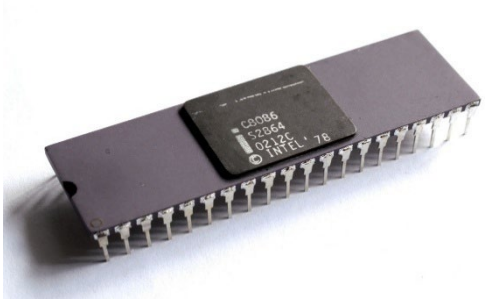
```
        jmp     start
        db      10      ; Hardware Timer Interrupt Vector
        db      00      ; Keyboard Interrupt Vector (unused)
        db      20      ; Numeric Keypad Interrupt Vector
```

```
; ===== Hardware Timer =======
        org     10
        nop             ; Do something useful here
        nop
        nop
        nop
        nop
        iret
; ============================

; ===== Keyboard Handler =====
        org     20
        CLI             ; Prevent re-entrant use
        push    al
        pushf

        in      08
        nop             ; Process the key press here
        nop
        nop
        nop
        nop

        popf
        pop     al
        STI
        iret
; ============================

; ===== Idle Loop ============
start:
        STI             ; Set (I) flag
        out     08      ; Make keypad visible
idle:
        nop             ; Do something useful here
        nop
        nop
        nop
        nop
        jmp     idle
; ============================
        end
; ============================
```

Znacznie więcej i konkretnie o programowaniu na symulatorze znajduje się w rozdziale 13.

Rozdział 6. Procesor 8086 firmy Intel

Na podstawie https://pl.wikipedia.org/wiki/Intel_8086



8086 – 16-bitowy mikroprocesor wprowadzony na rynek 8 czerwca 1978 roku.

Został zaprojektowany w technologii 3 µm HMOS (ang. High performance Metal-Oxide Semiconductor) jako rozszerzenie 8-bitowego procesora 8080/8085.
Jego zastosowanie (w szczególności późniejszej odmiany z 8-bitowym interfejsem – 8088) w pierwszych ogólnodostępnych komputerach osobistych (IBM PC) doprowadziło do wielkiej popularyzacji i dalszego rozwoju tej rodziny procesorów (architektura x86).

W związku z historycznym znaczeniem procesora 8086 firmie Intel przydzielono identyfikator 0x8086 na liście identyfikatorów (PCI ID) dostawców urządzeń dla magistrali PCI (ang. *Peripheral Component Interconnect* – magistrala komunikacyjna służąca do przyłączania kart rozszerzeń do płyty głównej w komputerach klasy PC). Po raz pierwszy została publicznie zaprezentowana w czerwcu 1992 r. jako rozwiązanie umożliwiające szybszą komunikację pomiędzy procesorem i kartami niż stosowane dawniej ISA. Dodatkową zaletą PCI jest to, że nie ma znaczenia, czy w gnieździe jest karta sterownika dysków (np. SCSI), sieciowa czy graficzna.

Każda karta pasująca do gniazda PCI funkcjonuje bez jakichkolwiek problemów, gdyż nie tylko sygnały, ale i przeznaczenie poszczególnych styków gniazda są znormalizowane.

6.1 Podstawowe parametry mikroprocesora 8086

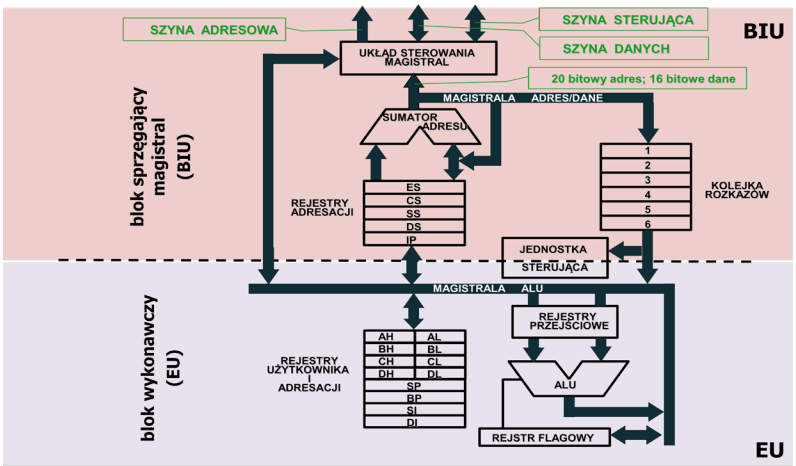
Jest w architekturze CISC. Przestrzeń adresowa pamięci to 1 MB w trybie rzeczywistym. 16-bitowa magistrala danych. 20-bitowa magistrala adresowa. Częstotliwość sygnału zegarowego do 10 MHz. 91 podstawowych typów rozkazów; samych rozkazów jest znacznie więcej. Przestrzeń adresowa urządzeń wejścia/wyjścia to 64 kB. Możliwość wykonywania operacji bitowych, bajtowych, o długości słowa i łańcuchowych. 7 trybów adresowania argumentów w pamięci. 16-bitowa jednostka arytmetyczno-logiczna (ALU). 16-bitowe rejestry ogólnego przeznaczenia. 6-bajtowa kolejka rozkazów.

Dwa tryby pracy – minimalny i maksymalny

W trybie minimalnym procesor steruje całym systemem mikrokomputerowym, pełniąc rolę kontrolera magistrali.
Zwykle system taki składa się z jednego obwodu drukowanego i kilku urządzeń peryferyjnych.

W trybie maksymalnym magistrala jest współdzielona pomiędzy mikroprocesor a procesory wspomagające.
Funkcje sterownika magistrali przejmuje wtedy osobny element systemu mikrokomputerowego zwany kontrolerem magistrali.
Tryb ten stosowany jest w przypadku systemów wieloprocesorowych (np. system, w którego skład wchodzi mikroprocesor wraz z koprocesorem matematycznym).

6.2 Schemat blokowy procesora 8086



Układ wykonawczy zawiera:

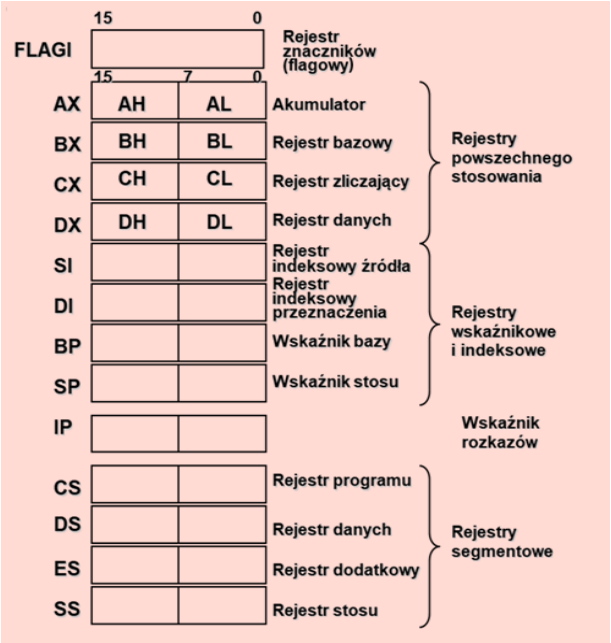
- 16-bitową jednostkę arytmetyczno-logiczną,
- układ sterowania z rejestrem rozkazów,
- cztery 16-bitowe rejestry użytkownika,
- cztery 16-bitowe rejestry adresacji,
- 16-bitowy rejestr wskaźników (rejestr flagowy).

Zadaniem układu wykonawczego jest dekodowanie i wykonywanie rozkazów gromadzonych w kolejce. W trakcie wykonywania rozkazów w układzie wykonawczym układ sprzęgający magistrali zewnętrznej otrzymuje zezwolenie na pobranie następnego rozkazu z pamięci RAM.
Układ sprzęgający magistrali zawiera:

- układ generacji adresu fizycznego,
- układ kolejkowania rozkazów,
- 16-bitowe rejestry adresacji (segmentowe i IP),
- bufor WE/WY.

Układ sprzęgający przesyła dane między procesorem a pamięcią operacyjną lub układami WE/WY pod nadzorem układu wykonawczego. W czasie, gdy układ wykonawczy realizuje kolejny rozkaz, BIU pobiera nowy rozkaz z pamięci operacyjnej i przekazuje go do kolejki.
Drugim istotnym zadaniem układu sprzęgającego BIU jest wyznaczania adresu fizycznego (m.in. na podstawie danych przekazywanych z EU).

6.3 Rejestr flagowy



***SF* (*sign flag*) – znacznik znaku** – równy najbardziej znaczącemu bitowi wyniku:

- *0* – wynik operacji dodatni,
- *1* – wynik operacji ujemny.

***ZF* (*zero flag*) – znacznik zera**:

- *0* – wynik operacji różny od zera,
- *1* – wynik operacji równy zeru.

***PF* (*parity flag*) – znacznik parzystości** – ustawiany w zależności od liczby jedynek w najmniej znaczących 8 bitach wyniku:

- *0* – liczba jedynek w wyniku operacji nieparzysta,
- *1* – liczba jedynek w wyniku operacji parzysta.

***AF* (*auxiliary carry flag*)** – znacznik przeniesienia połówkowego (pomocniczego):

- *0* – brak przeniesienia pomiędzy trzecim i czwartym bitem bajta (BCD),
- *1* – występuje przeniesienie.

***CF* (*carry flag*)** – znacznik przeniesienia:

- *0* – wynik operacji arytmetycznej nie powoduje powstania przeniesienia z najbardziej znaczącego bitu,
- *1* – wynik takie przeniesienie powoduje.

***OF* (*overflow flag*)** – znacznik nadmiaru:

- *0* – suma modulo 2 przeniesień z najbardziej znaczącej pozycji i pozycji przedostatniej jest równa 0,
- *1* – suma modulo 2 przeniesień z najbardziej znaczącej pozycji i pozycji przedostatniej jest równa 1 (przekroczenie zakresu w kodzie U2).

***IF* (*interrupt flag*)** – znacznik przerwań:

- *0* – brak zezwolenia na przyjmowanie przerwań z wejścia *INT*,
- *1* – zezwolenie na przyjmowanie przerwań.

***DF* (*direction flag*)** – znacznik kierunku – wskazuje, czy zawartości rejestrów SI i DI mają być zwiększane lub zmniejszane o jeden w czasie wykonywania operacji łańcuchowych:

- *0* – rejestry są zwiększane,
- *1* – rejestry są zmniejszane.

***TF* (*trap flag*)** – znacznik pułapki umożliwiającej pracę krokową:

- *0* – praca krokowa wyłączona,
- *1* – praca krokowa włączona; mikroprocesor po wykonaniu każdego rozkazu wykona skok do odpowiedniego podprogramu obsługi przerwania.

6.4 Rejestry ogólnego przeznaczenia: arytmetyczne

Rejestry arytmetyczne to cztery 16-bitowe rejestry ogólnego przeznaczenia: AX, BX, CX, DX. Każdy z tych rejestrów może również działać jako dwa niezależne rejestry 8-bitowe:

- *AX* lub *AH*, *AL*,
- *BX* lub *BH*, *BL*,
- *CX* lub *CH*, *CL*,
- *DX* lub *DH*, *DL*.

AX – akumulator (Accumulator)
Rejestr ten bezpośrednio współpracuje z jednostką arytmetyczno-logiczną. Niektóre operacje, których argumenty znajdują się w akumulatorze, wykonywane są szybciej niż ich odpowiedniki wykorzystujące inne rejestry. Takie rozkazy jak: mnożenie, dzielenie i operacje wejścia/wyjścia wymagają użycia akumulatora do przechowywania argumentu bądź też zapisu wyniku.

BX – baza (Base)
Rejestr ten może być używany do adresowania argumentu, znajdującego się w pamięci, stanowiąc bazę do obliczania adresu.

CX – licznik (Counter)
Rejestr ten jest używany jako licznik w operacjach łańcuchowych oraz pętlach. Po każdej iteracji jego zawartość jest automatycznie zmniejszana. W rozkazach przesunięć rejestr CL (mniej znaczący bajt rejestru CX) wykorzystywany jest jako licznik bitów.

DX – dane (Data)
Rejestr ten jest wykorzystywany w niektórych operacjach arytmetycznych do przechowywania części argumentu lub wyniku operacji (mnożenie i dzielenie 16-bitowe). Zawiera on także adres urządzenia w operacjach wejścia/wyjścia.

6.5 Rejestry ogólnego przeznaczenia: wskaźnikowe i indeksowe

16-bitowe rejestry adresowe: SP, BP, SI, DI
Ich głównym zadaniem jest wskazywanie miejsca w pamięci, w którym znajdują się argumenty rozkazu. Wykorzystywane są w adresowaniu indeksowym, bazowym oraz indeksowo-bazowym. Można ich również używać do przechowywania argumentu bądź wyniku operacji.

***SP* – wskaźnik stosu (*Stack Pointer*)**
Wskazuje adres ostatnio zapisanego słowa na stosie. Jego wartość jest automatycznie zwiększana lub zmniejszana w zależności od wykonywanej operacji (POP, PUSH).

BP* – wskaźnik bazy (*Base Pointer)
Pełni on funkcję wskaźnika ogólnego przeznaczenia. Wykorzystywany jest do adresowania bazowego danych w segmencie stosu.

***SI* – rejestr indeksowy źródła (*Source Index*)**
Uniwersalny rejestr indeksowy. Wykorzystywany dla dwuargumentowych operacji łańcuchowych do przetrzymywania adresu źródła danych. Po każdej kolejnej iteracji jego wartość jest zwiększana lub zmniejszana zależnie od ustawienia flagi kierunku.

***DI* – rejestr indeksowy przeznaczenia (*Destination Index*)**
Uniwersalny rejestr indeksowy. Jest wykorzystywany w czasie dwuargumentowych operacji łańcuchowych do przetrzymywania adresu przeznaczenia danych. Po każdej kolejnej iteracji jego zawartość zwiększana lub zmniejszana zależnie od ustawienia flagi kierunku.

6.6 Rejestry segmentowe

Są to 16-bitowe rejestry, dostępne dla programisty, których zawartość służy do obliczania adresu fizycznego komórki pamięci. Rejestry te zawierają adres początkowy danego segmentu pamięci. Mikroprocesor w zależności od rodzaju segmentu pamięci, do którego chce się odwołać, wykorzystuje odpowiedni rejestr. Programista ma możliwość zmiany automatycznie wykorzystywanego rejestru poprzez umieszczenie odpowiedniego prefiksu przed rozkazem, dla którego zmiana ma zostać zastosowana.
Mikroprocesor 8086 ma 20-bitową magistralę adresową. Pozwala ona na zaadresowanie do 1 MB pamięci operacyjnej.

Przestrzeń adresowa została podzielona na segmenty o długości 64 kB, rozpoczynające się co 16 bajtów (kolejne segmenty pamięci mogą nakładać się na siebie).

- ***CS* – rejestr segmentowy programu (*Code Segment register*).** Zawartość tego rejestru wyznacza początek aktualnie używanego segmentu programu. Wartość ta wykorzystywana jest do obliczania adresu fizycznego kolejnego rozkazu do pobrania z pamięci.
- ***DS* – rejestr segmentowy danych (*Data Segment register*).** Zawartość tego rejestru wyznacza początek aktualnie używanego segmentu danych. Wartość ta wy-

korzystywana jest do obliczania adresu fizycznego argumentu lub wyniku aktualnie wykonywanego rozkazu. Wyjątek stanowią rozkazy łańcuchowe, w których zawartość tego rejestru służy jedynie do obliczania adresu źródła danych.

- ***SS* – rejestr segmentowy stosu (*Stack Segment register*).** Zawartość tego rejestru wyznacza początek aktualnie używanego segmentu stosu. Wartość ta wykorzystywana jest do obliczania adresu fizycznego komórki pamięci, na którą wskazuje wskaźnik stosu (rejestr SP).
- ***ES* – rejestr segmentowy dodatkowy (*Extra Segment register*).** Zawartość tego rejestru wyznacza początek aktualnie używanego dodatkowego segmentu danych. Wartość ta wykorzystywana jest do obliczania adresu fizycznego przeznaczenia dla operacji łańcuchowych (np. rozkazu przenoszenia bloku danych).
- **Licznik rozkazów (Instruction Pointer – IP)** 16-bitowy rejestr, którego zawartość służy do obliczania adresu fizycznego następnego słowa rozkazu (instrukcji) do pobrania z pamięci. Stanowi on rejestr indeksowy dla rejestru CS, wyznaczającego segment z kodem programu. CS:IP to adres logiczny tej następnej instrukcji. Jego zawartość jest automatycznie zwiększana po pobraniu każdego bajta rozkazu (w przypadku pobrania słowa jego wartość wzrasta o 2). Programista ma możliwość zmiany zawartości licznika rozkazów poprzez zastosowanie rozkazu skoku.
- **Kolejka rozkazów** jest to 6-bajtowa pamięć zorganizowana w słowa (trzy 2-bajtowe komórki) wykorzystywane przez mikroprocesor do przechowywania pobranych wcześniej rozkazów.

6.7 Adresowanie segmentowe

Adres fizyczny komórek pamięci obliczany jest na podstawie dwóch 16-bitowych składników, tj. adresu segmentu oraz adresu efektywnego (przesunięcia).
Taki sposób adresowania nazywa się adresowaniem segmentowym.

Generator adresu fizycznego jest to 20-bitowy sumator służący do obliczania adresu fizycznego komórki pamięci, znajdujący się w jednostce interfejsowej.

Adres segmentu mnożony jest przez 16, co powoduje, że zostaje on przesunięty o 4 bity w lewo (zwolnione z prawej strony bity przyjmują wartość 0), a następnie dodaje się do niego adres efektywny, obliczony z zastosowaniem wszystkich modyfikatorów (w przypadku danych) lub zawartość licznika rozkazów (w przypadku rozkazów).

Przykład generowania adresu fizycznego:
Adres logiczny:

- **0010h:000Fh** (adres początku segmentu **:** wartość przesunięcia (tzw. adres efektywny))
  0010h * 0010h (16 dziesiętne) **= 00100h** (dwudziestobitowy adres początku segmentu)
  00100h + 000Fh = 0010Fh (adres fizyczny komórki pamięci)

Procesor 8086 przesuwa 16-bitowy segment tylko o cztery bity przed dodaniem go do 16-bitowego offsetu

- (16 × segment + offset),

tworząc w ten sposób 20-bitowy zewnętrzny (lub efektywny lub fizyczny) adres z 32-bitowej pary (segment : przesunięcie). W rezultacie do każdego adresu zewnętrznego można się odnieść przez **2^12 = 4096** różnych par (segment : przesunięcie).

```
  0110 1000 1000 0111 0000 Segment, 16 bits, shifted 4 bits left (or multiplied by 0x10)
+      0011 0100 1010 1001 Offset,    16 bits
-------------------------
  0110 1011 1101 0001 1001 Address, 20 bits
```

6.8 Tryby adresowania

Tryb adresowania to sposób wyznaczania adresu argumentów i wyników operacji. W mikroprocesorze 8086 każdy z rejestrów ogólnego przeznaczenia może służyć do przechowywania adresu lub jego składnika. Adres operandu obliczany jest zgodnie z równaniem

- EA = BR + IR + p

przy czym:

- EA – adres efektywny,
- BR – rejestr bazowy,
- IR – rejestr indeksowy,
- p – przemieszczenie.

Adres efektywny EA jest adresem logicznym „widzianym” przez program. Na podstawie EA układy segmentacji obliczają adres rzeczywisty w pamięci operacyjnej. Rejestrem bazowym może być rejestr BP lub BX, a rejestrem indeksowym może być rejestr SI lub DI. Przemieszczenie jest zawarte w rozkazie i może mieć długość ośmiu lub szesnastu bitów.

Mikroprocesor 8086 realizuje następujące tryby adresowania:

- natychmiastowe,
- rejestrowe,
- bezpośrednie,
- pośrednie,
- bazowe,
- indeksowe,
- indeksowo-bazowe.

Adresowanie natychmiastowe

W adresowaniu natychmiastowym argument pobierany jest bezpośrednio z rozkazu. W tym trybie wskazywany jest wyłącznie operand źródłowy. Np. MOV AX, 20 – w rejestrze AX zostanie zapisana liczba 20h.

Adresowanie rejestrowe
W adresowaniu rejestrowym operandy znajdują się w rejestrach wewnętrznych mikroprocesora. Jeżeli operand znajduje się w pamięci, to zespół wykonawczy EU oblicza jego 16-bitowy adres (przesunięcie) wewnątrz segmentu. Zespół BIU oblicza adres rzeczywisty na podstawie otrzymanego przesunięcia (adresu efektywnego EA) i zawartości wybranego rejestru segmentowego. Np. MOV AX, BX – w rejestrze AX zostanie zapisana zawartość rejestru BX.
W naszym symulatorze tak nie można.

Adresowanie bezpośrednie
W adresowaniu bezpośrednim adres operandu znajduje się bezpośrednio w rozkazie.
Np. MOV AX, [40] – w rejestrze AX zostanie zapisana zawartość komórki pamięci (segment danych) o adresie 40. Standardowy adres operandu jest przesunięciem w segmencie danych (DS), można to nadpisać poprzez wskazanie innego segmentu.
Np. MOV AX, CS:[40] – w rejestrze AX zostanie zapisana zawartość z komórki pamięci (segment PROGRAMU(kodu)) o offsecie 40.

Adresowanie pośrednie
W tym trybie odwołujemy się do jednego z rejestrów roboczych procesora (np. BX) lub do komórki pamięci (np. 19). W rejestrze (BX) zapisany jest numer komórki pamięci, do której trzeba sięgnąć aby odczytać tam zawarty adres i przenieść do drugiego rejestru (AX).
Dla adresowania pośredniego z pamięci odczytujemy numer komórki pamięci z dwóch komórek (komórki 19 i komórki 20) w taki sposób, że zawartość tej pierwszej (19) stanowi ważniejszą część tego numeru, zaś zawartość drugiej komórki (20) mniej ważną część tego numeru. Dalej postępujemy podobnie jak przy adresowaniu pośrednim z rejestru – przenosimy zawartość do rejestru AX. Np. MOV AX, [CX] – w rejestrze AX zostanie zapisana zawartość komórki pamięci o adresie, który znajduje się w rejestrze CX. Wszystkie rejestry wskazują offset w segmencie danych (DS), poza rejestrem BP, który jest przesunięciem w segmencie stosu (SS). Można to zmienić, określając segment w rozkazie.
Np. MOV AX, SS:[CX] – w rejestrze AX zostanie zapisana zawartość komórki pamięci z segmentu stosu o adresie, który znajduje się w rejestrze CX.

Adresowanie bazowe jest to rodzaj adresowania pośredniego, gdzie rozkaz wskazuje na jeden z rejestrów bazowych BX lub BP i może zawierać 8- lub 16-bitową wartość stanowiącą lokalne przemieszczenie.
Adresem efektywnym jest suma zawartości rejestru bazowego i lokalnego przemieszczenia. Np. MOV AX, [BP+2].

Adresowanie indeksowe jest rodzajem adresowania pośredniego, gdzie adres efektywny jest sumą zawartości rejestru indeksowego SI lub DI i lokalnego przemieszczenia. Np. MOV AX, [SI+3].

Adresowanie bazowo-indeksowe – adres efektywny jest sumą zawartości jednego z rejestrów bazowych, jednego z rejestrów indeksowych i lokalnego przemieszczenia.
Np. MOV AX, [SI+BP+4].

6.9 Rozkazy (instrukcje)

Rozkazy operujące na ciągach słów
Posługują się rejestrami indeksowymi. Rejestry SI i DI zawierają adresy efektywne pierwszego słowa odpowiednio w ciągu źródłowym i wynikowym. Po każdej transmisji rejestry indeksowe są automatycznie zwiększane lub zmniejszane w zależności od ustawienia bitu DF w rejestrze znaczników.

Rozkazy operujące na rejestrach WE/WY
Zawierają adres WE/WY (adres natychmiastowy) lub posługują się zawartością rejestru DX (adresowanie pośrednie).

Grupy rozkazów mikroprocesora 8086:

- arytmetyczno-logiczne,
- przesłań,
- skoków, obsługi pętli, wywołań i powrotów z podprogramu,
- dotyczące rejestrów segmentowych,
- wykonujące operacje na ciągach słów,
- wejścia/wyjścia,
- inne.

7	6	5	4	3	2	1	0
kod operacji						D	W
MOD		REG		R/M			

Liczba bajtów każdego rozkazu zależy od jego rodzaju i może wynosić od jednego do sześciu. Pierwszy bajt zawiera sześciobitowy kod operacji oraz dwa bity (kierunku i szerokości).

Bit D określa kierunek transmisji (0 – wynik operacji jest przesyłany z rejestru do pamięci, 1 – z pamięci do rejestru). W zależności od wartości tego bitu w rozkazie rozróżniane są operandy źródłowe i operandy przeznaczenia.

Bit W określa szerokość operandu danego rozkazu (0 – operacje bajtowe, 1 – operacje na słowie 16-bitowym).

Jeżeli rozkaz jest wielobajtowy, to drugi bajt rozkazu określa sposób adresowania argumentów. Zawiera trzy grupy bitów:

- dwubitowa grupa MOD określa tryb adresowania,
- trzybitowa grupa REG określa numer rejestru, w którym znajduje się operand,
- trzybitowa grupa R/M określa sposób wyznaczenia miejsca operandu.

Format rozkazu sześciobajtowego:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOD		REG			R/M			kod operacji						D	W
przemieszczenie															
dana															

Dwa ostatnie bajty to argument natychmiastowy dla tego rozkazu. Są także rozkazy jednobajtowe, np. rozkaz wymiany zawartości akumulatora z zawartością wybranego rejestru. W kodzie tego rozkazu pięć bitów stanowi kod operacji, a pozostałe trzy bity wskazują numer rejestru, którego ten rozkaz dotyczy.

Przykład prostego kodu

```
                        ; _memcpy(dst, src, len)
                        ; Copy a block of memory from one location to another.
                        ;
                        ; Entry stack parameters
                        ;       [BP+6] = len, Number of bytes to copy
                        ;       [BP+4] = src, Address of source data block
                        ;       [BP+2] = dst, Address of target data block
                        ;
                        ; Return registers
                        ;       AX = Zero

0000:1000                       org     1000h       ; Start at 0000:1000h

0000:1000               _memcpy proc
0000:1000 55                    push    bp          ; Set up the call frame
0000:1001 89 E5                 mov     bp,sp
0000:1003 06                    push    es          ; Save ES
0000:1004 8B 4E 06              mov     cx,[bp+6]   ; Set CX = len
0000:1007 E3 11                 jcxz    done        ; If len = 0, return
0000:1009 8B 76 04              mov     si,[bp+4]   ; Set SI = src
0000:100C 8B 7E 02              mov     di,[bp+2]   ; Set DI = dst
0000:100F 1E                    push    ds          ; Set ES = DS
0000:1010 07                    pop     es

0000:1011 8A 04         loop    mov     al,[si]     ; Load AL from [src]
0000:1013 88 05                 mov     [di],al     ; Store AL to [dst]
0000:1015 46                    inc     si          ; Increment src
0000:1016 47                    inc     di          ; Increment dst
0000:1017 49                    dec     cx          ; Decrement len
0000:1018 75 F7                 jnz     loop        ; Repeat the loop

0000:101A 07            done    pop     es          ; Restore ES
0000:101B 5D                    pop     bp          ; Restore previous call frame
0000:101C 29 C0                 sub     ax,ax       ; Set AX = 0
0000:101E C3                    ret                 ; Return
0000:101F                       end proc
```

Jeszcze prostszy kod

Kopiowane jest po jednym bajcie na raz. Poprzedni kod może być zastąpiony przez:

```
0000:1011 FC                    cld                 ; Copy towards higher addresses
0000:1012 F3            loop    rep                 ; Repeat until CX = 0
0000:1013 A4                    movsb               ; Move the data block
```

Procesor 8086 zapewnia dedykowane instrukcje do kopiowania ciągów bajtów. Instrukcje te zakładają, że dane źródłowe są przechowywane w DS:SI, dane docelowe są przechowywane w ES:DI, a liczba elementów do skopiowania jest przechowywana w CX.

Poniższa procedura wymaga, aby blok źródłowy i docelowy znajdowały się w tym samym segmencie, dlatego DS jest kopiowany do ES.

Jeszcze bardziej prosty kod
Można też REP MOVSB w jednej linii. Instrukcja REP powoduje, że następujący MOVSB powtarza się, aż CX wynosi zero, automatycznie zwiększając SI i DI oraz zmniejszając CX w miarę powtarzania. Alternatywnie, instrukcja MOVSW może być użyta do jednoczesnego kopiowania 16-bitowych słów (podwójne bajty); w takim przypadku CX zlicza liczbę skopiowanych słów zamiast liczby bajtów.

Ale te proste kody muszą być przetłumaczone na język maszynowy bezpośrednio do wykonania. Jest to robione w mikrokodzie procesora. W początkach rozwoju mikroprocesorów pamięć RAM była niewielka i bardzo droga; stąd też dążenie do zapisywania kodu w minimalnej ilości pamięci.

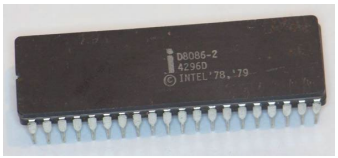
Czasy wykonania typowych instrukcji (w cyklach zegara)
EA = czas na obliczenie efektywnego adresu, w zakresie od 5 do 12 cykli. ALU to elementarne instrukcje arytmetyczno-logiczne

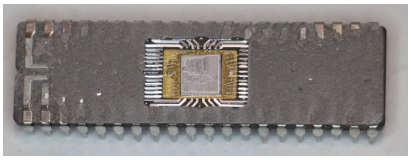
instruction	register-register	register immediate	register-memory	memory-register	memory-immediate
mov	2	4	8+EA	9+EA	10+EA
ALU	3	4	9+EA,	16+EA,	17+EA
jump	*register* ≥ 11 ; *label* ≥ 15 ; *condition,label* ≥ 16				
integer multiply	70~160 (depending on operand *data* as well as size) *including* any EA				
integer divide	80~190 (depending on operand *data* as well as size) *including* any EA				

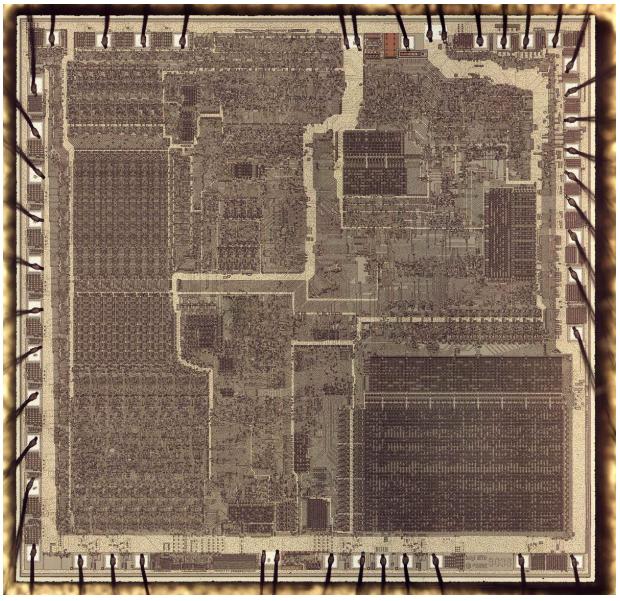
6.10 Co jest w środku procesora 8086

Na podstawie źródła http://www.righto.com/2020/06/a-look-at-die-of-8086-processor.html

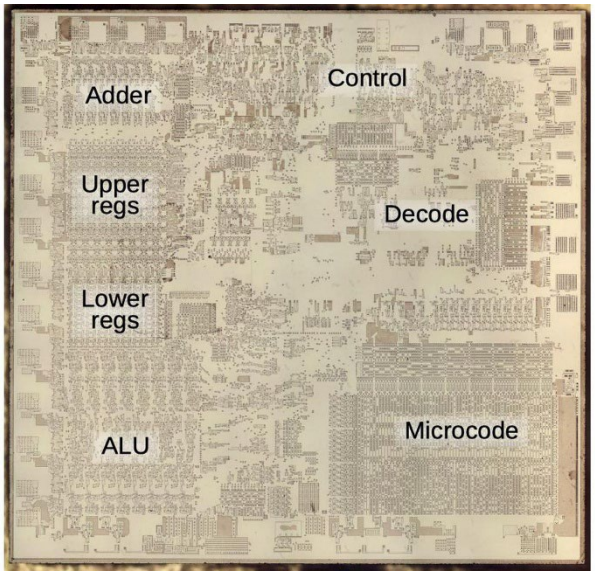
Po zdjęciu obudowy silikonowa matryca jest widoczna pośrodku.
Matryca jest połączona z metalowymi pinami chipa za pomocą maleńkich drutów łączących. Jest to 40-pinowy pakiet DIP, standardowe opakowanie dla mikroprocesorów w tamtym czasie. Sama matryca krzemowa zajmuje niewielką część rozmiaru chipa.
Pod mikroskopem widoczny jest numer katalogowy 8086 oraz data praw autorskich. Przewód łączący jest podłączony do podkładki. Część pamięci ROM z mikrokodem znajduje się na górze.







Wyżej widoczna jest metalowa warstwa chipa, w większości zasłaniająca znajdujący się pod nią krzem. Wokół krawędzi matrycy cienkie druty łączące zapewniają połączenia między padami na chipie a zewnętrznymi pinami. Podkładki zasilania i uziemienia mają po dwa przewody łączące, aby wspierać wyższy prąd. Układ był złożony jak na swoje czasy i zawierał 29 tys. tranzystorów.



Warstwy metalu i polikrzemu zostały usunięte, ukazując leżący pod spodem krzem z 29 tys. tranzystorów. Etykiety pokazują główne bloki funkcjonalne, oparte na tzw. inżynierii odwrotnej. Lewa strona układu zawiera 16-bitową ścieżkę danych: rejestry układu i obwody arytmetyczne.

Rejestry sumujące i rejestry górne tworzą Jednostkę Interfejsu Magistrali, która komunikuje się z pamięcią zewnętrzną RAM, podczas gdy rejestry dolne i ALU tworzą Jednostkę Wykonawczą, przetwa-

rzającą dane. Po prawej stronie chipa znajdują się obwody sterujące i dekodowanie instrukcji, a także mikrokod ROM, który kontroluje każdą instrukcję. Ważną cechą 8086 było pobieranie instrukcji z wyprzedzeniem, co poprawiało wydajność poprzez pobieranie instrukcji z pamięci, zanim były potrzebne.
Zostało to zaimplementowane przez moduł interfejsu magistrali w lewym górnym rogu, który miał dostęp do pamięci zewnętrznej.

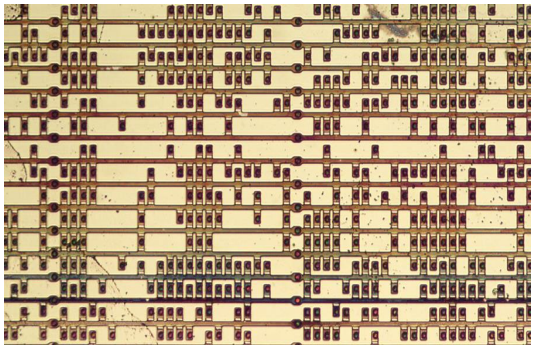
Górne rejestry obejmują niesławne rejestry segmentowe 8086, które zapewniały dostęp do większej przestrzeni adresowej niż 64 kilobajty dozwolone przez 16-bitowy adres. Dla każdego dostępu do pamięci dodano rejestr segmentowy i przesunięcie pamięci, aby utworzyć ostateczny adres pamięci.
Aby poprawić wydajność, 8086 miał oddzielny sumator (Adder) do tych obliczeń adresów pamięci, zamiast używać jednostki ALU. Górne rejestry zawierają również sześć bajtów bufora wstępnego pobierania instrukcji i licznik programu.

W lewym dolnym rogu chipa znajduje się Jednostka Wykonawcza, która wykonuje operacje na danych. Niższe rejestry obejmują rejestry ogólnego przeznaczenia i rejestry indeksowe, takie jak wskaźnik stosu. 16-bitowa jednostka ALU wykonuje operacje arytmetyczne (dodawanie i odejmowanie), operacje logiczne i przesunięcia. W ALU nie ma mnożenia ani dzielenia; operacje te są wykonywane przez sekwencję przesunięć i dodawania/odejmowania, więc są stosunkowo powolne.

Microcode
Jedną z najtrudniejszych części projektowania komputera jest stworzenie logiki sterującej, która mówi każdej części procesora, co zrobić, aby wykonać instrukcję.

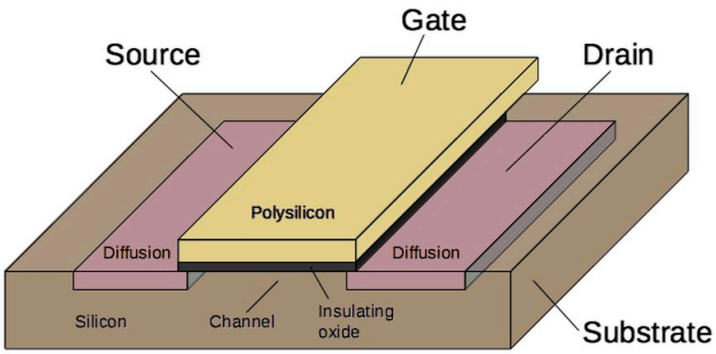
W 1951 roku Maurice Wilkes wpadł na pomysł mikrokodu: zamiast budować logikę sterowania ze złożonych obwodów bramki logicznej, logikę sterowania można było zastąpić specjalnym kodem, zwanym mikrokodem. Aby wykonać instrukcję, komputer wewnętrznie wykonuje kilka prostszych mikrorozkazów, które są zdefiniowane przez mikrokod. Dzięki mikrokodowi budowanie logiki sterującej procesorem staje się zadaniem programistycznym, a nie zadaniem projektowania logiki układu scalonego.

Mikrokod był powszechny w komputerach typu mainframe w latach 60., ale wczesne mikroprocesory, takie jak 6502 i Z-80, nie wykorzystywały mikrokodu, ponieważ wczesne chipy nie miały nań miejsca. Jednak późniejsze układy, takie jak 8086 i 68000, wykorzystywały mikrokod, korzystając z rosnącej gęstości układów. To pozwoliło 8086 na zaimplementowanie złożonych instrukcji (takich jak mnożenie i kopiowanie ciągów) bez komplikowania obwodów scalonych.



Część mikrokodu ROM widoczna pod mikroskopem. Bity można odczytać na podstawie obecności lub braku tranzystorów w każdej pozycji. Tranzystory to małe białe prostokąty nad i/lub pod każdym ciemnym prostokątem. Ciemne prostokąty to połączenia z poziomymi szynami wyjściowymi w warstwie metalowej.

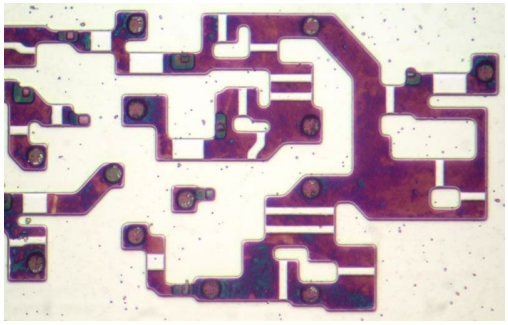
Pamięć ROM składa się z 512 mikroinstrukcji, każda na 21 bitach. Mikroinstrukcja określa przepływ danych między źródłem a miejscem docelowym. Definiuje również mikrooperację, która może być skokiem, operacją ALU, operacją pamięci, wywołaniem podprogramu mikrokodu lub zapisem mikrokodu.
Mikrokod jest dość wydajny; prosta instrukcja, taka jak inkrementacja lub dekrementacja, składa się z dwóch mikroinstrukcji, podczas gdy bardziej złożona instrukcja kopiowania ciągu jest zaimplementowana w ośmiu mikroinstrukcjach.



Struktura tranzystora MOSFET (ang. Metal-Oxide Semiconductor Field-Effect Transistor) w układzie scalonym

Układ 8086 został zbudowany z tranzystorów w technologii 3 µm HMOS (ang. High performance Metal-Oxide Semiconductor). Tranzystor można uznać za przełącznik, kontrolujący przepływ prądu między dwoma obszarami zwanymi źródłem i drenem. Są zbudowane przez domieszkowanie obszarów podłoża krzemowego zanieczyszczeniami w celu wytworzenia obszarów „dyfuzji" o różnych właściwościach elektrycznych. Tranzystor jest aktywowany przez bramkę wykonaną ze specjalnego rodzaju krzemu zwanego polikrzemem, ułożonego warstwowo nad podłożem krzemowym.

Tranzystory są połączone ze sobą metalową warstwą na górze, tworząc kompletny układ scalony. Podczas gdy współczesne procesory mogą mieć kilkanaście warstw metalu, 8086 miał pojedynczą warstwę metalu.



Zdjęcie krzemu (z mikroskopu elektronowego) pokazuje niektóre tranzystory z jednostki arytmetyczno-logicznej (ALU). Domieszkowany, przewodzący krzem ma ciemnofioletowy kolor. Białe paski to miejsce, w którym drut polikrzemowy przecina krzem, tworząc bramkę tranzystora. Można policzyć, że 23 tranzystory tworzą 7 bramek. Tranzystory mają skomplikowane kształty, aby układ był jak najbardziej wydajny. Ponadto mają różne rozmiary, aby zapewnić wyższą moc tam, gdzie jest to potrzebne. Sąsiednie tranzystory mogą współdzielić źródło lub dren, powodując ich połączenie. Kółka to połączenia (tzw. przelotki) pomiędzy warstwą krzemu a metalowym okablowaniem, podczas gdy małe kwadraty to połączenia pomiędzy warstwą krzemu a polikrzemem.

6.11 Historia procesora 8086

Droga do 8086 nie była tak bezpośrednia i zaplanowana, jak można by się spodziewać. Jego poprzednikiem był Datapoint 2200 (patrz obrazek obok z Wikipedii), komputer stacjonarny/terminal z 1970 roku. Datapoint 2200 był jeszcze przed erą mikroprocesorów; wykorzystywał 8-bitowy procesor zbudowany z płyty składającej się z pojedynczych układów scalonych TTL (Transistor-Transistor Logic). Firma Datapoint zwróciła się do Intel oraz do Texas Instruments, czy byłoby możliwe zastąpienie tej płyty chipów pojedynczym chipem.

Kopiując architekturę Datapoint 2200, firma Texas Instruments stworzyła procesor TMX 1795 (1971), a Intel – procesor 8008 (1972). Jednak Datapoint odrzucił te procesory, co było fatalną decyzją. Texas Instruments nie mogła znaleźć klienta na procesor TMX 1795 i porzuciła go, Intel zdecydował się sprzedać 8008 jako produkt, tworząc rynek mikroprocesorów. Intel rozwijał 8008 do procesorów 8080 (1974) i 8085 (1976).

W 1975 roku kolejnym wielkim planem Intela był procesor 8800, zaprojektowany jako główna architektura Intela na lata 80. Ten procesor został nazwany „micromainframe" ze względu na planowaną wysoką wydajność. Miał zupełnie nowy zestaw instrukcji zaprojektowany dla języków wysokiego poziomu, takich jak Ada, oraz obsługiwał programowanie obiektowe i *garbage collection* (usuwanie śmieci) na poziomie sprzętowym. Niestety, ten chip był zbyt ambitny i drastycznie spóźniał się z harmonogramem. Ostatecznie został wprowadzony na rynek w 1981 roku (jako iAPX 432) z kiepską wydajnością i okazał się komercyjną porażką.

Ponieważ projekt iAPX 432 był opóźniony, Intel zdecydował w 1976 roku, że potrzebuje prostego procesora typu stop-gap do sprzedaży, dopóki iAPX 432 nie będzie gotowy. W 1978 roku szybko zaprojektował i zrealizował 8086 jako 16-bitowy procesor, w pewnym stopniu kompatybilny z 8-bitowym 8080. 8086 spowodował wielki przełom (wręcz rewolucję) wraz z wprowadzeniem na rynek komputera osobistego IBM (PC) w 1981 roku. W 1983 roku IBM PC był najlepiej sprzedającym się komputerem i stał się standardem dla komputerów osobistych. Procesor w IBM PC to 8088, wariant 8086 z 8-bitową magistralą. Sukces IBM PC sprawił, że architektura 8086 stała się standardem, który nadal obowiązuje – niemal 50 lat później.

IBM PC 5150 z monochromatycznym monitorem 5151 CC BY-SA 3.0

IBM XT - 4,77 MHz, 128 KB RAM i 10 MB HDD. W Polsce po 1990 r.

Na 16-bitowym mikroprocesorze Intel 80286, 6 MHz, później 8 MHz, 16-bitowa szyna ISA.

Komputery kompatybilne (tzw. klony) z tym modelem często zawierały inne procesory i pracowały przy innych częstotliwościach zegara taktującego. Nosiły one nazwy takie jak PC/AT 386 lub PC/AT 286 12 MHz.

Dlaczego IBM PC wybrał procesor Intel 8088?

Według dr Davida Bradleya, jednego z pierwszych inżynierów IBM PC, kluczowym czynnikiem była znajomość (przez zespół IBM) oprogramowania Intela dla jego procesorów. Użyli procesora Intel 8085 we wcześniejszym komputerze stacjonarnym IBM Datamaster.

Inny inżynier, Lewis Eggebrecht, powiedział, że Motorola 68000 była godnym konkurentem, ale jej 16-bitowa magistrala danych znacznie podniosłaby koszty (jak w przypadku 8086). Podkreślił także lepsze wsparcie i narzędzia programistyczne firmy Intel. W każdym razie decyzja o zastosowaniu procesora 8088 ugruntowała sukces rodziny x86.

IBM PC AT (1984) został zaktualizowany do kompatybilnego, ale mocniejszego procesora 80286.

W 1985 r. linia x86 przeszła na 32-bitową wersję 80386, a następnie 64-bitową w 2003 r. z architekturą Opteron firmy AMD.

Architektura x86 jest wciąż rozszerzana o funkcje, takie jak operacje wektorowe AVX--512 (2016). Ale mimo wszystkich tych zmian zachowuje zgodność z oryginalnym 8086.

Procesor 8086 był pomyślany jako tymczasowy procesor typu „stop-gap”, dopóki Intel nie wypuścił swojego flagowego układu iAPX 432, i był potomkiem procesora zbudowanego z płyty pełnej układów TTL. Jednak od tych skromnych początków architektura 8086 (x86) niespodziewanie zdominowała komputery stacjonarne i serwerowe aż do chwili obecnej. Intel 8086 jest oparty na CISC.

Jest jeszcze architektura RISC!

Rozdział 7. ISA – CISC – RISC

7.1 ISA – Instruction Set Architecture

Na podstawie https://pl.wikipedia.org/wiki/Lista_rozkaz%C3%B3w_procesora

Architektura zestawu instrukcji procesora (ang. *instruction set architecture*, **ISA**) to **model programowy procesora**. Jest to ogólna definicja (specyfikacja) dotycząca organizacji, funkcjonalności i zasad działania procesora, widocznych z punktu widzenia programisty jako dostępne mechanizmy programowania.
Na model programowy procesora składają się między innymi:

- lista rozkazów procesora,
- typy danych,
- dostępne tryby adresowania,
- zestaw rejestrów dostępnych dla programisty,
- zasady obsługi wyjątków i przerwań.

Procesory posiadające ten sam model programowy są ze sobą kompatybilne, co oznacza, że mogą wykonywać te same programy i generować te same rezultaty. W początkowej historii procesorów model programowy procesora zależał od fizycznej implementacji procesora i niejednokrotnie całkowicie z niej wynikał. Obecnie tendencja jest odwrotna i stosuje się bardzo różne implementacje fizyczne (mikroarchitektury) pochodzące od różnych producentów, natomiast realizujące ten sam ISA. ISA to również abstrakcyjny model komputera, zwany również architekturą komputera.

Realizacja ISA nazywana jest implementacją. ISA pozwala na wiele implementacji, które mogą różnić się wydajnością, rozmiarem fizycznym i kosztami (między innymi), ponieważ ISA służy jako interfejs między oprogramowaniem a sprzętem. Oprogramowanie napisane dla ISA może działać w różnych implementacjach tego samego ISA. Umożliwiło to łatwe osiągnięcie zgodności binarnej między różnymi generacjami komputerów oraz rozwój rodzin komputerów.

Oba te rozwiązania pomogły obniżyć koszty komputerów i zwiększyć ich zastosowanie. Z tych powodów ISA jest jedną z najważniejszych abstrakcji w dzisiejszej informatyce. ISA definiuje wszystko, co programista w języku maszynowym musi wiedzieć, aby zaprogramować komputer.

7.2 CISC – Complex Instruction Set Computing

CISC to typ architektury zestawu instrukcji procesora o następujących cechach:

- występowanie złożonych, specjalistycznych rozkazów (instrukcji), które do wykonania wymagają od kilku do kilkunastu cykli zegara, oraz
- szeroka gama trybów adresowania.

W przeciwieństwie do architektury RISC (patrz następny podrozdział), rozkazy mogą operować bezpośrednio na pamięci (zamiast przesłania wartości do rejestrów i operowania na nich). Powyższe założenia powodują, iż dekoder rozkazów jest skomplikowany. Istotą architektury CISC jest to, iż pojedynczy rozkaz mikroprocesora wykonuje kilka operacji niskiego poziomu, jak na przykład pobranie z pamięci, operację arytmetyczną i zapisanie do pamięci.

Przed powstaniem procesorów RISC wielu architektów komputerowych próbowało zaprojektować zestawy rozkazów, które wspierałyby języki programowania wysokiego poziomu przez dostarczenie rozkazów wysokiego poziomu np. wywołania funkcji i zwrócenia jej wartości, instrukcje pętli czy kompleksowe tryby adresowania. Rezultatem tego były programy o mniejszym rozmiarze i z mniejszą ilością odwołań do drogiej (w tym czasie) i niewielkiej pamięci, co w tamtym czasie było istotne z punktu widzenia wydajności, przy jednoczesnym dążeniu do obniżenia kosztów pojedynczego komputera.

Przykłady rodzin procesorów o architekturze CISC to:

- IBM System/360,
- VAX,
- PDP-11,
- x86.

Współczesne procesory zgodne z x86, produkowane przez firmy Intel, AMD i VIA, przetwarzają rozkazy procesora x86 na proste mikropolecenia pracujące według idei RISC, często wykonywane równolegle.

Zanim podejście RISC stało się atrakcyjne, wielu architektów komputerowych próbowało wypełnić tak zwaną lukę semantyczną, tj. zaprojektować zestawy instrukcji, które bezpośrednio obsługują konstrukcje programowania wysokiego poziomu, takie jak wywołania procedur, sterowanie pętlami i złożone tryby adresowania, tworzenie struktur danych oraz dostępy do tablic, które można połączyć w pojedyncze instrukcje.

Instrukcje są również zazwyczaj wysoko zakodowane w celu dalszego zwiększenia gęstości kodu.

Kompaktowy charakter takich zestawów instrukcji skutkuje mniejszymi rozmiarami programów i mniejszą liczbą dostępów do pamięci głównej (które były często powolne i drogie), co w tamtych czasach (na początku lat 60. i później) skutkowało ogromnymi oszczędnościami na kosztach pamięci komputera i przechowywania dysków, jak również powodowało szybsze wykonanie. Oznaczało to również dobrą produktywność programowania, nawet w języku asemblerowym, ponieważ języki wysokiego poziomu, takie jak Fortran lub Algol, nie zawsze były dostępne lub odpowiednie.

Mikroprocesory w architekturze CISC są wciąż używane dla pewnych typów krytycznych aplikacji.

7.3 RISC – Reduced Instruction Set Computer

Na podstawie https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html
RISC – komputer z ograniczonym zestawem instrukcji – to rodzaj architektury mikroprocesorowej, która wykorzystuje mały, wysoce zoptymalizowany zestaw instrukcji, a nie bardziej wyspecjalizowany zestaw instrukcji często spotykany w innych typach architektur jak CISC. Pierwsze projekty RISC pochodziły z IBM, Stanford i UC-Berkeley pod koniec lat 70. i na początku lat 80. XX wieku. IBM 801, Stanford MIPS oraz Berkeley RISC 1 i 2 zostały zaprojektowane zgodnie z podobną architekturą, która stała się znana jako RISC.

Niektóre cechy konstrukcyjne były charakterystyczne dla większości procesorów RISC:

- czas wykonania jednego cyklu: procesory RISC mają CPI (zegar na instrukcję) jednego cyklu. Wynika to z optymalizacji każdej instrukcji na procesorze i techniki zwanej *pipelining*;
- jest to technika pozwalająca na jednoczesne wykonywanie części lub etapów instrukcji w celu wydajniejszego przetwarzania instrukcji;
- duża liczba rejestrów: projektowanie RISC na ogół obejmuje większą liczbę rejestrów, aby zapobiec dużej liczbie interakcji z pamięcią.

Najpierw szczegółowo przyjrzymy się MIPS jako przykładowi wczesnej architektury RISC, aby lepiej zrozumieć cechy i projekt architektury RISC. Następnie omówimy potokowanie, aby zobaczyć korzyści wydajnościowe takiej techniki. Dalej przyjrzymy się zaletom i wadom takiej architektury opartej na architekturze RISC w porównaniu z architekturami CISC. Na koniec omówimy niektóre z ostatnich osiągnięć i przyszłych kierunków rozwoju technologii procesorowej RISC w szczególności oraz technologii procesorowej jako całości.

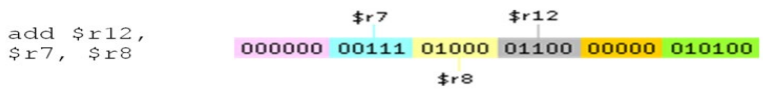
7.3.1 RISC – MIPS

Procesor MIPS (akronim od *Microprocessor without Interlocked Pipeline Stages*) został opracowany w ramach programu badawczego VLSI na Uniwersytecie Stanforda na początku lat 80. Profesor John Hennessy, były rektor tego uniwersytetu, rozpoczął rozwój MIPS od zajęć – burzy mózgów dla doktorantów. Odczyty i sesje pomysłów pomogły w rozpoczęciu rozwoju procesora, który stał się jednym z pierwszych procesorów RISC. IBM i Berkeley rozwijały podobne procesory mniej więcej w tym samym czasie.

Architektura MIPS
Grupa badawcza ze Stanford miała duże doświadczenie w kompilatorach, co doprowadziło ich do opracowania procesora, którego architektura reprezentowałaby obniżenie poziomu kompilatora do poziomu sprzętowego, w przeciwieństwie do podnoszenia sprzętu do poziomu oprogramowania (patrz CISC), co miało miejsce wówczas od dawna w branży sprzętowej. W ten sposób procesor MIPS zaimplementował mniejszy, prostszy zestaw instrukcji. Każda z tych instrukcji działała w jednym cyklu zegara.

W procesorze zastosowano technikę zwaną potokowaniem, aby wydajniej przetwarzać instrukcje. MIPS używał 32 rejestrów, każdy o wielkości 32 bitów (wzorzec bitowy tego rozmiaru jest określany jako słowo).

Zestaw instrukcji MIPS składa się z około 111 instrukcji, z których każda jest reprezentowana w 32 bitach. Przykład instrukcji MIPS znajduje się poniżej.



asembler (po lewej) i binarna (po prawej) reprezentacja instrukcji dodawania MIPS

Instrukcja nakazuje procesorowi obliczenie sumy wartości w rejestrach 7 oraz 8, a następnie zapisanie wyniku w rejestrze 12. Znaki dolara są używane do wskazania operacji na rejestrze. Kolorowa reprezentacja binarna po prawej stronie ilustruje 6 pól instrukcji MIPS. Procesor identyfikuje typ instrukcji za pomocą cyfr binarnych w pierwszym i ostatnim polu. W tym przypadku procesor rozpoznaje, że ta instrukcja jest dodawaniem od zera w jego pierwszym polu i 20 w jego ostatnim polu. Operandy są reprezentowane w niebieskich i żółtych polach, a żądana lokalizacja wyniku jest przedstawiona w czwartym (fioletowym) polu. Pomarańczowe pole reprezentuje wielkość przesunięcia, coś, co nie jest używane w operacji dodawania.

Zestaw instrukcji składa się z różnych podstawowych instrukcji, w tym:

21 instrukcji arytmetycznych (+, -, *, /, %),
8 instrukcji logicznych (&, |, ~),
8 instrukcji manipulacji bitów,
12 instrukcji porównawczych (>, <, =, >=, <=, ¬),
25 instrukcji rozgałęzień/skoków,
15 instrukcji ładowania – pobierania z pamięci,
10 instrukcji do zapisywania w pamięci,
8 instrukcji do kopiowania,
4 inne instrukcje.

Listę podstawowych instrukcji MIPS można znaleźć na stronie:
ftp://ftp.mkp.com/COD2e/Web_Extensions/survey.htm#I.3

MIPS dzisiaj

Firma MIPS Computer Systems Inc. została założona w 1984 roku i korzystała z badań przeprowadzonych na Uniwersytecie w Stanford, gdzie powstał pierwszy chip MIPS. Została zakupiona przez Silicon Graphics Inc. w 1992 roku i wydzielona, jako MIPS Technologies Inc., w 1998 roku. Obecnie MIPS ma zastosowanie w wielu urządzeniach elektroniki użytkowej. Oprócz Johna L. Hennessy'ego oraz Chrisa Rowena należy wymienić też Skipa Strittera, dawniej w Motoroli, oraz Johna Moussourisa poprzednio w IBM.

7.3.2 RISC – ARM

ARM (ang. *Advanced RISC Machine*, pierwotnie *Acorn RISC Machine*) – rodzina architektur (modeli programowych) procesorów 32-bitowych oraz 64-bitowych typu RISC. Różne wersje rdzeni ARM są szeroko stosowane w systemach wbudowanych i systemach o niskim poborze mocy ze względu na ich energooszczędną architekturę. Procesory z architekturą ARM są jednymi z najczęściej stosowanych procesorów na świecie. Używa się ich między innymi w dyskach twardych, telefonach komórkowych, routerach, kalkulatorach, a nawet w zabawkach dziecięcych. Obecnie zajmują one ponad 75% rynku 32-bitowych CPU dla systemów wbudowanych.

Najpopularniejszym projektem ARM był rdzeń ARM7TDMI, szeroko stosowany w telefonach komórkowych. Moc obliczeniowa architektury ARM umożliwia instalacje na procesorze systemu operacyjnego z zaimplementowanymi mechanizmami wielowątkowości, z możliwością wykorzystania zawartego w systemie stosu TCP/IP czy systemu plików (np. FAT32).

Projekt pierwszego procesora ARM powstał w 1983 roku w brytyjskiej firmie Acorn Computers Ltd. jako projekt rozwojowy. Grupa inżynierów kierowana przez Rogera Wilsona i Steve'a Furbera rozpoczęła projektowanie ulepszonej wersji procesora MOS 6502 firmy MOS Technology. Acorn produkował w tym czasie komputery w oparciu o mikroprocesor MOS 6502, więc celem projektu było opracowanie nowego, potężniejszego mikroprocesora programowalnego w podobny sposób.

Pierwsza wersja testowa, nazywana ARM1, opracowana została w 1985 roku, a rok później ukończono wersję produkcyjną ARM2. ARM2 wyposażony był w 32-bitową szynę danych, 26-bitową przestrzeń adresową oraz w szesnaście 32-bitowych rejestrów. Był to w tym czasie najprostszy szeroko stosowany 32-bitowy mikroprocesor, zawierający tylko 30 tysięcy tranzystorów. Prostota wynikała głównie z braku mikrokodu i, jak w większości procesorów w tym czasie, braku cache (pamięci podręcznej). ARM2 miał z tego powodu bardzo niski pobór mocy i jednocześnie szybkość przetwarzania większą od procesora Intel 80286. Następna wersja ARM3 produkowana była z 4 KB cache, co jeszcze bardziej poprawiło wydajność.

W późnych latach osiemdziesiątych XX w. Apple Computer rozpoczęło współpracę z Acorn Computers w projektowaniu nowszej wersji jądra ARM. Projekt był na tyle istotny, że Acorn wydzielił grupę projektową, tworząc w 1990 roku Advanced RISC Machines (ARM Ltd.). Wynikiem tej współpracy był procesor ARM6, udostępniony w roku 1990. Apple użył opartego na ARM6 procesora ARM610 w palmtopie (PDA) o nazwie Apple Newton.

Jądro procesora ARM6 zawiera około 35 tysięcy tranzystorów i jest tylko niewiele większe od jądra ARM2 (30 tysięcy tranzystorów). Dzięki swej prostocie jądro ARM może być łączone z dodatkowymi blokami funkcjonalnymi, tworząc w jednej obudowie mikroprocesor dostosowany do konkretnych wymagań. Jest to możliwe, gdyż podstawą

działalności ARM Ltd. jest sprzedaż licencji na zaprojektowane jądra. Dzięki temu powstały także mikrokontrolery oparte na architekturze ARM.

Digital Equipment Corporation (DEC) zakupiło licencję na architekturę ARM i na jej podstawie zaprojektowało procesor StrongARM. Przy częstotliwości 233 MHz procesor ten pobierał tylko 1 W mocy (najnowsze wersje StrongARM pobierają znacznie mniej). Projekt ten został następnie przejęty przez Intel na podstawie umowy między przedsiębiorstwami. Dla Intela była to szansa na zastąpienie przestarzałej architektury i960 nową architekturą StrongARM. Na podstawie StrongARM Intel zaprojektował bardzo wydajny mikroprocesor o nazwie XScale.

Zgodnie z założeniami architektury RISC rozkazy procesorów ARM są tak skonstruowane, aby wykonywały jedną określoną operację i były przetwarzane w jednym cyklu maszynowym. Interesującą zmianą w stosunku do innych architektur jest użycie 4-bitowego kodu warunkowego na początku każdej instrukcji. Dzięki temu każda instrukcja może być wykonana warunkowo. Ogranicza to dostępną pamięć RAM, na przykład dla instrukcji przeniesień w pamięci, ale z drugiej strony nie ma potrzeby stosowania instrukcji rozgałęzień dla kodu zawierającego wiele prostych instrukcji warunkowych.

Klasycznym przykładem jest implementacja algorytmu Euklidesa wyznaczania największego wspólnego dzielnika. W języku C wygląda to tak:

```
    while (i != j)
    {
       if (i > j)
           i -= j;
       else
           j -= i;
    }
```

W asemblerze procesora ARM będzie miała postać następującej pętli:

```
loop    CMP    Ri, Rj       ; porównaj i z j, ustawiając flagi warunkowe:
                            ;   - "GT" dla (i > j)
                            ;   - "LT" dla (i < j)
                            ;   - "NE" dla (i != j)
        SUBGT  Ri, Ri, Rj   ; jeśli "GT" (większa niż ang. greater than),
wykonaj i := i - j;
        SUBLT  Rj, Rj, Ri   ; jeśli "LT" (mniejsza niż ang. less than),
wykonaj j := j - i;
        BNE    loop         ; jeśli "NE" (nie równe ang. not equal),
wykonaj skok do etykiety 'loop'
```

Inną unikatową cechą zestawu instrukcji procesora ARM jest łączenie operacji przesunięcia i obrotu bitów w rejestrze z instrukcjami arytmetycznymi, logicznymi czy też przesłania danych z rejestru do rejestru. Dzięki temu wyrażenie języka C „a += (j << 2);” może zostać przetłumaczone przez kompilator na pojedynczą instrukcję asemblera.

Przedstawione cechy powodują, że typowy program zawiera mniej linii kodu niż w przypadku innych procesorów RISC. W rezultacie jest mniejsza liczba operacji pobrania/zapisania argumentów instrukcji, więc potokowość jest bardziej efektywna.

Pomimo że procesory ARM są taktowane zegarem o stosunkowo niskiej częstotliwości, są konkurencyjne w stosunku do znacznie bardziej złożonych procesorów.

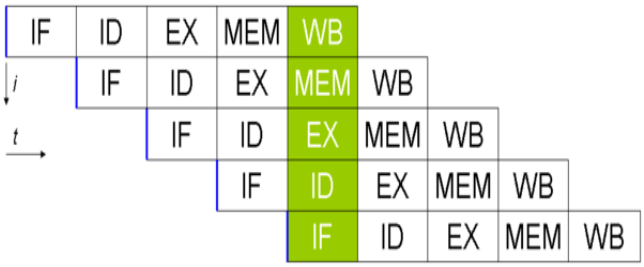
7.3.3 RISC – potokowanie (*pipelininig*)

Pipelining, standardowa funkcja procesorów RISC, przypomina linię montażową. Ponieważ procesor pracuje na różnych etapach instrukcji w tym samym czasie, więcej instrukcji może być wykonanych w krótszym czasie.

Potokowość to technika budowy procesorów polegająca na podziale logiki procesora odpowiedzialnej za proces wykonywania programu (rozkazów procesora) na specjalizowane grupy w taki sposób, aby każda z grup wykonywała część pracy związanej z wykonaniem rozkazu. Grupy te są połączone sekwencyjnie – w potok (ang. *pipe*) – i wykonują pracę równocześnie, pobierając dane od poprzedniego elementu w sekwencji. W każdej z tych grup rozkaz jest na innym stadium wykonania. Można to porównać do taśmy produkcyjnej. W uproszczeniu potok wykonania instrukcji procesora może wyglądać następująco:

Uproszczony schemat potokowości. Części rozkazów oznaczone na zielono wykonywane są równocześnie:

- pobranie instrukcji z pamięci – ang. *instruction fetch* (IF),
- zdekodowanie instrukcji – ang. *instruction decode* (ID),
- wykonanie instrukcji – ang. *execute* (EX),
- dostęp do pamięci – ang. *memory access* (MEM),
- zapisanie wyników działania instrukcji – ang. *store*, *write back* (WB).



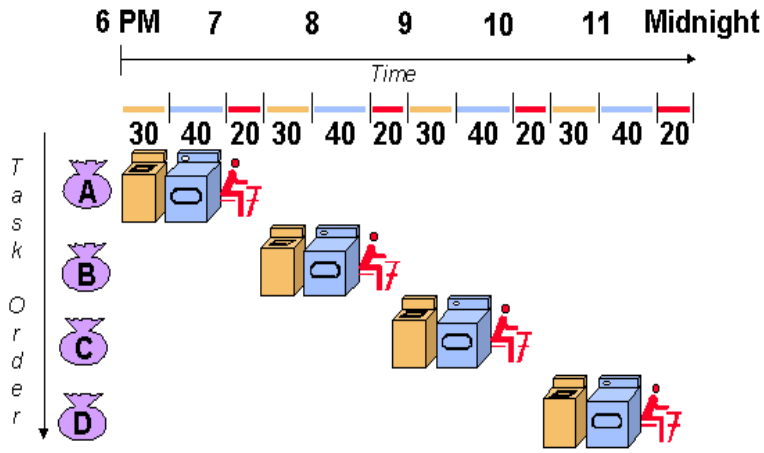
W powyższym pięciostopniowym potoku przejście przez wszystkie stopnie (wykonanie jednej instrukcji) zabiera co najmniej pięć cykli zegarowych. Jednak ze względu na równoczesną pracę wszystkich stopni potoku jednocześnie wykonywanych jest pięć rozkazów procesora, każdy w innym stadium. Oznacza to, że taki procesor w każdym cyklu zegara rozpoczyna i kończy wykonanie jednej instrukcji i statystycznie wykonuje rozkaz w jednym cyklu zegara. Każdy ze stopni potoku wykonuje mniej pracy w porównaniu do pojedynczej sekwencji, dzięki czemu może wykonać ją szybciej – z większą częstotliwością – tak więc dodatkowe zwiększenie liczby stopni umożliwia osiągnięcie coraz wyższych częstotliwości pracy.

Podstawowym mankamentem techniki potoku są rozkazy skoku, powodujące w najgorszym wypadku potrzebę przeczyszczenia całego potoku i wycofania rozkazów, które następowały zaraz po instrukcji skoku, i rozpoczęcie zapełniania potoku od początku od adresu, do którego następował skok. Taki rozkaz skoku może powodować ogromne opóźnienia w wykonywaniu programu – tym większe, im większa jest długość potoku. Dodatkowo szacuje się, że dla modelu programowego x86 taki skok występuje co kilkanaście rozkazów. Z tego powodu niektóre architektury programowe (np. SPARC) zakładały zawsze wykonanie jednego lub większej liczby rozkazów następujących po rozkazie skoku, tak zwany skok opóźniony.

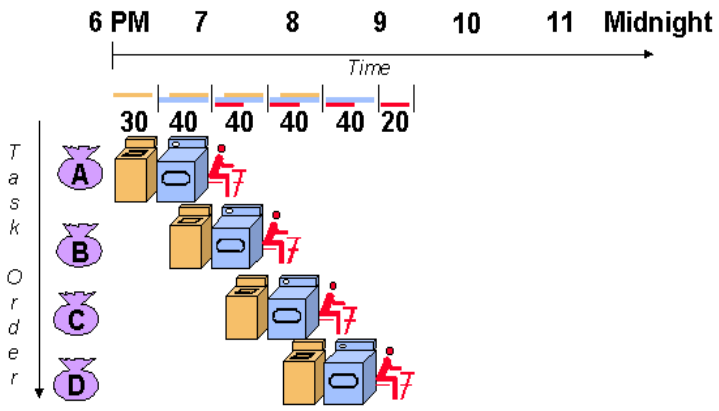
Stosuje się także skomplikowane metody predykcji skoku lub metody programowania bez użycia skoków.

Przykład: ***pranie*** (źródło http://www.ece.arizona.edu/~ece462/Lec03-pipe/)

Załóżmy, że są cztery ładunki brudnej odzieży do prania, które trzeba wyprać, wysuszyć i złożyć. Pierwszy ładunek mogliśmy włożyć do pralki na 30 min, suszyć przez 40 min i złożyć ubrania przez 20 min Następnie weź drugi ładunek i upierz, wysusz, złóż; powtórz to dla trzeciego i czwartego ładunku. Przypuśćmy, że zaczęliśmy o 18.00 i pracowaliśmy tak wydajnie, jak to możliwe, nadal robilibyśmy pranie do północy.



Jednak sprytniejszym podejściem do problemu (patrz rysunek niżej) byłoby włożenie drugiego ładunku brudnego prania do pralki po tym, jak pierwszy był już czysty i szczęśliwie wirował się w suszarce. Następnie, podczas gdy pierwszy ładunek był składany, a drugi ładunek wysychał, trzeci ładunek można było dodać do procesu prania. Przy tej metodzie pranie byłoby gotowe do godziny 9.30 PM (tj. 21.30).



Potoki (*pipelines*) w RISC

Potok procesora RISC działa w bardzo podobny sposób, ale etapy potoku są różne. Chociaż różne procesory mają różną liczbę kroków, są to zasadniczo odmiany tych pięciu, używane w procesorze MIPS R3000:

- pobierz instrukcję z pamięci,
- odczytaj rejestry i dekoduj instrukcję,
- wykonaj instrukcję lub oblicz adres,
- dostęp do operandu w pamięci danych,
- zapisz wynik do rejestru.

Wracając do schematu potoku w pralni, zauważmy, że chociaż pralka kończy pracę w pół godziny, to suszarka zajmuje dodatkowe dziesięć minut, a zatem mokre ubrania muszą czekać dziesięć minut, aż suszarka się zwolni. Zatem długość potoku zależy od długości najdłuższego kroku. Ponieważ instrukcje RISC są prostsze, niż te używane w procesorach pre-RISC (opartych na CISC), więc bardziej sprzyjają potokom.

Podczas gdy instrukcje CISC różnią się długością, wszystkie instrukcje RISC mają tę samą długość i można je pobrać w jednej operacji.

W idealnym przypadku każdy z etapów potoku procesora RISC powinien trwać 1 cykl zegara, tak aby procesor kończył instrukcję w każdym cyklu zegara.

Problemy z potokami

W praktyce jednak procesory RISC działają w więcej niż jednym cyklu na instrukcję. Procesor może czasami zatrzymywać się z powodu zależności danych i instrukcji skoku. Zależność od danych występuje, gdy instrukcja zależy od wyników poprzedniej instrukcji. Instrukcja może wymagać danych w rejestrze, które nie zostały jeszcze tam zapisane, ponieważ jest to zadanie poprzedzającej instrukcji, która nie osiągnęła jeszcze tego kroku w potoku.

Przykład:

```
add     $r3,    $r2,    $r1
add     $r5,    $r4,    $r3
```

… dalej więcej instrukcji niezależnych od dwóch pierwszych

Pierwsza instrukcja: dodaj zawartości rejestrów r1 i r2 i zapisz wynik w rejestrze r3.
Druga: dodaj r3 i r4 i zapisz sumę w r5.

Umieszczamy ten zestaw instrukcji w potoku. Gdy druga instrukcja jest w drugim etapie, procesor będzie próbował odczytać r3 i r4 z rejestrów. Pierwsza instrukcja jest tylko o krok przed drugą, więc dodawana jest zawartość r1 i r2, ale wynik nie został jeszcze zapisany do rejestru r3. Dlatego druga instrukcja nie może czytać z rejestru r3, ponieważ nie została jeszcze zapisana i musi czekać, aż dane, których potrzebuje, zostaną zapisane. W konsekwencji potok zostaje zatrzymany i pewna liczba pustych instrukcji (znanych jako bąbelki) trafia do potoku.

Zależność danych wpływa na długie potoki bardziej niż na krótsze, ponieważ potrzeba więcej czasu, zanim instrukcja osiągnie końcowy etap zapisu do rejestru w długim potoku. Rozwiązaniem MIPS dla tego problemu jest zmiana kolejności kodu.

Jeśli, jak w powyższym przykładzie, poniższe instrukcje nie mają nic wspólnego z pierwszymi dwoma, kod może zostać przeorganizowany tak, aby te instrukcje były wykonywane pomiędzy dwoma zależnymi instrukcjami, a potok mógł działać wydajnie. Zadanie zmiany kolejności kodu jest generalnie pozostawione kompilatorowi, który rozpoznaje zależności danych i stara się zminimalizować przerwy.

Instrukcje rozgałęzienia to te, które mówią procesorowi, aby podjął decyzję o tym, jaka kolejna instrukcja ma zostać wykonana na podstawie wyników innej instrukcji. Instrukcje rozgałęzienia mogą być kłopotliwe w potoku, jeśli rozgałęzienie jest uzależnione od wyników instrukcji, która nie zakończyła jeszcze swojej ścieżki w potoku.

Przykład:

```
Loop :
add $r3, $r2, $r1
sub $r6, $r5, $r4
beq $r3, $r6, Loop
```

dodaj r1 i r2 i umieścił wynik w r3; odejmij r4 od r5, przechowując różnicę w r6; beq oznacza skok na warunek (rozgałęzienie) – jeśli r3 i r6 są równe, to procesor powinien skoczyć do etykiety „Loop”. W przeciwnym razie powinien przejść do następnej instrukcji. W tym przykładzie procesor nie może podjąć decyzji, którą gałąź wybrać, ponieważ do rejestrów r3 i r6 nie zostały jeszcze zapisane wartości z poprzednich instrukcji.

Procesor może się zawiesić. Bardziej wyrafinowaną metodą radzenia sobie z instrukcjami rozgałęzień jest przewidywanie rozgałęzień. Procesor zgaduje, którą ścieżkę wybrać – jeśli się pomyli, to wszystko, co zostało zapisane w rejestrach, musi zostać wyczyszczone, a potok musi zostać uruchomiony ponownie z poprawną instrukcją.

Niektóre metody przewidywania gałęzi zależą od zachowania stereotypowego. Gałęzie skierowane do tyłu (pętle) zajmują około 90% czasu i często znajdują się na dole pętli. Z drugiej strony, gałęzie skierowane do przodu są tylko w około 50% przypadków. Dlatego logiczne byłoby, aby procesory zawsze podążały za gałęzią, gdy wskazuje ona wstecz, ale nie – gdy wskazuje do przodu.

Inne metody przewidywania gałęzi są mniej statyczne: procesory korzystające z przewidywania dynamicznego przechowują historię dla każdej gałęzi i używają jej do przewidywania przyszłych gałęzi. Te procesory mają rację w swoich przewidywaniach w 90% przypadków.

Jeszcze inne procesory rezygnują z całej próby przewidywania gałęzi. System RISC/6000 pobiera i rozpoczyna dekodowanie instrukcji z obu stron gałęzi. Kiedy określa, którą gałąź należy zastosować, wysyła odpowiednie instrukcje w dół potoku do wykonania.

Aby procesory były jeszcze szybsze, opracowano różne metody optymalizacji potoków. *Superpipelining* odnosi się do podziału potoku na więcej etapów. Im więcej jest etapów potoku, tym szybszy jest potok, ponieważ każdy etap jest wtedy krótszy. W idealnym przypadku potok z pięcioma etapami powinien być pięć razy szybszy niż procesor bez potoku (lub raczej potok z jednym etapem). Instrukcje są wykonywane z szybkością, z jaką każdy etap jest zakończony, a każdy etap zajmuje jedną piątą czasu, jaki zajmuje instrukcja niepotokowa. W ten sposób procesor z 8-stopniowym potokiem (MIPS R4000) będzie nawet szybszy niż jego 5-stopniowy odpowiednik.

MIPS R4000 dzieli swój potok na więcej części, dzieląc kilka kroków na dwa. Na przykład pobieranie instrukcji odbywa się teraz w dwóch etapach, a nie w jednym. Etapy są przedstawione niżej:

- Instruction Fetch (First Half),
- Instruction Fetch (Second Half),
- Register Fetch,
- Instruction Execute,
- Data Cache Access (First Half),
- Data Cache Access (Second Half),
- Tag Check,
- Write Back.

To samo po polsku:

- pobieranie instrukcji (pierwsza połowa),
- pobieranie instrukcji (druga połowa),
- zarejestruj się, pobierz,
- wykonaj instrukcję,
- dostęp do pamięci podręcznej danych (pierwsza połowa),
- dostęp do pamięci podręcznej danych (druga połowa),
- sprawdzanie tagów,
- zapisz do pamięci podręcznej danych.

7.4 Superskalarność

Źródło https://pl.wikipedia.org/wiki/Superskalarno%C5%9B%C4%87

Superskalarność to jednoczesne wykonywanie kilku rozkazów maszynowych, realizowane poprzez zwielokrotnienie skalarnych CPU. Pierwszym procesorem Intela z rodziny x86 wykorzystującym fragmentaryczną superskalarność był procesor Pentium, który miał dwa CPU, z czego jeden mógł wykonywać tylko proste instrukcje. Wykorzystanie wszystkich CPU zależy od powiązań między kolejnymi instrukcjami, tj. czy następna instrukcja nie ma (jako parametru) wyniku poprzedniej.

Dla przykładu:

```
a = b + 5
c = a + 10
```

nie mogą zostać wykonane równolegle. Wartość c zależy od a wyliczanej wcześniej. Jeśli kod przepiszemy równoważnie:

```
a = b + 5
c = b + 15
```

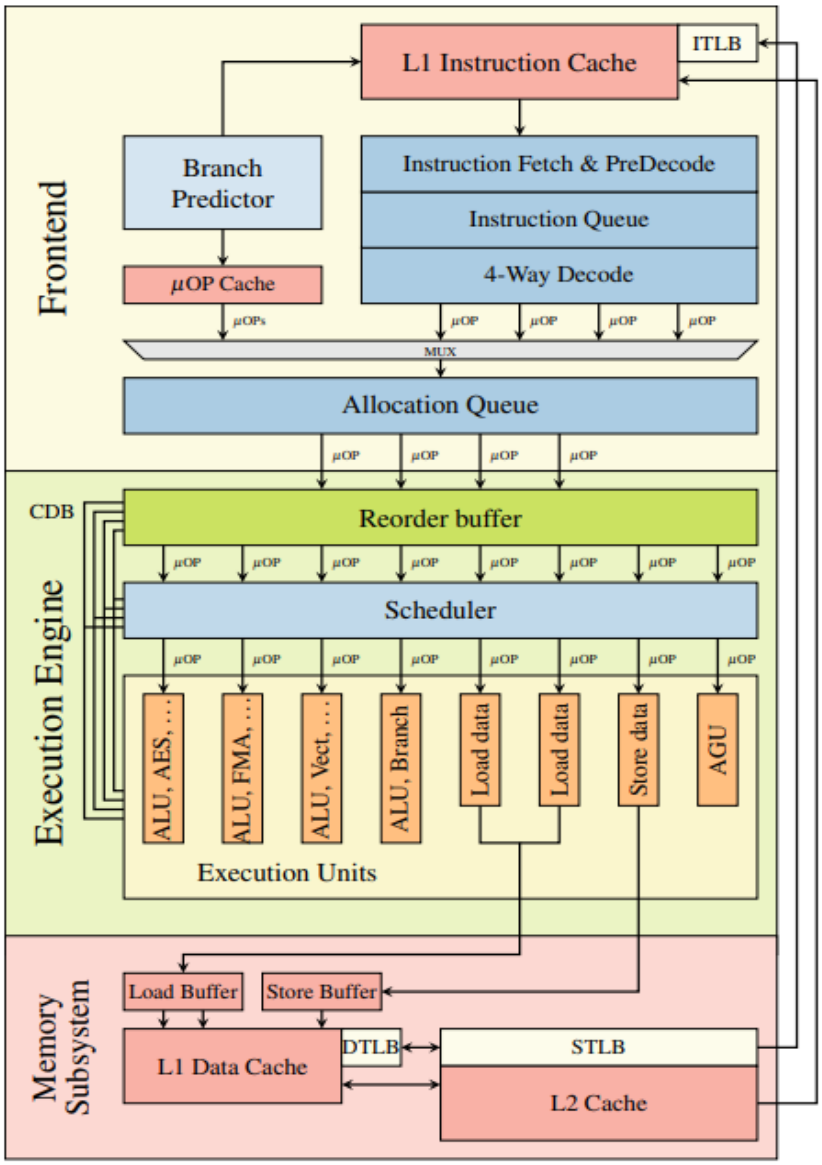
, to superskalarne wykonanie tych instrukcji jest możliwe.

O taki kod powinien zadbać programista oraz/lub kompilator.

CDC 6600 Seymoura Craya z 1964 roku był pierwszym superskalarnym komputerem typu mainframe. IBM System/360 Model 91 z 1967 roku był kolejnym. Następne to: mikroprocesory Motorola MC88100 (1988), Intel i960CA (1989) i AMD 29000 z serii 29050 (1990).

Potoki superskalarne składają się z wielu równoległych potoków. Często jednak superskalarne tworzenie potoków odnosi się do wielu kopii wszystkich etapów potoku (w przykładzie prania oznaczałoby to cztery pralki, cztery suszarki i cztery osoby składające ubrania). Powoduje to znaczne skomplikowanie procesora.

Niżej uproszczona ilustracja pojedynczego rdzenia mikroarchitektury Intel Skylake. Instrukcje są dekodowane na µOP-y i wykonywane poza kolejnością w silniku wykonawczym przez poszczególne jednostki wykonawcze.



7.4.1 Wykonywanie poza kolejnością

Wykonywanie poza kolejnością pozwala maksymalnie wykorzystać wszystkie rdzenie procesora. Zamiast przetwarzać instrukcje ściśle w sekwencyjnej kolejności programu, procesor wykonuje je, gdy tylko dostępne są wszystkie wymagane zasoby. Gdy jeden rdzeń jest zajęty, inne rdzenie mogą wykonywać operacje. Jeśli instrukcje nie korzystają ze wspólnych danych, to mogą być wykonywane równolegle.
Wykonywanie poza kolejnością może być spekulatywne.

7.5 Wykonywanie spekulatywne

https://pl.wikipedia.org/wiki/Wykonywanie_spekulatywne

Wykonywanie spekulatywne to wykonywanie instrukcji znajdujących się już po skoku warunkowym, co do którego jeszcze nie wiadomo, czy nastąpi, a więc czy kolejne instrukcje zostaną kiedykolwiek wykonane. Te wykonania z wyprzedzeniem mogą być albo zatwierdzone, albo odrzucone, w zależności warunku tego skoku.

Przykład w pseudokodzie

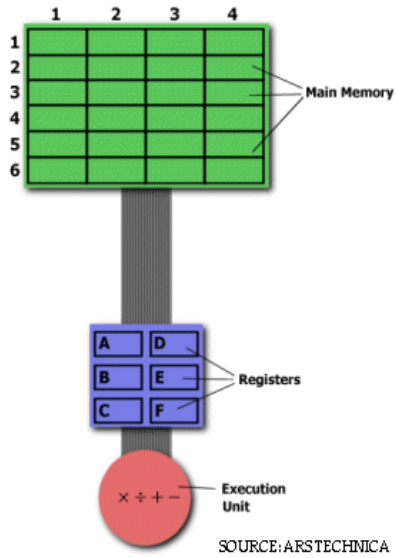
adres xxx: oznacza konkretne miejsce w RAM. Po nim są instrukcje zapisane od tego miejsca.

```
    a := 0
adres 100:
    a := a + 1
    c := c - 1
    ...
    jeśli a < 10 wtedy skok_do_adresu_100
    jeśli c = 20 wtedy skok_do_adresu_200
adres 120:
    b := 0
    ...
adres 200:
    c := 0
```

W tym przykładzie, jeśli jest duże prawdopodobieństwo (wyliczone przez odpowiedni mechanizm) wykonania skoku do adresu 200, to do potoku wykonawczego procesora zostaną wstawione instrukcje od adresu 200. Ale może się okazać, że skoku do adresu 200 nie będzie (c nie jest równe 20). Wtedy ten potok zostaje wyczyszczony, a wyniki anulowane. Następnie wykonywane są instrukcje od adresu 120.

7.6 RISC versus CISC

Najprostszym sposobem zbadania zalet i wad architektury RISC jest porównanie jej z poprzedniczką – architekturą CISC (Complex Instruction Set Computers).



Mnożenie dwóch liczb w pamięci RAM

Obok diagram przedstawiający schemat pamięci dla zwykłego komputera. Pamięć główna podzielona jest na lokalizacje ponumerowane od (wiersz) 1: (kolumna) 1 do (wiersz) 6: (kolumna) 4. Jednostka wykonawcza odpowiada za wykonanie wszystkich obliczeń. Jednak jednostka wykonawcza może działać tylko na danych,

które zostały załadowane do jednego z sześciu rejestrów (A, B, C, D, E lub F). Załóżmy, że chcemy znaleźć iloczyn dwóch liczb — jeden przechowywany w lokalizacji 2:3, a drugi w lokalizacji 5:2 — a następnie przechowywać go z powrotem w lokalizacji 2:3.

7.6.1 CISC

Podstawowym celem architektury CISC jest wykonanie zadania w jak najmniejszej liczbie linii kodu asemblera. Osiąga się to poprzez sprzętową konstrukcję procesora, który jest w stanie zrozumieć i wykonać szereg operacji. Do tego konkretnego zadania procesor CISC byłby przygotowany z określoną instrukcją (nazwiemy ją „MULT"). Przy wykonaniu instrukcja ta ładuje dwie wartości do oddzielnych rejestrów, mnoży operandy w jednostce wykonawczej, a następnie przechowuje produkt w odpowiednim rejestrze. W ten sposób całe zadanie mnożenia dwóch liczb można wykonać za pomocą jednej instrukcji:

MULT 2:3, 5:2

MULT to „złożona instrukcja". Działa bezpośrednio na komórkach pamięci komputera i nie wymaga od programisty jawnego wywoływania funkcji ładowania lub przechowywania. Bardzo przypomina polecenie w języku wyższego poziomu.

Na przykład, jeśli „a" reprezentuje wartość 2:3, zaś „b" reprezentuje wartość 5:2, to polecenie to jest identyczne z poleceniem w języku C

„a = a * b".

Jedną z głównych zalet tego systemu jest to, że kompilator musi wykonać bardzo mało pracy, aby przetłumaczyć instrukcję języka wysokiego poziomu na asembler. Ponieważ długość kodu jest stosunkowo krótka, do przechowywania instrukcji wymagana jest bardzo mała pamięć RAM. Nacisk kładziony jest na budowanie skomplikowanych instrukcji bezpośrednio w sprzęcie.

7.6.2 Podejście RISC

Procesory RISC używają tylko prostych instrukcji, które można wykonać w jednym cyklu zegara. Tak więc opisane powyżej polecenie „MULT" można podzielić na trzy oddzielne polecenia:

„LOAD", które przenosi dane z pamięci do rejestru,
„PROD", które znajduje iloczyn dwóch argumentów znajdujących się w rejestrach, oraz
„ STORE", które przenosi dane z rejestru do pamięci.

Aby wykonać dokładnie serię kroków opisanych w podejściu CISC, programista musiałby zakodować cztery linie asemblera:

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

Na pierwszy rzut oka może się to wydawać znacznie mniej wydajnym sposobem wykonania operacji. Ponieważ jest więcej linii kodu, więcej pamięci RAM jest potrzebne do przechowywania instrukcji poziomu asemblera. Kompilator musi również wykonać więcej pracy, aby przekonwertować instrukcję języka wysokiego poziomu na kod tej postaci.

7.6.3 Porównanie CISC versus RISC

CISC	RISC
Nacisk na sprzęt.	Nacisk na oprogramowanie.
Zawiera multi-zegar.	Pojedynczy zegar.
Złożone instrukcje.	Tylko skrócona instrukcja.
Pamięć do pamięci: "LOAD" oraz "STORE" włączone do instrukcji.	Rejestr do rejestru: "LOAD" i "STORE" są niezależnymi instrukcjami.
Małe rozmiary kodów.	Duże rozmiary kodu.
Wiele cykli zegara na instrukcję.	Instrukcja na cykl.
Tranzystory (*microcode*) używane do przechowywania złożonych instrukcji.	Więcej tranzystorów w rejestrach pamięci.
Wiele operacji wielocyklowych.	Wykonanie jednocyklowe.
Mikrokodowane wielocyklowe operacje.	Sterowanie hardwarowe.
Operacje na pamięci: register-memory oraz memory-memory.	Architektura Load/Store.
Wiele trybów. Wiele formatów i długości operacji.	Tylko kilka trybów adresowania pamięci. Format instrukcji/operacji o stałej długości.
Ręczne składanie kodu, aby uzyskać dobre wykonanie.	Poleganie na kompilatorze w celu optymalizacji wykonania.
Niewielka ilość rejestrów, również specjalnego przeznaczenia	Wiele rejestrów (dobre dla kompilatorów).

Podejście RISC niesie ze sobą również kilka bardzo ważnych korzyści. Ponieważ każda instrukcja wymaga tylko jednego cyklu zegara do wykonania, cały program zostanie wykonany w przybliżeniu w tym samym czasie co wielocyklowe polecenie „MULT”. Te „zredukowane instrukcje” RISC wymagają mniej tranzystorów w chipie procesora niż instrukcje złożone, pozostawiając więcej miejsca na rejestry ogólnego przeznaczenia. Ponieważ wszystkie instrukcje są wykonywane w jednakowym czasie (tj. jeden zegar), możliwe jest polokowanie.

Oddzielenie instrukcji „LOAD” i „STORE” zmniejsza ilość pracy, którą musi wykonać komputer. Po wykonaniu polecenia „MULT” w stylu CISC procesor automatycznie kasuje rejestry. Jeśli jeden z operandów ma być użyty do innego obliczenia, procesor musi ponownie załadować dane z pamięci do rejestru. W RISC operand pozostanie w rejestrze, dopóki w jego miejsce nie zostanie załadowana inna wartość.

CISC próbuje zminimalizować liczbę instrukcji na program kosztem zwiększenia liczby cykli na instrukcję. RISC działa odwrotnie, minimalizuje liczbę cykli (do jednego cyklu) na instrukcję kosztem zwiększenia liczby instrukcji na program.

RISC-V

- ≈ 200 instrukcji po 32 bity,
- 4 formaty,
- wszystkie operandy w rejestrach,
- prawie wszystkie mają po 32 bity,
- ≈ 1 tryb adresowania: Mem[reg + wartość natychmiastowa].

x86 = ClSC

- 1000 instrukcji,
- każda od 1 do 15 bajtów,
- operandy w dedykowanych rejestrach ogólnego przeznaczenia,
- rejestry, pamięć, na stosie, … może mieć 1, 2, 4, 8 bajtów, ze znakiem lub bez,
- 10 trybów adresowania,
- np. Mem[segment + reg + reg * skala + przesunięcie].

7.6.4 Przeszkody na drodze RISC

Pomimo zalet przetwarzania opartego na RISC, chipy RISC czekały ponad dekadę, aby zdobyć przyczółek w świecie komercyjnym. Wynikało to w dużej mierze z braku wsparcia oprogramowania. Chociaż linia Power Macintosh firmy Apple zawierała chipy oparte na RISC, a Windows NT był kompatybilny z RISC, Windows 3.1 i Windows 95 zostały zaprojektowane z myślą o procesorach CISC.

Wiele firm nie chciało zaryzykować nowej technologii RISC. Bez zainteresowania komercyjnego, wytwórcy procesorów nie byli w stanie wyprodukować chipów RISC w wystarczająco dużych ilościach, aby ich cena była konkurencyjna. Kolejną poważną przeszkodą była obecność Intela. Chociaż ich chipy CISC stawały się coraz bardziej nieporęczne i trudne do opracowania, Intel miał zasoby, aby przebić się przez dalszy rozwój i produkować potężne procesory. Chociaż chipy RISC mogły przewyższyć procesory Intela w określonych obszarach, różnice nie były wystarczająco duże, aby przekonać kupujących do zmiany technologii.

Obecnie Intel x86 jest prawdopodobnie jedynym układem, który zachowuje architekturę CISC. Wynika to przede wszystkim z postępu w innych obszarach technologii komputerowej. Cena pamięci RAM drastycznie spadła. W 1977 roku 1 MB pamięci DRAM kosztował około 5 000 USD. Do 1994 roku ta sama ilość pamięci kosztowała tylko 6 USD (po uwzględnieniu inflacji). Kompilatory stały się również bardziej wyrafinowane, dzięki czemu metody wykorzystania pamięci RAM w architekturze RISC i nacisk na oprogramowanie stały się wręcz idealne.

Co zrobiły wtedy firmy AMD i Intel ze swoimi procesorami x86 opartymi na CISC?:
Na podstawie https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext

Wykorzystały one 500-osobowe zespoły projektowe i doskonałą technologię półprzewodnikową, aby zniwelować różnicę w wydajności między procesorami x86 i RISC. Zainspirowani korzyściami wynikającymi z potokowania prostych i złożonych instrukcji, zaprojektowali dekoder instrukcji tłumaczący na bieżąco złożone instrukcje x86 na wewnętrzne mikroinstrukcje typu RISC. Następnie AMD i Intel zrealizowały w swoich procesorach potokowe wykonanie mikroinstrukcji RISC.

Wszystkie pomysły, które projektanci RISC wykorzystywali dla wydajności – oddzielenie pamięci podręcznej (cache) instrukcji i danych, pamięci podręcznej drugiego poziomu na chipie, głębokie potoki oraz pobieranie i wykonywanie kilku instrukcji jednocześnie, mogły zostać włączone do x86.

AMD i Intel dostarczały około 350 milionów mikroprocesorów x86 rocznie w szczytowym momencie ery PC w 2011 roku. Duże wolumeny i niskie marże przemysłu PC oznaczały również niższe ceny niż komputery oparte na RISC.

Biorąc pod uwagę setki milionów komputerów PC (na procesorach x86) sprzedawanych każdego roku na całym świecie, oprogramowanie na x86 stało się gigantycznym rynkiem. Podczas gdy dostawcy oprogramowania dla rynku Unix oferowaliby różne wersje oprogramowania dla różnych komercyjnych RISC ISAsAlpha, HP-PA, MIPS, Power i SPARC, rynek komputerów PC był zdominowany całkowicie przez jedno ISA. Tj. CISC, więc twórcy oprogramowania dostarczali oprogramowanie, które było kompatybilne binarnie tylko z ISA x86. Znacznie większa baza oprogramowania, podobna wydajność i niższe ceny sprawiły, że do 2000 roku procesory x86 zdominowały zarówno rynek komputerów stacjonarnych, jak i małych serwerów.

Apple pomógł zapoczątkować erę post-PC z iPhonem w 2007 roku. Zamiast kupować mikroprocesory, producenci smartfonów zbudowali własne systemy na chipie (SoC) przy użyciu projektów innych firm, w tym procesorów RISC firmy ARM. Projektanci urządzeń mobilnych cenili powierzchnię matrycy i efektywność energetyczną tak samo jak wydajność, co niekorzystnie wpływało na CISC ISA. Co więcej, pojawienie się internetu rzeczy znacznie zwiększyło zarówno liczbę procesorów, jak i wymagania w zakresie rozmiaru matrycy, zużycia energii, kosztów i wydajności.

Tendencja ta zwiększyła znaczenie czasu i kosztów projektowania, co jeszcze bardziej niekorzystnie wpływało na procesory CISC. W dzisiejszej erze post-PC dostawy x86 spadały o prawie 10% rocznie od szczytu w 2011 roku, podczas gdy chipy z procesorami RISC wzrosły do 20 miliardów. Obecnie 99% 32-bitowych i 64-bitowych procesorów to procesory RISC stosowane w systemach wbudowanych i smartfonach.

Podsumowując ten historyczny przegląd, możemy powiedzieć, że rynek rozstrzygnął debatę RISC – CISC. CISC wygrał późne etapy ery PC, ale RISC wygrywa erę post-PC. Od dziesięcioleci nie było nowych certyfikatów CISC ISA. Dzisiaj RISC jest uważany powszechnie za najlepsze ISA dla procesorów ogólnego przeznaczenia, ponad 40 lat po jego wprowadzeniu.

Najnowocześniejsza technologia procesorowa znacznie się zmieniła od czasu wprowadzenia chipów RISC na początku lat 80. Ponieważ wiele ulepszeń jest używanych zarówno przez procesory RISC, jak i CISC, granice między tymi dwiema architekturami zaczęły się zacierać. Wydaje się, że obecnie obie architektury niemalże przyjęły strategie drugiej strony. Ponieważ szybkość procesora wzrosła, układy CISC są teraz w stanie wykonać więcej niż jedną instrukcję w ramach jednego cyklu zegara. Pozwala to również chipom CISC na korzystanie z potoków.

Dzięki innym ulepszeniom technologicznym możliwe jest teraz umieszczenie o wiele więcej tranzystorów na jednym chipie. Daje to procesorom RISC wystarczająco dużo miejsca na wprowadzanie bardziej skomplikowanych poleceń, podobnych do CISC. Chipy RISC wykorzystują również bardziej skomplikowany sprzęt, wykorzystując dodatkowe jednostki funkcyjne do wykonywania superskalarnego.

Wszystkie te czynniki skłoniły niektóre grupy do twierdzenia, że jesteśmy teraz w erze „post-RISC", w której oba style stały się tak podobne, że rozróżnienie między nimi nie ma już znaczenia. Należy jednak zauważyć, że chipy RISC nadal zachowują kilka ważnych cech: ściśle wykorzystują jednolite instrukcje jednocyklowe. Zachowują również architekturę typu rejestr-rejestr, ładowanie/przechowywanie (Load/Store). I pomimo rozszerzonych zestawów instrukcji, chipy RISC nadal mają dużą liczbę rejestrów ogólnego przeznaczenia.

Jednoczesna wielowątkowość (*Simultaneous Multi-Threading – SMT*) stała się ważnym rozwiązaniem; umożliwia jednoczesne wykonywanie wielu wątków. Wątki to szereg zadań wykonywanych naprzemiennie przez procesor. Normalne wykonywanie wątków wymaga włączania i wyłączania wątków na procesorze, ponieważ pojedynczy proces (wątku) dominuje nad procesorem przez chwilę. Pozwala to na bardziej wydajne wykonywanie niektórych zadań, które wymagają oczekiwania (na dostęp do dysku lub użycie sieci). SMT umożliwia jednoczesne wykonywanie wątków, przeciągając instrukcje do potoku z różnych wątków. W ten sposób wiele wątków działa w swoich procesach i żaden wątek nie dominuje nad procesorem w danym momencie.

Przewidywanie wartości jaką wygeneruje dana instrukcja ładowania jest następnym rozwiązaniem. Dane ładowane z pamięci na ogół nie są losowe, a około połowa instrukcji ładowania w programie pobiera tę samą wartość co w poprzednim wykonaniu. Zatem przewidywanie, że wartość obciążenia będzie taka sama jak ostatnio, przyspiesza procesor, ponieważ umożliwia komputerowi kontynuowanie pracy bez konieczności

oczekiwania na dostęp do pamięci. Ponieważ ładowanie jest zwykle jedną z najwolniejszych i najczęściej wykonywanych instrukcji, to ulepszenie powoduje znaczną różnicę w szybkości procesora.

7.6.5 Konwergencja CISC i RISC

Współpraca Intela i RISC-V zmieniła zasady gry. Dzisiaj RISC-V z interesującego podejścia do architektury procesorów staje się architekturą dominującą. Architektura x86 została zaimplementowana w procesorach Intel, Cyrix, AMD, VIA Technologies i wielu innych firmach; istnieją również implementacje otwarte, takie jak platforma Zet SoC (obecnie nieaktywna). Niemniej jednak spośród nich tylko Intel, AMD, VIA Technologies i DM&P Electronics posiadają licencje na architekturę x86, ale tylko dwie pierwsze z nich nadal produkują nowoczesne procesory 64-bitowe.

Od 2021 r. większość sprzedawanych komputerów stacjonarnych, laptopów i konsol do gier (z wyjątkiem Nintendo Switch) opiera się na architekturze x86, podczas gdy urządzenia mobilne, takie jak smartfony lub tablety, są zdominowane przez ARM; x86 nadal dominuje w segmentach stacji roboczych intensywnie korzystających z mocy obliczeniowej i w chmurze obliczeniowej.

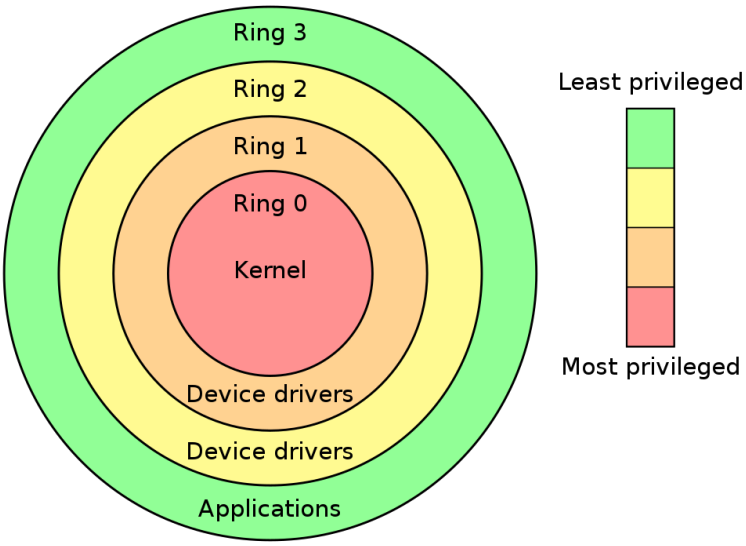
Najszybszy (według TOP500) w 2020 roku superkomputer Fugaku został zbudowany przez Fujitsu i Riken w oparciu o 64-rdzeniowy mikroprocesor Fujitsu A64FX, wykorzystujący architekturę procesora ARM w wersji 8.2A. Trzy kolejne też nie były już oparte na x86.

7.6.6 Przeoczone bezpieczeństwo

W latach siedemdziesiątych XX wieku architekci procesorów skupili się na zwiększeniu bezpieczeństwa komputera za pomocą pierścieni ochronnych. Wiedzieli, że większość błędów będzie znajdować się w oprogramowaniu, ale wierzyli, że wsparcie architektoniczne może pomóc. Funkcje te były w dużej mierze nieużywane przez systemy operacyjne, które celowo koncentrowały się na rzekomo łagodnych środowiskach (takich jak komputery osobiste), a ponieważ wiązały się one wówczas ze znacznym kosztem, więc zostały wyeliminowane.

W społeczności programistów wiele osób uważało, że weryfikacja formalna i techniki, takie jak mikrojądra, zapewnią skuteczne mechanizmy tworzenia wysoce bezpiecznego oprogramowania. Niestety, skala współczesnych systemów informacyjnych i dążenie do wydajności oznaczało, że takie techniki nie nadążały za wydajnością procesora. W rezultacie duże systemy oprogramowania nadal mają wiele luk w zabezpieczeniach, a efekt został spotęgowany przez ogromną i rosnącą ilość danych osobowych w Internecie oraz korzystanie z przetwarzania w chmurze, które udostępnia fizyczny sprzęt potencjalnym włamywaczom.

Na podstawie https://en.wikipedia.org/wiki/Protection_ring#/media/File:Priv_rings.svg



Pierścienie uprawnień dla architektury x86, dostępne w trybie chronionym, zazwyczaj są wymuszane sprzętowo przez niektóre architektury procesorów, które zapewniają różne tryby procesora na poziomie sprzętu lub mikrokodu.

Inżynierowie IT powoli zdawali sobie sprawę z rosnącego znaczenia bezpieczeństwa i zaczęli uwzględniać obsługę sprzętową maszyn wirtualnych i szyfrowanie. Niestety, spekulacyjne wykonania wprowadziły nieznaną, ale istotną lukę w zabezpieczeniach wielu procesorów. W szczególności luki w zabezpieczeniach Meltdown i Spectre doprowadziły do powstania nowych, które wykorzystują luki w mikroarchitekturze, umożliwiając szybki wyciek chronionych informacji niewidocznych na poziomie ISA.

W 2018 roku pokazano, jak wykorzystać jeden z wariantów Spectre do wycieku informacji przez sieć bez wczytywania przez atakującego kodu na docelowy procesor. Chociaż ten atak, zwany NetSpectre, ujawnia informacje powoli, faktem jest, że pozwala on na zaatakowanej maszynie w tej samej sieci lokalnej (lub w tym samym klastrze w chmurze) tworzyć wiele nowych luk w zabezpieczeniach. Następnie zgłoszono dwie kolejne luki w architekturze maszyn wirtualnych.

Jedna z nich, o nazwie Foreshadow, umożliwia penetrację mechanizmów bezpieczeństwa Intel SGX, zaprojektowanych w celu ochrony danych o największym ryzyku (takich jak klucze szyfrowania). Niemal co miesiąc odkrywane są nowe luki w zabezpieczeniach.

Takie sposoby ataku nie są nowe, ale w większości wcześniejszych przypadków to luka w oprogramowaniu pozwalała na powodzenie ataku. W atakach Meltdown, Spectre i innych jest to wada w implementacji sprzętu, która ujawnia chronione informacje.

Istnieje zasadnicza trudność w sposobie, w jaki architekci procesorów określają, jaka jest prawidłowa implementacja ISA, ponieważ standardowa definicja nie mówi nic o skutkach wydajności wykonania sekwencji instrukcji, a jedynie o widocznym przez ISA stanie wykonania.

Architekci muszą przemyśleć definicję prawidłowej implementacji ISA, aby zapobiec takim błędom w zabezpieczeniach. Jednocześnie powinni więcej uwagi poświęcić na bezpieczeństwo komputerów, i współpracę z projektantami oprogramowania w celu wdrożenia bezpieczniejszych systemów.

Ale ciągle (rok 2024) twa wyścig pomiędzy Intelem a AMD o miano najlepszego komputera PC (patrz www.laphard.pl/pl/n/2024/AMD-vs.-Intel-jaki-procesor-wybrac-w-2024-roku/140), również jako elementu klastrów w chmurach. Trzeciego konkurenta po prosu nie ma na rynku.

AMD Ryzen 9 9950X – 16 rdzeni i 32 wątki, taktowanie 4,3 GHz (bazowe) i 5,7 GHz (boost), pamięć cache 80 MB i TDP (Thermal Design Power) 170 W, premiera – sierpień 2024 r. Procesory Intel 15. generacji o nazwie kodowej „Arrow Lake" mają zostać wprowadzone na rynek między październikiem a grudniem 2024 roku i będą konkurować z serią AMD Ryzen 9000.

Ten wyścig powoduje, że wypuszczane są do produkcji coraz bardziej złożone i nie do końca sprawdzone procesory. Błędy (czasami bardzo poważne) pojawiają się w trakcie eksploatacji.

There is no fix for Intel's crashing 13th and 14th Gen CPUs — any damage is permanent. https://www.theverge.com/2024/7/26/24206529/intel-13th-14th-gen-crashing-instability-cpu-voltage-q-a, https://www.pcworld.com/article/2415697/intels-crashing-13th-14th-gen-cpu-nightmare-explained.html

Mój służbowy PC (z AMD Ryzen 7 4800H, Radeon Graphics, 2.90 GHz, 16 GB RAM) od czasu do czasu zawiesza się (czarny ekran) zaraz po starcie.

7.7 Wniosek końcowy

***Najciemniejsza godzina jest tuż przed świtem*.**
Tomasz Fuller, 1650

Twórcy nowych architektur komputerowych powinni zrozumieć, że postępy w zakresie oprogramowania mogą również inspirować ich samych. Podniesienie poziomu abstrakcji interfejsu sprzęt/oprogramowanie daje możliwości dla takich nowych przełomowych architektur. To rynek ostatecznie rozstrzyga debaty na temat architektury komputerowej. Przykłady Intel iAPX 432 oraz Itanium ilustrują, w jaki sposób inwestycje w archi-

tekturę mogą przekroczyć ewentualne zyski, podczas gdy S/360, 8086 i ARM zapewniają wysokie zyski przez dziesięciolecia, a końca jak na razie nie widać.

Nowym podejściem do projektowania ISA dla procesorów ogólnego przeznaczenia staje się RISC, który przetrwał próbę czasu. Ale prawdziwym wyzwaniem jest nie von Neumanowska architektura komputera, postulowana już od co najmniej końca lat 70. przez Johna Backusa.

Spodziewajmy się wiec, że w następnych latach (dekadach?) nastąpi kambryjska eksplozja nowatorskich architektur komputerowych, czyli ekscytujące czasy dla architektów komputerowych zarówno w środowisku akademickim, jak i przemysłowym.

Rozdział 8. Pamięć

Pamięć to moduł komputera służący do przechowywania danych i programów. Razem z procesorem są głównymi elementami architektury komputera.

Pamięć główna – Random Access Memory (RAM) – jest bezpośrednio lub pośrednio połączona z centralną jednostką przetwarzania (CPU) za pośrednictwem magistrali komunikacyjnej, w skład której wchodzą: szyna adresowa, szyna danych i szyna sterująca. W cyklu odczytu pamięci CPU najpierw ustawia szynę adresową, tak by wskazywała żądaną lokalizację (adres) danych w pamięci. Dane są kopiowane na szynę danych, procesor odczytuje je z szyny danych i przekazuje do swoich rejestrów. W trakcie cyklu zapisu do pamięci procesor wstawia dane na szynę i przekazuje informacje o wykonywanej operacji zapisu na szynie sterującej.

Oprócz dużej pamięci głównej RAM są jeszcze dwie: rejestry procesora oraz pamięć podręczna (ang. cache). Rejestry procesora znajdują się wewnątrz procesora. Każdy rejestr zwykle zawiera słowo danych (często 32 lub 64 bity). Rejestry są najszybszymi pamięciami. Pamięć podręczna procesora jest pomiędzy ultraszybkimi rejestrami i znacznie wolniejszą pamięcią RAM. Została wprowadzona do poprawy szybkości komputerów. Dane w RAM, które są najczęściej używane, są kopiowane do pamięci podręcznej, która jest szybsza, ale o znacznie mniejszej wielkości. Podstawowa pamięć podręczna jest niewielka, ale bardzo szybka i znajduje się wewnątrz procesora. Dodatkowa pamięć podręczna jest większa i wolniejsza.

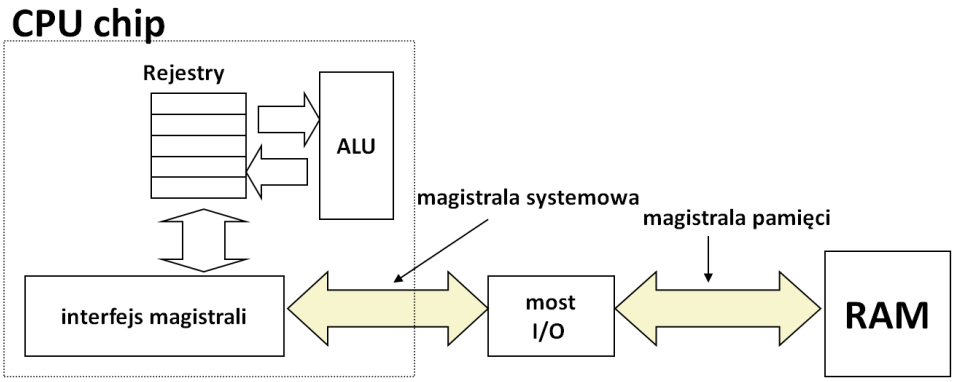
8.1 Pamięć: pisanie i czytanie

Opracowane na podstawie Computer Systems: A Programmer's Perspective, Second Edition, by Randal E. Bryant and David R O'Hallaron.

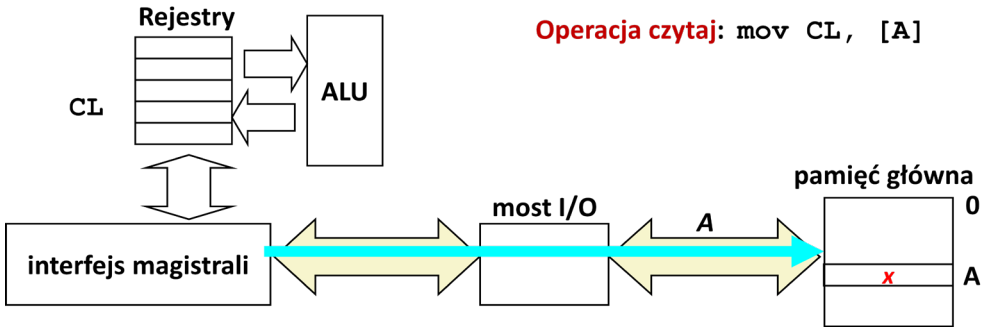
Transfer danych z CPU do pamięci RAM. W asemblerze odpowiada za to instrukcja
mov [CL], AL
tj. zapisz to, co jest w rejestrze **AL**, w komórce pamięci RAM o adresie, który jest w rejestrze **CL**.

Transfer danych z pamięci RAM do CPU. W asemblerze odpowiada za to instrukcja
mov AL, [CL]
tj. pobierz zawartość z komórki pamięci RAM o adresie umieszczonym w rejestrze **CL**, i umieść ją (tę zawartość) w rejestrze **AL**.

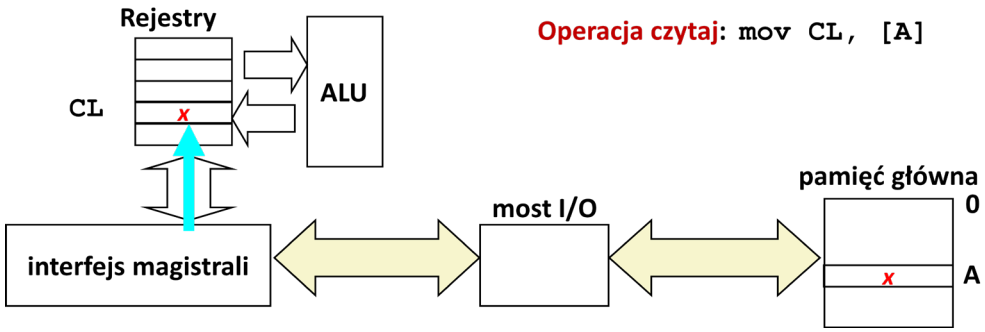
Klasyczna struktura magistrali (ang. bus) łączącej procesor i pamięć jest przedstawiona na poniższym rysunku. Magistrala to zestaw równoległych przewodów, które przenoszą adresy, dane i sygnały sterujące. Magistrale są zwykle wspólne dla wielu urządzeń.



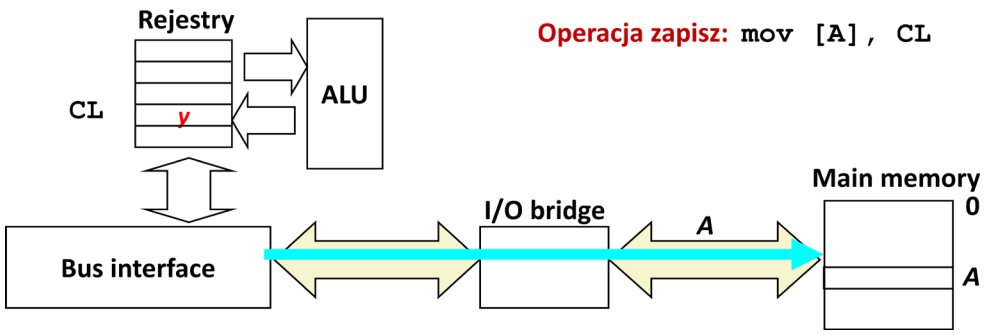
Czytanie z RAM: CPU wstawia adres A do magistrali.



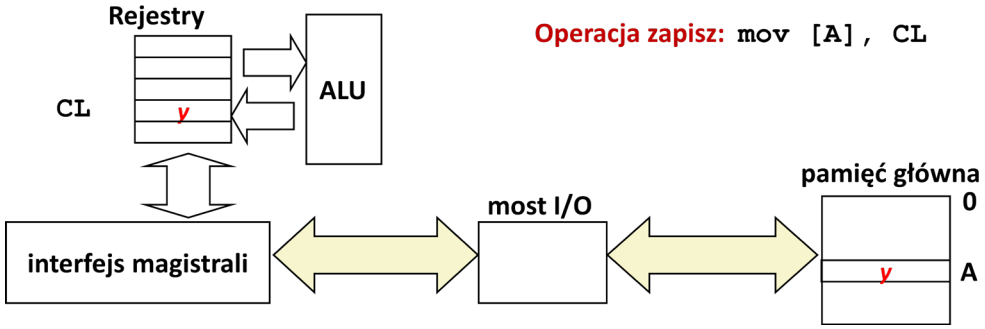
Pamięć główna odczytuje adres A z magistrali pamięci, sięga do komórki pod tym adresem, czyta x i umieszcza je na magistrali danych. CPU czyta x z magistrali danych i kopiuje do rejestru CL



Wpisywanie do RAM: CPU wstawia adres A na magistralę. RAM czyta adres i czeka na dane do zapisania pod tym adresem



CPU wstawia dane z rejestru CL na magistralę.
RAM czyta y z magistrali i zapisuje do komórki o adresie A.



Pamięć RAM jest realizowana jako osobny układ scalony lub jako część procesora. Podstawową jednostką pamięci jest zwykle komórka (jeden bit na komórkę). Razem tworzą pamięć RAM. Występuje w dwóch odmianach:

- SRAM (statyczna pamięć RAM)
  - 6 tranzystorów / bit,
  - utrzymuje stale stan.
- DRAM (dynamiczna pamięć RAM)
  - 1 tranzystor + 1 kondensator / bit,
  - stan musi być cykliczne odświeżany.

	Tranzystory na bit	Czas dostępu	Odświeżanie	EDC	Koszt	Zastosowanie
SRAM	6 lub 8	1x	Nie	Może	100x	pamięć cache
DRAM	1	10x	Tak	Tak	1x	pamięć główna, bufory

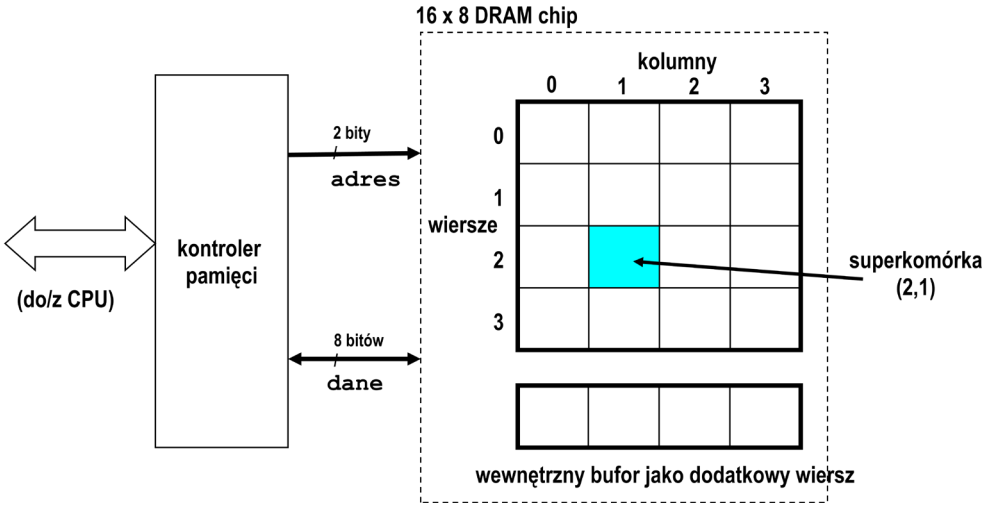
EDC: Error detection and correction (wykrywanie i korekta błędu)

Trendy rozwoju:
SRAM ewoluuje wraz z technologią półprzewodników. Skalowanie pamięci DRAM jest ograniczone potrzebą minimalnej pojemności kondensatorów.
Działanie DRAM nie zmieniło się od czasu jej wynalezienia i komercjalizacji przez Intel w 1970 roku.

Odmiana DRAM z lepszą logiką interfejsu i szybszym We/Wy:
Synchroniczna pamięć DRAM (SDRAM) wykorzystuje konwencjonalny sygnał zegarowy zamiast sterowania asynchronicznego. Powoduje to podwojenie szybkości przesyłania danych. W zależności od wielkości małego bufora pobierania wstępnego występują następujące typy SDRAM: DDR (2 bity), DDR2 (4 bity), DDR3 (8 bitów), DDR4 (16 bitów). Do 2010 roku był to standard dla większości serwerów i komputerów stacjonarnych PC. Intel Core i7 obsługuje DDR3 i DDR4 SDRAM

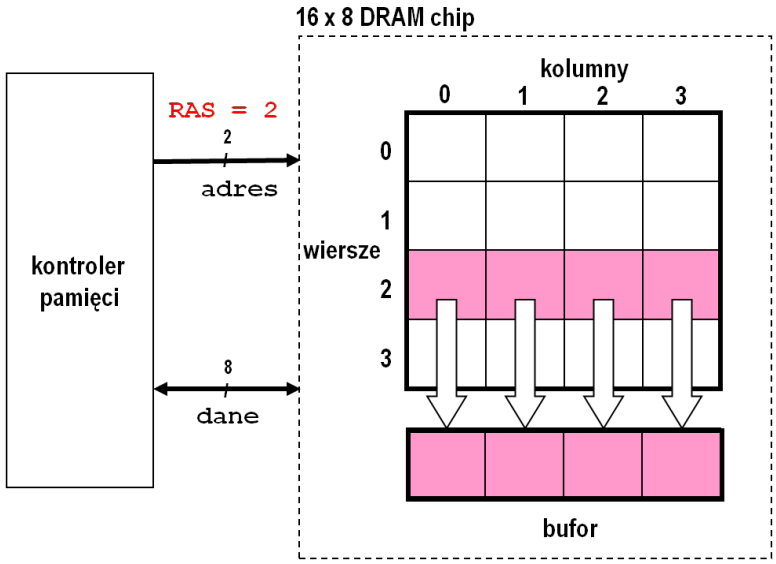
Klasyczna struktura DRAM:

W ***d*** superkomórkach (supercells) po **w** bitów każda; razem ***d*** · ***w*** bitów.



Czytanie: DRAM Supercell (2,1)

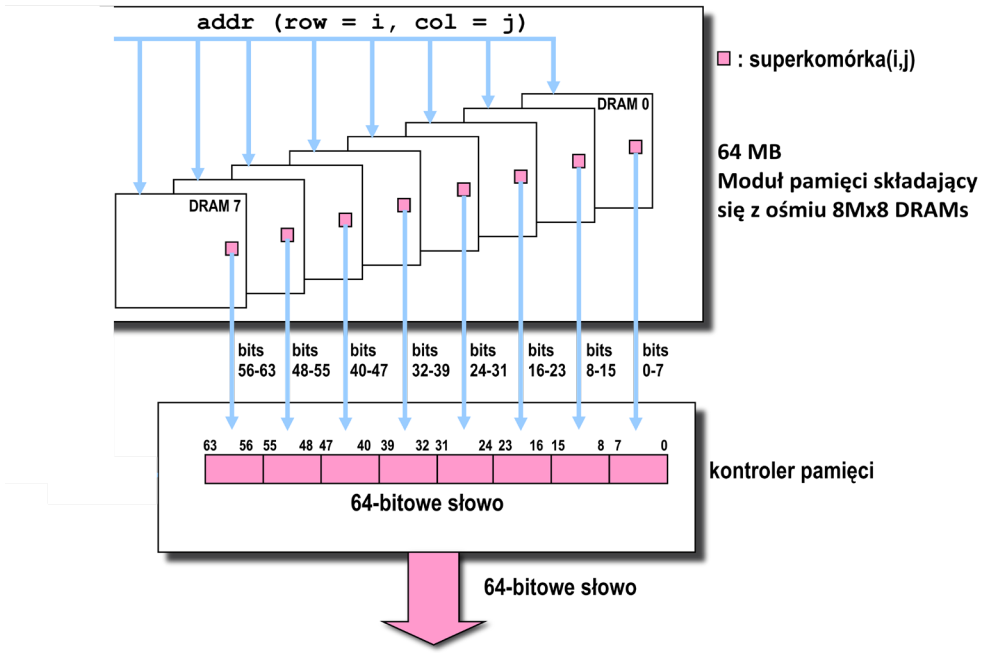
Krok 1: Row access strobe (**RAS**) wybiera wiersz numer 2, który jest następnie kopiowany do bufora.



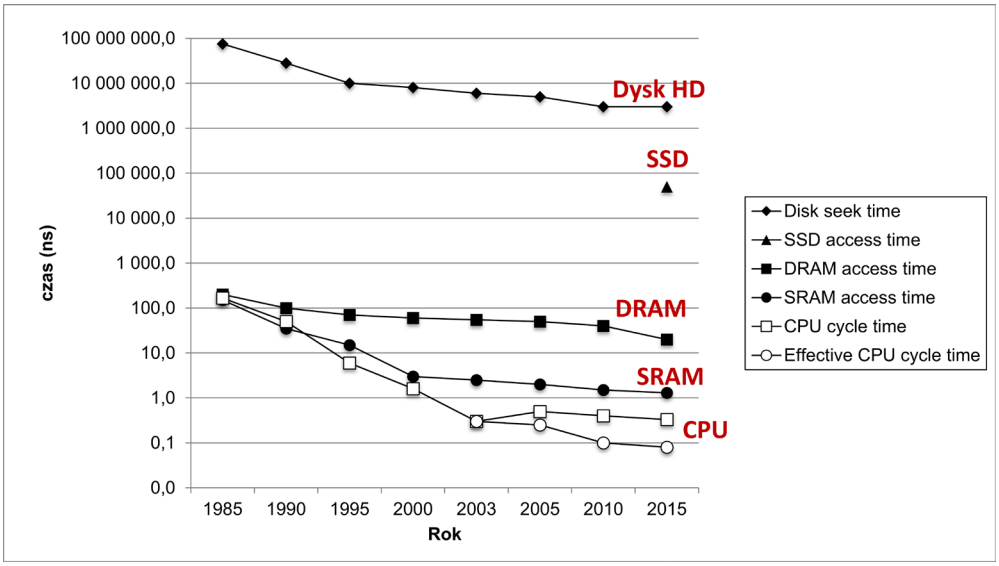
Krok 2: Column access strobe (**CAS**) wybiera kolumnę numer 1 z bufora, która następnie jest kopiowana i przesyłana do kontrolera pamięci (Memory Controller) a potem do CPU.

Krok 3: Dane z bufora odświeżają zawartość wiersza numer 2.

Moduły pamięci



Luka szybkości pomiędzy CPU a pamięcią.



Jak tę lukę minimalizować?

Można próbować na poziomie programowania. Kod jest zapisywany w pamięci. Jeśli ten kod ma własność *lokalności*, to można.

Zasada lokalności: Aplikacja (w trakcie wykonywania instrukcji) używa adresów danych i adresów instrukcji, które są blisko zapisane w pamięci. Jeśli są skoki, to nie powinny być duże.

Czasowa (temporalna) lokalność: Ostatnio używane odnośniki są często używane w kontynuacji wykonania.

Przestrzenna lokalność: Adresy danych i instrukcje zapisane (w pamięci) blisko siebie są często używane w bliskich chwilach czasu.

Przykłady lokalności w kodzie

Referencje do danych:

- do elementów tablicy jeden po drugim. Czyli jest to lokalność przestrzenna;
- do zmiennej **sum** po każdej iteracji – jest to lokalność temporalna.

Referencje do instrukcji:

- według kolejności w kodzie – jest to lokalność przestrzenna;
- cykle w pętli – jest to lokalność temporalna.

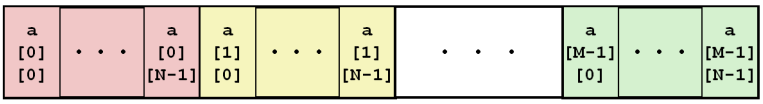
```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Jakościowa ocena lokalności kodu

Fakt: profesjonalny programista potrafi ocenić jakościową lokalność, patrząc na kod.
Pytanie: czy funkcja niżej ma dobrą lokalność z względu na tablicę a?

Odpowiedź: TAK

```
int sum_array_rows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

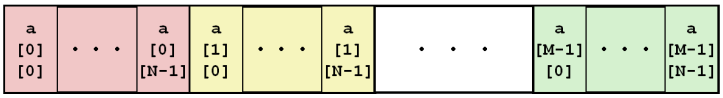


Pytanie: a czy poniższa funkcja też posiada dobrą lokalność ze względu na tablicę?

```
int sum_array_cols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Odpowiedź: Nie, chyba że…

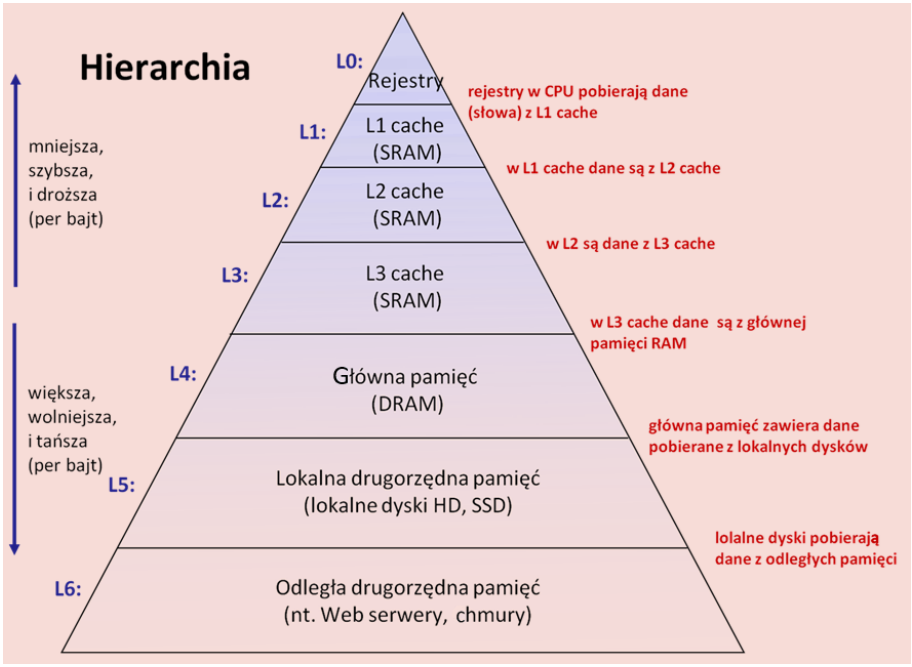
M jest małe



Kluczowe czynniki w rozwoju sprzętu i oprogramowania są następujące. Technologie szybkich pamięci są drogie, a więc dotyczą mniejszych pojemności, wymagają więcej energii, co powoduje wydzielanie więcej ciepła i grzanie.

Luka pomiędzy CPU (wiele rdzeni) a główną pamięcią poszerza się; jest to słynne wąskie gardło architektury von Neumanna. Dobry kod (z lokalnością) pozwała zmniejszyć nieco tę lukę. Konieczna jest hierarchiczna organizacja pamięci.

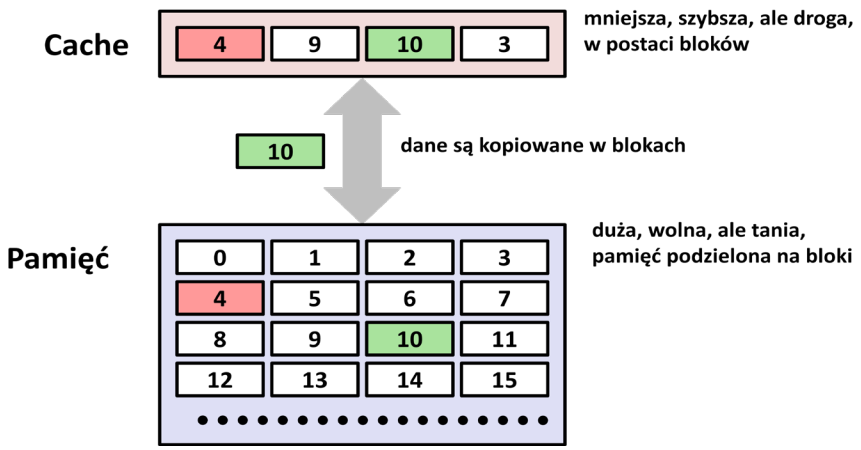
8.2 Hierarchia pamięci



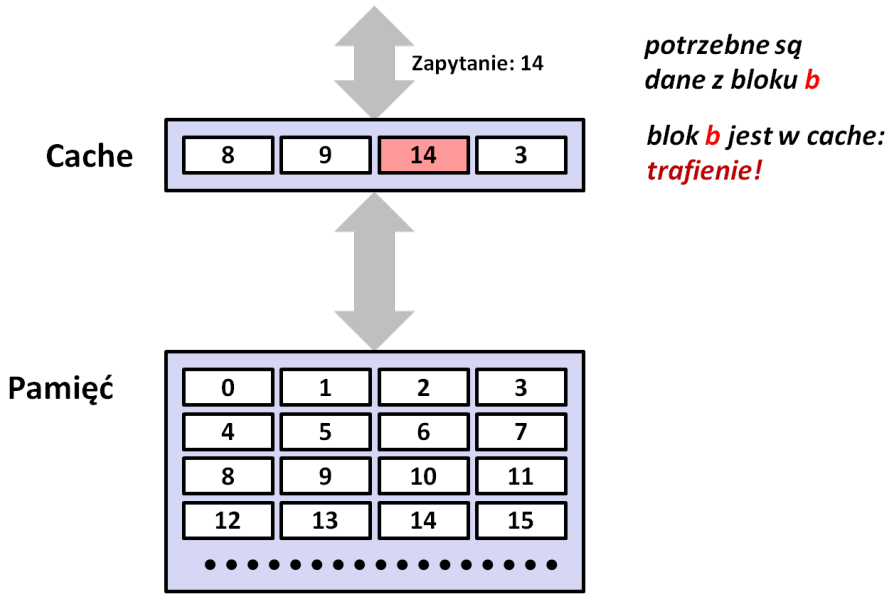
8.3 Cache

Cache: mniejsza, ale szybsza pamięć
Fundamentalna idea: Dla każdego k szybsza i mniejsza pamięć na poziomie k służy jako cache dla większej, ale wolniejszej pamięci na poziomie k+1.

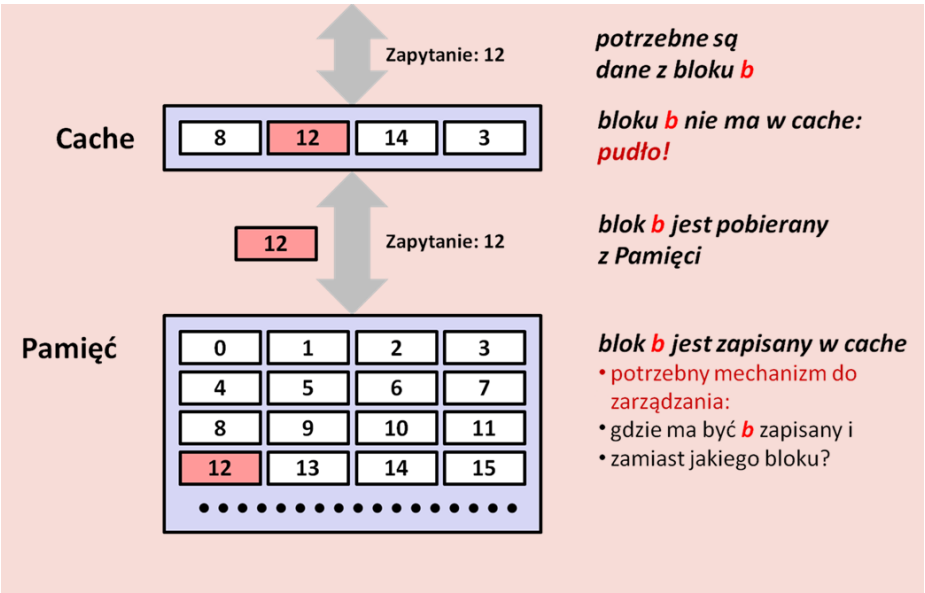
Jaki jest cel? Lokalność kodu powoduje, że instrukcje mogą sięgać po **dane** w cache na poziomie k, chociaż, oryginalnie, mogą być na znacznie głębszym poziomie.
Idealnie: Te **dane** powinny być umieszczone jak najwyżej, tj. jak najbliżej CPU.



Cache : trafienie



Cache: pudło



Cache: 3 typy niepowodzeń

- Przy starcie: na początku cache jest pusty, więc pierwsze zapytania będą nietrafione.
- Za mały cache: potrzebnych bloków jest więcej niż wielkość cache.

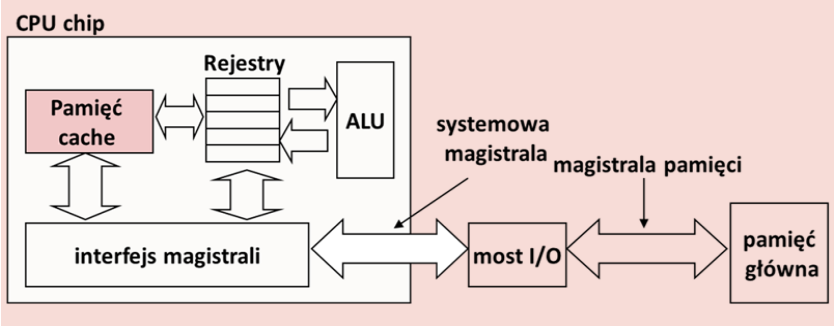
- Konflikt kolejnych zapytań: zazwyczaj bloki na poziomie głębszym są większe niż te na wyższym. Konflikt występuje, jeśli zapytania zmieniają się kolejno, np.:
  - 0 (nietrafione, zamiana 8 na 0),
  - 8 (nietrafione, zamiana 0 na 8),
  - 0 (nietrafione, zamiana 8 na 0),
  - 8 (nietrafione, zamiana 0 na 8).

Translacyjny bufor podręczny (*Translation Lookup Buffor* – TLB) to pamięć podręczna używana do skrócenia czasu potrzebnego na dostęp do lokalizacji w pamięci użytkownika. Jest to część jednostki zarządzania pamięcią (*Memory Management Unit* – MMU) układu.

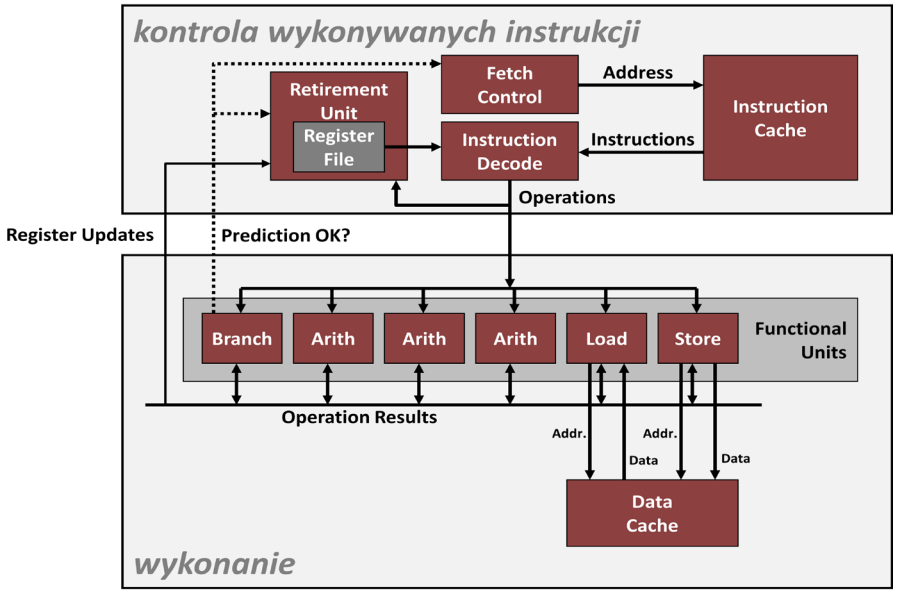
Pamięć cache to mała, szybka pamięć oparta na SRAM, zarządzana automatycznie w hardware. Zawiera bloki z pamięci głównej (RAM) często używane przez aktualnie wykonywany proces (procesy). CPU zwraca się najpierw do cache.

typ cache	rozmiar bloków	zapisane	opóźnienie (w cyklach)	zarządzane przez
Registers	słowa 4-8 bajtowe	CPU core	0	kompilator
TLB	translacja adresów	On-Chip TLB	0	Hardware MMU
L1 cache	bloki 64-bajtowe	On-Chip L1	4	Hardware
L2 cache	bloki 64-bajtowe	On-Chip L2	10	Hardware
Virtual Memory	strony 4-KB	pamięć główna	100	Hardware + OS
Buffer cache	kawałki pliku	pamięć główna	100	OS
Disk cache	sektory dysku	kontroler dysku	100,000	Disk firmware
Network buffer cache	kawałki pliku	Lokalny dysk	10,000,000	NFS client
Browser cache	strony WWW	Lokalny dysk	10,000,000	Web browser
Web cache	strony WWW	odległe serwery dyskowe	1,000,000,000	Web proxy server

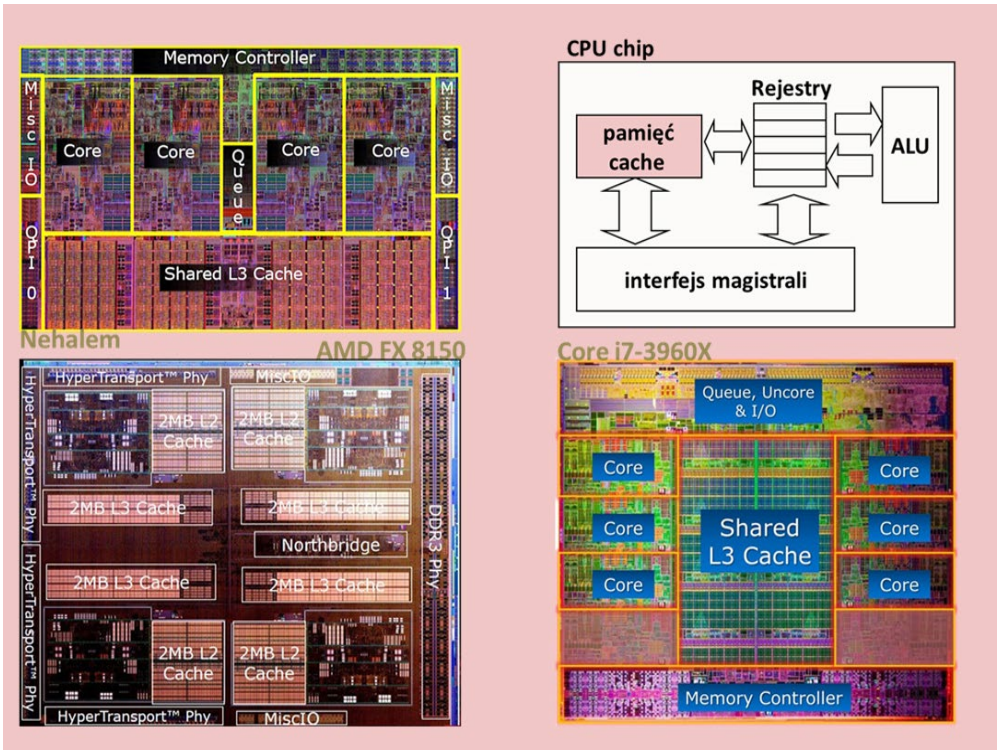
Typowa architektura z pamięcią główną RAM:

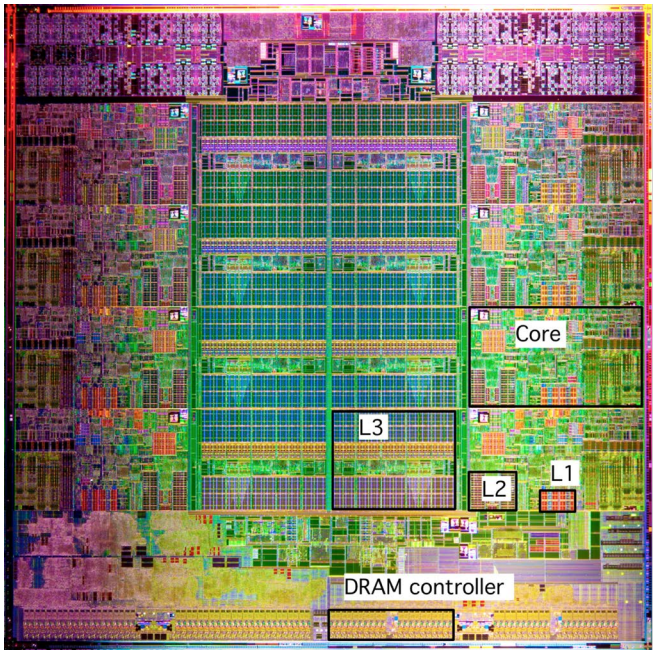


Współczesna architektura CPU:



Jak to jest w czipach?





Intel Sandy Bridge
Processor Die

L1: 32KB Instruction + 32KB Data
L2: 256KB
L3: 3–20MB

8.3 Pamięci trwałe

Dyski magnetyczne, ROM i SSD.

8.3.1 Dyski magnetyczne

Dostęp jest elektromagnetyczny.

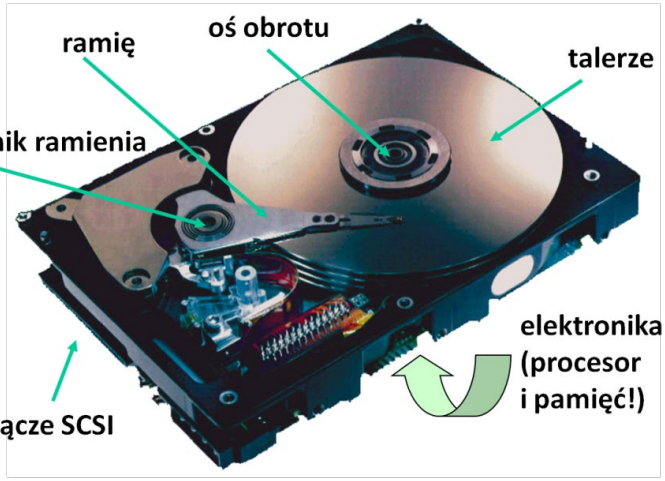
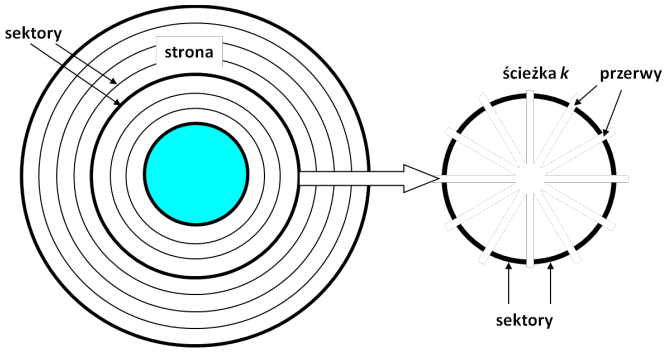


Image courtesy of Seagate Technology

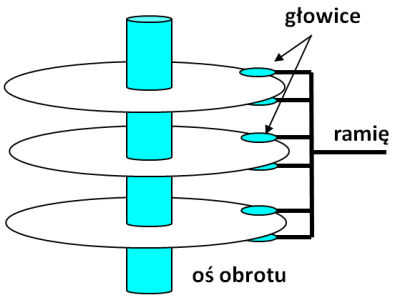
Dysk składa się z talerzy, każdy ma dwie strony. Strona składa się z koncentrycznych pierścieni zwanych ścieżkami (ang. tracks). Ścieżka składa się z sektorów oddzielonych przerwami.



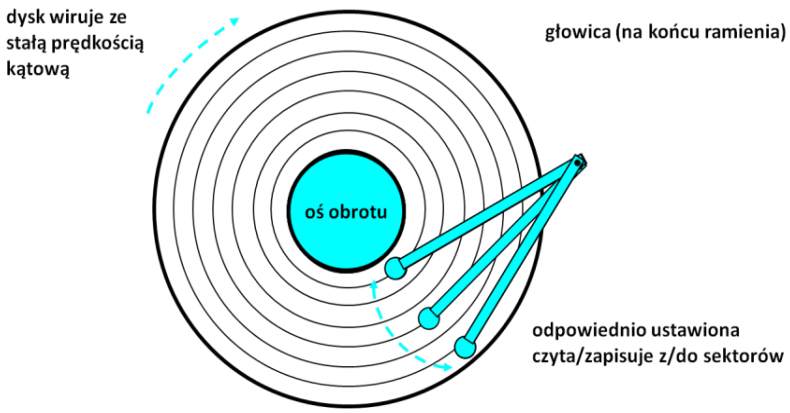
Pojemność dysku to maksymalna ilość bajtów, jaka może być na nim zapisana (gigabajty (GB), terabajty (TB), przy czym 1 GB = 10^9 bajtów oraz 1 TB = 10^{12} bajtów), zależy od:

- gęstości zapisu (bity/cal),
- gęstości upakowania sektorów (sektory/cal),
- gęstości na powierzchni (bity/cal^2).

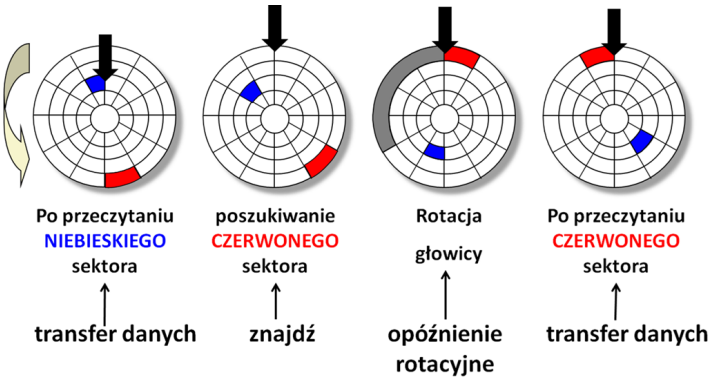
Talerzy może być więcej, a każdy ma dwie strony:



Zapisywanie/czytanie:



Dostęp do dysku – czasy poszczególnych operacji:



Czas dostępu do dysku:

- **Średni czas dostępu to sektora docelowego jest wyznaczany przez :**
  - $T_{dostęp} = T_{znajdź} + T_{rotacja} + T_{transfer}$
- **Czas poszukiwania ($T_{znajdź}$)**
  - Czas potrzebny do ustawienie głowicy nad docelowym sektorem
  - zwykle $T_{znajdź}$ to około 3—9 ms
- **Opóźnienie rotacyjne ($T_{rotacja}$)**
  - Czas oczekiwania na pierwszy skopiowany bit z sektora docelowego dostarczony do głowicy
  - $T_{rotacja}$ = 1/2 x 1/RPMs x 60 sec/1 min
  - typowa prędkość obrotowa = 7 200 RPMs
- **Czas transferu ($T_{transfer}$)**
  - czas potrzebny do przeczytania bitów z docelowego sektora T
  - $T_{transfer}$ = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min

czas jednej rotacji-obrotu (w minutach) — ułamek obrotu do przeczytania

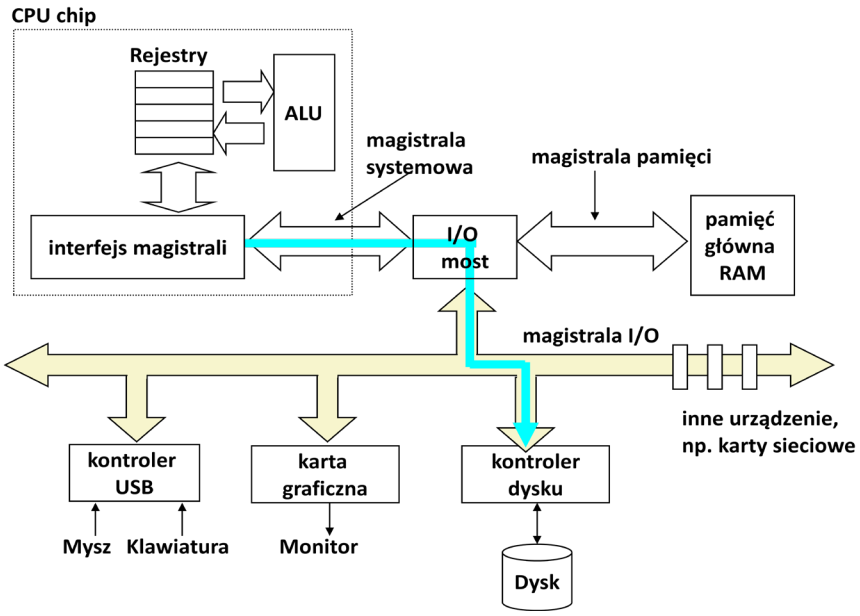
Przykład:

- **Dane są:**
  - prędkość obrotowa = 7 200 RPM
  - średni czas wyszukiwania = 9 ms
  - średnia liczba sektorów na ścieżkę (# sectors/track) = 400
- **Liczymy:**
  - $T_{rotacja}$ = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms
  - $T_{transfer}$ = 60/7200 x 1/400 x 1000 ms/sec = 0,02 ms
  - $T_{dostęp}$ = 9 ms + 4 ms + 0,02 ms
- **Ważne uwagi:**
  - czas dostępu zdominowany przez czas wyszukiwania, opóźnienie rotacyjne
  - pierwszy bit w sektorze jest czasowo najdroższy, reszta prawie za darmo
  - *czas dostępu dla SRAM to ok. 4 ns/double-word, dla DRAM to ok. 60 ns*
    - *Dyski HD są ok. 40 000 razy wolniejsze od SRAM,*
    - *2 500 razy wolniejsze od DRAM.*

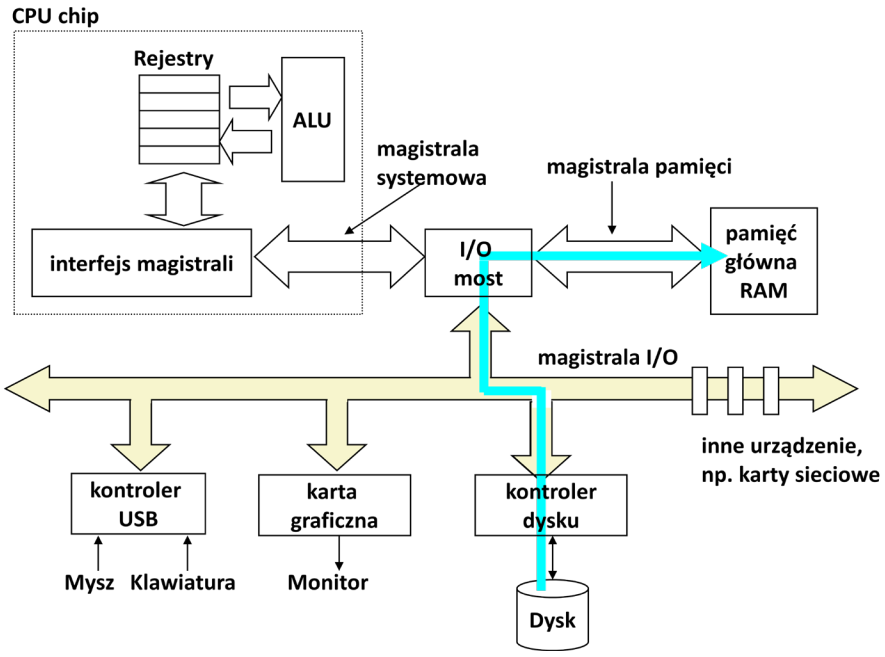
8.3.2 Magistrala I/O

Czytanie z dysku

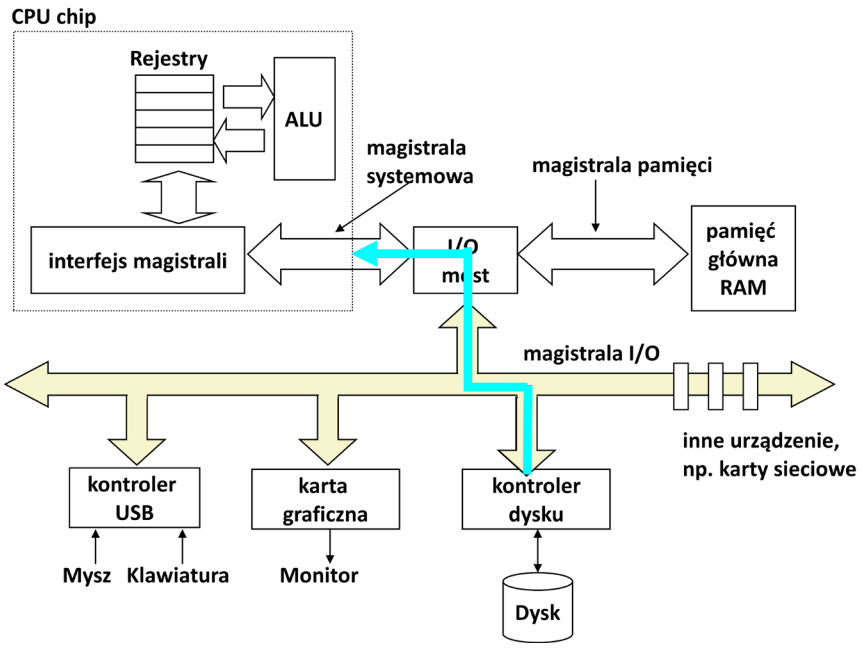
CPU inicjuje czytanie poprzez komendę OUT z numerem portu (do dysku) skierowaną do kontrolera dysku:



Kontroler czyta odpowiedni sektor Direct Memory Access (DMA) i przekazuje dane do pamięci głównej:



Po wykonaniu DMA kontroler powiadamia o tym CPU poprzez tzw. przerwanie (ang. *interrupt):*



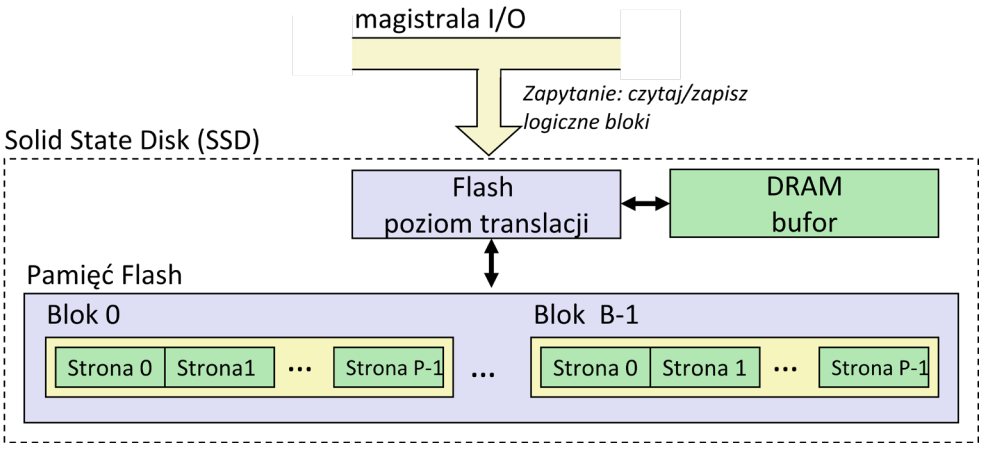
8.3.3 ROM

DRAM oraz SRAM są pamięciami nietrwałymi; dane są tracone po wyłączeniu zasilania. Trwałe pamięci zachowują dane po wyłączeniu zasilania. Są to:

- Read-only memory (ROM): na stałe podczas ich wytwarzania,
- Electrically Eraseable PROM (EEPROM): można zapisywać i nadpisywać,
- pamięć Flash: EEPROM, można zmieniać zapis do ok. 100 000 razy,
- 3D XPoint (Intel Optane) & emerging NVMs.

Trwałe pamięci potrzebne są do oprogramowania sprzętowego, BIOS-u, kontrolerów dysków, kart sieciowych, kart graficznych, podsystemów bezpieczeństwa itd.

Solid State Disks (SSDs) to pamięć typu Flash. Składa się ze stron wielkości od 512 KB do 4 KB. Blok zawiera od 32 do 128 stron. Czytanie/wpisywanie dokonywane jest na pojedynczej stronie. Wpisywanie jest możliwe tylko jeśli uprzednio jej blok został wymazany z poprzedniego wpisu. Blok jest przeznaczony na ok 100 000 zmian wpisów.

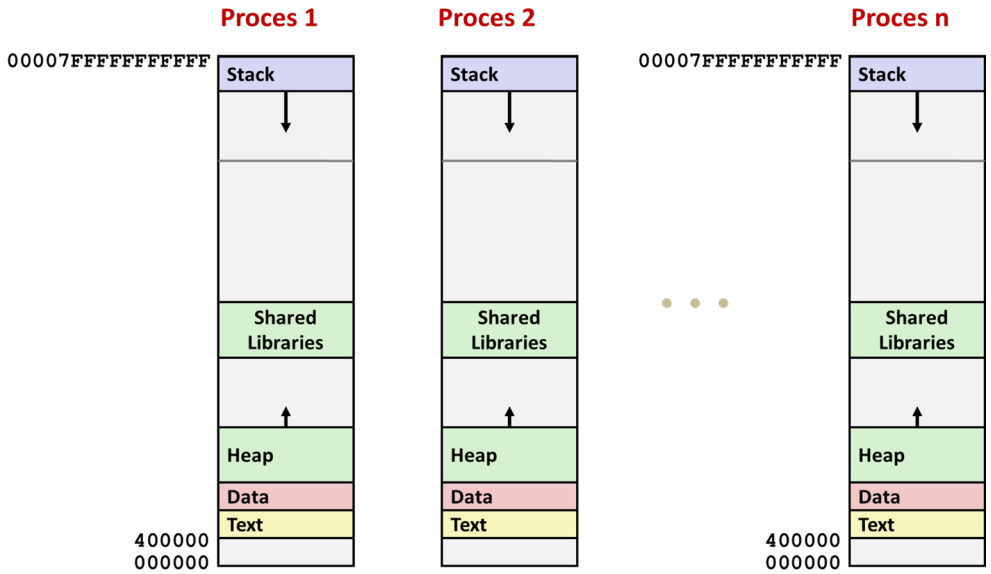


8.4 Podsumowanie

Luka pomiędzy CPU a pamięcią (zwłaszcza masową) zwiększa się coraz bardziej. Dobrze napisany kod z lokalnością może nieco tę lukę zmniejszyć. Hierarchiczna organizacja pamięci i keszowanie pomaga, wykorzystując lokalność. Pamięci Flash są coraz większe i coraz częściej zastępują magnetyczne dyski HD. Ale HD są bardziej trwałe! Wraz z postępującą miniaturyzacją mogą też zastępować DRAM oraz SRAM.

Rozdział 9. Pamięć wirtualna

Wiele procesów korzysta z tej samej pamięci na dysku HD. Pamięć wirtualna to rozwiązanie, które pozwala na to, żeby te procesy nie przeszkadzały sobie wzajemnie.



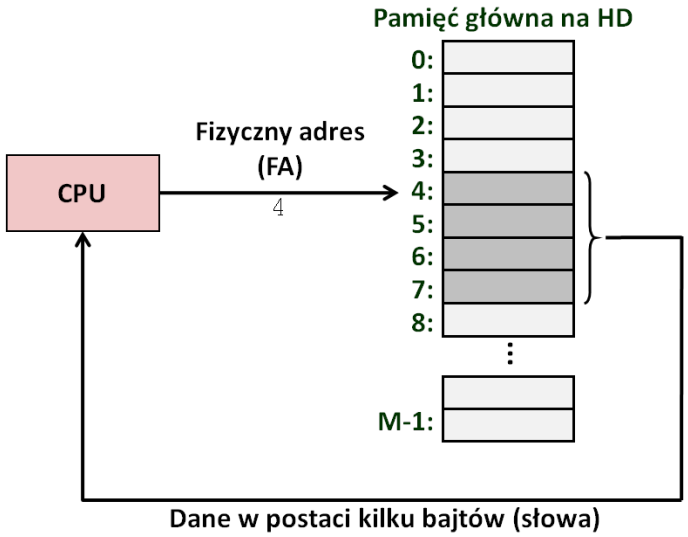
Rozwiązanie: wirtualna pamięć (ang. Virtual Memory)

9.1 Przestrzenie adresowe (ang. Address Spaces)

Wirtualne przestrzenie adresowe dla procesów pozwalają na izolację tych procesów poprzez tłumaczenie tych wirtualnych adresów na adresy fizyczne. Wirtualna przestrzeń adresowa jest podzielona na zestaw stron. Każdą z tych stron można odwzorować do pamięci fizycznej za pomocą wielopoziomowej tablicy translacji stron. Tabele translacji, oprócz odwzorowania wirtualnego na fizyczne, zawierają kontrolę uprawnień dotyczących odczytywania i zapisu. Adres aktualnie używanej tablicy translacji jest w specjalnym rejestrze CPU. Proces może odwoływać się tylko do danych należących do jego wirtualnej przestrzeni adresowej. Wirtualna przestrzeń adresowa jest podzielona na część użytkownika i jądro. Proces może uzyskać dostęp do jądra tylko wtedy, gdy procesor działa w trybie uprzywilejowanym.

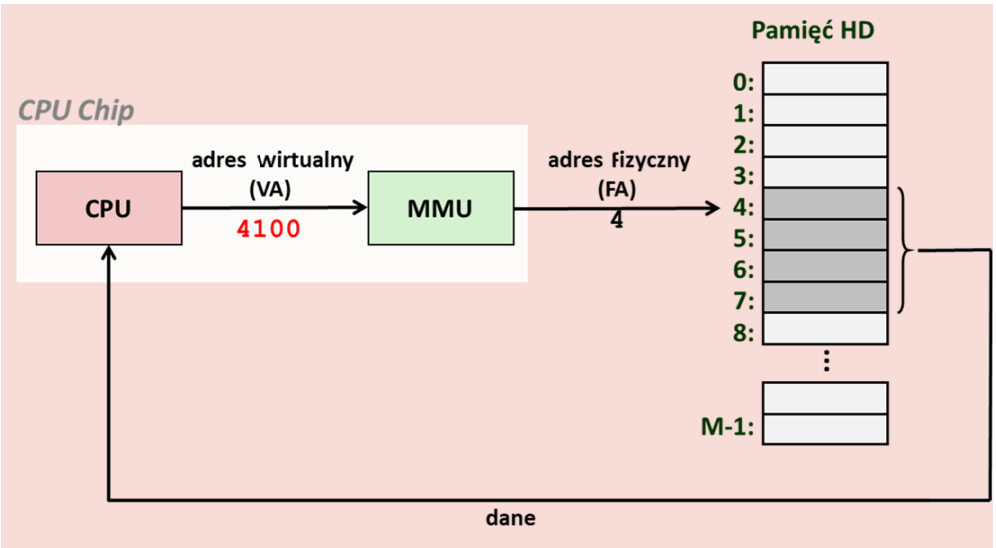
Translacyjny bufor podręczny (*Translation Lookup Buffor* – TLB) to pamięć podręczna używana do skrócenia czasu potrzebnego na dostęp do lokalizacji w pamięci użytkownika. Jest to część jednostki zarządzania pamięcią (*Memory Management Unit* – MMU) układu. TLB przechowuje ostatnie translacje pamięci wirtualnej do pamięci fizycznej i można ją nazwać pamięcią podręczną translacji adresów.

Fizyczna (prawdziwa) adresacja pamięci:



Taka adresacja jest używana w prostych systemach wbudowanych, takich jak wbudowane mikrokontrolery w urządzeniach, np. samochodach.

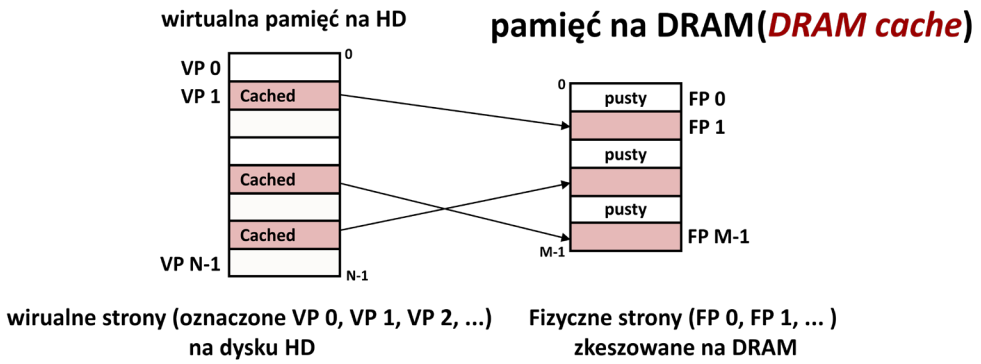
Wirtualne adresowanie:



Pamięć na HD jest jedna, ale jej adresacji może być wiele. Specjalny moduł MMU realizowany jest raczej w OS niż w hardware. Jeden to jeden z ważniejszych wynalazków w IT. Upraszcza i usprawnia zarządzanie pamięcią na dysku HD. Każdy proces otrzymuje osobną wirtualną adresację. Procesy nie mogą „mieszać" w pamięciach przydzielonych innym procesorom. Aplikacje użytkowników nie mają dostępu do zastrzeżonych obszarów pamięci, takich jak jądro systemu operacyjnego i kod.

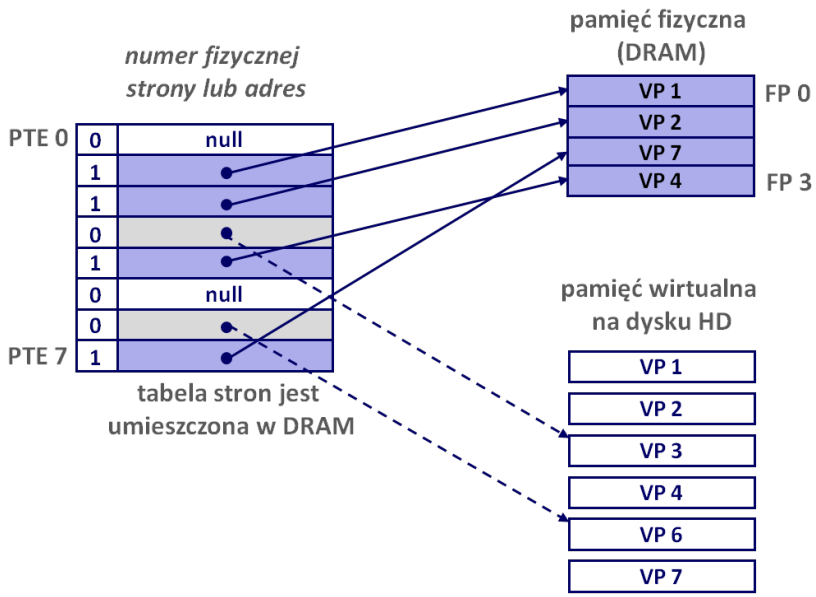
9.1.1 Wirtualna pamięć (VM) jako narzędzie do keszowania (*caching*)

Koncepcyjnie wirtualna pamięć jest tablicą złożoną z N bloków bajtów, umieszczoną w pamięci HD; te bloki są nazwane stronami (ang. *pages*).

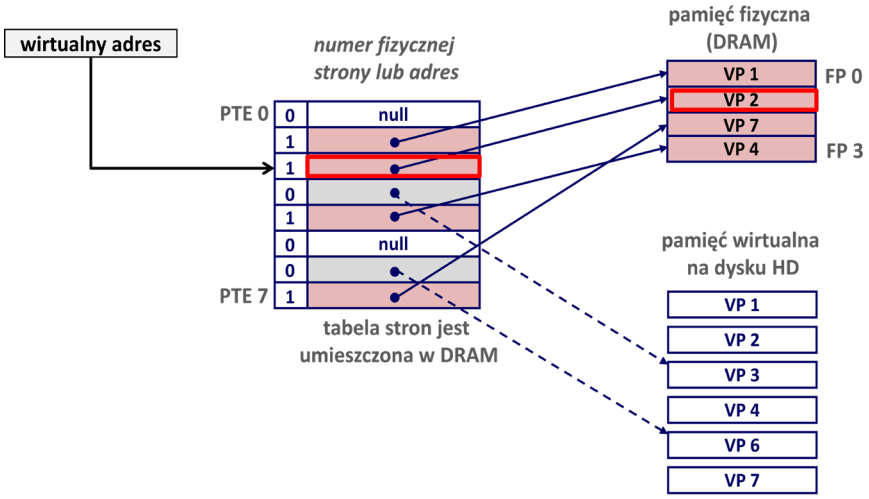


Dlaczego DRAM cache? Co prawda pamięć DRAM jest 10 razy wolniejsza niż SRAM, ale dostęp do HD jest 10 000 razy wolniejszy niż do DRAM. Czas załadowania jednego bloku z dysku HD > 1ms (> 1 miliona cyklów zegara). W tym czasie CPU może wykonać mnóstwo obliczeń.

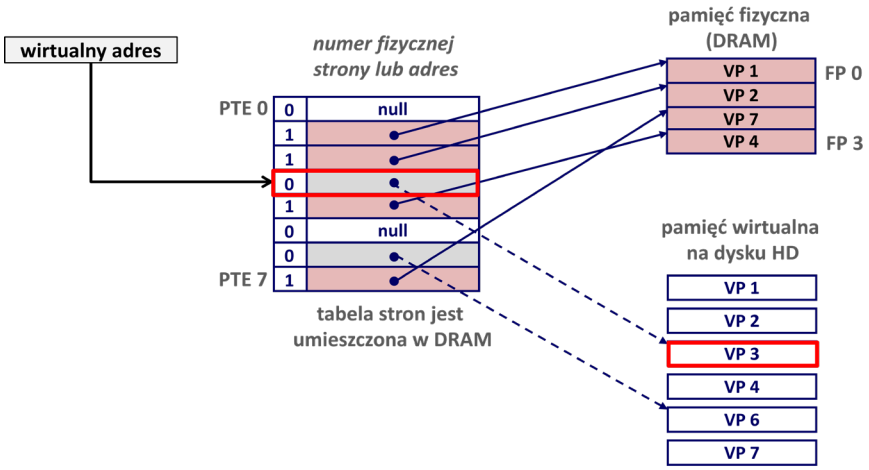
Tabela stron (*page table*) jest tablicą składającą się z elementów (PTE 0, PTE 1, PT 2, ...), które odwzorowują wirtualne strony (VP) na fizyczne strony (FP) dla każdego procesu osobno.



Referencja do strony: sukces

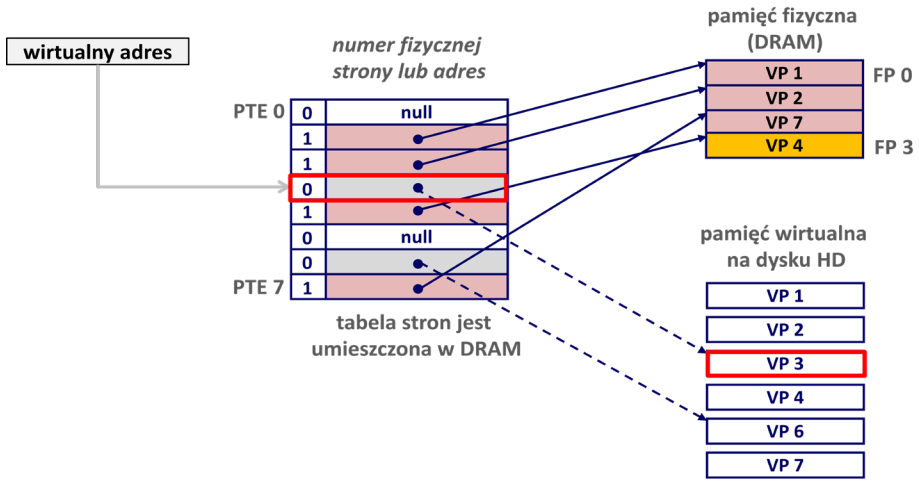


Nie ma jej w DRAM cache:

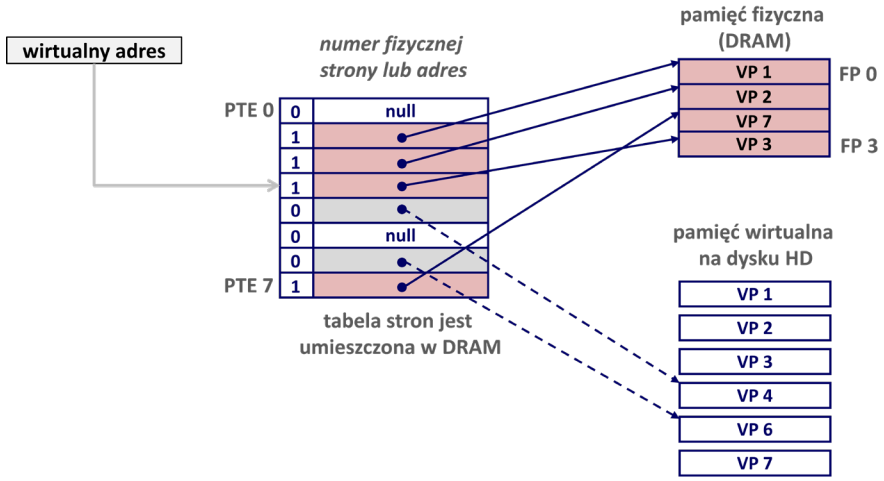


Jak system sobie z tym radzi? Generuje wyjątek.

Wyznaczona jest „ofiara”, tj. strona do usunięcia, tutaj jest to FP 3 o zawartości VP 4.



W jej miejsce wstawiana jest VP 3



Lokalność kodu może pomóc! Pamięć wirtualna wydaje się być bardzo nieefektywna, ale sprawdza się, jeśli kod ma własność lokalności. W każdym momencie wykonania aplikacja ma dostęp do aktualnych aktywnych wirtualnych stron nazywanych *working set* (zestaw roboczy).

Jeśli kod ma większą lokalność, to tych stron jest mniej.

Jeśli zestaw roboczy jest mniejszy od wielkości głównej pamięci, to następuje efektywne szybkie wykonanie jednego procesu (po kilku niepowodzeniach).

Jeśli zbiór roboczy jest większy od wielkości głównej pamięci, to występuje wtedy tzw. *thrashing* (szamotanie procesów). Następuje wówczas tzw. krach wydajnościowy, polegający na ciągłym zamienianiu i kopiowaniu stron.

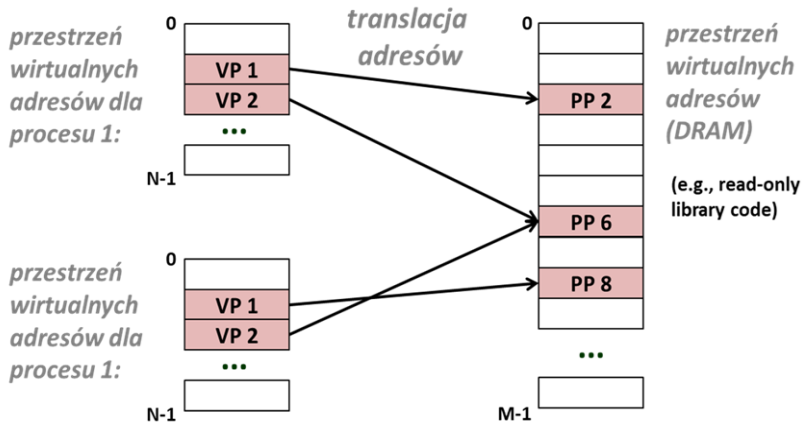
Jeśli jednocześnie uruchomionych jest zbyt wiele procesów, to ich całkowity rozmiar zestawu roboczego jest większy od rozmiaru pamięci głównej i powoduje takie szamotanie.

Wirtualna pamięć (VM) to kluczowe narzędzie do zarządzania pamięcią. Polega to na tym, że każdy proces ma własną wirtualną przestrzeń adresową i może operować na niej jak na zwykłej pamięci fizycznej, nie będąc świadomym, że jest ona wirtualna. Funkcja mapowania rozprasza adresy w pamięci fizycznej i może poprawić lokalność.

Uproszczenie alokacji pamięci

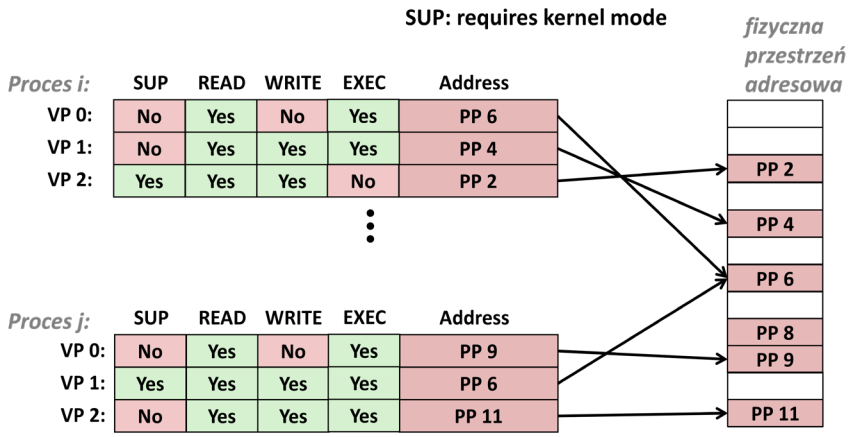
Każda strona wirtualna może być mapowana na dowolną stronę fizyczną. Strona wirtualna może być przechowywana na różnych stronach fizycznych w różnym czasie. Umożliwia udostępnianie kodu i danych między procesami.

Odwzoruj strony wirtualne na tę samą stronę fizyczną (tutaj: PP 6)

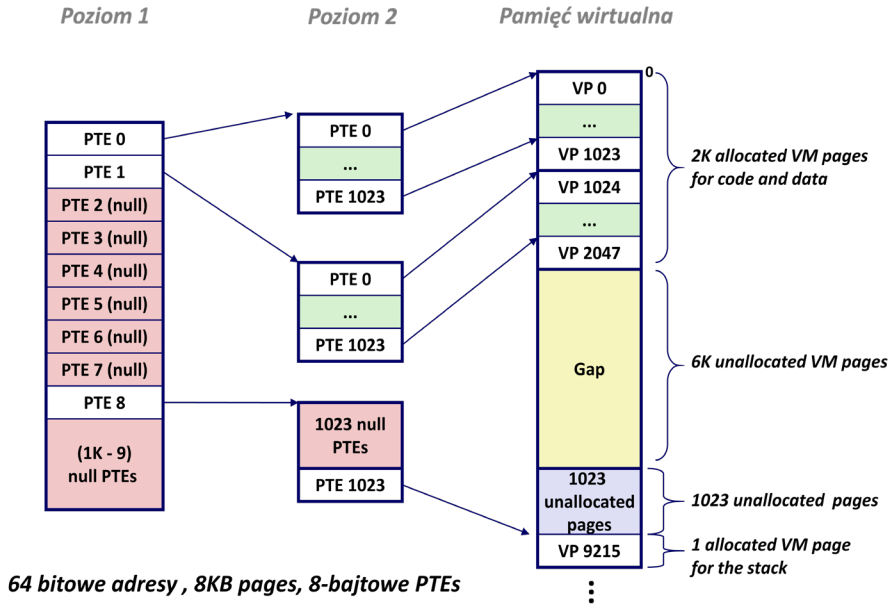


VM jako narzędzie do ochrony pamięci

Rozszerza PTE (strony w tabeli) o bity uprawnień. MMU sprawdza te bity przy każdej próbie dostępu. SUP to Supervisor-mode Access.



Dwupoziomowa hierarchia tabeli stron



9.2 Podsumowanie wirtualnej pamięci

Punkt widzenia programisty na pamięć wirtualną jest następujący: każdy proces ma własną prywatną liniową przestrzeń adresową. Ta prywatna pamięć nie może być zmieniana przez inne procesy.

System operacyjny wydajnie wykorzystuje pamięć wirtualną tylko jeśli jest lokalność kodu. Upraszcza to zarządzanie pamięcią i samo programowanie. Upraszcza również ochronę, zapewniając wygodne sprawdzanie uprawnień. Wirtualna pamięć jest implementowana jako kombinacja hardware & software:

- **hardware** -- MMU, TLB (Translation Lookaside Buffer), exception handling mechanisms,
- **software** -- page fault handlers, TLB management.

Rozdział 10. Architektura DATA FLOW (przepływ danych)

W architekturze von Neumanna:

- kolejna instrukcja jest pobierana i wykonywana z programu zapisanego w RAM
- zgodnie ze wskaźnikiem instrukcji IP.
- Wykonanie jest sekwencyjne, chyba że bieżąca instrukcja to skok.

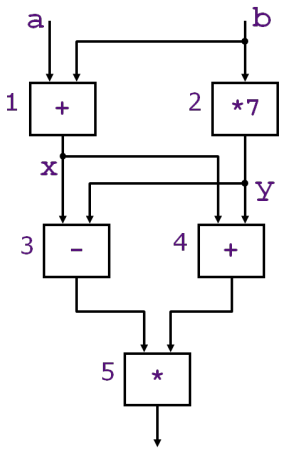
W architekturze przepływu danych:

- instrukcja jest wykonywana w kolejności zgodnej z przepływem danych, tzn. kiedy na jej wejściu (*input*) są wszystkie potrzebne dane,
- nie ma wskaźnika instrukcji IP,
- kolejność wykonywanych instrukcji określona jest dynamicznie przez zależności przepływu danych,
- każda instrukcja określa „kto" (następna instrukcja) powinien otrzymać jej wynik (*output*); być może w zależności od wyniku,
- wiele instrukcji może być wykonywanych jednocześnie, równolegle.

Przepływ danych, jako program, składa się z węzłów przepływu danych z kierunkiem przepływu danych. Jest to graf skierowany. Węzeł jest uruchamiany (pobierany i wykonywany), gdy wszystkie jego dane wejściowe są dostarczone.

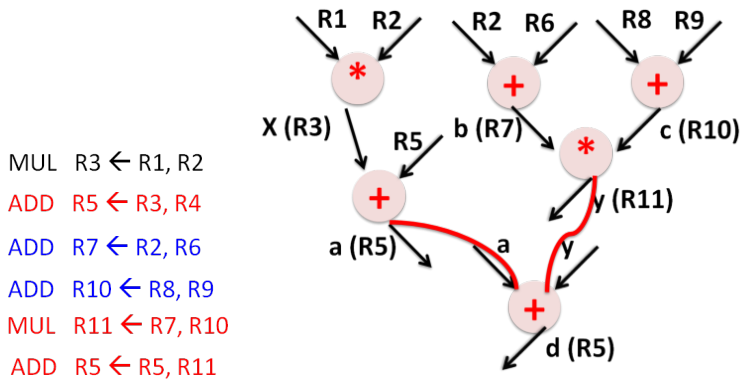
Przykład

	kod operacji	argument 1	argument 2	wynik 1	wynik 2
1	+			3L	4L
2	*			3P	4P
3	−			5L	
4	+			5P	
5	*			out	



Graf przepływu danych może być budowany z fragmentu klasycznego programu. Czy możemy to zrobić dla całego programu? Należy wzbogacić węzły o warunki i relacje oraz węzły synchronizacyjne. Ale to nie wystarcza, bo konieczne są pętle, żeby mieć rekursję.

Prosty przykład sekwencyjnego programu i jego realizacji w postaci grafu przepływu danych:



Architektura przepływu danych jest wykorzystana w firmie Maxeler (https://www.maxeler.com/technology/dataflow-computing/), specjalizującej się w High Performance Computing. Cytat:

> *Nasze rozwiązania wykorzystują przepływ danych — rewolucyjny sposób wykonywania obliczeń, całkowicie różny od obliczeń z wykorzystaniem konwencjonalnych procesorów. Komputery przepływu danych koncentrują się na optymalizacji przepływu danych w aplikacji i wykorzystują ogromny paralelizm między tysiącami maleńkich „rdzeni przepływu danych", aby zapewnić większe (o rząd wielkości) korzyści w zakresie wydajności, zużycia przestrzeni i energii. Analogią do przejścia od przepływu sterowania do przepływu danych jest model produkcji samochodów Forda, w którym drogich, wysoko wykwalifikowanych rzemieślników (rdzenie procesora przepływu sterowania) zastępuje linia fabryczna, przesuwająca samochody do kolejnych pracowników, każdy o jednej prostej umiejętności (rdzenie przepływu danych).*

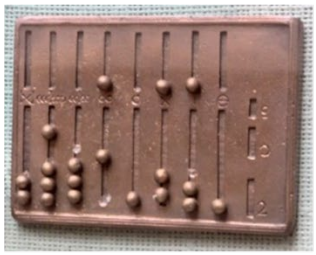
Pomimo szeregu wdrożeń w firmach z listy Fortune 500 oraz akademickich badań, to podejście nie jest stosowane na szeroką skalę. Wraz z pojawieniem się architektury opartej na mikroserwisach w chmurach zainteresowanie tą architekturą, opartą na przepływie danych, znacznie osłabło.

Rozdział 11. Historia komputerów

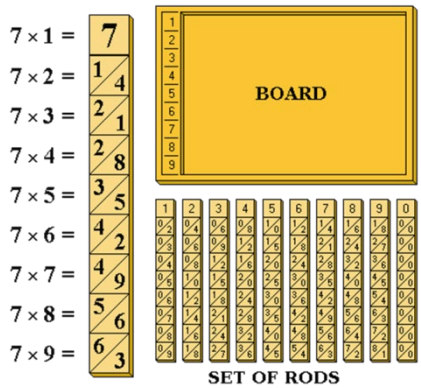
Historia ta jest bardzo ciekawa i pouczająca, więc warto ja poznać.
Z wykorzystaniem materiałów na https://en.wikipedia.org/wiki/Computer

Przed XX wiekiem obliczenia były wykonywane przez ludzi, ewentualnie wspomaganych mniej lub bardziej sprawnymi narzędziami. Pierwsze narzędzia do obliczeń nazywane były kalkulatorami (łac. *calculus*), co odnosiło się do małych kamieni służących właśnie do liczenia. Człowiek używający tych narzędzi nazywany był „liczący", po angielsku *computer*.

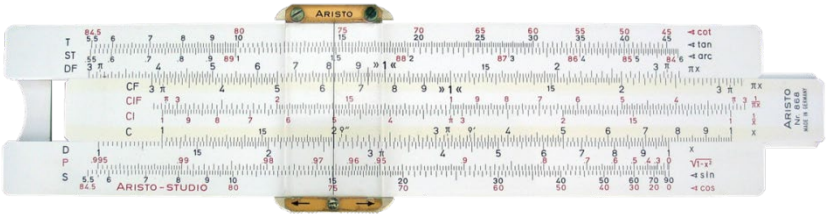
W czasach prehistorycznych człowiek używał do liczenia: palców (stąd system dziesiętny, lub dwunastkowy), znaków na ścianach jaskiń, kości, kamieni itp. Powstawały wtedy liczydła (abacus): chińskie (suanpan), rzymskie, rosyjskie. Sposoby liczenia za pomocą suanpan są bardzo rozwinięte: dodawanie, mnożenie, odejmowanie, dzielenie a nawet (?) podobno pierwiastki kwadratowe i sześcienne.


Szkocki matematyk i fizyk John Napier odkrył w 1617 r., że mnożenie i dzielenie liczb może być wykonywane jako dodawanie i odejmowanie logarytmów tych liczb. Stąd też powstały tabele logarytmów; żeby je wyliczyć Napier potrzebował wielu mnożeń i aby ułatwić sobie pracę, wynalazł tzw. kości Napiera, będące odmianą liczydła.

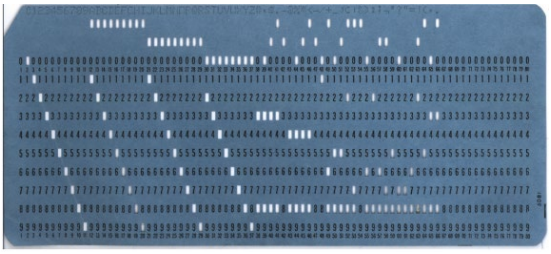
Współczesną wersją był suwak logarytmiczny, powszechnie używany przez inżynierów do czasu wprowadzenia kalkulatorów, a potem komputerów PC.



Około 1820 r. Charles Xavier Thomas de Colmar skonstruował pierwszy w pełni funkcjonalny i masowo produkowany kalkulator – Arithmometer (arytmometr). Realizował cztery działania arytmetyczne. Oparty był na pracach Leibniza.

Mechaniczne kalkulatory były w użyciu do lat 1970., a panie kasjerki używały ich wtedy z napędem elektrycznym.

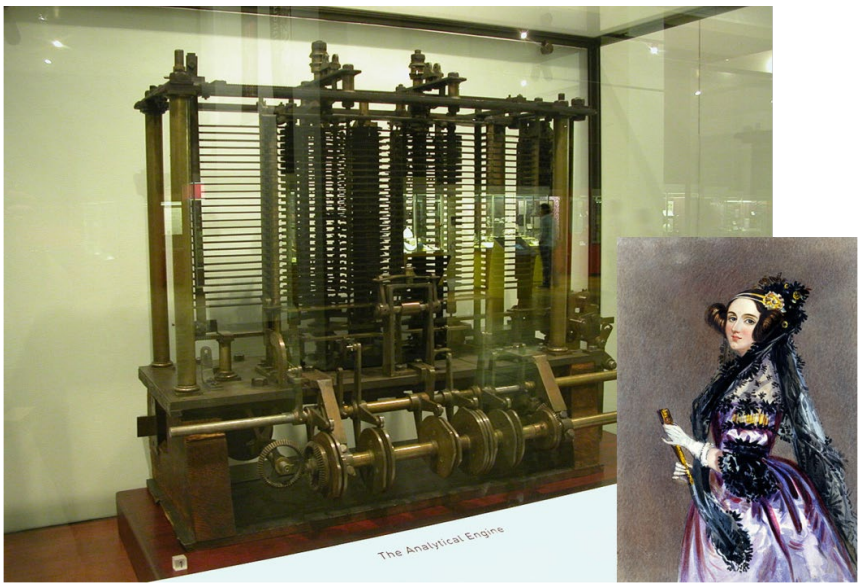
W późnych latach 1880. Amerykanin Herman Hollerith wynalazł sposób na przechowywanie danych oparty na kartach perforowanych (*punched cards*), które mogły być czytane przez odpowiednio do tego skonstruowane maszyny. Do lat 1980. był to nośnik danych i programów. Ta technika została użyta podczas powszechnego spisu ludności USA w 1890 roku i sprawdziła się – było znacznie szybciej i taniej. Firma, którą założył Hollerith, była zalążkiem IBM.



Kalkulator Curta został skonstruowany przez austriackiego wynalazcę Curta Herzstarka w 1948 r. Opracował on projekt tego kalkulatora, będąc (jako więzień) w obozie w Buchenwald. Herzstark przeżył obóz i wyjechał do Liechtensteinu, gdzie dokończył swoje dzieło. Był mały, poręczny (mieścił się w dłoni) i był godnym następcą kalkulatora Leibniza oraz arytmometru. Wyprodukowano 80 000 sztuk typu I oraz 60 000 sztuk typu II; ostatni w listopadzie 1970 roku.

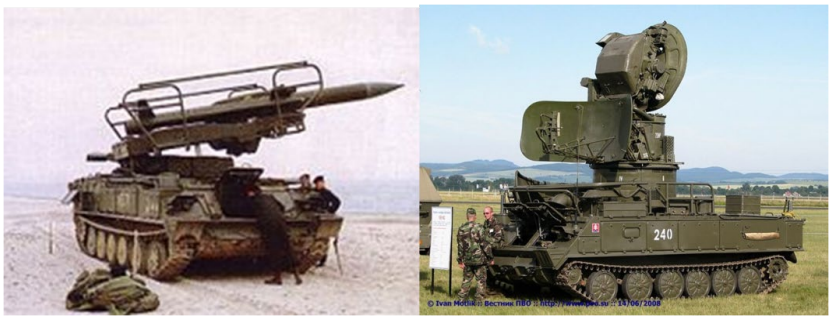
Charles Babbage i jego maszyna (1837–1871)

Maszyna analityczna (ang. *analytical engine*) – zaprojektowana, ale niedokończona – to pierwszy programowalny komputer ogólnego zastosowania. Zakładała rozdzielenie pamięci („magazynu”, ang. *store*) i jednostki obliczeniowej („młyna”, ang. *mill*) – podobnie jak we współczesnych komputerach. Urządzenie pozwalało na wykorzystanie konstrukcji znanych z dzisiejszych języków programowania takich jak pętle, instrukcje warunkowe. Napęd – silnik parowy, dane wprowadzane przez karty perforowane. Był to pierwszy „komputer”, dla którego zostały napisane programy. Pierwszy z nich, autorstwa Ady Lovelace (córka lorda Byrona), miał obliczać liczby Bernoulliego.



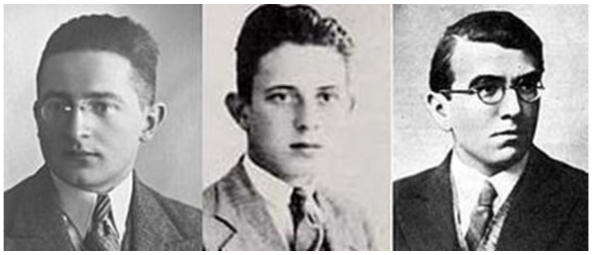
Komputery analogowe

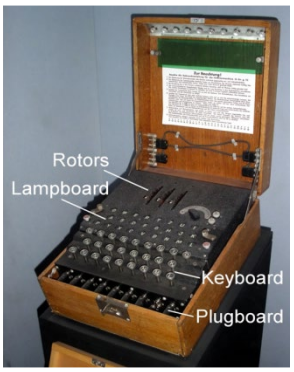
W pierwszej połowie XX wieku uważano, że komputery analogowe są przyszłością automatycznych obliczeń. Opierały się one na przetwarzaniu sygnałów elektrycznych (ale również mechanicznych czy hydraulicznych) nie w sposób dyskretny, ale ciągły, np. rozwiązywanie równań różniczkowych, transformacje np. Fouriera itp. Brak jest publikacji po 1950 roku na ten temat (?). Zastosowania militarne – np. poniższe:



Nasi kryptolodzy – złamali Enigmę

Marian Rejewski (1905-80) ok. 1932 r. „złamał" Enigmę. Jerzy Różycki (1909-42) i Henryk Zygalski (1908-78) opracowali automatyczne łamanie szyfru Enigmy w oparciu o teorię cykli i tzw. bomby kryptologicznej. Używali sześć sprzężonych polskich kopii Enigmy napędzanych silnikiem elektrycznym. Jedna bomba kryptologiczna pozwalała na odkodowanie klucza dziennego w ciągu około dwóch godzin i zastępowała pracę około 100 ludzi.

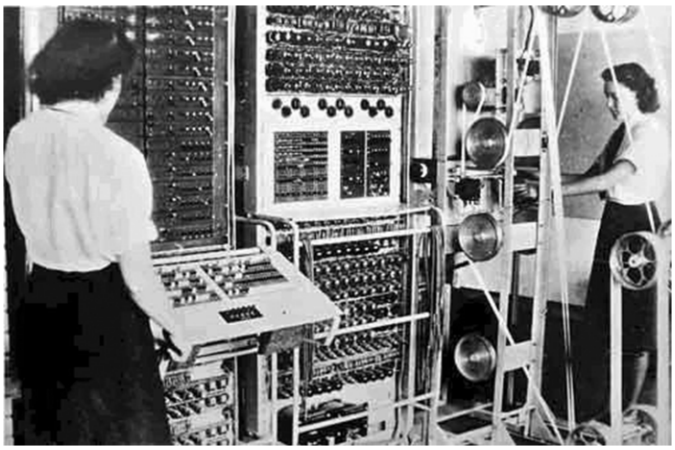




Colossus

Wyniki polskich kryptologów zostały przekazane Brytyjczykom i Francuzom 25 lipca 1939 r. Prace były kontynuowane w Bletchley Park pod kierunkiem Alana Turinga. Colossus to seria programowalnych maszyn cyfrowych (konstruktorzy: Max Newman i Tommy Flowers) zbudowana do łamania szyfrów używanych przez Niemców w czasie II wojny światowej. Pierwszym szyfrem była Enigma, łamana za pomocą elektromechanicznych bomb. Niemcy używali również szyfru Lorenz SZ 40/42 do komunikacji w wysokim szczeblu dowodzenia. Ten szyfr, nazwany Tunny (mały tuńczyk), został złamany przez Brytyjczyków.

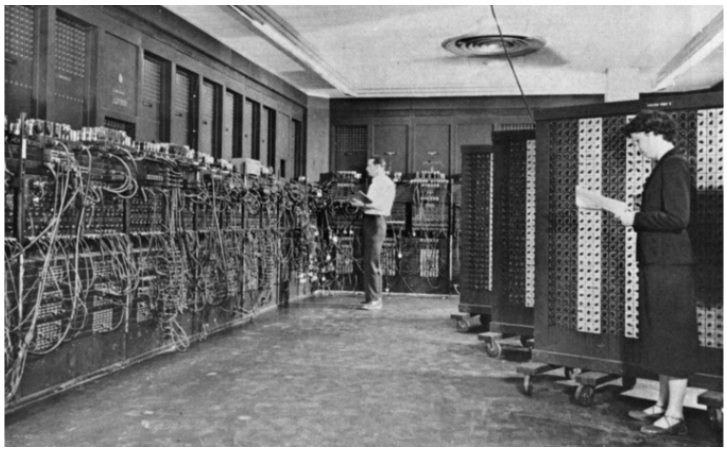
Bez Colossusa alianci byliby pozbawieni możliwości odczytywania zaszyfrowanych telegraficznych wiadomości pomiędzy niemieckim Sztabem Generalnym a Wehrmachtem w okupowanej Europie. Szczegóły dotyczące Colossusa były utrzymywane w ścisłej tajemnicy do 1975 roku (zgodnie z prawodawstwem brytyjskim) do tego stopnia, że Winston Churchill osobiście nakazał jakoby zniszczenie (?) wszystkiego, co dotyczyło Colossusa i złamania szyfru Lorenz SZ przez Brytyjczyków, żeby nikt (np. ZSRR) się o tym nie dowiedział. W rezultacie Colossus nie był uwzględniany w historii komputerów.



ENIAC (Electronic Numerical Integrator and Computer) 1946

Powstał z rezultacie supertajnego projektu „PX". Miał jakoby służyć do liczenia trajektorii pocisków artyleryjskich, faktycznie na potrzeby budowy bomby atomowej i wodorowej. Skonstruowany w latach 1943-1945 przez J.P. Eckerta i J.W. Mauchly'ego na Uniwersytecie Pensylwanii w USA. Zaprzestano jego używania w 1955 r. Podobny do Colossusa, ale znacznie szybszy i wszechstronniejszy. Podobnie jak w Colossusie „program" był „lutowany" w przełącznikach i wstawiany do komputera. Wykorzystane zostały doświadczenia z konstrukcji ABC (od ang. Atanasoff-Berry Computer), zbudowanrgo w Iowa State University przez Johna Vincenta Atanasoffa i Clifforda Berry'ego w latach 1937-42.

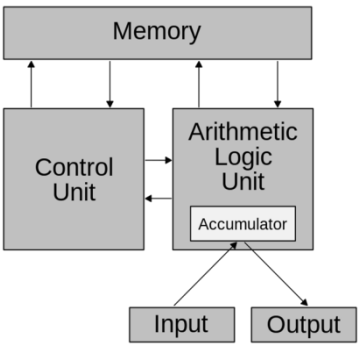
ENIAC składał się z 42 czarnych szaf z blachy stalowej, każda o wymiarach 300 × 60 × 30 cm. Posiadał 18 800 lamp elektronowych szesnastu rodzajów; ponadto 6000 komutatorów, 1500 przekaźników oraz 50 000 oporników. Całość wymagała ręcznego wykonania 0,5 mln lutowań. Maszyna ważyła 30 ton i pobierała 140 kW mocy. Jej system wentylacyjny miał wbudowane dwa silniki Chryslera o łącznej mocy 24 KM.



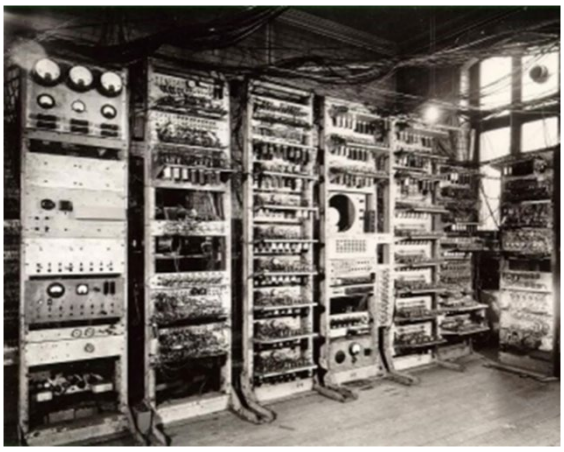
W każdej szafie zainstalowany był termostat, który wstrzymywał pracę komputera, gdy temperatura przekraczała 48°C. Egzemplarz ENIAC-a można na zobaczyć w Smithsonian Museums, Mall Washington DC.

Von Neumann computer architecture (1947)

Idea powstała na podstawie teoretycznych prac Warrena McCullocha i Waltera Pittsa, *A logical calculus of the ideas immanent in nervous activity*, Bull. Math. Biophysics, Vol. 5 (1943), pp. 115–133. Sama architektura została opracowana przez Johna von Neumanna i opublikowana w 1945 roku w technicznym raporcie First Draft of a Report on the EDVAC.



The Manchester Mark 1 (1948)



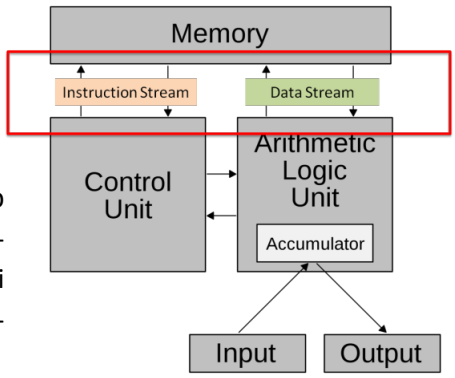
W Wielkiej Brytanii po II wojnie światowej została kadra i doświadczenia po Colossusie. Mark 1 był pierwszym komputerem, w którym program był zapisywany w pamięci, a nie lutowany na przełącznikach. Był prototypem Ferranti Mark 1, pierwszego komercyjnego i dostępnego komputera uniwersalnego przeznaczenia. EDSAC to pierwszy nowocześnie zaprojektowany i zrealizowany komputer przez Maurice'a Wilkesa i jego zespół w University of Cambridge, Mathematical Laboratory in England w 1949 roku.

John von Neumann urodził się 28 grudnia 1903 r. w Budapeszcie. W czasie pobytu w Niemczech nazywał się Johann von Neumann, dziś znany jest jednak przede wszystkim pod swym amerykańskim imieniem John. Jako sześciolatek potrafił szybko dzielić w pamięci ośmiocyfrowe liczby, posiadał fotograficzną pamięć, która pozwalała mu po krótkim spojrzeniu na stronę książki cytować dokładnie jej zawartość. Po uzyskaniu matury studiował na kilku europejskich uniwersytetach (ETH Zürich, uniwesytety: Budapeszt, Getynga, Hamburg, Berlin).

John von Neumann wniósł znaczący wkład do wielu dziedzin matematyki, m.in. logiki matematycznej, teorii mnogości, teorii liczb, udowodnił twierdzenie min-max. Oprócz tego był jednym z pionierów informatyki. Od 1943 r. uczestniczył również w projekcie Manhattan. Z tego czasu pochodzi też rozwój architektury komputerowej zwanej architekturą von Neumanna.

"Von Neumann Bottleneck" (VNB)

„Wąskie gardło von Neumanna" – ograniczona przepustowość pomiędzy CPU (procesorem) a pamięcią w porównaniu z rozmiarem pamięci. CPU jest zmuszony czekać (3/4 swojego czasu lub więcej) na dane z pamięci / zapis danych w pamięci. Jest to ograniczenie, widoczne zwłaszcza jeśli duża porcja danych ma być przetworzona w stosunkowo prosty i szybki sposób przez CPU.



Coraz bardziej zwiększa się szybkość CPU (rzędu 3 GHz) oraz wielkość pamięci (rzędu TB), ale przepustowość pomiędzy CPU a pamięcią nie za bardzo. Delegowanie części

przetwarzania do karty graficznej czy urządzeń input-output niewiele daje. Problem z wąskim gardłem (*von Neumann bottleneck*) staje się coraz bardziej poważny. Architektura hardwarowa staje się coraz bardziej złożona.

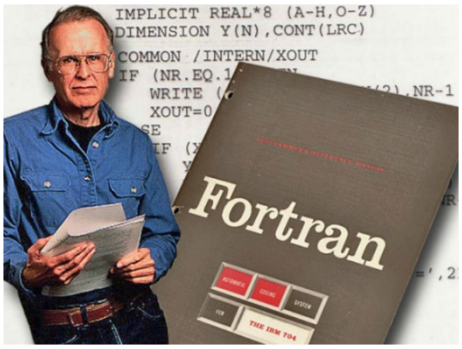
Von Neumann language – architektura komputera według von Neumanna jest dominująca (i jedyna) od samego początku, czyli od lat 40. XX wieku.
Język programowania von Neumanna to taki, który jest izomorficzny (na wysokim poziomie abstrakcji) z architekturą komputera według von Neumanna.

Izomorfizm ten polega na następujących równoważnościach:

- zmienne programowe ↔ komputerowe komórki pamięci,
- instrukcje sterujące ↔ instrukcje warunkowe (flagi) i skoki,
- instrukcje przypisania ↔ pobieranie, przechowywanie,
- instrukcje przetwarzania ↔ odniesienia (adresy) do pamięci i instrukcje arytmetyczne.

Metafora Johna Backusa
Instrukcje przypisania w językach von Neumanna dzielą programowanie na dwa światy:

1. **Pierwszy świat** składa się z wyrażeń, uporządkowanej przestrzeni matematycznej z potencjalnie użytecznymi właściwościami algebraicznymi: większość obliczeń ma miejsce tutaj.
2. **Drugi świat** składa się z instrukcji przypisania.

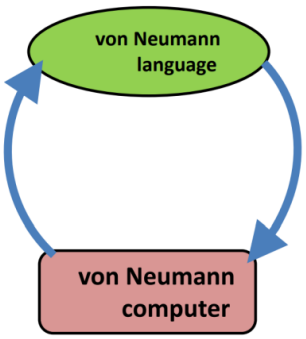


John Backus był współtwórcą notacji Backus-Naur form (BNF), szeroko stosowanej do definiowania formalnej składni języków programowania. Wąskie gardło von Neumanna zostało opisane przez Johna Backusa w jego wykładzie ACM Turing Award z 1977 roku.

Według Backusa:
Z pewnością musi istnieć mniej prymitywny sposób dokonywania dużych zmian w pamięci niż poprzez kopiowanie i zapisywanie ogromnej liczby słów tam i z powrotem przez wąskie gardło von Neumanna, tj. magistralę adresową i magistralę danych.

Intelektualne wąskie gardło:
To ograniczenie polegające na tym, że programowanie musi być tylko na poziomie przetwarzania słów (bajtów).
Dlatego programowanie polega zasadniczo na planowaniu i szczegółowym opisywaniu ogromnego ruchu słów przez wąskie gardło von Neumanna, a większość tego ruchu dotyczy nie istotnych danych, ale tego, gdzie te dane zapisać i potem odnaleźć.

Von Neumann vicious cycle (błędne koło):
Języki programowania (von Neumannowskie) odzwierciedlają dokładnie architekturę komputera von Neumanna. Po to żeby stworzyć nową architekturę, konieczny jest nowy, nie- von Neumannowski język programowania. I odwrotnie, żeby wymyślić nowy język, potrzebna jest nowa architektura.



Programy na poziomie wartości to takie, które opisują, jak połączyć różne wartości (tj. liczby, symbole, ciągi bajtów itp.), aby utworzyć inne wartości, aż do uzyskania wartości wynikowych. Pierwotne wartości to obiekty typów prostych, np. Int, String, Boolean. Nowe wartości są konstruowane z istniejących wartości przez zastosowanie różnych funkcji do wartości, takich jak dodawanie, konkatenacja, odwracanie macierzy itd. Zwykłe programy (w językach von Neumannowskich) są na poziomie wartości: wyrażenia, po prawej stronie instrukcji przypisania, dotyczą wyłącznie liczenia wartości, która ma być następnie przechowywana w pamięci.

John Backus: programowanie na poziomie funkcji
Zasadnicza idea to programy jako obiekty matematyczne. Program ma być zbudowany bezpośrednio z podprogramów, które są podane na początku. Podejście na poziomie funkcji według Backusa jest inne niż tak zwane programowanie funkcjonalne, oparte na rachunku lambda i logice kombinatorycznej, np. w językach: Haskell, F# etc.

Bardzo ciekawy i pouczający jest **życiorys naukowy Johna Backusa**.
Na podstawie biografii *John W. Backus 3 December 1924 - 17 March 20*07, autorstwa Rene Gabriels, Dirk Gerrits, Peter Kooijmans, May 29, 2007.

John Backus na krótko przed śmiercią powiedział, że jego

> *praca nad funkcjonalnymi językami programowania zakończyła się niepowodzeniem i prawdopodobnie musiała zakończyć się niepowodzeniem, ponieważ łatwo było robić skomplikowane rzeczy, ale niezwykle trudno było robić rzeczy proste.*

Inne cytaty:

> *... nie wchodź w oprogramowanie. To tak skomplikowany bałagan, że po prostu męczysz swój mózg, próbując zrobić coś wartościowego ...*
>
> *Co roku oblewałem. Nigdy nie studiowałem. Nienawidziłem się uczyć. Po prostu się kręciłem. Miało to cudowną konsekwencję, że co roku chodziłem do letniej szkoły w New Hampshire, gdzie żeglowałem w lato, miło spędzając czas.*
>
> *Nienawidziłem tego. Nie lubią myśleć w szkole medycznej. Zapamiętują – to wszystko, co chcą, żebyś zrobił. Nie wolno ci myśleć.*

Lubił muzykę, chciał mieć dobry system hi-fi, ale nie mógł znaleźć takiego, więc postanowił sam go zbudować. Uczęszczał do szkoły radiotechnicznej i tam wykonał obliczenia dla jednego z nauczycieli, spodobała mu się matematyka. To skłoniło go do studiowania matematyki na Uniwersytecie Columbia. W 1949 roku, kiedy prawie skończył studia licencjackie, ale nadal nie miał pojęcia, co robić dalej, odwiedził centrum komputerowe IBM przy Madison Avenue w Nowym Jorku. Był tam Selective Sequence Electronic Calculator (SSEC). Pani, która go oprowadzała, nalegała, aby porozmawiał z dyrektorem o zatrudnieniu. Zrobił to i został od razu przyjęty. Jego praca w IBM rozpoczęła się we wrześniu 1950 roku. Pierwszym zadaniem, jakie dostał Backus, było programowanie dla SSEC.

Był to jedyny w swoim rodzaju komputer, zaprojektowany i zbudowany przez IBM w latach 1946–1948 do wykonywania obliczeń naukowych. Komputer był częściowo elektromechaniczny (zawierał 21 400 przekaźników) i częściowo elektroniczny (zawierał 12 500 lamp próżniowych), mieścił się w 60-metrowej szafie umieszczonej w kształcie litery U wzdłuż ściany.

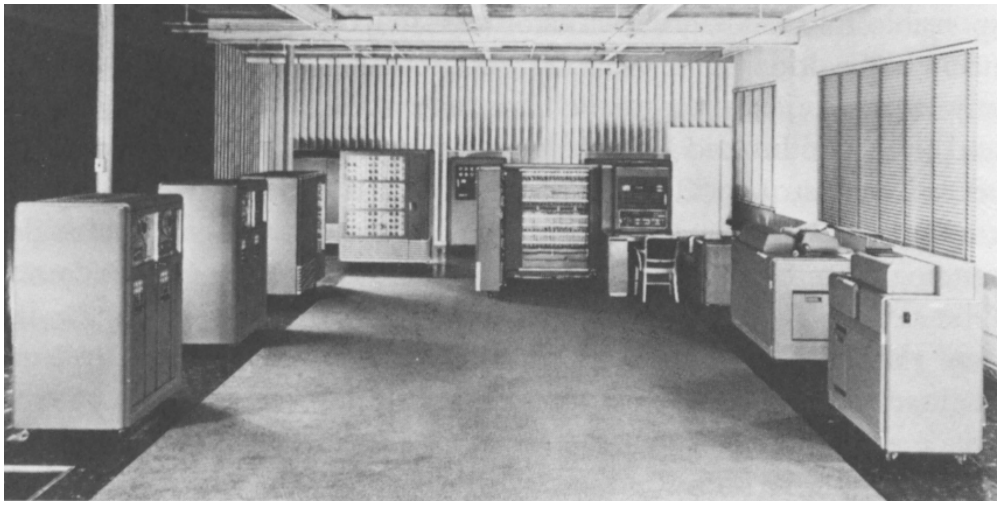
Jednostka arytmetyczna SSEC mogła wykonać mnożenie 14 × 14 bitów w około 67 ms, dzielenie w 33 ms, a dodawanie lub odejmowanie w 0,3 ms. Jego pamięć składała się z lamp próżniowych (szybkich) i przekaźników (wolnych). Karty perforowane i taśmy papierowe mogły być używane do wprowadzania i wyprowadzania programów i danych. Ponadto zawierała drukarkę i terminal kontrolny. Kod programu był zapisywany na kartach lub taśmach perforowanych, które zawierały binarną reprezentację programu zrozumiałą dla komputera.

Ponieważ SSEC nie był komputerem z zapisanym programem w swojej pamięci, programy były uruchamiane bezpośrednio z taśmy lub karty. Aby wielokrotnie powtarzać program na taśmie dziurkowanej, taśma musiała być sklejona w pętli. Pewnego razu

program zachowywał się dziwnie: naprzemiennie zachowywał się poprawnie i niewłaściwie. Okazało się, że taśma została sklejona we wstęgę Möbiusa.

SSEC służył do wykonywania obliczeń naukowych. Na przykład był używany w fizyce jądrowej, w tym w obliczeniach dotyczących pierwszej amerykańskiej bomby wodorowej. Głównym projektem, który John Backus wykonał na maszynie, było obliczenie tabel orbit księżycowych. Tabele te zostały później wykorzystane w projekcie Apollo.

Następcą IBM SSEC był IBM 701 o kryptonimie „Defense Calculator”. Był to pierwszy komercyjny komputer elektroniczny na dużą skalę, zaprojektowany i zbudowany przez IBM. Został stworzony, aby pomóc Stanom Zjednoczonym w działaniach wojennych w Korei w tamtym czasie, stąd jego nazwa „Defense Calculator".



Ostatecznie zbudowano 19 maszyn IBM 701, które wysłano do uniwersytetów, laboratoriów badawczych i dużych firm. Maszyna ta była pierwszym w pełni elektronicznym komputerem IBM z programami przechowywanymi w pamięci komputera. Zawierał procesor, który mógł wykonać 36 × 36-bitowe mnożenie lub dzielenie w 456 mikrosekund oraz dodawanie lub odejmowanie w 60 mikrosekund. Pamięć składała się z 72 lamp Williamsa o łącznej pojemności 2048 słów po 36 bitów. Do trwałego zapisywania danych i wejścia/wyjścia miał dysk magnetyczny, czytniki i nagrywarki taśm magnetycznych, czytniki kart i dziurkacze oraz drukarkę.

SSEC oraz 701 obsługiwał tylko arytmetykę liczb całkowitych, podczas gdy większość problemów, z jakimi borykał się wtedy Backus, to obliczenia naukowe, które wymagały dodatkowego „współczynnika skalującego” (wymyślonego przez Johna von Neumanna). Liczba wraz ze współczynnikiem skalującym to teraz liczba zmiennoprzecinkowa. Aby pisanie programów obliczeniowych było szybsze i mniej podatne na błędy, a przez to tańsze, Backus wynalazł system Speedcoding. Ten system miał interpreter liczb zmiennoprzecinkowych.

W 1953 roku było powszechnie wiadomo, jak zbudować komputer, jaki znamy dzisiaj, oparty na architekturze von Neumanna. W skład takiego komputera wchodziły: procesor, pamięć zawierającą zarówno dane, jak i programy oraz różne urządzenia wejścia-wyjścia do trwałego zapisywania danych i komunikacji z otoczeniem. Programowanie tych urządzeń odbywało się poprzez ładowanie kodu maszynowego z jakiegoś nośnika pamięci (karty perforowane lub taśmy magnetyczne) do pamięci głównej RAM.

Większość kodu maszynowego była pisana ręcznie lub przy pomocy prymitywnego asemblera. Istniały jednak systemy „automatycznego programowania", które zwykle próbowały obejść ograniczenia maszyny, dostarczając interpreter dla maszyny wirtualnej, która miała pożądane funkcje. Przykładem takiego systemu był Speedcoding Backusa.

Inny przykład to system A-2 stworzony przez Grace Murray Hopper (https://www.komputerswiat.pl/artykuly/redakcyjne/grace-hopper-wizjonerka-ktora-zmieniala-swiat-programowania/t7jrcq7). Był podobny, z tą różnicą, że program był raczej kompilowany do kodu maszynowego niż interpretowany. Oba podejścia były powolne, ponieważ większość czasu spędzano na symulowaniu operacji zmiennoprzecinkowych.

W tym czasie było kilka obiecujących teoretycznych podejść do tego, co dziś nazwalibyśmy językami programowania wysokiego poziomu. Pojawił się nawet opis kompilatora dla języka, ale brakowało działającego kompilatora dla prawdziwego komputera. Języki należące do tej klasy to: Plankalkul (autorstwa Zuse), Flow Diagrams (autorstwa Goldstine i von Neumanna), Composition (autor: Curry), IntermediatePL (autor: Burks), Klammerausdrucke (autor: Rutishauser) oraz Formules (autor: Bohm).

Istniały wówczas również eksperymentalne języki programowania „wysokiego poziomu", które miały działający kompilator. Języki te zazwyczaj nie były zbyt ekspresyjne i były zależne od maszyny. Co więcej, kompilator nie tworzył bardzo wydajnego kodu maszynowego. Przykłady to: Short Code (autor: Mauchly), A-2 (autor: Hopper) i Algebraic Interpreter (autor: Laning i Zierler).

Kompilatory te generowały niewydajny kod maszynowy, więc programiści podchodzili do nich bardzo sceptycznie. Ręczne pisanie programów dla maszyn było samo w sobie wystarczająco trudne, a cóż dopiero poprawianie kompilatora.

Następcą IBM 701 był IBM 704, miał większą pamięć.

Backus nalegał na włączenie sprzętowej obsługi zmiennoprzecinkowej i indeksacji, ponieważ Speedcoding był sporym sukcesem już dla IBM 701. Początkowo Gene Amdahl, główny projektant 704, nie był przekonany, więc Backus postanowił sam zaprojektować sprzęt zmiennoprzecinkowy. Przedstawił swój projekt projektantom 704 i chociaż jego projekt był bardzo niezdarny, przekonał ich, że sprzęt zmiennoprzecinkowy powinien być częścią 704. Oprócz procesora obsługującego system zmiennoprzecinkowy i indeksację, komputer ten zawierał pamięć z rdzeniem magnetycznym zamiast wolniejszych lamp Williamsa. Procesor był około dwa razy szybszy niż jego poprzednik, chociaż operacje zmiennoprzecinkowe były wolniejsze niż operacje na liczbach stałoprzecinkowych. Inne komponenty były podobne do IBM 701: dyski magnetyczne, taśmy magnetyczne i karty perforowane; drukarka i terminal operacyjny do sterowania maszyną.

Backus był przekonany, że nowe automatyczne programowanie będzie zaakceptowane tylko wtedy, jeśli będzie generować programy prawie tak wydajne, jak programy pisane ręcznie. Według Backusa koszt zaprojektowania oprogramowania był już wtedy wyższy niż zakup sprzętu i że różnica ta będzie się zwiększać, jak tylko komputery staną się szybsze i tańsze, a programy staną się większe.

W grudniu 1953 roku zaproponował szefowi IBM (Cuthbert Hurd) zaprojektowanie takiego systemu komputera IBM 704. Zostało to zatwierdzone przez IBM. W listopadzie 1954 r. powstał wstępny raport: Specications for The IBM Mathematical FORmula –

TRANslating System FORTRAN. Ten język programowania operował na dwóch typach danych: liczbach stałych (całkowitych) i liczbach zmiennoprzecinkowych, oba o skończonym rozmiarze. Dla obu typów zdefiniowano składnię literałów i zmiennych. Zmienne mogą mieć maksymalnie 2 znaki, z czego pierwszy znak określał typ zmiennej: \i" do \n" określał zmienną całkowitą, podczas gdy wszystkie pozostałe znaki alfanumeryczne określały zmienną zmiennoprzecinkową. Można było operować na tych typach danych, tworząc wyrażenia, używając elementarnych operatorów i funkcji matematycznych.

Najważniejszą częścią języka były:

- instrukcje przypisania (nazywane w raporcie formułami arytmetycznymi), które przypisywały wartość wyrażeniu do zmiennej,
- wykonanie warunkowe i iteracyjne.

Warunkowa instrukcja wykonania IF była bardzo ograniczona: tylko kilka prostych wyrażeń porównujących liczby. Na przykład:

```
10 IF I > 0 12,11
11 I = -I
12 J = J + I
```

Iteracyjna instrukcja wykonania DO

Iteracja była kontrolowana przez zmienną, która zaczynała się od określonej wartości, a następnie sukcesywnie zwiększana aż do osiągnięcia wartości końcowej. To bardzo przypomina `for` w językach takich jak Pascal. Na przykład:

```
10 DO 11,12,13 J=1,10,1
11 C = C + A(J)
12 D = D + B(J)
13 E = C / D
```

Oprócz tego były tam: skok bezwarunkowy GO TO, STOP (program przestaje się wykonywać), instrukcje odczytu i zapisu do wykonywania operacji we/wy. Brak (w pierwszej wersji) definicji procedur i funkcji.

Fortran I miał już większość podstawowych składników współczesnego imperatywnego języka programowania. Miał zmienne, wyrażenia, przypisania, struktury kontrolne do operowania na tablicach liczb całkowitych oraz funkcje zdefiniowane przez użytkownika. Brakowało struktury blokowej; wszystkie zmienne były globalne i nie było dostępnej pamięci dynamicznej (takiej jak stos), więc rekurencja była wtedy jeszcze niemożliwa.

Obecnie FORTRAN III to pełny język programowania do obliczeń w fizyce i technice.

Pioneer Day Banquet, National Computer Conference, Houston, Texas, June 9, 1982. Od prawej strony: Richard Goldberg, Robert Nelson, Lois Haibt, Roy Nutt, Irving Ziller, Sheldon Best, Harlan Herrick, John Backus i Peter Sheridan.

Backus 1995:

> *Nie wiedzieliśmy, czego chcemy i jak to zrobić. Po prostu robiliśmy to. Pierwsza walka toczyła się o to, jak będzie wyglądał język. Następnie jak parsować wyrażenia – to był duży problem, a to, co zrobiliśmy, wygląda teraz zdumiewająco niezgrabnie.*

Język ten był ważnym krokiem w programowaniu: wykazał, że można zaimplementować język programowania wysokiego poziomu w celu wygenerowania wydajnego kodu maszynowego. Fortran spowodował dalszy rozwój języków programowania. Położył podwaliny pod rozwój innych języków programowania wysokiego poziomu, z których najważniejszymi są Algol, Cobol i Lisp, które z kolei miały ogromny wpływ na ich następców.

11.1 CDC & CRAY

W USA, w firmie Control Data Corporation (CDC), powstawały superkomputery projektowane przez Seymoura Craya (fot.) używające innowacyjnych rozwiązań oraz równoległości od osiągnięcia rewelacyjnych (jak na te czasy) szybkości obliczeniowych. Jego zespół (kilkunastu inżynierów) pracował w wynajętym garażu, daleko od centrum i biur CDC.

Komputer CDC 6600, zrealizowany w 1964, jest uważany za pierwszy superkomputer. Był 3 razy szybszy od IBM 7030 Stretch, miał wydajność rzędu 1 megaFLOPS, (FLoating-point Operations Per Second). CDC 6600 był najszybszy od 1964 do 1969 r., kiedy zastąpił go CDC 7600. Dla przykładu – współczesny procesor single-core 2,5 GHz ma wydajność rzędu 10 gigaFLOPS.

Do 1960 roku Cray zaprojektował CDC 1604, poprawiona wersja niedrogiego (?) w stosunku do wydajności ERA 1103. Cray pracował dalej nad nowymi (CDC 3000 series), lecz szefowie firmy chcieli niedrogiego komputera do użytku biznesowego.

Ambicją Cray'a było tworzenie najszybszych komputerów na świecie. Po zaprojektowaniu CDC 3000 series, zaczął pracować nad CDC 6600. Porównując, hardware CDC 6600 nie był najlepszy, ale genialne były rozwiązania Craya dotyczące architektury procesora i I/O .

Słowa Cray'a: *Anyone can build a fast CPU. The trick is to build a fast system*.

CDC 6600 był pierwszym komercyjnym superkomputerem, bijącym na głowę konkurencję; sprzedaż pierwszego egzemplarza zwracała wszystkie koszty projektowe; był drogi, ale najszybszy! Potężne IBM próbowało konkurować (Stretch/IBM 7030), ale nie dało rady!!! Powód to tzw. *imprecise interrupts*. W CDC 6600 obsługa przerwań I/O została delegowana do 12 peryferyjnych procesorów (*built-in mini-computers*) odpowiedzialnych za transfer do i z centralnej pamięci. Optymalizacja tej metody doprowadziła do 5 razy szybszego CDC 7600.

11.2 Pionier polskiej informatyki – Jacek Karpiński

Inż. Jacek Karpiński, twórca K-202, czyli pierwszego polskiego minikomputera, zmarł w wieku 82 lat, 22 lutego 2010 roku. Odznaczony był Krzyżem Walecznych za zasługi w Szarych Szeregach. K-202 to polski 16-bitowy minikomputer, opracowany i skonstruowany w latach 1970–1973. To pierwszy polski komputer zbudowany z użyciem układów scalonych, w kooperacji polskich zakładów MERA Metroneks z firmami angielskimi: Data-Loop oraz M.B. Metals. Według opinii dr. hab. Piotra J. Durki K-202 przewyższał pod względem szybkości pierwsze IBM PC oraz umożliwiał wielozadaniowość, wielodostępność i wieloprocesorowość.

Jacek Karpiński był uznanym elektronikiem. Maszyna AAH do prognoz pogody oraz AKAT-1 i pierwszy analizator równań różniczkowych korzystający z tranzystorów – to właśnie jego dzieła.

Rozdział 12. Podsumowanie

Komputery są bezużyteczne. Mogą ci tylko dać odpowiedzi.

Pablo Picasso (1881-1973). Hiszpański malarz, rzeźbiarz, grafik i ceramik.

Czy to znaczy, że zadawanie pytań (nawet głupich) jest esencją intelektu?
Owszem, komputery dają tylko odpowiedzi, ale robią to bardzo szybko! Jednak nie na wszystkie pytania – tylko na niewiele pytań te odpowiedzi są sensowne. Trzeba wiedzieć, jakie to pytania i jak je formułować, żeby odpowiedzi były sensowne. Te pytania to „dobry" software.

Współczesne Open AI czy ChatGPT oraz Copilot dają „dobre" odpowiedzi, bazując na olbrzymiej ilości informacji (niekoniecznie spójnej, nie mówiąc już, czy prawdziwej) gromadzonej latami w Data Centers (inaczej *clouds*) należących do AWS, Google, Microsoftu i nie tylko. To są oczywiście aplikacje (software) oparte na heurystycznych metodach, takich jak np. *deep learning*.
Co z tego wyniknie? Czy jest to następna bańka, podobna do tej z 2000 roku?
Czas pokaże. Na pewno przyniesie znaczny postęp (być może i technologiczny) w obszarach, w których nie można się tego teraz spodziewać.

Czy mogę do ciebie oddzwonić później? Właśnie kupiliśmy nowy komputer i próbujemy go skonfigurować zanim stanie się przestarzały
(ok. 2000 r.)

Niżej jest trochę o zagrożeniach, jakie przyniósł „postęp" w rozwoju współczesnych procesorów. Intencyjnie w oryginale, bez tłumaczenia.
https://www.theverge.com/2024/7/26/24206529/intel-13th-14th-gen-crashing-instability-cpu-voltage-q-a

There is no fix for Intel's crashing 13th and 14th Gen CPUs — any damage is permanent.

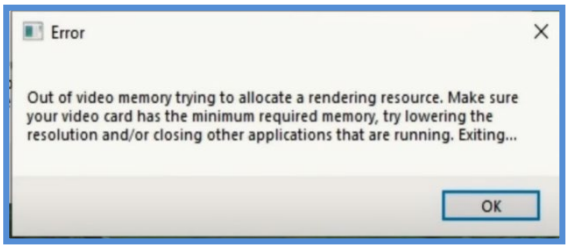
https://www.pcworld.com/article/2415697/intels-crashing-13th-14th-gen-cpu-nightmare-explained.html

"Intel has determined that elevated operating voltage is a primary cause of the instability issues in some 13th and 14th Gen desktop processors", an Intel spokesman told PCWorld. "Analysis of

returned processors confirms that the elevated operating voltage is stemming from a micro-code algorithm resulting in incorrect voltage requests to the processor".

How do you know if you're affected?

Application crashes, blue screens, system crashes—these are all symptoms of critical PC issues, but they're usually vague and infrequent. Not in this case, though. The typical error message for affected Intel CPUs often looks like this:



The error message apparently indicates a problem with the graphics card, but in reality the CPU is the culprit, thanks to the flawed microcode algorithm.

The microcode is the CPU firmware that defines important parameters, such as how much voltage the CPU requires from the motherboard. And this is exactly where the problem lies, because the voltage specifications are apparently too high. According to Intel, a too-high CPU voltage supply can occur not just under load, but also when idle.

Rozdział 13. Zadania na kurs *laboratorium z programowania w asemblerze na symulatorze sms32v50*

Rozwiązaniem każdego zadania jest kod w asemblerze uruchomiony na symulatorze https://nbest.co.uk/Softwareforeducation/sms32v50/index.php (do ściągnięcia i uruchomienia na własnym PC). Zadania są tak kolejno ułożone, że samodzielne rozwiązanie przez studenta zadania o numerze n+1 znaczy tyle, że potrafi on rozwiązać zadanie o numerze n.

UWAGA: są używane TYLKO liczby w systemie heksadecymalnym od 00 do FF.
Działania arytmetyczne są w **U2** na 8-bitowej reprezentacji.

Najbardziej znaczący bit (ten, który stoi po lewej stronie) określa znak liczby. Jeśli jest to **jedynka,** to liczba jest **ujemna**, w przeciwnym razie jest ona **dodatnia.** Więcej – patrz rozdział **4.1 Programowanie.**

Tabela kodów ASCII jest przydatna do programowania w asemblerze.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Zadanie 1

Wyświetlić na monitorze VDU swoją datę urodzenia w formacie dd-mm-rrrr. Należy użyć tabeli ASCII. Znaleźć tam kody znaków (liczby heksadecymalne dwucyfrowe) i skopiować do kolejnych komórek pamięci, poczynając of adresu C0.

; kody ASCII kolejnych cyfr
30
31
32
...
39
; kod ASCII myślnika - to 2D
Np. dla daty urodzenia 1 maja 2000:

MOV AL, 30
MOV [C0], AL
MOV AL, 31
MOV [C1], AL
MOV AL, 2D
MOV [C2], AL
…
Na VDU pojawi się napis „01-”
Itd.

Zadanie 2

Napisać program, który w komórkach RAM pod adresami [C0], [C1], [C2], [C3], [C4], ... itd. zapisze kody ASCII kolejnych liter w zdaniu „ala ma kota”. Po uruchomieniu programu na wyświetlaczu VDU powinno być wyświetlone to zdanie.

Wskazówka 1: należy użyć kolejno instrukcji.

Kod ASCII litery „a” to 61.

MOV AL, 61 ; wstaw 61 do rejestru AL; 61 jest to liczba heksadecymalna
; kodująca znak „a”

MOV [C0], AL ; kopiuj to, co jest w AL, do komórki pamięci o adresie C0;
jest to pierwsza komórka pamięci VDU (wyświetlacza – ekranu), na wyświetlaczu VDU w pierwszym wierszu od góry od lewej strony pojawi się znak „a”. Następna komórka w VDU ma adres [C1], potem [C2], ... , aż do [FF], a jest to ostatni (czwarty) wiersz w VDU i pierwszy znak od prawej strony w tym wierszu.

MOV AL, 6C ; wstaw 6C do rejestru AL; 6C jest to liczba heksadecymalna
kodująca znak „l”

MOV [C1], AL ; kopiuj to, co jest w AL, do komórki pamięci o adresie C1; jest to kolejna (druga) komórka pamięci VDU ... itd.

Wskazówka 2: można prościej, bez odwoływania się do tabeli ASCII.

```
JMP START
DB A1                ; w RAM, w komórce o adresie 02, czyli w komórce [02]
                     ; będzie liczba heksydecymalna A1
DB ”ala ma kota”     ; w RAM, począwszy od adresu 03, będzie to zdanie

START:
MOV AL, [03]
MOV [C0], AL
MOV AL, [04]
MOV [C1], AL
                     ; i tak dalej aż do adresu [13]
END
```

Zadanie 2a

Wypisać na VDU zdanie „12+34= ”

```
MOV DL, 31
MOV [C0], DL     ; jedynka wstawiona do VDU
MOV DL, 32
MOV [C1], DL     ; dwójka wstawiona do VDU
MOV DL, 2B
MOV [C2], DL     ; + wstawiony do VDU
...
ltd.
```

Zadanie 3

To samo, co w zadaniu 2a, tylko wpisywanie bezpośrednio z klawiatury rzeczywistej za pomocą instrukcji IN 00, która powoduje wpisanie kodu ASCII kolejnego wciśniętego klawisza z klawiatury do rejestru AL.

```
IN 00            ; czyta kod ASCII znaku z klawiatury do AL
MOV [C0], AL
IN 00
MOV [C1], AL
IN 00
MOV [C2], AL
IN 00
MOV [C3], AL
IN 00
MOV [C4], AL
IN 00
MOV [C5], AL     ; na VDU powinno być „12+34= ”
```

Następnie obliczyć w kodzie sumę, a wynik wyświetlić na VDU.

Wskazówka: należy zamienić dziesiętne liczby 12 oraz 34 na układ szesnastkowy w notacji U2. Dodać je w kodzie.

```
MOV AL, [C0]
SUB AL, 30      ; z kodu ASCII dostajemy tę cyfrę
MUL AL, 0A      ; mnożymy przez 10, heksadecymalnie przez 0A

MOV BL, [C1]
SUB BL, 30      ; z kodu ASCII dostajemy tę cyfrę
ADD AL, BL      ; w AL jest 0C, czyli heksadecymalnie to samo co dziesiętnie 12

MOV CL, [C3]
SUB CL, 30      ; z kodu ASCII dostajemy tę cyfrę
MUL CL, 0A      ; mnożymy przez 10, heksadecymalne przez 0A

MOV DL, [C4]
SUB DL, 30      ; z kodu ASCII dostajemy tę cyfrę
ADD CL, DL      ; w CL jest 22, czyli heksadecymalne to samo co dziesiętnie 34

ADD AL, CL      ; w AL jest 2E ; to samo co dziesiętnie 12+34
```

Następnie należy obliczyć (w kodzie) cyfrę dziesiątek tej sumy, a potem cyfrę jedności.

Nie można kopiować z rejestru bezpośrednio do innego rejestru, tak jak np. MOV BL, AL. Można użyć np. ostatniej komórki w RAM, tj. [FF].

```
MOV [FF], AL
MOV BL, [FF]            ; w BL jest 2E
```

Wyświetlić kolejno kody ASCII tych cyfr na VDU po znaku „=”.

```
DIV AL, 0A      ; dzielenie całkowite przez 10, wynik (cyfra) jest w AL
ADD AL, 30      ; w AL jest kod ASCII tej cyfry
MOV [C6], AL    ;
MOD BL, 0A      ; dzielenie modulo, czyli reszta z dzielenia przez 10
ADD BL, 30      ; w BL jest kod ASCII tej reszty
MOV [C7], BL    ;
END
```

Zadanie 4

Zamiast do pamięci VDU, wpisujemy z klawiatury zdanie „ala ma kota” na stos, który ma zarezerwowane komórki pamięci RAM o adresach kolejno B0, B1, B2, ... , BD, BE, BF (czyli całą linię B w RAM).

Instrukcja PUSH AL powoduje wpisanie do kolejnej wolnej komórki na stosie tego, co jest w rejestrze AL. Z tym, że kierunek jest odwrotny, tj. najpierw do komórki o adresie BF, następnie do komórki o adresie BE, potem do komórki o adresie BD, ... itd. aż do B0.

Instrukcja POP AL powoduje zdjęcie ze stosu tego, co ostatnio zostało tam włożone, do rejestru AL.

```
IN 00
PUSH AL
IN 00
PUSH AL
                ;... itd.
                ; a następnie ze stosu ma VDU
POP AL
MOV [C0], AL
POP AL
MOV [C1], AL
POP AL
MOV [C2], AL
END
```

Zadanie 4a

To samo co w zadaniu 4, tylko że wczytywanie DOWOLNEGO ZDANIA (nie więcej niż 16 znaków) za pomocą klawiatury. **Pierwszym i ostatnim wczytanym znakiem jest Enter** (kod ASCII to 0D). Należy zastosować pętle.

Po każdym wczytanym znaku z klawiatury (IN 00) sprawdzać, czy nie jest to Enter, czyli czy kod tego znaku nie jest równy 0D, tj. wykonywać instrukcję:

```
CMP AL, 0D
```

a następnie sprawdzać instrukcją:

```
JZN skocz_do
```

„skocz_do” jest etykietą, którą należy umieścić na początku linii w kodzie, gdzie należy wykonać skok, jeśli wczytanym znakiem (z klawiatury) NIE JEST Enter.

W kodzie jest to następująca pierwsza pętla:

```
IN 00                   ; pierwszy znak Enter
PUSH AL
skocz_do_1:
IN 00
PUSH AL
CMP AL, 0D
JNZ skocz_do_1  ; jeśli jest Enter, to przejdź do następnej instrukcji
```

Dalej wczytywanie ze stosu do pamięci VDU, to druga pętla z użyciem licznika w rejestrze CL:

```
MOV CL, C0          ; licznik CL ustawiony na adres pierwszej komórki VDU
POP AL              ; pomijamy ostatni Enter
skocz_do_2:
POP AL
MOV [CL], AL        ; wpisujemy do kolejnej komórki pamięci VDU to, co jest w
                    ; rejestrze AL. UWAGA: trzeba użyć licznika adresów tych
                    ; kolejnych komórek, tutaj rejestru CL, i za każdy razem
                    ; zwiększać ten rejestr o 1, tj. INC CL
INC CL
CMP AL, 0D
JNZ skocz_do_2
END
```

Można też tak, bez założenia, że pierwszy wczytany znak to ENTER. Licznik DL – ile razy na stos.

```
MOV DL, 00          ; wyzerowany licznik DL
Skocz_do1:
IN 00
PUSH AL
INC DL
CMP AL, 0D
JNZ Skocz_do1
MOV CL, C0          ; licznik CL ustawiony na adres pierwszej komórki VDU
Skocz_do2:          ; ze stosu na VDU
POP AL
MOV [CL], AL
INC CL
DEC DL
CMP DL, 00
JNZ Skocz_do2
END
```

Zadanie 4b

Zamień liczbę dodatnią x mniejszą niż 128 (wczytaną z IN 00 jako znaki ASCII) na ujemną –x i wyślij na VDU.

Np.

- wczytaj dziesiętnie 26 (zamień na notację U2 w reprezentacji hex, tj. 1A) . Oblicz liczbę szesnastkową Y odwrotną do 1Al, tj. 1A + Y = 0. Zamień Y na -26. Wyświetl na VDU.

Wskazówka: Załóżmy, że wczytaną liczba dziesiętną jest x, zaś hex(x) to jej reprezentacja w U2. Szukana liczba heksadecymalna Y (odwrotna do hex(x)) ma być w rejestrze DL. Czyli powinien być spełniony warunek hex(x) + DL=0.

Jak zamienić wczytane cyfry liczby dziesiętnej na hex jest w zadaniu 3. Załóżmy, że hex(x) jest już w rejestrze BL.

```
MOV AL, 00        ; sprawdzaj od AL=0 co -1
skok:
PUSH BL
POP DL
DEC AL
ADD DL, AL
CMP DL, 00
JNZ skok
END               ; w DL jest Y
                  ; zamień na dziesiętna reprezentację: pamiętaj o znaku minus
                  ; zamień na znaki ASCI i wyświetl na VDU
END
```

Można inaczej, wykorzystując wzór na U2 na bajcie (8 bitach), gdzie lewy skrajny bit jest bitem znaku -1, i operację bitową NOT.

Załóżmy, że hex(x) jest już w rejestrze BL.

```
NOT BL
ADD DL, 01                ; w DL jest Y
```

Zadanie 5

Liczby heksadecymalne wpisywane kolejno do RAM, np. początek kodu:

```
JMP Start
DB 10             ; liczba heksydecymalna 10 jest wpisywana (jako bajt) do komórki
                  ; RAM o adresie 02

DB A5             ; liczba heksydecymalna A5 jest wpisywana (jako bajt) do komórki
                  ; RAM o adresie 03

DB 5C             ; liczba heksydecymalna 5C jest wpisywana (jako bajt) do komórki
                  ; RAM o adresie 04
DB ''przepelnienie'' ; wpisanie do RAM kodów ASCII kolejnych znaków w wyrazie „przepelnienie”,
                     począwszy od adresu 05

DB ''MINUS"
DB ''ZERO"
DB ...    ; itd. ewentualnie inne dane
Start:    ; jest to etykieta, od której procesor zaczyna wykonywanie programu
```

Wpisać do tabeli danych: dwie dowolne liczby heksadecymalne x oraz y, a następnie wyraz ”przepelnienie”, oraz ‘’MINUS” i ‘’ZERO”.

Wykonać na nich działania: dodawania, odejmowania, mnożenia, dzielenia całkowitego oraz reszty z dzielenia, czyli instrukcje ADD, SUB, MUL, DIV, oraz MOD.

Obsługa flagi przepełnienia O za pomocą instrukcji skoku JO oraz JNO.

Jeśli flaga O jest ustawiona po wykonaniu działania, to należy wyświetlić na VDU w wierszu C, wyraz ”przepelnienie” wpisany poprzednio do tabeli danych.

Jeśli wynik jest ujemny (sprawdzenie flagi S za pomocą instrukcji skoku JS lub JNS), to wypisać na VDU w wierszu D wyraz ‘’MINUS’’ z DB.

Jeśli wynik jest zero (sprawdzenie flagi Z za pomocą instrukcji skoku JZ lub JNZ), to wypisać na VDU w wierszu E, wyraz ‘’ZERO’’ z DB.

Zadanie 6. Prosty kalkulator na liczbach dziesiętnych

Zaczynamy od liczb jednocyfrowych. Wpisać do kolejnych (poczynając od adresu 02) dwóch komórek dwie dowolne liczby heksadecymalne x oraz y **(z zakresu: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; czyli x oraz y są cyframi dziesiętnymi)**,

JMP Start

DB 05

DB 09

Start:

a następnie wyświetlić ich kody ASCII (dodać 30) na VDU dla działania arytmetycznego (+, -, *, /, mod) w postaci:

$x_{(10)} + y_{(10)} =$; dodawanie

$x_{(10)} - y_{(10)} =$; odejmowanie

$x_{(10)} * y_{(10)} =$; mnożenie

$x_{(10)} / y_{(10)} =$; dzielnie całkowite

$x_{(10)}$ (mod $y_{(10)}$) = ; reszta z dzielenia x przez y

$x_{(10)}$ oraz $y_{(10)}$ to odpowiednio kody ASCII cyfr x oraz y.

Żeby otrzymać $x_{(10)}$ należy: skopiować zawartość komórki o adresie 02 (tam, gdzie został wpisany x) do rejestru AL

MOV AL, [02]

a następnie dodać 30, czyli

ADD AL, 30

wtedy w rejestrze AL jest $x_{(10)}$ czyli kod ASCII cyfry x.

Analogicznie dla cyfry y oraz $y_{(10)}$

MOV BL, [03]

a następnie dodać 30, czyli

ADD BL, 30

Do VDU skopiować $x_{(10)}$ oraz $y_{(10)}$. Następnie wstawić w odpowiednie miejsce kod ASCII znaku działania (+, -, *, /, mod z nawiasami) oraz kod ASCII znaku =

Z DB skopiować x oraz y do odpowiednich rejestrów.

MOV AL, [02]

MOV BL, [03]

Wykonać odpowiednie działanie, tj. jedną z instrukcji ADD, SUB, MUL, DIV, lub MOD. Np. dla mnożenia jest to:

MUL AL, BL.

Wynik działania dla mnożenia oraz dodawania może być dwucyfrową liczbą dziesiętną.

Cyfry otrzymujemy, wykonując następujące działanie i zakładając, że wynik jest w rejestrze AL.

```
PUSH AL
POP BL          ; kopiujemy zawartość AL do BL
MOD BL, OA      ; 0A jest to dziesięć w systemie dziesiętnym; po wykonaniu
                ; działania w rejestrze BL jest cyfra jedności (reszta z dzielenia przez 0A)
DIV AL, 0A      ; po wykonaniu instrukcji w rejestrze AL jest cyfra dziesiątek.
```

Do tego, co jest w rejestrach AL oraz BL, dodajemy 30 i wyświetlamy na VDU po znaku równości =.

MOV [C5], AL

MOV [C6], BL

A jak to zrobić dla odejmowania? Problem jest, jak wynik jest ujemny! Można to sprawdzić poprzez flagę S.

JNS koniec

MOV CL, (kod ASCII znaku -)

MOV [...], CL ; w komórce przed wpisaniem wyniku odejmowania

koniec:

Zadanie 6a

Działania na liczbach dwucyfrowych.

Działanie jest wypisane z klawiatury rzeczywistej (IN 00) na VDU od początku wiersza C w postaci (dla dodawania):

xy + zv =

Przykładowy pseudokod:

```
MOV CL, 09      ; licznik wpisywanych znaków z klawiatury, łącznie ze spacjami
                ; za każdym wpisanym znakiem będzie zmniejszany o 1
MOV DL, C0      ; licznik adresów komórek w VDU
petla1:
IN 00
MOV [DL], AL
```

```
INC DL
DEC CL
CMP CL, 00      ; sprawdzanie, ile znaków zostało już wprowadzonych z klawiatury
JNZ petla1

MOV AL, [C0]
SUB AL, 30
MUL AL, 0A
PUSH AL

MOV AL, [C1]
SUB AL, 30
PUSH AL

MOV AL, [C4]
SUB AL, 30
MUL AL, 0A
PUSH AL

MOV AL, [C4]
SUB AL, 30
PUSH AL


POP AL
POP BL
ADD BL, AL      ; druga liczba jest w BL

POP AL
POP DL
ADD DL, AL      ; pierwsza liczba jest w DL
ADD BL, DL      ; dodajemy te liczby, wynik jest w BL

                ; dalej należy wyliczyć cyfry setek, dziesiątek i jedności
                ; dodać 30 i otrzymać znaki ASCII.
                ; Wypisać kolejno w komórkach [CA], [CB] i [CC] na VDU
END
```

Zadanie 7

Prosty kalkulator na liczbach dziesiętnych dwucyfrowych **x** oraz **y** z zakresu od -128 do +127. Wczytywanie liczb x oraz y z klawiatury numerycznej symulowanej (numer przerwania 04). Dalej, podobnie jak w zadaniu 6, dodatkowo obsługa flagi przepełnienia za pomocą instrukcji skoku

JO oraz JNO. Jeśli wystąpiło przepełnienie, to jest wyświetlany na VDU tylko komunikat błędu, tj. wyraz ”przepelnienie” po znaku =.

Zadanie 8

Prosty kalkulator na dowolnych liczbach nieujemnych dziesiętnych dwucyfrowych (trzycyfrowych) **x** oraz **y.** Należy opracować własną metodę obliczeń, ponieważ instrukcje ADD, MUL mogą prowadzić do przepełnień. Dla liczb trzycyfrowych reprezentacja jednobajtowa nie wystarczy.

Program ma działać w następujący sposób:

wczytać z klawiatury (instrukcja IN 00) do VDU:

kolejne cyfry liczby x,

potem znak działania,

następnie dwie kolejne cyfry liczby y,

potem znak równości = .

Wykonać działanie, a wynik wyświetlić na VDU po znaku =.

Dla wszystkich działań arytmetycznych może nie wystarczyć pamięci RAM. Wystarczy zrobić dla dodawania i mnożenia.

Zadanie 9

Wpisać do tabeli danych zdanie: ‘’ala ma kota’’ .

Skopiować to zdanie z RAM do wyświetlacza VDU.

Kod jest z grubsza następujący:

```
JMP Start
DB ”ala ma kota” ; wpisanie do RAM kodów ASCII kolejnych znaków w zdaniu ”ala ma kota”,
                   począwszy od adresu 02. Zdanie ma 11 znaków (razem ze spacjami). Czyli
                   ostatni znak będzie wpisany do komórki pamięci RAM o adresie 0C (jest to 12
                   dziesiętnie).
Start:             ; jest to etykieta, od której procesor zaczyna wykonywanie programu.
MOV BL, 02         ; adres początku (pierwszego znaku) zdania ”ala ma kota” w RAM
                   jest skopiowany do rejestru BL, który będzie licznikiem adresów kolejnych zna-
                   ków w tym zdaniu.
MOV CL, C0         ; adres pierwszej komórki VDU, tj. C0 jest skopiowany do rejestru CL.
Petla:
MOV DL, [BL]       ; rejestr DL jest tutaj pomocniczy, bo nie można bezpośrednio kopiować
                   z komórki do komórki
MOV [CL], DL       ; zawartość komórki o adresie [BL] zostanie skopiowana (poprzez rejestr DL)
                   do komórki o adresie [CL]
INC BL             ; zwiększ zawartość rejestru BL o 1
INC CL             ; zwiększ zawartość rejestru CL o 1
```

```
CMP BL, 0D      ; 0C to jest adres ostatniego znaku zdania ”ala ma kota” w RAM, następna
                  komórka RAM ma adres 0D
JNZ Petla       ; jeśli zawartość rejestru BL jest różna od 0D, to skocz do etykiety Petla. Jeśli
                  jest równa, to wykonaj następną instrukcję
END             ; koniec kodu programu.
```

Zadanie 9a

Wpisać do tabeli danych dwa zdania: ”kota ma ala” oraz ”ala ma kota”.

Skopiować te dwa zdania z RAM do kolejnych dwóch linii wyświetlacza VDU.

Kod z użyciem procedury jest następujący:

```
JMP Start
DB ”ala ma kota’’ ; wpisanie do RAM kodów ASCII kolejnych znaków w zdaniu ”ala ma kota”
DB ”kota ma ala” ; wpisanie do RAM kodów ASCII kolejnych znaków w zdaniu ”kota ma ala”
Start:
ORG 30          ; deklaracja procedury (zapisanej w RAM) zaczyna się od komórki o adresie 30
Petla:
MOV DL, [BL]    ; rejestr DL jest tutaj pomocniczy, bo nie można bezpośrednio kopiować
                  z komórki do komórki
MOV [CL], DL    ; zawartość komórki o adresie [BL] zostanie skopiowana (poprzez rejestr DL)
                  do komórki o adresie [CL]
INC BL          ; zwiększ zawartość rejestru BL o 1
INC CL          ; zwiększ zawartość rejestru CL o 1
CMP BL, AL      ; porównaj zawartości rejestrów BL oraz AL, czyli czy to jest ostatni znak do
                  przepisania
JNZ Petla       ; jeśli zawartość rejestru BL jest różna od zawartości AL, to skocz do etykiety
                  Petla. Jeśli jest równa, to wykonaj następną instrukcję
RET             ; koniec procedury
                ; dokończyć kod, co należy wstawić do rejestrów AL, BL, CL, żeby po dwu-
                  krotnym wywołaniu procedury (instrukcja CALL 30) w pierwszej linii VDU by-
                  ło zadanie ”kota ma ala”, zaś w drugiej linii było zdanie ”ala ma kota”
Rozwiązanie:
                ; parametry: 02, 0C, C0 dla zdania ”ala ma kota”
MOV AL, 02
MOV BL, 0C
MOV CL, C0
CALL 30         ; wywołanie procedury dla powyższych wartości rejestrów

                ; następne parametry: 0D, 18, D0 ‘’kota ma ala”
MOV AL, 0D
MOV BL, 18
```

```
MOV CL,D0
CALL 30          ; wywołanie procedury dla powyższych wartości rejestrów
END              ; koniec programu
```

Zadanie 9b

Rozwiązania zadań 5 i 6 przepisać z użyciem procedur

Zadanie 10

Wczytać z klawiatury zwykłej (IN 00) na VDU (pierwsza linia) **dowolne** zdanie trzy- (lub więcej) wyrazowe. Następnie w drugiej linii VDU wyświetlić w odwrotnej kolejności wyrazów. Np. „ala ma kota" w pierwszej linii, zaś w drugiej powinno być „kota ma ala".

Wskazówka:

Na początku wpisujemy to zdanie do pierwszej linii VDU.

IN 00 ; do rejestru AL wpisywany jest kod ASCII znaku z klawiatury. Kończymy znakiem „Enter".

Adres ostatniego wpisanego znaku w pierwszej linii jest ważny.

Z tej pierwszej linii VDU wczytywać kolejne znaki (w odwrotnej kolejności, poczynając od ostatniego znaku zdania) na stos aż do pierwszego znaku spacji. W ten sposób ostatni wyraz zdania zostanie zapisany na stosie. Należy zdjąć ten wyraz ze stosu i umieścić w drugiej linii VDU, począwszy od adresu D0. Ten ostatni wyraz z pierwszej linii będzie teraz pierwszym wyrazem w drugiej linii.

Następnie należy wczytać kolejny od końca (drugi) wyraz z pierwszej linii VDU (zaczynając od ostatniej litery tego wyrazu) na stos (aż nie napotkamy kodu ASCII spacji). Zdejmujemy teraz ze stosu i kopiujemy do kolejnych komórek w drugiej linii VDU. Kolejny od końca (drugi) wyraz z pierwszej linii VDU jest następnym (drugim) w drugiej linii VDU.

Powtarzamy to samo dla kolejnego (pierwszego) wyrazu w pierwszej linii VDU, który będzie teraz wpisany do drugiej linii VDU jako następny (trzeci, ostatni) wyraz w drugiej linii VDU.

Można to zrobić dla zdania z dowolną ilością wyrazów.

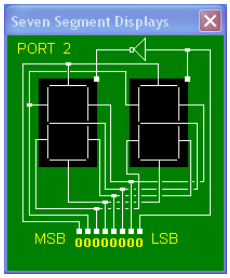
Wkładanie na stos (PUSH) wyrazu z pamięci RAM i zdejmowanie (POP) w inny obszar pamięci RAM powtarzane jest po trzy razy, więc warto zadeklarować procedury i wywoływać je po trzy razy.

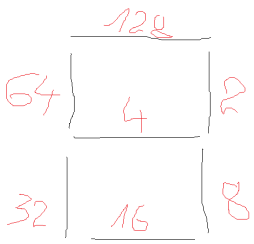
Zadanie 10a

Wczytać z klawiatury symulowanej (numer przerwania 03) na VDU (pierwsza linia) **dowolne** zdanie trzy- (lub więcej) wyrazowe. Następnie w drugiej linii VDU wyświetlić w odwrotnej kolejności wyrazów. Np. "ala ma kota" w pierwszej linii.

Zadanie 11

Obliczyć kody (liczby heksadecymalne bez znaku) poszczególnych cyfr (od 0 do 9) dla prawej i lewej części wyświetlacza siedmiosegmentowego na podstawie następującego rysunku:





Napisać i uruchomić kod na symulatorze, który wyświetla kolejno te cyfry dla lewej strony oraz dla prawej strony.

```
MOV AL, ...     ; kod cyfry jest wpisywany do AL
OUT 02          ; wysyła zawartość AL do portu 02 (wyświetlacz siedmiosegmentowy)

                ; cyfra 1 po lewej stronie 2+8 = 10 = 0A
                ; cyfra 1 po prawej stronie 0A+1 = 0B
                ; cyfra 4 po lewej stronie 64+ 4+2+8 = 78 = 4E
                ; cyfra 4 po prawej stronie 4E+1 = 4F
```

Itd.

Zadanie 12

(a) Wpisać kolejno dwie cyfry z klawiatury (IN 00 lub (trudniejsza wersja) z klawiatury numerycznej symulowanej z przerwaniem sprzętowym), a następnie wyświetlić je na wyświetlaczu siedmiosegmentowym.

(b) Wpisać w RAM za pomocą DB dwie liczby dwucyfrowe heksadecymalne dodatnie, wykonać dzielenie (DIV), resztę z dzielenia (MOD) i wypisać wynik jako liczbę dziesiętną na wyświetlaczu siedmiosegmentowym.

Kod dla (a); łatwiejsza wersja, bez przerwania sprzętowego:

```
JMP Start
DB ....         ; kod cyfry 0 (w wyświetlaczu siedmiosegmentowym) w komórce pamięci [02],
                  tj. o adresie 02
DB ....         ; kod cyfry 1 w komórce pamięci [03], tj. o adresie 03
DB ....         ; kod cyfry 2
...
DB ....         ; kod cyfry 9, w komórce [0B]

Start:          ; jest to etykieta, od której procesor zaczyna wykonywanie programu
IN 00           ; z klawiatury dostajemy na AL kod ASCII cyfry, należy odjąć 30, tj.
SUB AL, 30
ADD AL, 02      ; kod z wyświetlacza dla cyfry x jest w komórce [x+2]
PUSH AL,
POP BL
MOV AL, [BL]
```

```
OUT 02          ; cyfra wyświetlana po prawej stronie
IN 00           ; z klawiatury dostajemy na AL kod ASCII drugiej cyfry, należy odjąć 30, tj.
SUB AL, 30
ADD AL, 02
PUSH AL,
POP BL
MOV AL, [BL]
ADD AL, 01      ; można też tak INC AL; wyświetlamy cyfrę po lewej stronie
OUT 02
END             ; koniec kodu programu
```

Ponieważ kod się (prawie) powtarza od IN 00 do OUT 02, to można (I NALEŻY) zastosować procedurę i wywołać ją 2 razy z różnym parametrem CL, tj. najpierw równym 0, a potem równym 1.

```
ADD AL, CL ; przed instrukcją OUT 02
```

...podwójnie pechowe!

Zadanie 14

Wyświetlać **(z przerwaniem zegarowym co 1 sekunda)** kolejno na wyświetlaczu siedmiosegmentowym liczby 00, 01, 02, ... 10, 11, 12, ... 58, 59, a następnie od 00, 01, ... i tak w kółko.

Zamiast przerwania zegarowego (w wersji łatwiejszej) można, pomiędzy wyświetlanymi liczbami, użyć

```
NOP
NOP
NOP     ; tyle razy ile trzeba
```

Kod:

```
JMP start
DB 20           ; dla ewentualnego przerwania zegara co 1 sek.
DB FA           ; 0, teraz w komórce [3]
DB 0A           ; 1
DB B6           ; 2
DB 9E           ; 3
DB 4E           ; 4
DB DC           ; 5
DB FC           ; 6
DB 8A           ; 7
DB FE           ; 8
DB DE           ; 9
start:
MOV DL, 00      ; rejestr DL to licznik sekundnika
start1:
```

```
MOV [90], DL      ; komórka [90] jest tutaj pomocnicza, do przechowywania stanu licznika DL
MOV BL, [90]
MOD BL, 0A
ADD BL, 03
MOV AL, [BL]
INC AL
OUT 02
MOV BL, [90]
DIV BL, 0A
ADD BL, 03
MOV AL, [BL]
OUT 02
NOP
NOP
NOP
INC DL
CMP DL, 3C        ; 3C to dziesiętnie 60
JNZ koniec
MOV DL, 00
JMP start1
koniec:
JMP start1
END
```

Zadanie 15

Wczytywać na VDU z klawiatury symulowanej (numer przerwania 03, numer portu 07) kolejne cztery (dokładnie 4) zdania najwyżej 15-znakowe. Klawisz Enter oznacza „od nowej linii” na VDU.

Wersja zadania 15: dodatkowo obsłużyć też klawisz „Backspace”, ale tylko dla linii

```
CMP AL, „kod Backspace”
JNZ Skok
MOV [CL], 00
DEC CL
Skok:
                  ; Obsługa Enter:
                  Rejestr CL to licznik na VDU
DIV CL, 10
INC CL
MUL CL, 10
```

Zadanie 16

Wypisywanie cyfr (od 0 do 9) dla prawej i lewej części wyświetlacza siedmiosegmentowego na przemian. Wypisywanie powinno być ciągłe, tj**. nie na jeden raz**.

```
MOV AL, ...     ; kod cyfry wyświetlacza wpisywany do AL
OUT 02          ; wysyła zawartość AL do portu 02 (wyświetlacz siedmiosegmentowy)
```

Wypisywać kolejno dwie cyfry z klawiatury symulowanej numerycznej (IN 08, numer przerwania 04), a następnie wyświetlić je na wyświetlaczu siedmiosegmentowym i tak w kółko, raz po prawej, raz po lewej itd.

```
JMP Start
DB 10           ; adres kodu obsługi przerwania zegarowego
DB 20           ; NIE BĘDZIE UŻYTA klawiatura symulowana pełna
DB 30           ; adres kodu obsługi przerwania klawiatury numerycznej symulowanej
DB ....         ; kod cyfry 0 (w wyświetlaczu siedmiosegmentowym) w komórce pamięci [05],
                  tj. o adresie 05
DB ....         ; kod cyfry 1 w komórce pamięci [06], tj. o adresie 06
DB ....         ; kod cyfry 2
;...
DB ....         ; kod cyfry 9

ORG 10          ; procedura obsługi przerwania zegarowego o numerze 02
NOP
IRET            ; koniec obsługi przerwania zegarowego

ORG 20          ; procedura obsługi przerwania klawiatury pełnej symulowanej numerze 03
NOP
IRET            ; koniec procedury

ORG 30  ; procedura obsługi przerwania klawiatury numerycznej o numerze 04
CLI             ; uniemożliwienie przerwań sprzętowych
CMP CL, 00
JNZ skok
IN 08           ; z klawiatury numerycznej dostajemy na AL kod ASCII cyfry, należy odjąć 30, tj.
SUB AL, 30
ADD AL, 05      ; można SUB AL, 2B zamiast odejmowania i tego dodawania
                ; kod dla cyfry x jest w komórce [x+5]
PUSH AL
POP BL
MOV AL, [BL]
OUT 02
INC CL
```

```
JMP Skok1
Skok:
IN 08           ; z klawiatury dostajemy na AL kod ASCII cyfry, należy odjąć 30, tj.
SUB AL, 30
ADD AL, 05
PUSH AL,
POP BL
MOV AL, [BL]
ADD AL, 01      ; INC AL
OUT 02
DEC CL
Skok1:
STI             ; umożliwienie przerwania sprzętowego
IRET            ; koniec procedury przerwania klawiatury 04

Start:          ; jest to etykieta, od której procesor zaczyna wykonywanie programu
MOV CL, 00
OUT 08          ; pokazanie wyświetlacza siedmiosegmentowego
STI             ; umożliwienie przerwania sprzętowego
Petla:          ; bezczynności
NOP
NOP
NOP
JMP Petla
END             ; koniec kodu programu

Prostszy kod:
JMP Start
DB 10
DB 00
DB 30
DB ....         ; kod cyfry 0 (w wyświetlaczu siedmiosegmentowym) w komórce pamięci [05],
                  tj. o adresie 05
DB ....         ; kod cyfry 1 w komórce pamięci [06], tj. o adresie 06
DB ....         ; kod cyfry 2
...
DB ....         ; kod cyfry 9

ORG 10  ; procedura obsługi przerwania zegarowego o numerze 02
NOP
IRET            ; koniec obsługi przerwania zegarowego
```

```
ORG 30  ; procedura obsługi przerwania klawiatury numerycznej o numerze 04
CLI
IN 08           ; z klawiatury numerycznej dostajemy na AL kod ASCII cyfry, należy odjąć 30, tj.
SUB AL, 30
ADD AL, 05
PUSH AL,
POP BL
MOV AL, [BL]
ADD AL, CL
OUT 02
CMP CL, 00
JNZ Skok
INC CL
JMP Skok1
Skok:
DEC CL
Skok1:
STI
IRET           ; koniec procedury przerwania klawiatury 04

Start:         ; jest to etykieta, od której procesor zaczyna wykonywanie programu
STI
MOV CL, 00
OUT 08
Petla:
NOP
NOP
NOP
JMP Petla
END            ; koniec kodu programu
```

Jeszcze bardziej prosty kod: autor – były student Aleksander Ferens , instrukcja XOR

```
JMP Start
DB 10
DB 00
DB 30
DB ....        ; kod cyfry 0 (w wyświetlaczu siedmiosegmentowym) w komórce pamięci [05],
               tj. o adresie 05
DB ....        ; kod cyfry 1 w komórce pamięci [06], tj. o adresie 06
DB ....        ; kod cyfry 2
; ...
DB ....        ; kod cyfry 9
```

```
ORG 10      ; procedura obsługi przerwania zegarowego o numerze 02
NOP
IRET        ; koniec obsługi przerwania zegarowego

ORG 30      ; procedura obsługi przerwania klawiatury numerycznej o numerze 04
CLI
IN 08           ; z klawiatury numerycznej dostajemy na AL kod ASCII cyfry, należy odjąć 30, tj.
SUB AL, 30
ADD AL, 05
PUSH AL,
POP BL
MOV AL, [BL]
ADD AL, CL
OUT 02
XOR CL, 01
STI
IRET            ; koniec procedury przerwania klawiatury 04
Start:          ; jest to etykieta, od której procesor zaczyna wykonywanie programu
STI
MOV CL, 00
OUT 08
Petla:
NOP
NOP
NOP
JMP Petla
END             ; koniec kodu programu
```

13.1 Zadania na kolokwia

Zadanie 1K. Wersja zadania. Z klawiatury symulowanej numerycznej (numer przerwania 04 oraz IN 08) wpisywać kolejno cyfry. Sprawdzać, czy wprowadzony kod ASCII, np. x, jest kodem cyfry dziesiętnej, tj. czy heksadecymalnie 2F < x < 3A, a następnie wyświetlać je kolejno raz po prawej, a następnie po lewej stronie wyświetlacza 7-segmentowego OUT 02.

Zadanie 1Ka. Z klawiatury symulowanej numerycznej (numer przerwania 04 oraz IN 08) wpisywać kolejno cyfry heksadecymalne. Wprowadzić dodatkowe kody dla cyfr A, B, C, D, E i F. Problem może być z B oraz D; są takie same jak dla 8 i 0. Można odjąć segment z dołu, z lewej strony, o wartości dziesiętnej 32. Następnie wyświetlać je kolejno raz po prawej, później po lewej stronie wyświetlacza 7-segmentowego OUT 02.

Zadanie 2K. Jest to uproszczona wersja zadania 19. Wypisywanie z klawiatury symulowanej (numer przerwania 03, IN 07) tekstu na VDU; bez Enter (tj. od nowej linii), ale z Backspace, z kursorem migającym, z użyciem zegara (przerwanie zegarowe, numer 02).

Zadanie 2Ka. Wypisywanie z klawiatury symulowanej (numer przerwania 03, IN 07) tekstu na VDU z Enter (tj. od nowej linii) oraz z Backspace.

Zadanie 2Kb. Wypisywanie z klawiatury rzeczywistej (IN 00) tekstu na VDU bez Enter (tj. od nowej linii), ale z Backspace, z kursorem migającym, z użyciem zegara (przerwanie zegarowe, numer 02).

Zadanie 3K. Uproszczona wersja zadania 20. Wypisanie z klawiatury symulowanej (numer przerwania 03, IN 07) na VDU działania (dodawania) dwóch liczb dwucyfrowych dziesiętnych: xy + zv =

Wykonać dodawanie i sprawdzić flagę Overflow. Jeśli nie ma przepełnienia, to wynik powinien być wypisany po znaku = . Jeśli jest przepełnienie, to wpisujemy literę P po znaku =.

Zadanie 4K. Prosta wersja zadania 18. Wczytać z klawiatury symulowanej (numer przerwania 03, IN 07) na VDU, do pierwszej linii, dwa wyrazy przedzielone spacją. Następnie w drugiej linii VDU wyświetlić w odwrotnej kolejności wyrazów. Np. ”mala kotka” w pierwszej linii, zaś w drugiej powinno być ”kotka mala”.

Zadanie 4Ka. Wczytać z klawiatury rzeczywistej IN 00 na VDU, do pierwszej linii, trzy wyrazy przedzielone spacją. Następnie w drugiej linii VDU wyświetlić w odwrotnej kolejności wyrazów. Np. ”mala rozowa kotka” w pierwszej linii, zaś w drugiej powinno być ”kotka rozowa mala”.

Zadanie 5K. Z klawiatury rzeczywistej IN 00 wpisywać na VDU: dwie cyfry heksadecymalne, znak minus -, następne dwie cyfry heksadecymalne, znak równości =. Każdą z tych dwóch cyfr należy potraktować jako liczbę w notacji U2. Po znaku = wykonać odejmowanie i sprawdzić flagę Overflow. Jeśli nie ma przepełnienia, to wypisać wynik jako dwucyfrową liczbę heksadecymalną. Jeśli jest przepełnienie, to wpisać literę P.

13.2 Zadania na egzamin

Zadanie 1E. Jest to wersja zadania 14a Sekundnik. Wyświetlać kolejno ma wyświetlaczu siedmiosegmentowym liczby 00, 01, 02, ... 58, 59, a następnie od 00, 01, ... i tak w kółko w odstępach sekundowych.

Jako przerwy pomiędzy wyświetlanymi liczbami użyć przerwania zegarowego, numer przerwania 02.

Wskazówka: kolejną liczbę dwucyfrową xy (np. 45) wstawić do AL oraz BL

```
DIV AL, 0A      ; daje jako wynik cyfrę dziesiątek do wyświetlenia (po zamianie na kod tej
                  cyfry) po prawej stronie
MOD BL, 0A      ; daje jako wynik cyfrę jednostek do wyświetlenia (po zamianie na kod +1) po
                  lewej stronie
```

Zwiększać liczbę o 1 aż do 60, potem powrót do zera itd.

```
MOD AL, 3C      ; reszta z dzielenia licznika sekund AL przez 60 (dziesiętnie).
```

Dodatkowo: ustawianie „budzika” wczytanie liczby x od 0 do 60 z klawiatury numerycznej symulowanej. Uruchomienie sekundnika. Po osiągnięciu x, miganie światłami sygnalizacyjnymi: czerwone, żółte, zielone i tak w kółko.

Zadanie 2E. Jest to wersja zadania 10a. Odwracanie wyrazowe zdania wpisanego z klawiatury symulowanej, numer przerwania 03.

Wczytać z klawiatury symulowanej (wpisujemy na VDU – pierwsza linia) **dowolne** zdanie trzy-(lub więcej) wyrazowe. Następnie w drugiej linii VDU wyświetlić w odwrotnej kolejności wyrazów. Np. ”ala ma kota” w pierwszej linii, zaś w drugiej powinno być ”kota ma ala”.

Schemat kodu:

```
JMP Start
DB 10
DB 20

ORG 10          ; procedura obsługi przerwania zegarowego o numerze 02
NOP
IRET            ; koniec obsługi przerwania zegarowego

ORG 20          ; procedura obsługi przerwania klawiatury pełnej symulowanej
                 o numerze przerwania 03
CLI
IN 07           ; z klawiatury dostajemy na AL kod ASCII znaku zamiast IN 00
CMP AL, 0D
JZ Skok
MOV [CL], AL
INC CL
STI
IRET            ; koniec procedury przerwania klawiatury 03

Start:          ; jest to etykieta, od której procesor zaczyna wykonywanie programu
STI
Petla:
NOP
NOP
NOP
JMP Petla
Skok:
                ; tutaj wstawić odwracanie kolejności wyrazów w wczytanym zdaniu na VDU
END             ; koniec kodu programu
```

Zadanie 3E. Edytor tekstowy (podobnie jak prosty Notepad) na VDU; od nowej linii, Backspace, kursor migający z użyciem zegara (przerwanie zegarowe, numer 02). Znaki wpisywane z klawiatury symulowanej, numer przerwania 03.

> ***Uwaga:*** *Dla pozycji kursora na początku linii (tj. D0, E0, F0) Backspace musi wrócić do poprzedniej linii, w miejsce, gdzie poprzednio nastąpił Enter.*

Zadanie 4E. KALKULATOR: Wczytywanie na VDU z klawiatury symulowanej numerycznej (numer przerwania 4) dwóch liczb (dziesiętnych) dwucyfrowych przedzielonych znakiem dodawania lub mnożenia. Po znaku „=" ma być wyświetlony wynik działania jako liczba dziesiętna.

Wersja zadania 4E: dwie liczby heksadecymalne dwucyfrowe U2 wprowadzone za pomocą DB. Wyświetlenia na VDU: jako liczb dziesiętnych, wykonanie działania (do wyboru): dodawania (ADD) lub mnożenia (MUL) na liczbach heksadecymalnych z obsługą przepełnienia. Po znaku „=" ma być wynik działania w postaci liczby dziesiętnej albo litera „P" wskazująca na przepełnienie.

Literatura

Najbardziej poczytne (2024 rok) podręczniki o architekturze komputerów:

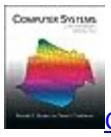

William Stallings. Organizacja i architektura systemu komputerowego. Tom 1: Projektowanie systemu a jego wydajność. Wydawnictwo Naukowe PWN 2022, EAN: 9788301222000.

Smruti R. Sarangi. Basic Computer Architecture Version 3.01 Version 3.02, November 25, 2024, wolny dostęp w pdf https://www.cse.iitd.ac.in/~srsarangi/archbook/archbook.pdf

Shuangbao Paul Wang. Computer Architecture and Organization: Fundamentals and Architecture Security. Published by John Wiley & Sons Singapore Pte. Ltd., Higher Education Press 2013. ISBN 978-1-118-16881-3.
https://www.iqytechnicalcollege.com/Computer%20Architecture%20and%20Security.pdf


Computer Architecture: A Quantitative Approach (Paperback) by John L. Hennessy. 704 pages, September 27, 2006 by Morgan Kaufmann. ISBN 9780123704900.


Computer Organization & Design: The Hardware/Software Interface by David A. Patterson. Morgan Kaufmann Publishers Inc; Edycja 2. (1 sierpnia 1997). 1000 ss. ISBN-10: 9781558604285, ISBN-13: 978-1558604285.


Computer Systems: A Programmer's Perspective (Hardcover) by Randal Bryant and David O'Hallaron. Pearson; 3rd edition (March 2, 2015), 1128 pages, ISBN-10: 013409266X, ISBN-13: 978-0134092669.


Digital Design and Computer Architecture by David Money Harris. Morgan Kaufmann; 2nd edition (August 7, 2012), 720 pages, ISBN-10: 9789382291527, ISBN-13: 978-0123944245.


Code: The Hidden Language of Computer Hardware and Software by Charles Petzold. Microsoft Press US; Edycja 1. (11 października 2000), 400 ss., ISBN-10: 9780735611313, ISBN-13: 978-0735611313.


The Elements Of Computing Systems: Building A Modern Computer From First Principles (by Noam Nisan. The MIT Press (25 stycznia 2008), 342 ss., ISBN-10: 0262640686, ISBN-13: 978-0262640688.

Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture by Jon Stokes. No Starch Press; Reprint edition (December 1, 2006), 320 pages, ISBN-10: 1593276680, ISBN-13: 978-1593276683.


Structured Computer Organization. by Andrew S. Tanenbaum. Pearson; 6th edition (July 25, 2012), 808 pages, ISBN-10: 0132916525, ISBN-13: 978-0132916523.

Computer Organization and Architecture by William Stallings. Cengage Learning; Revised edition (March 1, 2013), 800 pages, ISBN-10: 9781111987084, ISBN-13: 978-1111987084.


Computer System Architecture by M. Morris Mano, Pearson; 3rd edition (October 19, 1992), 544 pages, ISBN-10: 0131755633, ISBN-13: 978-0131755635.


Introduction to Computing Systems: From Bits & Gates to C & Beyond by Yale N. Patt and Sanjay J. Patel, McGraw Hill; 2nd edition (August 5, 2003), 656 pages, ISBN-10: 0072467509, ISBN-13: 978-0072467505.


Write Great Code: Volume 1: Understanding the Machine (Paperback) by Randall Hyde, 440 pages, October 25, 2004 by No Starch Press. ISBN 9781593270032 (ISBN10: 1593270038).


Modern Computer Architecture and Organization: Learn processor architecture including RISC-V, and design of PCs, cloud servers, and smartphones by Jim Ledin, Packt Publishing; Edition Illustrated (30 kwietnia 2020), 560 ss., ISBN-10: 1838984399, ISBN-13: 978-1838984397.

But How Do It Know? The Basic Principles of Computers for Everyone by John Clark Scott; First Edition (July 4, 2009), 222 pages, ISBN-10: 0615303765, ISBN-13: 978-0615303765.


Computer Architecture and Organization by John P. Hayes, William C Brown Pub; Subsequent edition (December 1, 1997), 624 pages, ISBN-10: 0070273553, ISBN-13: 978-0070273559.

Computer Architecture: Concepts and Evolution (Hardcover) by Gerritt A. Blaauw and Frederick P. Brooks Jr., Addison-Wesley; First Edition (January 1, 1997), 1213 pages, ISBN-10: 0201105578, ISBN-13: 978-0201105575.


Computer Organization and Design MIPS Edition: The Hardware/Software Interface by David A. Patterson and John L. Hennessy, Morgan Kaufmann; 5th edition (October 10, 2013), 800 pages, ISBN-10: 0124077269, ISBN-13: 978-0124077263.

Autor monografii:

dr hab. Stanisław Ambroszkiewicz, prof. uczelni

Uniwersytet w Siedlcach
Wydział Nauk Ścisłych i Przyrodniczych, Instytut Informatyki

Recenzja naukowa:

prof. dr hab. inż. Mieczysław A. Kłopotek

https://doi.org/10.34739/68355.58.1

e-ISBN 978-83-68355-58-1

Wydawnictwo Naukowe UwS

www.wydawnictwo-naukowe.uws.edu.pl
08-110 Siedlce | ul. Żytnia 17/19 | tel. 25 643 15 20