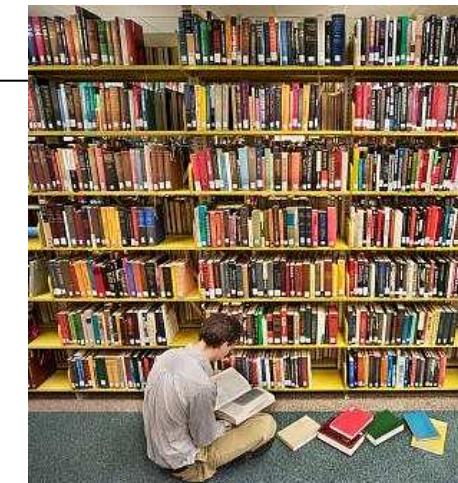


Organizacja i Architektura Komputerów

Wprowadzenie

prof. dr hab. inż. Ryszard Pełka

e-mail: rpelka@wsisiz.edu.pl



Literatura

- [1] W. Stallings, *Organizacja i architektura komputerów*, WNT, 2004 (tłum. wydania 4)
- [2] B.S. Chalk, *Organizacja i architektura komputerów*, WNT, 1998
- [3] P. Metzger, *Anatomia PC*, Helion, 2007
- [4] A. Kowalczyk, *Asembler*, Croma, 1999
- [5] A. Skorupski, *Podstawy budowy i działania komputerów*, WKŁ, 2005

Literatura uzupełniająca

- [1] D.A. Patterson, J.L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 4th edition, 2009
- [2] J.P. Hayes, *Computer Architecture Organization*, McGraw-Hill, 5th edition, 2008
- [3] W. Stallings, *Computer Organization and Architecture*, Prentice Hall, 8th edition, 2010



Internet

- [1] Computer Architecture WWW Home Page
<http://www.cs.wisc.edu/~arch/www>
strona Uniwersytetu Wisconsin

- [2] AMD web site
<http://www.amd.com>

- [3] Intel Developer's Site
<http://developer.intel.com>

Plan przedmiotu

- Wprowadzenie

- Podstawowe pojęcia, terminologia
- Co to jest **architektura** komputerów?
- Co to jest **organizacja** komputerów?
- Bloki funkcjonalne komputerów
- Podstawowe zasady współpracy bloków funkcjonalnych

Plan przedmiotu (cd.)

- Historia i ewolucja komputerów
 - wpływ technologii na rozwój komputerów
 - **wydajność komputerów**
 - programy testujące wydajność (benchmarki)
- **Arytmetyka komputerów**
 - kody liczbowe, konwersja kodów
 - operacje arytmetyczne
 - arytmetyka zmiennopozycyjna
- Układy i bloki cyfrowe
 - bramki, przerzutniki, rejstry, liczniki, multipleksery
 - półsumator i sumator
 - jednostka arytmetyczno-logiczna ALU
 - układy mnożenia i dzielenia

Plan przedmiotu (cd.)



- Procesor (CPU)
 - standardowa architektura procesora
 - jednostka wykonawcza i sterująca
 - cykl wykonania rozkazu
 - stos
 - przerwania
- Architektury współczesnych procesorów
 - ogólna architektura procesorów IA-32 i IA-64
 - rejesty
 - segmentacja pamięci

Plan przedmiotu (cd.)



- **Język asemblera**
 - język asemblera i wysokiego poziomu
 - tryby adresowania argumentów
 - formaty instrukcji
 - lista instrukcji (ISA), grupy instrukcji
 - instrukcje przesłań
 - operacje arytmetyczne
 - operacje logiczne, operacje na bitach
 - skoki, pętle, instrukcje sterujące
 - przykłady programów
 - makroasemblerы
 - debugery

Plan przedmiotu (cd.)

- Pamięć

- technologie wytwarzania pamięci
- pamięci RAM, SRAM, DRAM
- pamięci ROM, PROM, EEPROM i Flash
- współpraca procesora z pamięciami
- bezpośredni dostęp do pamięci (DMA)
- segmentacja i stronicowanie
- pamięć wirtualna

- Pamięć cache

- hierarchia systemu pamięci
- metody odwzorowywania pamięci cache
- spójność pamięci cache



Plan przedmiotu (cd.)

- **Potok instrukcji (*pipeline*)**
 - działanie przetwarzania potokowego
 - przewidywanie skoków
 - optymalizacja potoku instrukcji
 - przykłady implementacji
- **Architektura superskalarna**
 - przykłady procesorów

Plan przedmiotu (cd.)

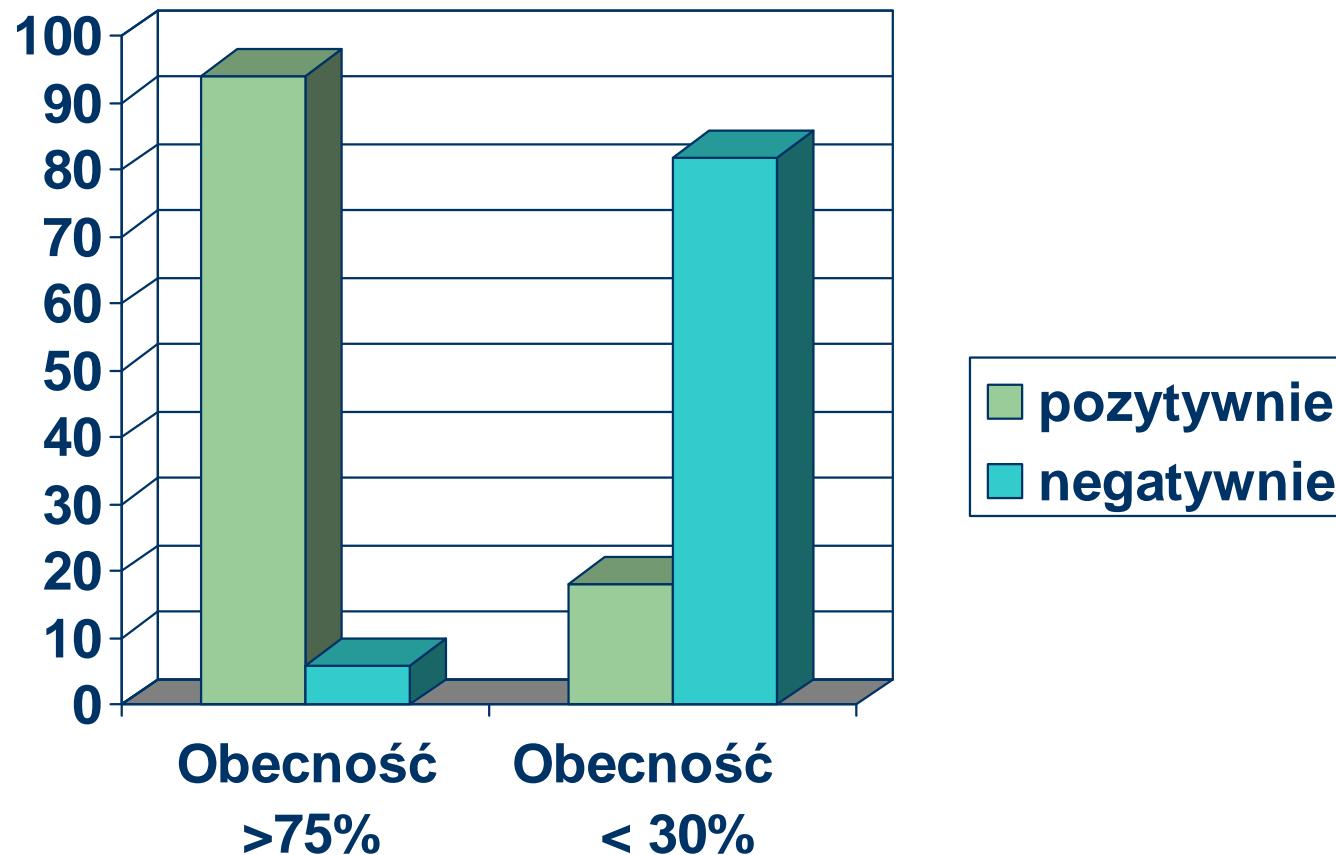
- **Architektura RISC**
 - Podstawowe własności
 - Banki i okna rejestrów
 - Przetwarzanie potokowe w architekturze RISC
 - Przykłady procesorów
 - Porównanie architektur RISC i CISC
- **Architektura VLIW**
- **Zagadnienia fakultatywne:**
 - Przetwarzanie równoległe, systemy wieloprocesorowe
 - **Architektury wielordzeniowe**
- Podsumowanie



Egzamin

- pisemny test
 - około 10 zadań sprawdzających znajomość głównych zagadnień poruszanych na wykładach
 - zadania koncepcyjne i rachunkowe
 - tematyka w całości związana z wykładem
 - szczegółowe zagadnienia egzaminacyjne i przykłady zadań – **w trakcie wykładu**

Wyniki i obecność na wykładach



Dane z roku akademickiego 2009/2010 – dla wybranych grup studentów poddanych ankietyzacji

Architektura i organizacja (1)

- **Architektura** to atrybuty komputera „widoczne” dla programisty:
 - lista instrukcji, liczba bitów stosowana do reprezentacji danych, mechanizm I/O, techniki adresowania
 - np. czy dostępna jest operacja mnożenia?
- **Organizacja** oznacza sposób implementacji architektury
 - sygnały sterujące, interfejsy, technologia wykonania pamięci itp.
 - np. czy mnożenie jest realizowane sprzętowo, czy przez sekwencję dodawań?

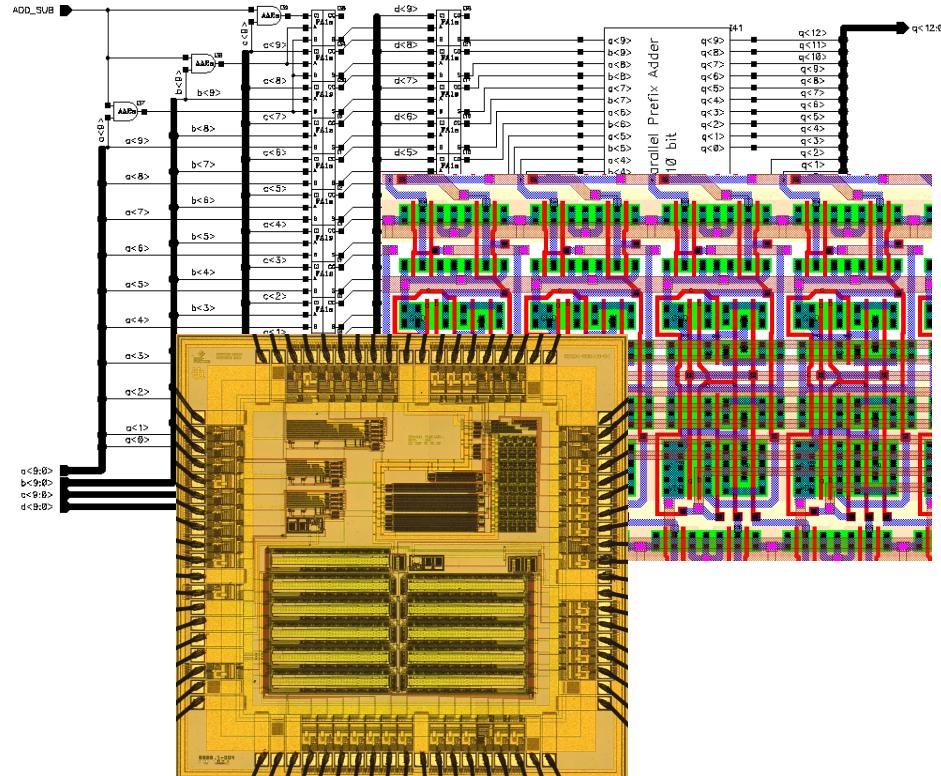
Architektura i organizacja cd.

... the attributes of a [computing] system as seen by the programmer, *i.e.* the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.

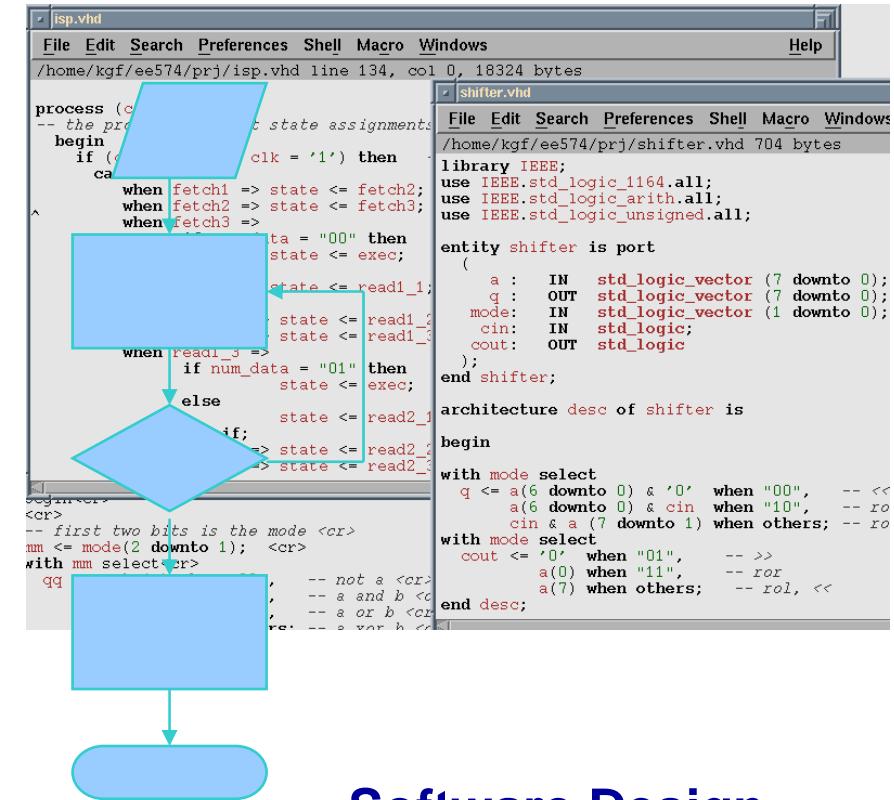
– *Amdahl, Blaaw, and Brooks, 1964*

(...atrybuty systemu komputerowego widziane przez programistę, tzn. koncepcyjna struktura i zachowanie funkcjonalne, w odróżnieniu od organizacji przepływu danych, sterowania układami logicznymi i fizycznej implementacji)

Architektura i organizacja cd.



Hardware Design



Software Design

Architektura i organizacja cd.

- wszystkie procesory Intel x86 mają podobną podstawową architekturę
- to zapewnia zgodność programową
 - przynajmniej w dół (*backward compatibility*)
- organizacja poszczególnych wersji procesorów Intel x86 jest różna

Architektura i organizacja cd.

Architektura komputera =
architektura listy instrukcji + ...

Ewolucja pojęcia „architektura”

- 1950s to 1960s: Computer Architecture Course:
[Computer Arithmetic](#)
- 1970s to mid 1980s: Computer Architecture Course:
[Instruction Set Design, especially ISA appropriate for compilers](#)
- 1990s: Computer Architecture Course:
[Design of CPU, memory system, I/O system, Multiprocessors, Networks](#)
- 2010s: Computer Architecture Course:
[Self adapting systems? Self organizing structures?
DNA Systems/Quantum Computing?](#)

John Kubiatowicz, Berkeley University

Przykłady ISA (Instruction Set Architecture)

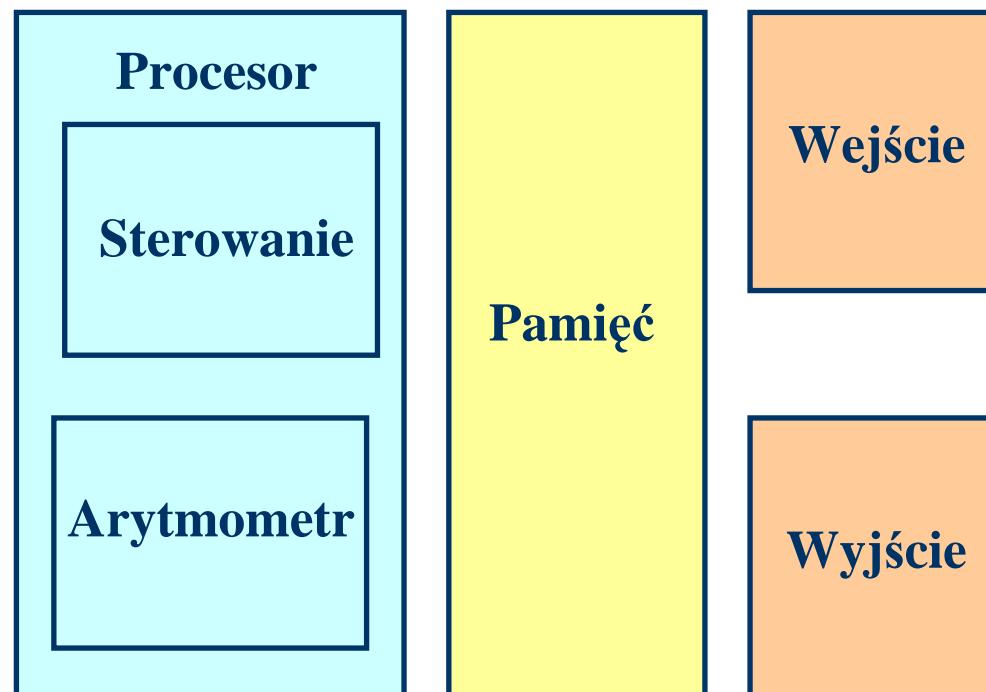
- **Digital Alpha** v1, v3 ... 1992
- **HP PA-RISC** v1.1, v2.0 1986
- **Sun Sparc** v8, v9 1987
- **MIPS / ARM** MIPS I, II, III, IV, V, 10k, 20k, ARM7, ARM9, ARM11, Cortex 1986
- **Intel / AMD** 8086, 80286, 80386, 80486, Pentium, MMX, Pentium II, III, IV, ... 1978

Funkcje i główne bloki komputera

- Funkcje
 - przechowywanie danych
 - przesyłanie danych
 - przetwarzanie danych
 - funkcje sterujące
- Bloki funkcjonalne
 - pamięć
 - podsystem I/O
 - procesor (arytmometr)
 - procesor (układ sterujący)

Bloki funkcjonalne komputera

Od 1946 r. komputery niezmiennie składają się z 5 tych samych bloków:

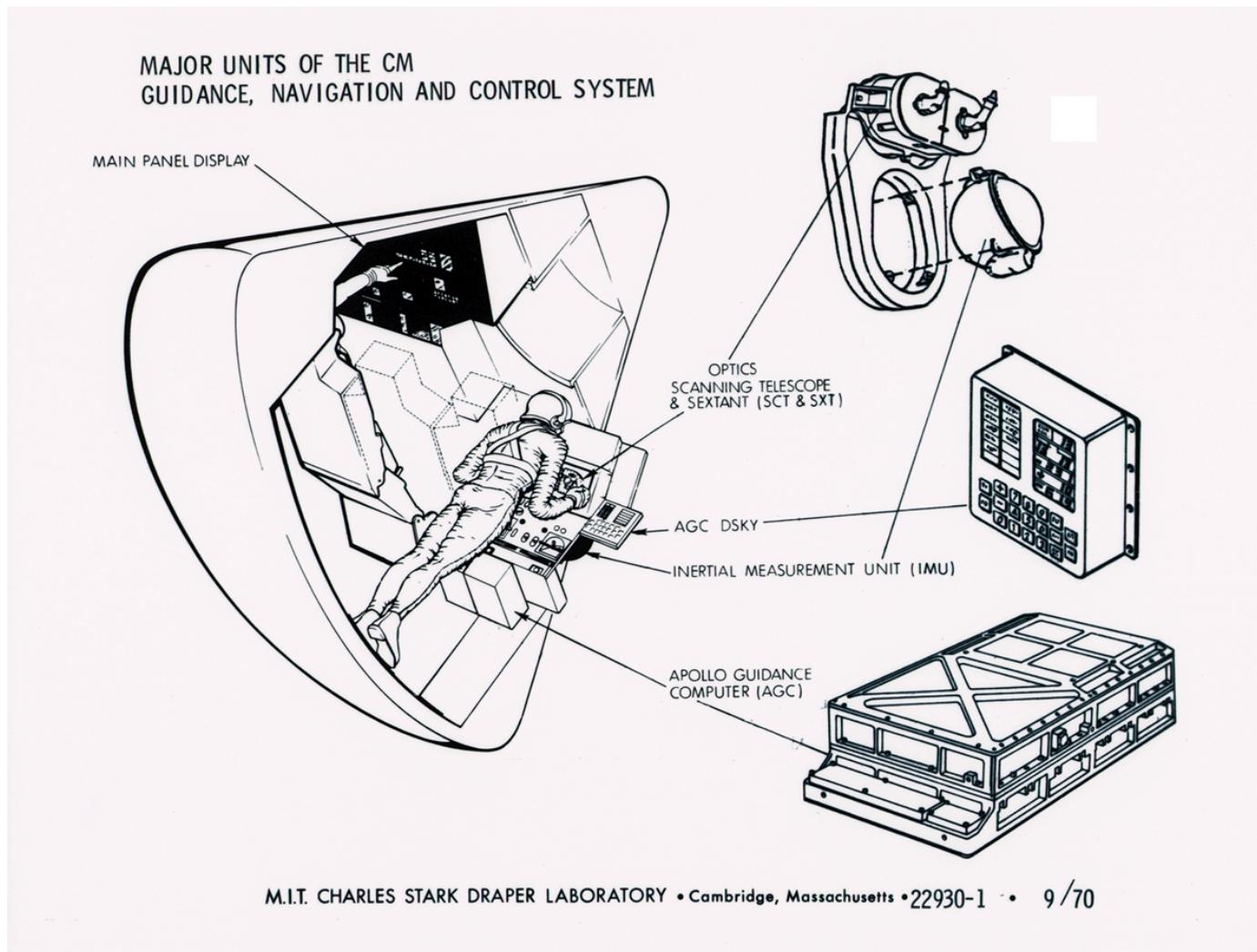


... ale różnią się wydajnością

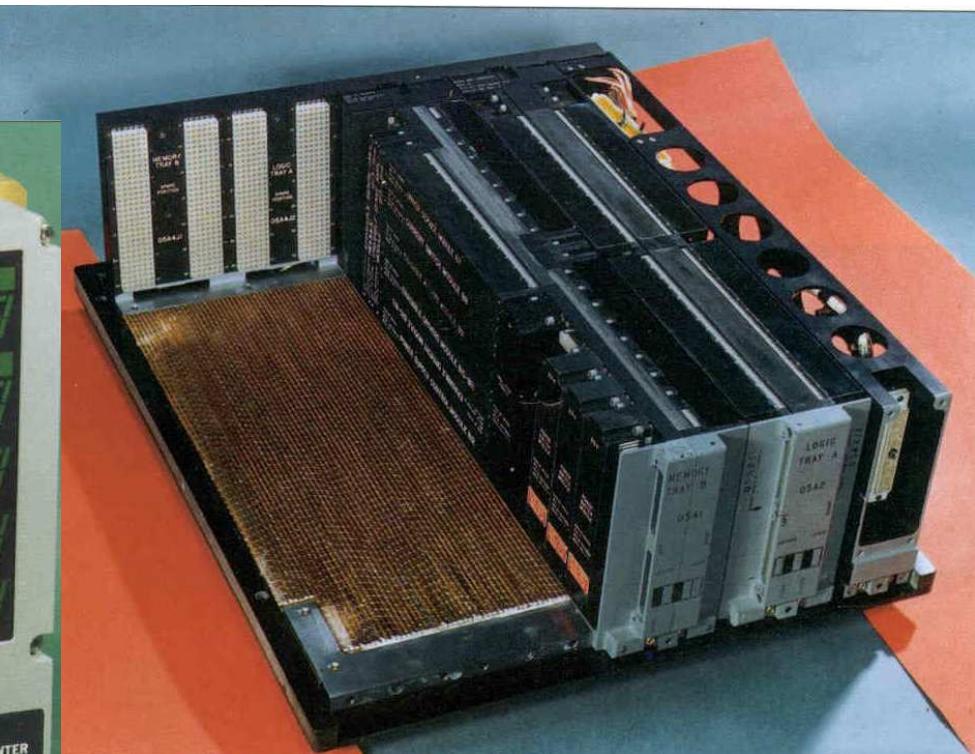
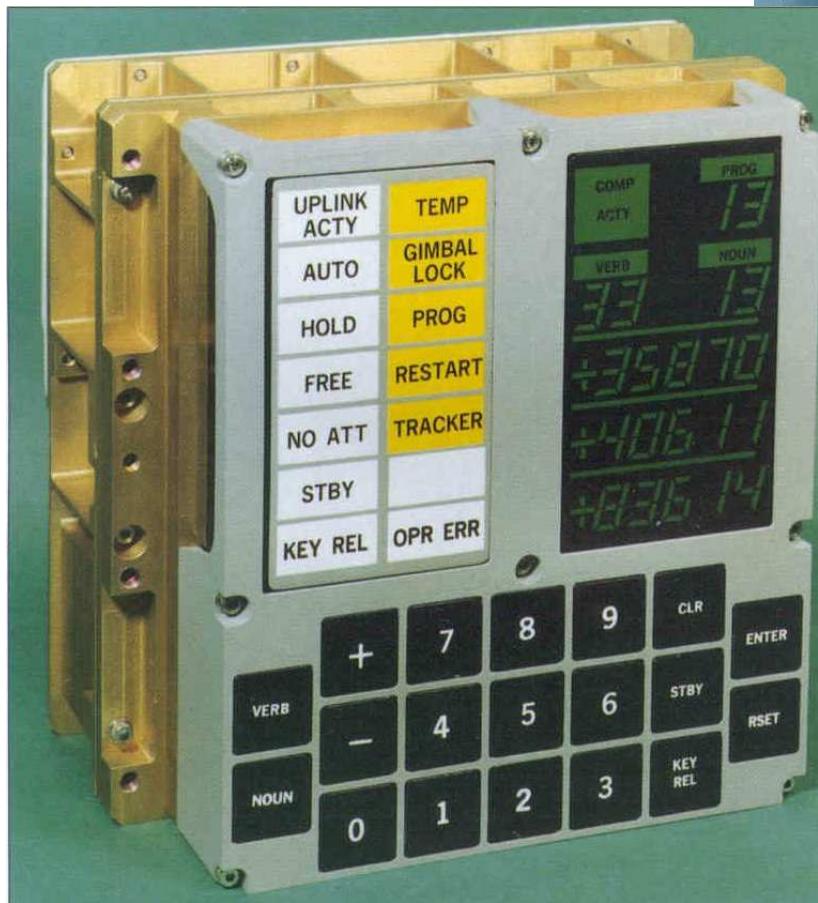


<http://hrst.mit.edu/hrs/apollo/public/index.htm>

Komputer statku Apollo



Komputer statku Apollo cd.

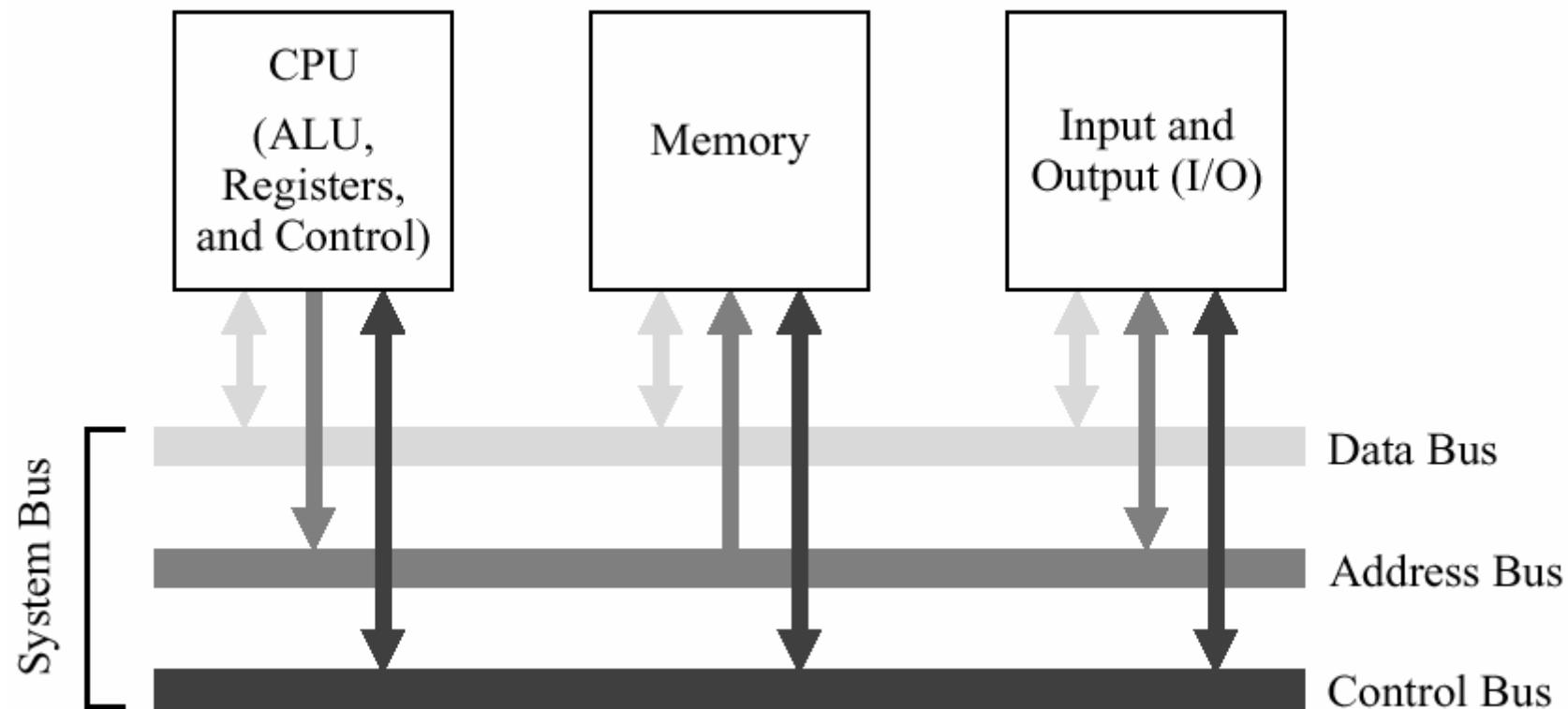


Komputer statku Apollo cd., źródło: M.I.T.

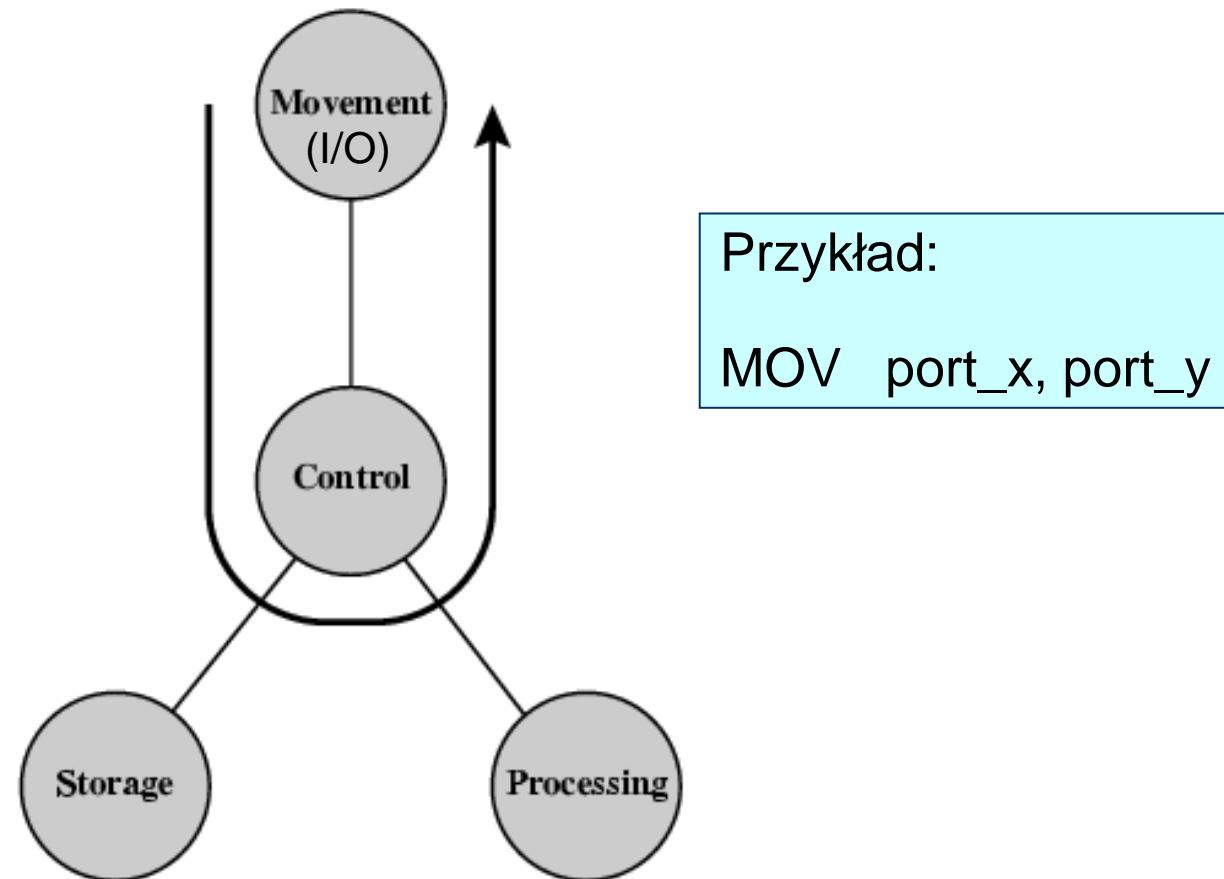
Apollo Guidance Computer (AGC)

- Word length: 16 bits (15 bits data + parity bit)
- First computer using integrated circuits (ICs)
- Magnetic core memory
 - Fixed memory (ROM): 36,864 words
 - Erasable memory (RAM): 2,048 words
- Number of instructions: 34
- Cycle time: 11.7 µsec
 - Clock frequency: 85 kHz (!)
- Number of logic gates: 5,600 (2,800 packages)
- Weight: 30 kg
- Power consumption: 70 W

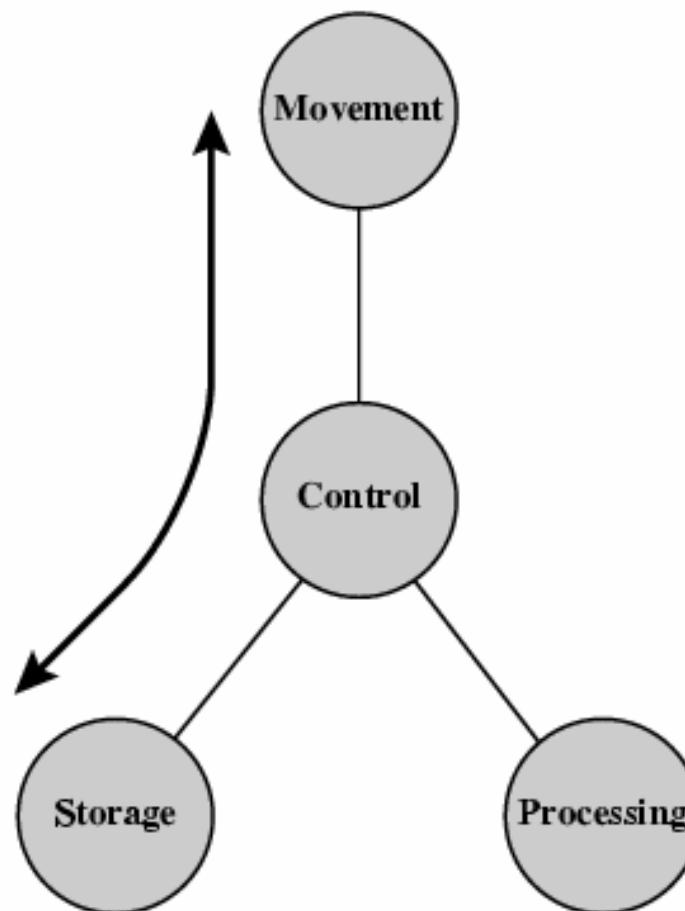
Struktura komputera



Operacja 1 – przesyłanie danych



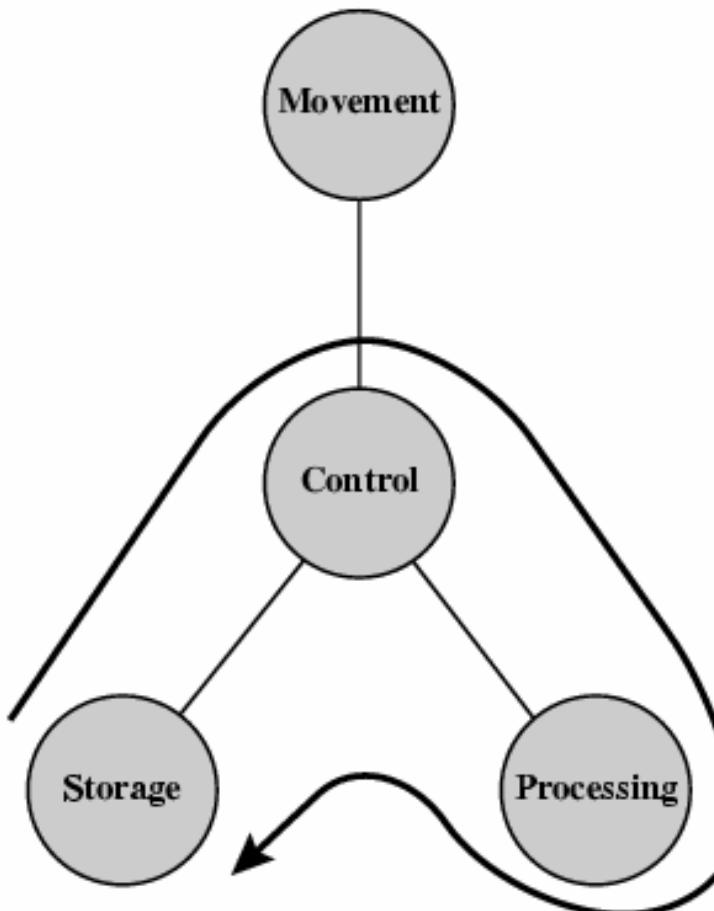
Operacja 2 – pamiętanie danych



Przykład:

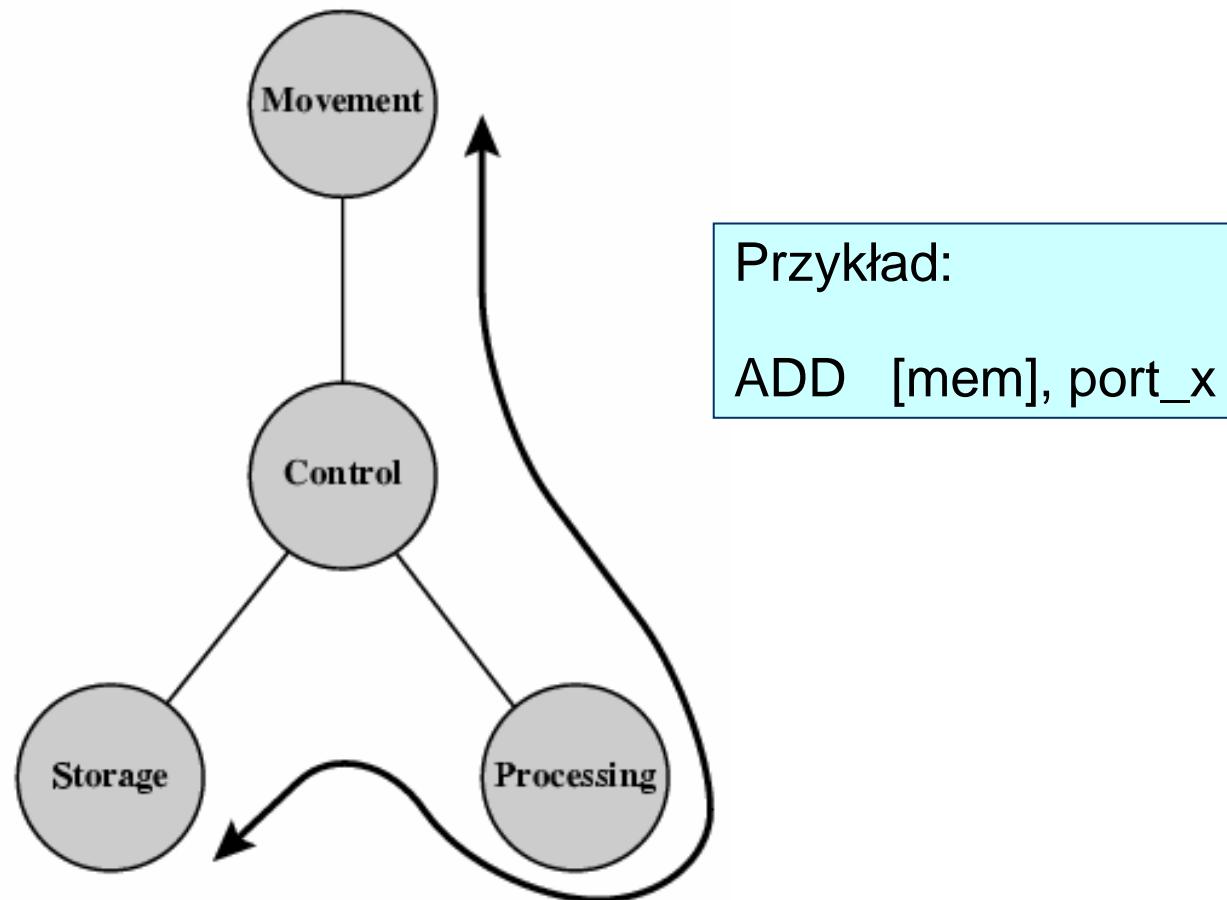
```
MOV port_x, [mem]  
MOV [mem], port_y
```

Przetwarzanie z/do pamięci

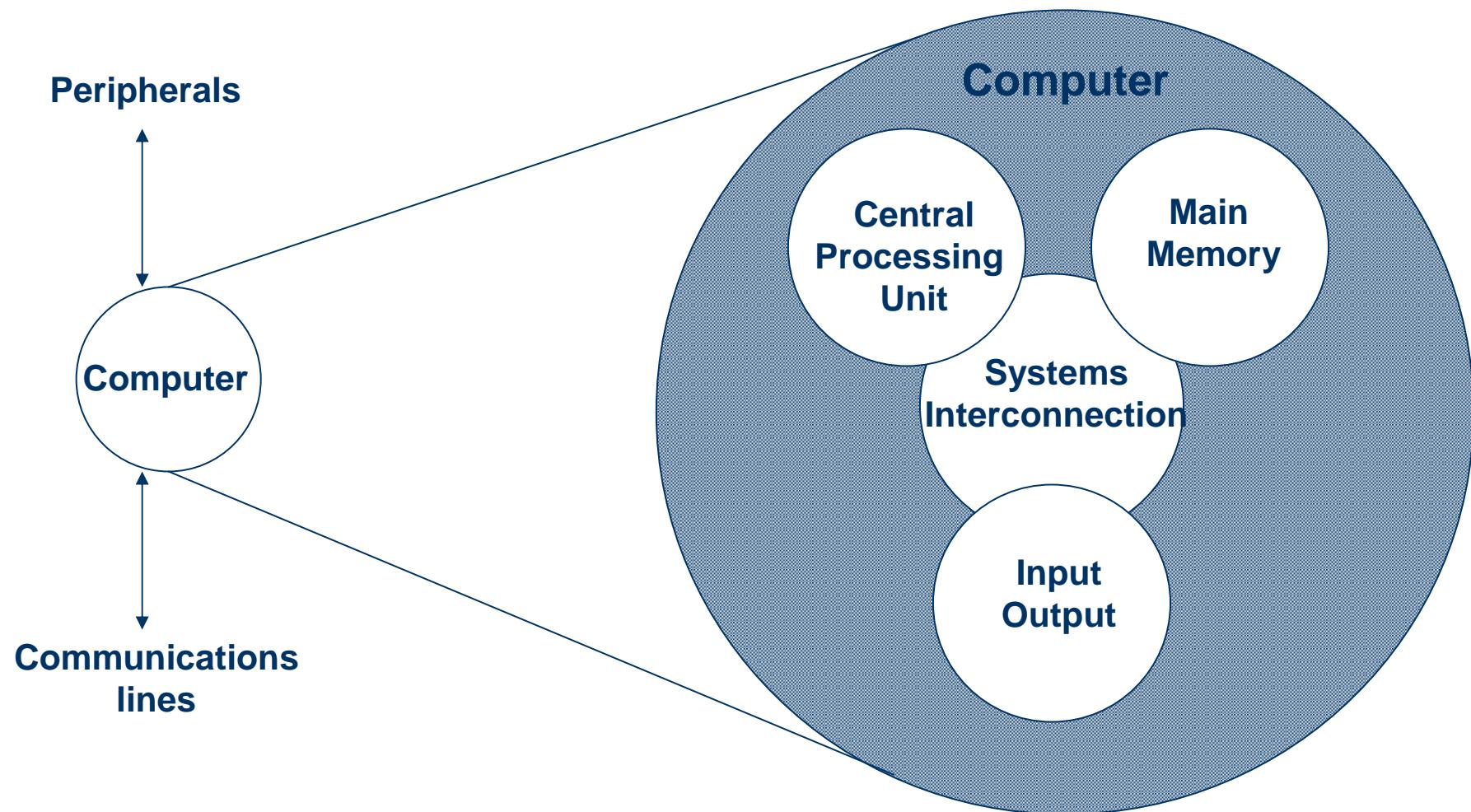


Przykład:
ADD [mem], data

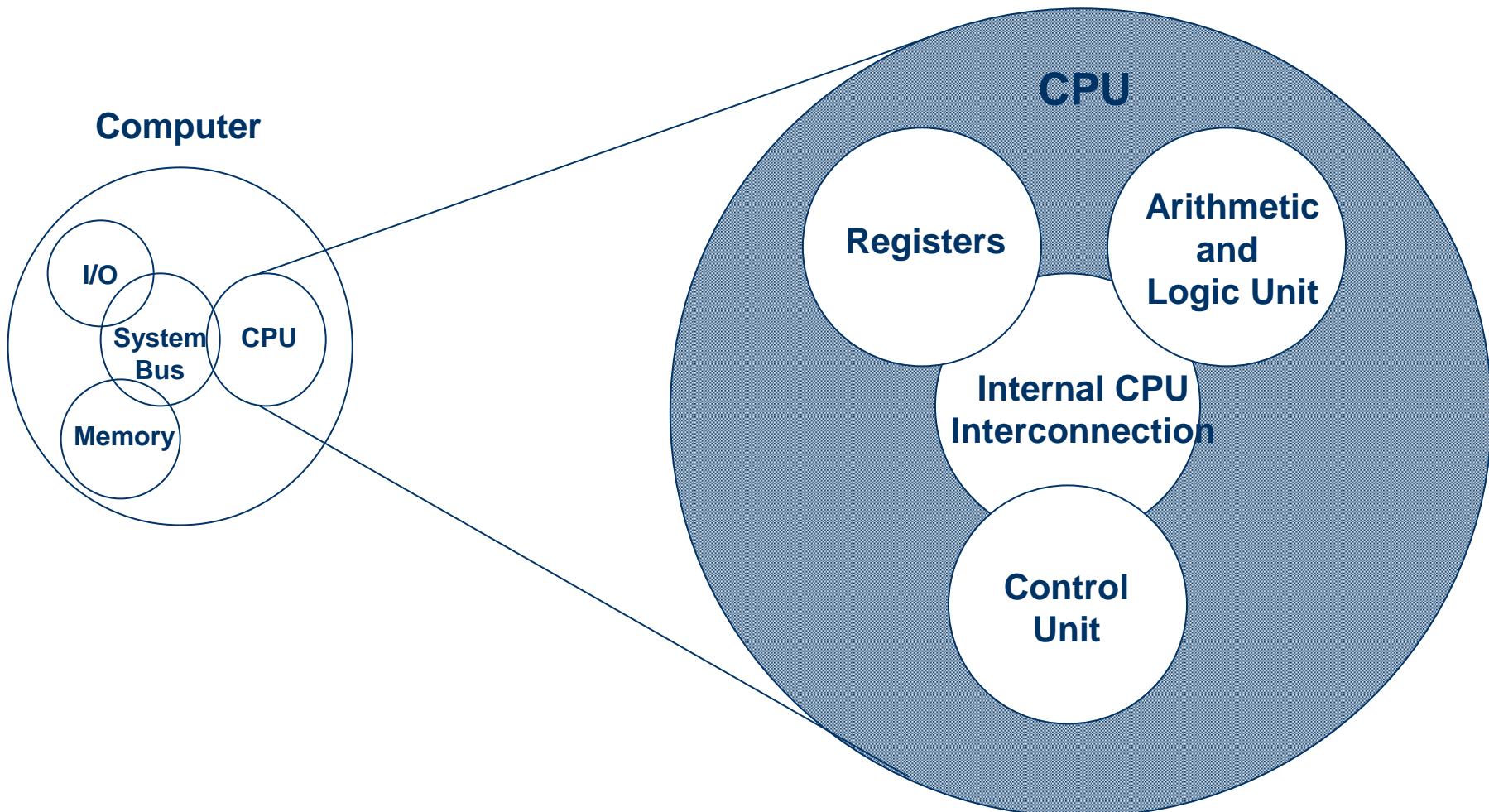
Przetwarzanie z pamięci do I/O



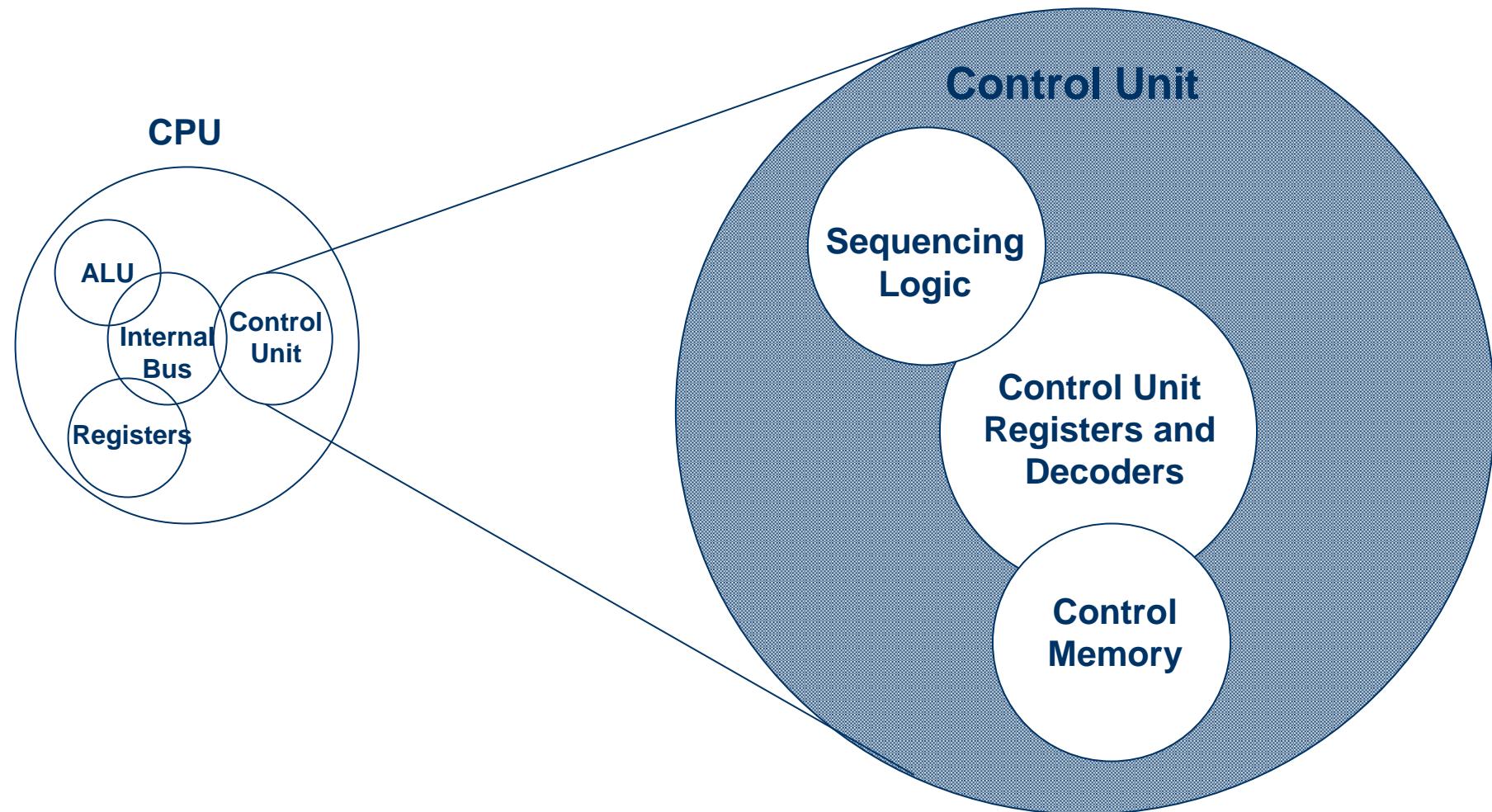
Struktura komputera – wysoki poziom



Struktura CPU

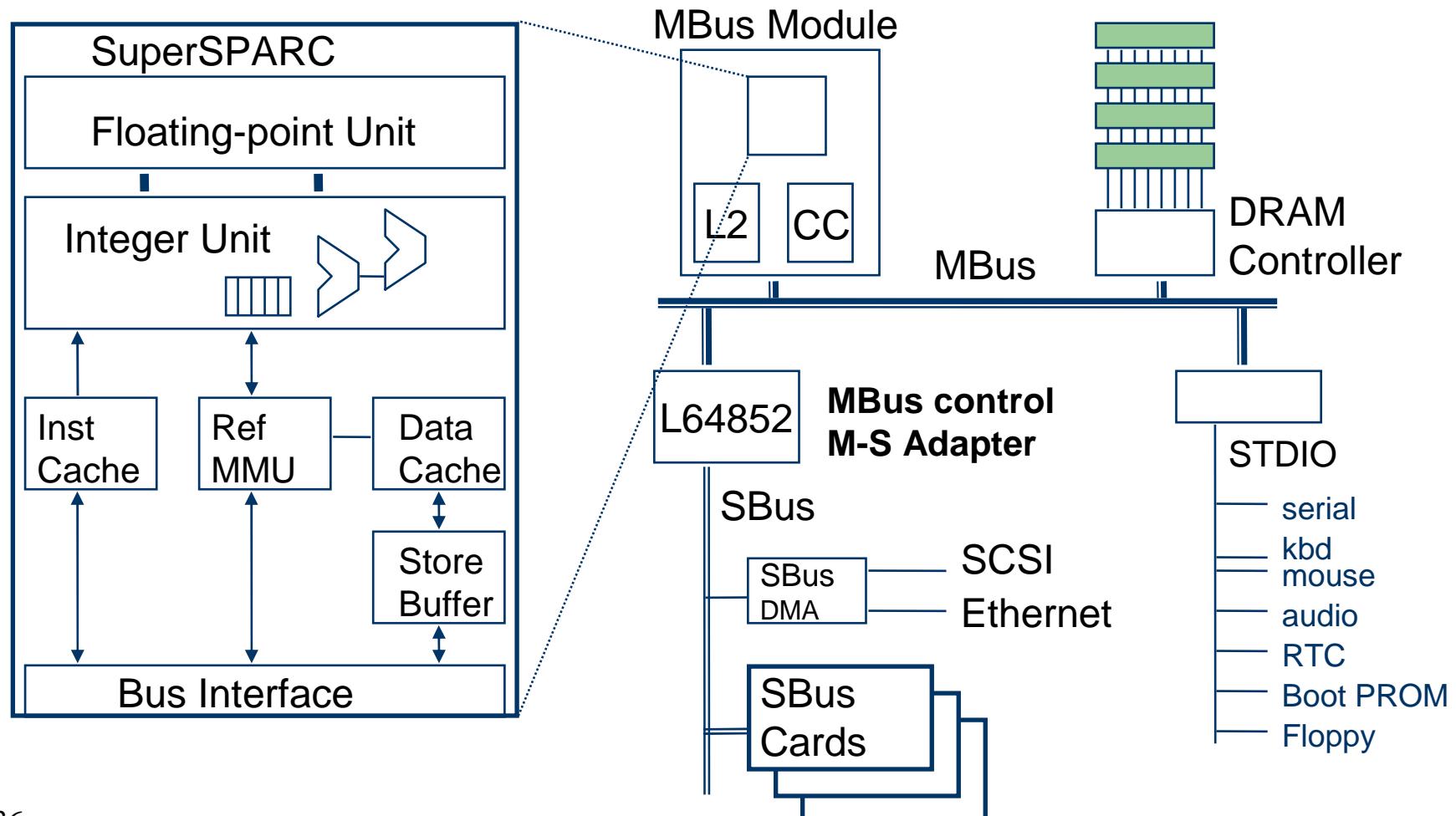


Struktura układu sterowania



Przykładowa organizacja

TI SuperSPARC™ TMS390Z50 in Sun SPARCstation20



Podstawowe pojęcia

- **Pamięć**

- w ogólnym znaczeniu – blok funkcjonalny do przechowywania informacji na określonym nośniku (zwykle magnetycznym lub półprzewodnikowym)
- pamięć RAM – pamięć półprzewodnikowa zbudowana z przerzutników (SRAM) lub tranzystorów MOS (DRAM)

- **Rejestr**

- pamięć o niewielkiej pojemności (najczęściej 8, 16, 32 lub 64 bitów) wbudowana do procesora lub innego układu cyfrowego VLSI, wykonana zwykle jako SRAM (zbiór przerzutników)

- **ALU (*arithmetic-logic unit*)**

- arytmometr, główny element bloku wykonawczego procesora zdolny do realizacji podstawowych operacji arytmetycznych i logicznych

- **FP (*floating-point*)**

- zmiennoprzecinkowy zapis liczb (znak, mantysa, wykładnik)

Podstawowe pojęcia cd.

- **Pamięć cache**
 - nazywana podręczną lub kieszeniową – dodatkowa pamięć buforowa umieszczona między główną pamięcią komputera (operacyjną) a procesorem; jej zadaniem jest przyspieszenie komunikacji procesora z pamięcią główną
- **Potok instrukcji (pipeline)**
 - ciąg instrukcji (zwykle kilku lub kilkunastu) pobranych przez procesor i wykonywanych etapami, tak jak na taśmie produkcyjnej
- **Procesor superskalarny**
 - procesor z wieloma potokami instrukcji i wieloma układami wykonawczymi (arytmometrami)
- **Zegar**
 - generator impulsów taktujących bloki funkcjonalne systemu cyfrowego

Podstawowe pojęcia cd.

- **Język asemblera**
 - język programowania operujący elementarnymi instrukcjami zrozumiałymi dla procesora
- **Język wysokiego poziomu** (C, C++, C#, Python, Java)
 - język programowania operujący złożonymi instrukcjami, niezrozumiałymi dla procesora; program przed wykonaniem musi być przetłumaczony na język asemblera
- **Szyna, magistrala**
 - zbiór linii przesyłających sygnały między blokami cyfrowymi systemu (adresy, dane, sygnały sterujące)
- **I/O – input/output**
 - układy wejścia/wyjścia do komunikacji z otoczeniem

Podstawowe pojęcia cd.

- **Architektura CISC** (np. Pentium)
 - *Complex Instruction Set Computer*
 - procesor wykorzystujący bogaty zestaw (listę) instrukcji i liczne sposoby adresowania argumentów; charakterystyczną cechą jest zróżnicowana długość instrukcji (w bajtach) i czas ich wykonania
- **Architektura RISC** (np. PowerPC, MIPS, ARM)
 - *Reduced Instruction Set Computer*
 - procesor wykorzystujący niewielki zbiór elementarnych rozkazów o zunifikowanej długości i takim samym czasie wykonania

Podsumowanie

- Architektura komputera oznacza jego właściwości „widziane” przez programistę (struktura rejestrów, lista instrukcji itp..)
- Organizacja komputera oznacza konkretną implementację (realizację) architektury przy użyciu określonych układów i bloków cyfrowych
- Komputer składa się z 5 podstawowych elementów: układu wykonawczego, układu sterowania, pamięci, układów wejścia i wyjścia
- Układ wykonawczy i sterujący tworzą jednostkę centralną (CPU)

Organizacja i Architektura Komputerów

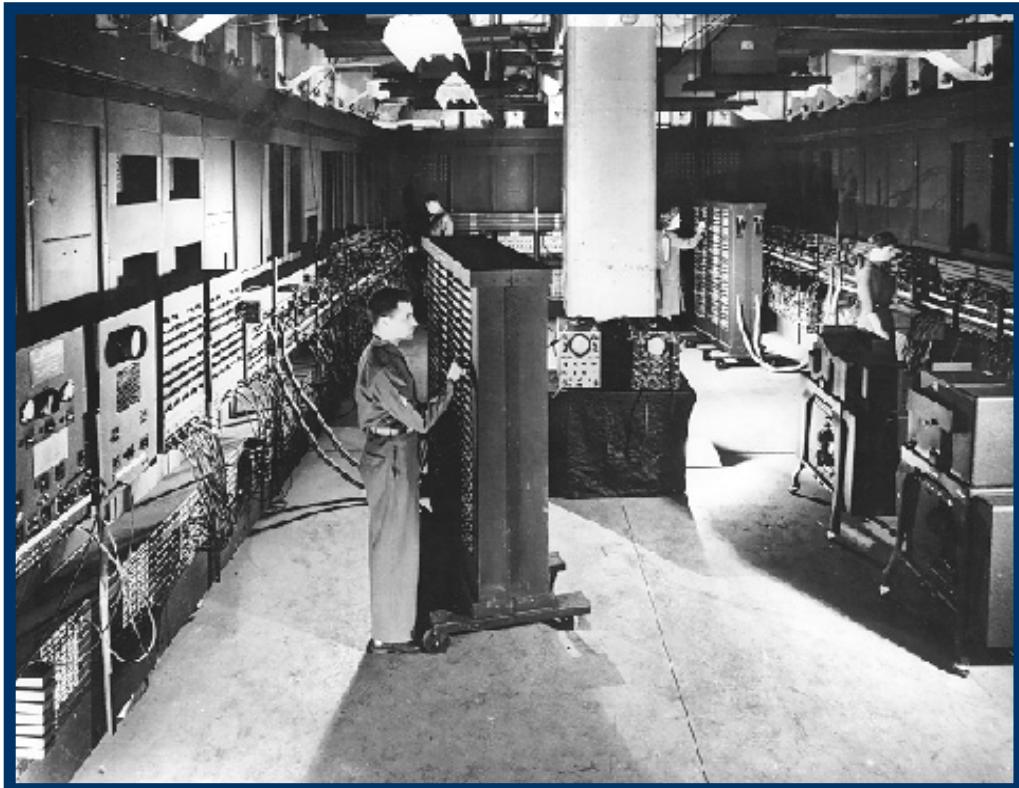
Ewolucja komputerów i ich architektury

ENIAC – pierwszy komputer

- Electronic Numerical Integrator And Computer
- konstruktorzy: Eckert and Mauchly
- University of Pennsylvania
- obliczenia trajektorii pocisków (balistyka)
- początek projektu: 1943
- zakończenie projektu: 1946
 - za późno by wykorzystać ENIAC w II wojnie światowej
- używany do 1955

ENIAC

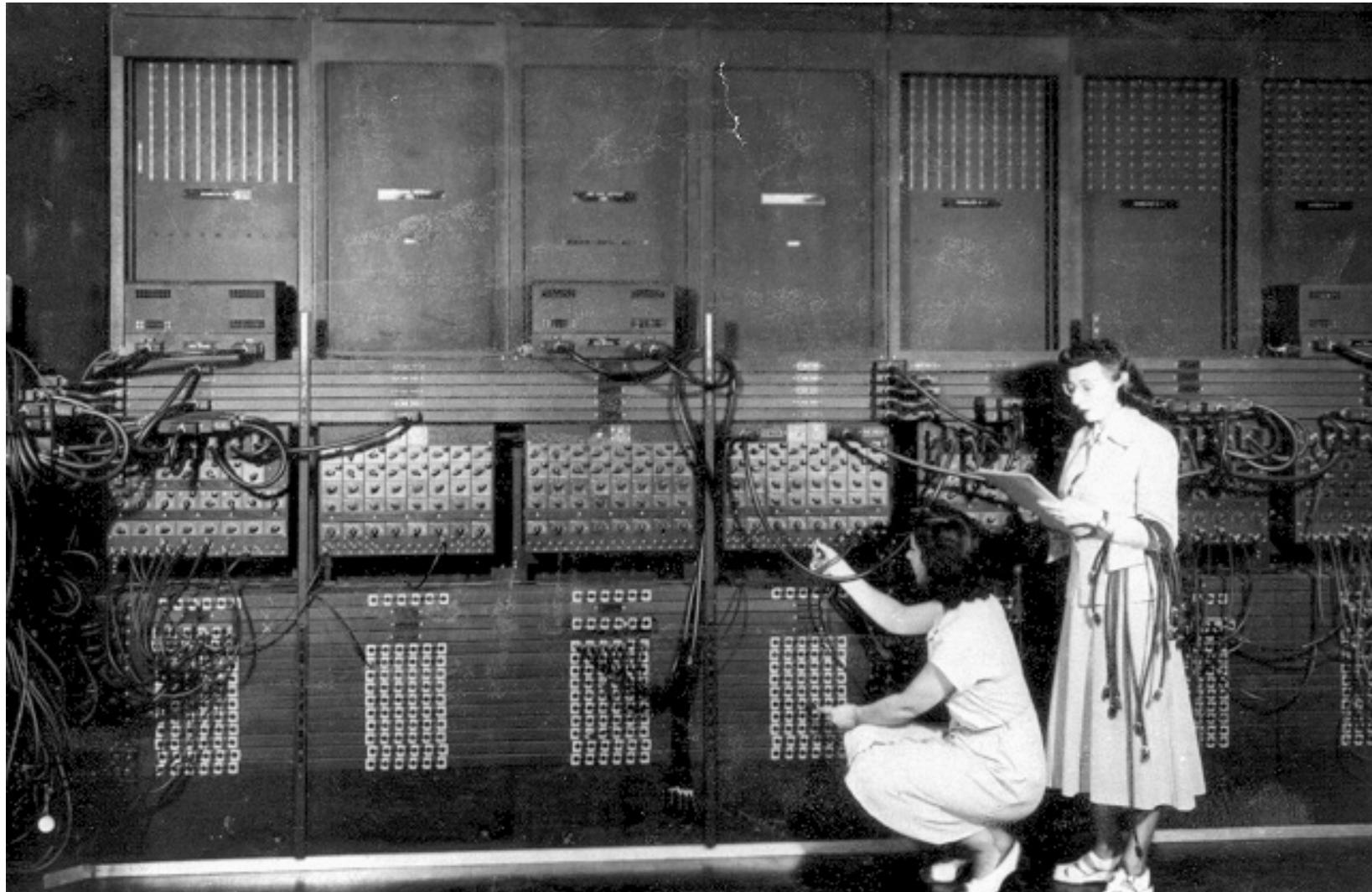
Eckert and Mauchly



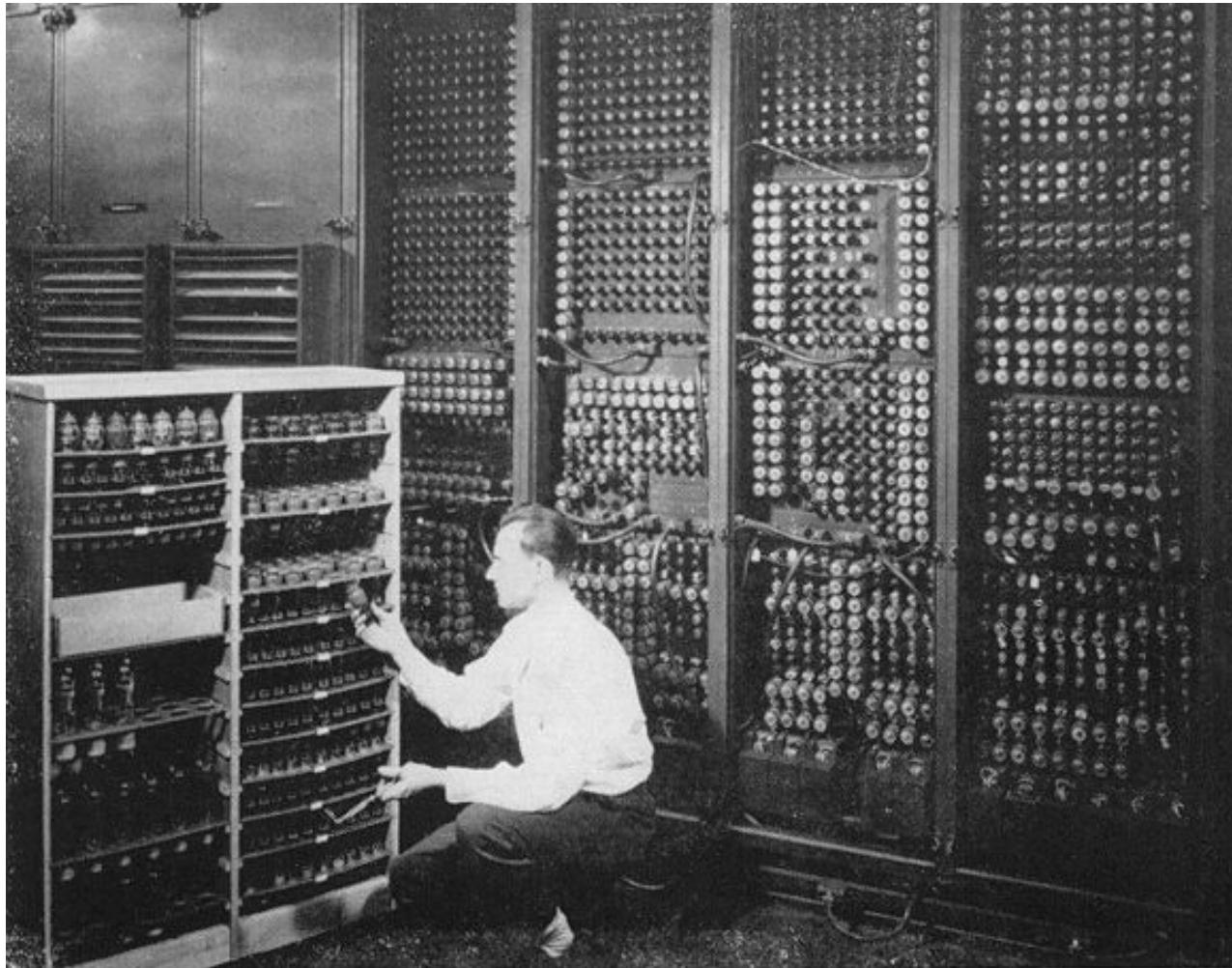
- 1st working electronic computer (1946)
- 18,000 Vacuum tubes
- 1,800 instructions/sec
- 3,000 ft³

ENIAC

ENIAC – programowanie



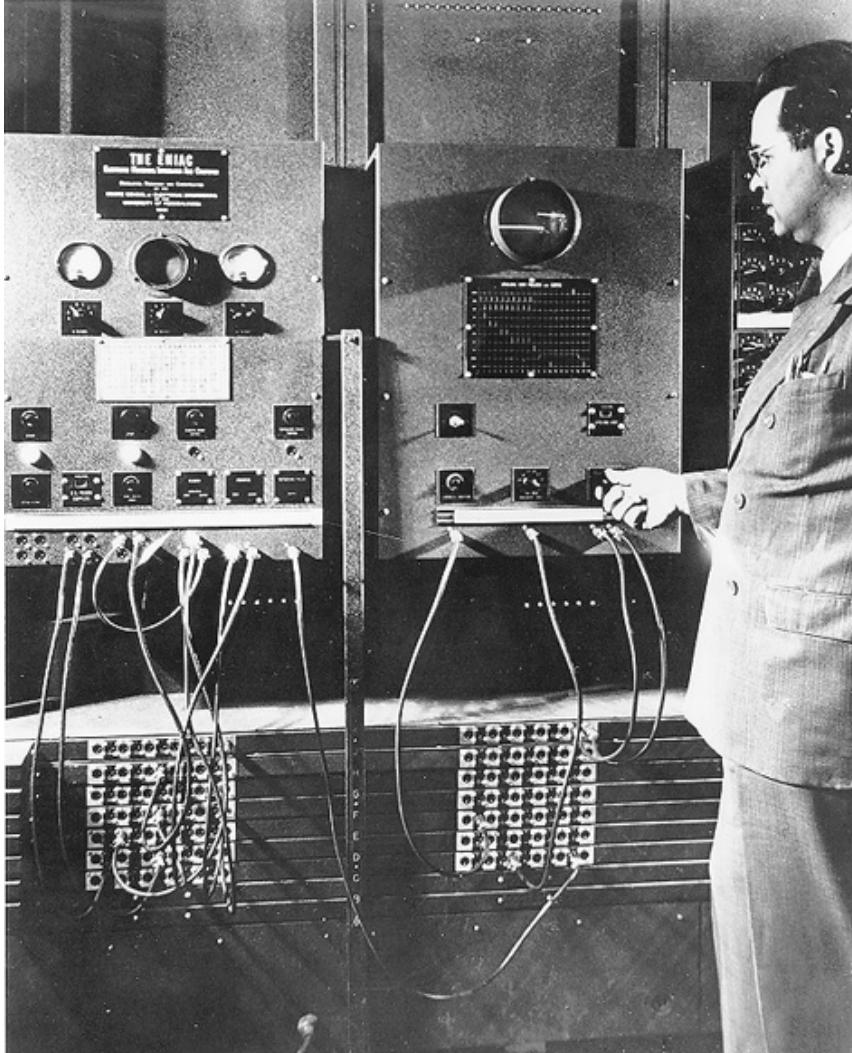
ENIAC – serwis



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

Wymiana
uszkodzonej
lampy
wymagała
sprawdzenia
18 000
możliwości

Testowanie komputera



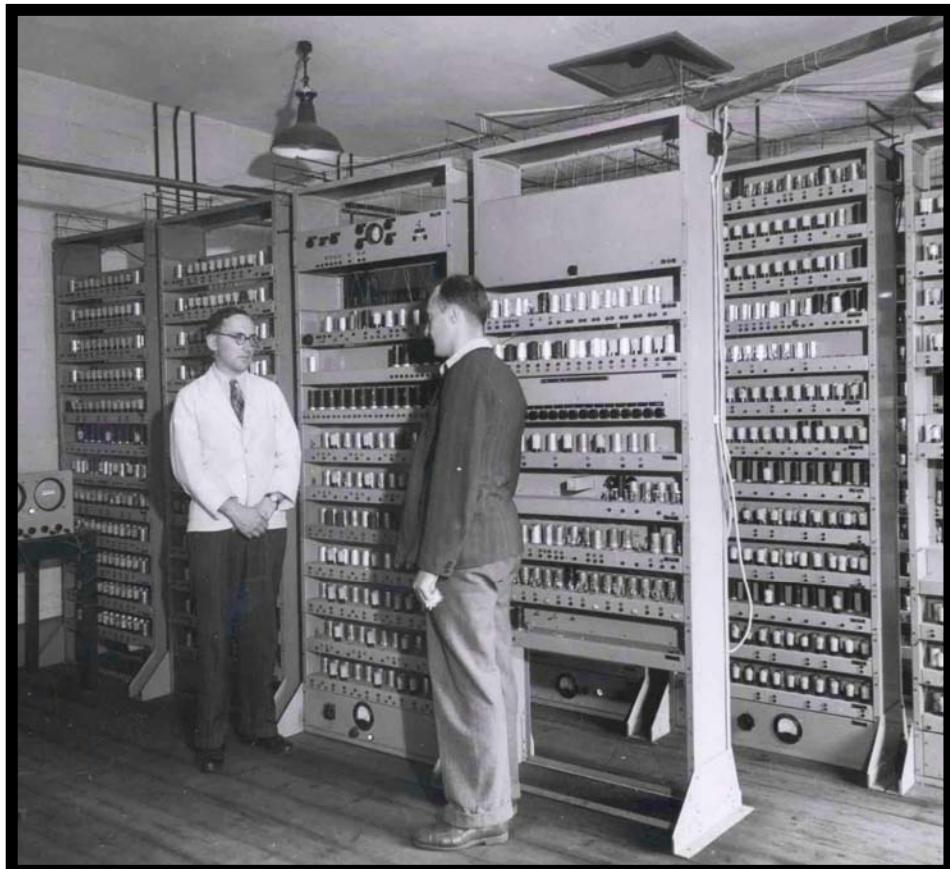
głównym narzędziem jest
oscyloskop

stosuje się specjalne
matryce do generowania
testowych sekwencji
bitów

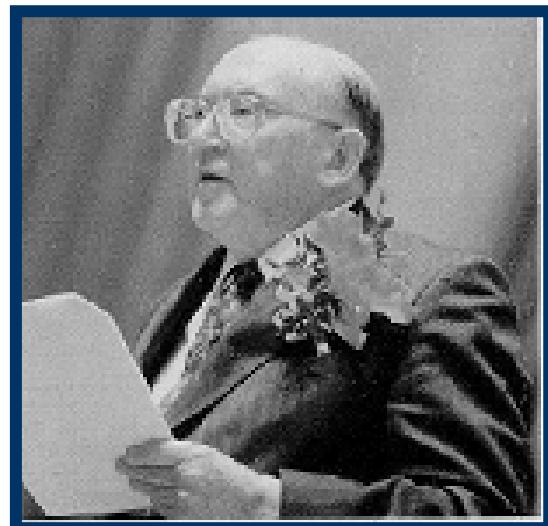
ENIAC – parametry techniczne

- arytmetyka dziesiętna (nie dwójkowa)
- 20 akumulatorów po 10 cyfr w każdym
- programowany ręcznie przełącznikami
- 18,000 lamp, 70,000 rezystorów, 10,000 kondensatorów, 6,000 przełączników
- masa 30 ton
- powierzchnia 1,500 stóp², (30 x 50 stóp)
- pobór mocy 140 kW
- 5,000 operacji dodawania na sekundę (0,005 MIPS)

EDSAC 1 - 1949

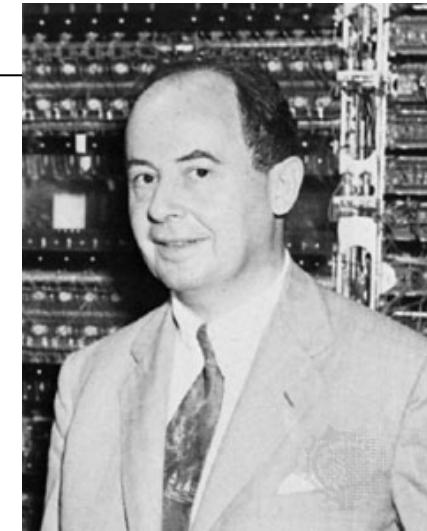


- Maurice Wilkes



1st store program computer
650 instructions/sec
1,400 ft³

<http://www.cl.cam.ac.uk/UoCCL/misc/EDSAC99/>



Architektura von Neumanna

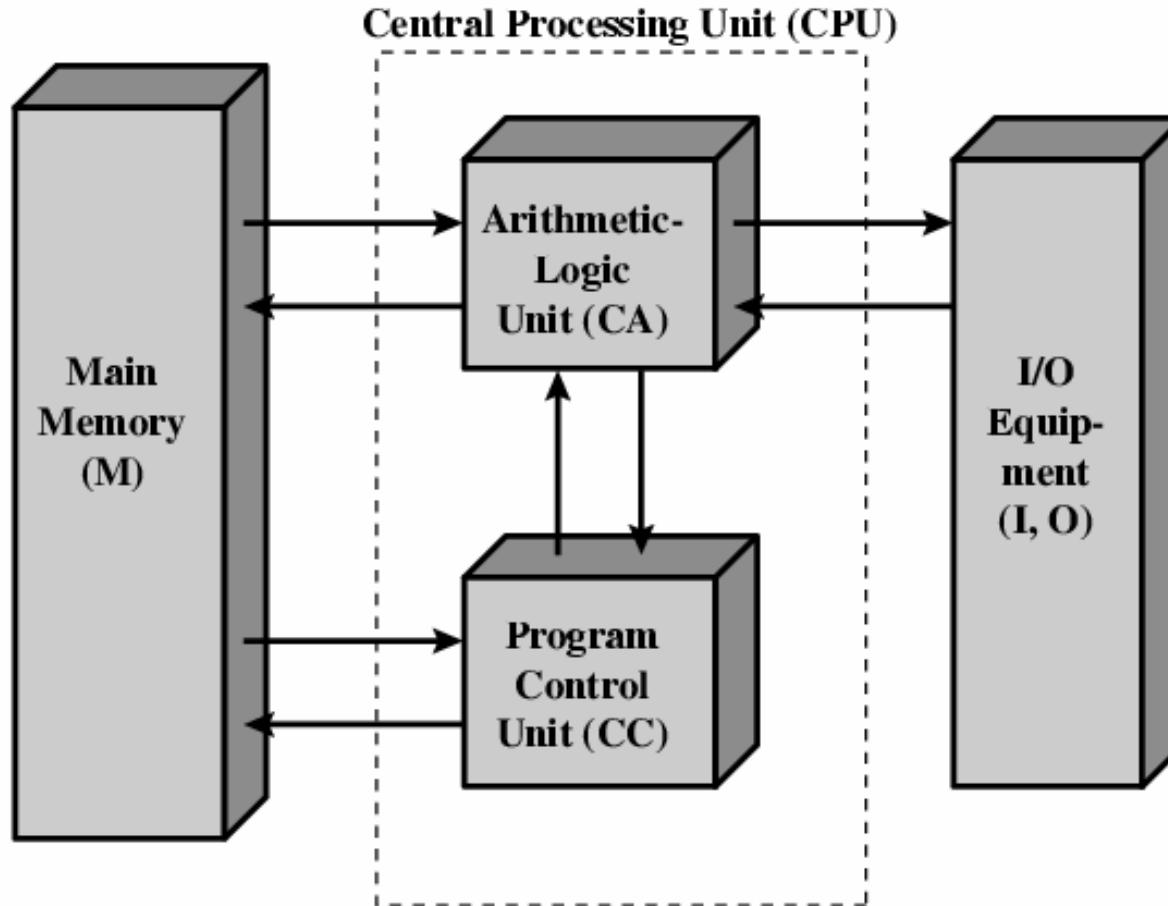
- program przechowywany w pamięci
- program i dane rezydują we wspólniej pamięci
- ALU pracuje na danych dwójkowych
- jednostka sterująca interpretuje i wykonuje rozkazy pobierane z pamięci
- jednostka sterująca zarządza systemem I/O
- Princeton Institute for Advanced Studies
 - IAS (tak samo nazwano prototypowy komputer)
- zakończenie projektu: 1952

John von Neumann



- 1903 – 1957
- matematyk amerykański pochodzenia węgierskiego
- od 1933 profesor ISA w Princeton
- od 1937 członek Akademii Nauk w Waszyngtonie
- od 1954 członek Komisji do Spraw Energii Atomowej USA
- 1945 – 1955: dyrektor Biura Projektów Komputerów Cyfrowych w USA

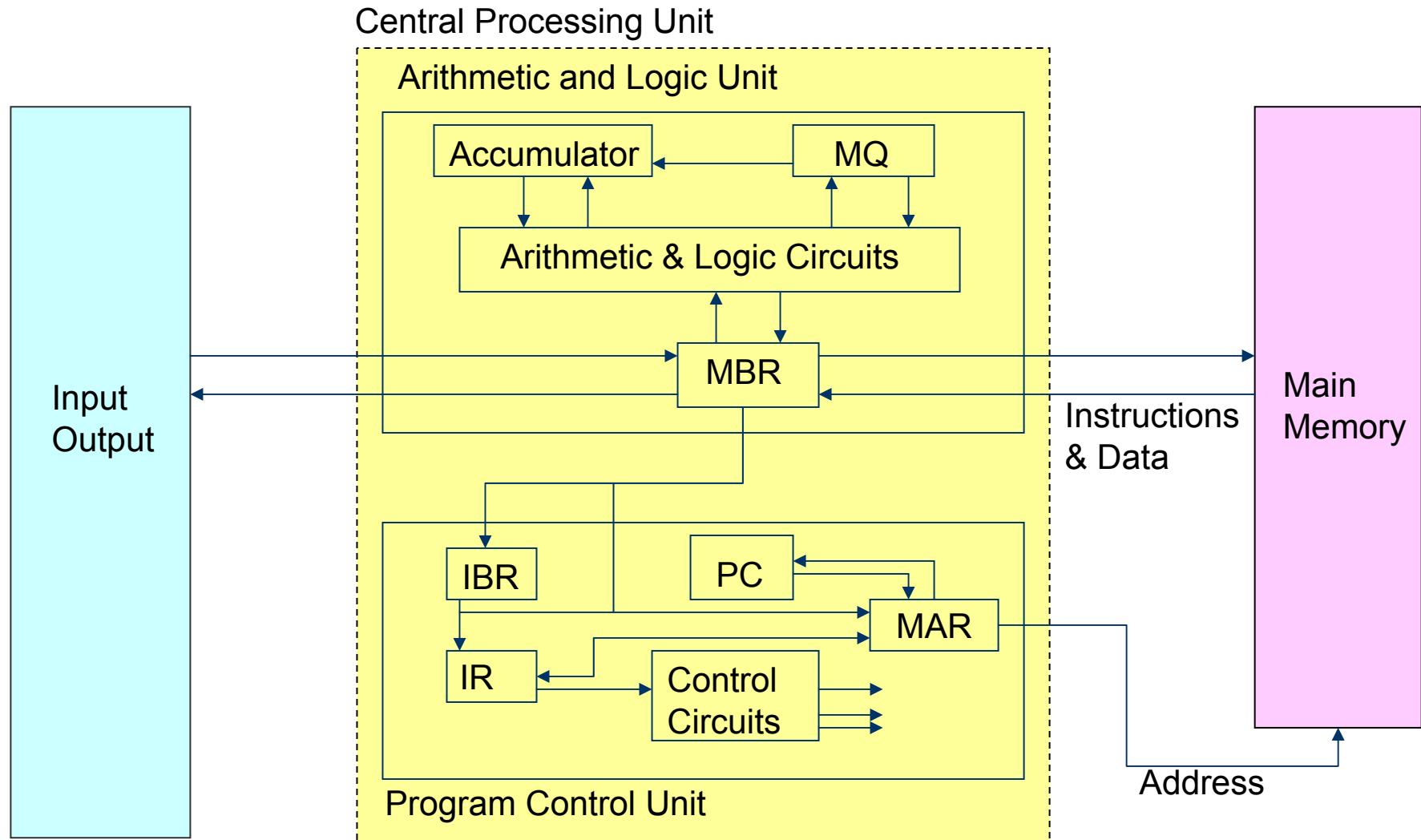
Architektura von Neumanna



IAS – parametry

- 1000 słów po 40 bitów
 - arytmetyka dwójkowa
 - format instrukcji 2 x 20 bitów
- Zbiór rejestrów (zawarty w CPU)
 - Memory Buffer Register (MBR - rejestr bufora pamięci)
 - Memory Address Register (MAR - rejestr adresowy)
 - Instruction Register (IR - rejestr instrukcji)
 - Instruction Buffer Register (IBR - rejestr bufora instrukcji)
 - Program Counter (PC - licznik rozkazów)
 - Accumulator (A - akumulator)
 - Multiplier Quotient (MQ - rejestr ilorazu)

IAS – struktura



Pierwsze komputery komercyjne

- 1947 - Eckert-Mauchly Computer Corporation
- UNIVAC I (Universal Automatic Computer)
 - 1950 - US Bureau of Census, obliczenia biurowe
- powstaje Sperry-Rand Corporation
- koniec lat 50-tych - UNIVAC II
 - większa szybkość
 - większa pamięć

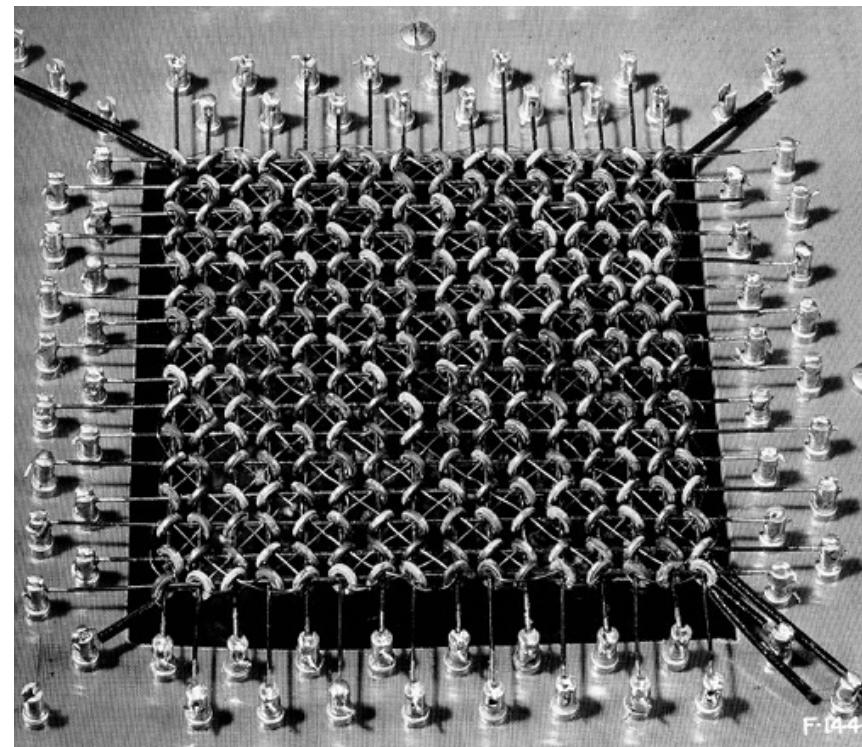


IBM

- produkcja urządzeń do perforowania kart
- 1953 - model 701
 - pierwszy komputer IBM oferowany na rynku
 - obliczenia naukowe
- 1955 - model 702
 - zastosowania biurowe (banki, administracja)
- początek serii komputerów 700/7000

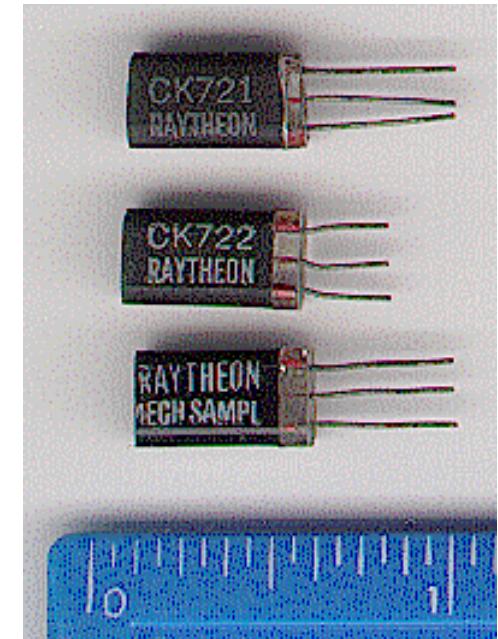
Pamięć ferrytowa

- radykalna poprawa niezawodności
- brak ruchomych części
- pamięć nieulotna
- relatywnie niski koszt
- większe pojemności



Tranzystor

- następca lamp
- znacznie mniejszy
- tańszy
- mniejszy pobór mocy
- wykonany z półprzewodnika
- wynaleziony w 1947 w Bell Labs
- William Shockley *et al.*

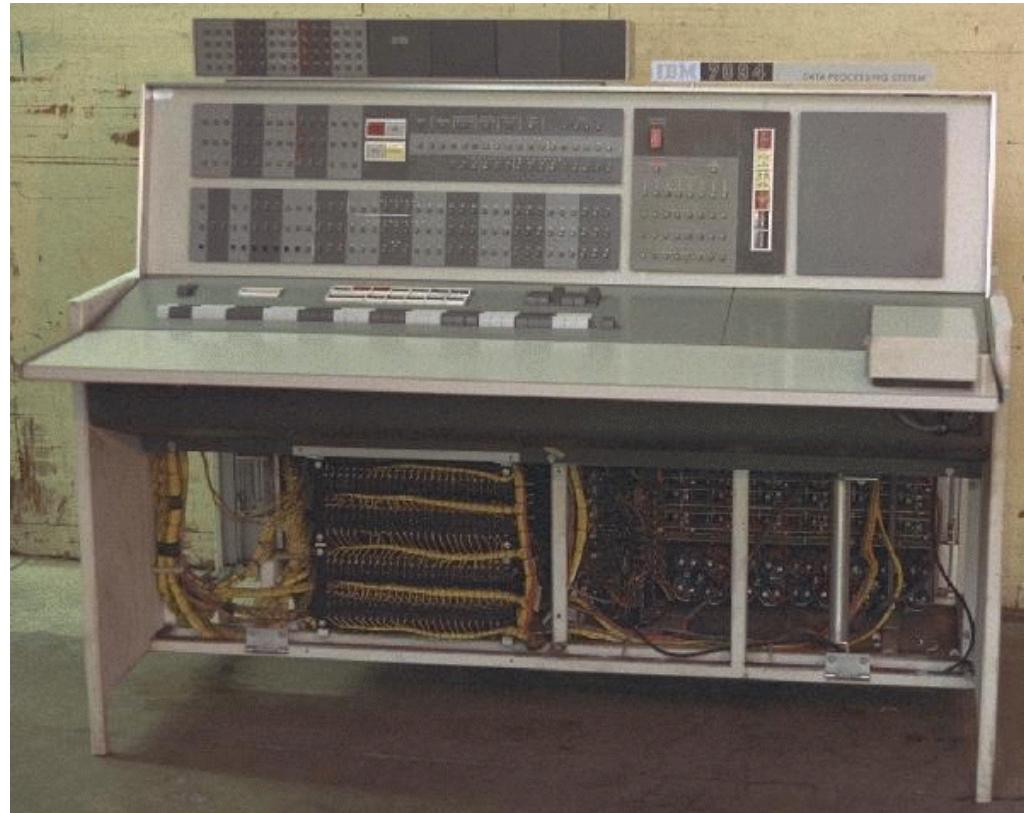


Raytheon CK722

1954

2 generacja komputerów

- podstawowym elementem jest tranzystor
- NCR & RCA – „małe” komputery
- IBM 7000
- DEC - 1957
 - model PDP-1



Minikomputery – IBM 360

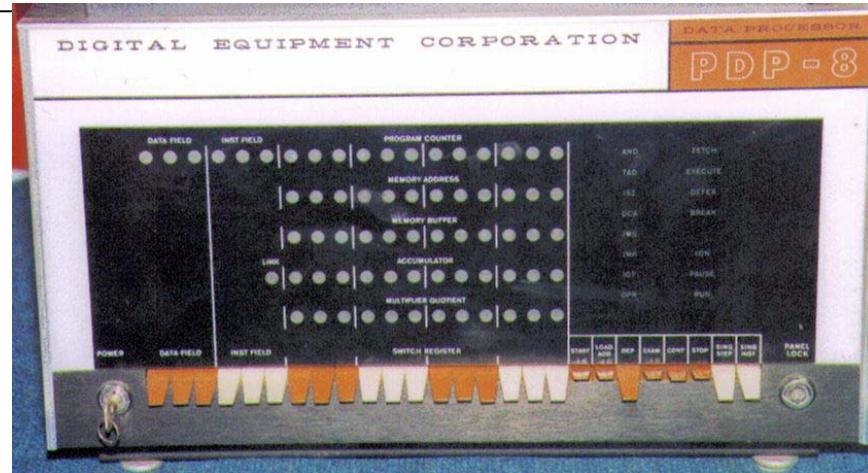
- 1964
- IBM 360 zastępuje serię 7000
 - brak kompatybilności
- pierwsza rodzina popularnych komputerów
 - podobne lub identyczne listy instrukcji
 - podobny lub taki sam OS
 - rosnąca szybkość
 - wzrost liczby układów I/O (terminali)
 - wzrost pojemności pamięci
 - kolejne modele kosztują coraz więcej
- pojawiają się emulatory dla serii 7000 - /1400



- 1964 rok
- 1.6 MHz CPU
- 32 – 256 KB RAM
- \$ 225 000

DEC PDP – 8

- 1964
- pierwszy minikomputer
- klimatyzacja pomieszczenia
nie jest wymagana
- mieści się na biurku
- cena \$16,000
 - dla porównania IBM 360 kosztuje \$100k+
- wbudowane aplikacje & OEM
- nowatorska struktura szyn (OMNIBUS)

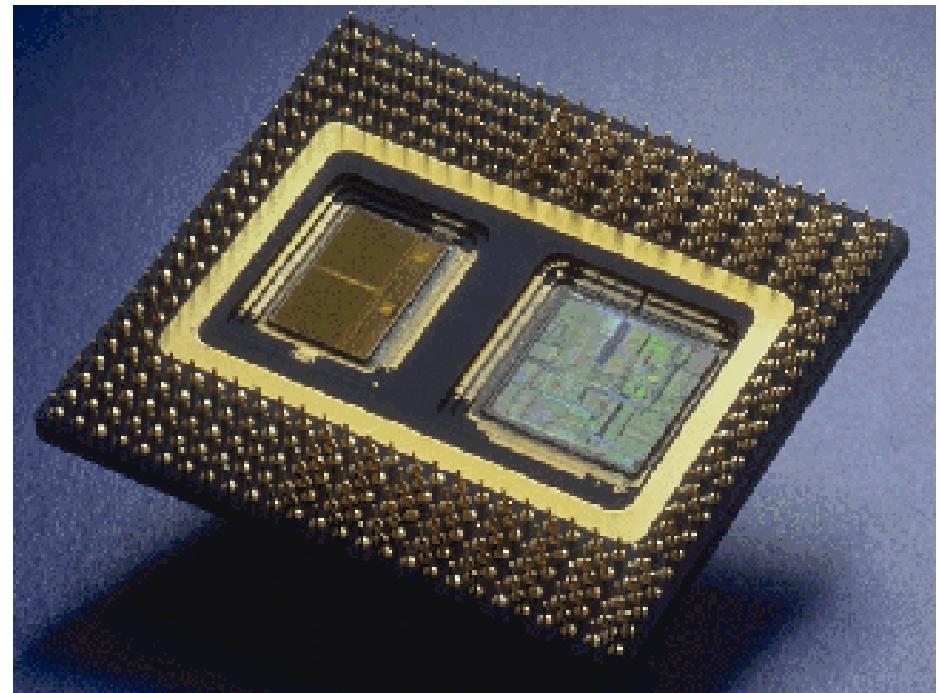


Pamięć półprzewodnikowa

- 1970
- Fairchild
- rozmiar pojedynczego rdzenia ferrytowego
 - cała pamięć ma teraz taki rozmiar jak wcześniej komórka 1-bitowa
- pojemność 256 bitów
- odczyt nie niszczy zawartości
- znacznie większa szybkość niż dla pamięci ferrytowych
- pojemność podwaja się co rok

Rozwój mikroelektroniki

- podstawowy element – układ scalony
- układy SSI, potem MSI, wreszcie LSI i VLSI
- seryjna produkcja układów scalonych na bazie krzemu



Generacje układów elektronicznych

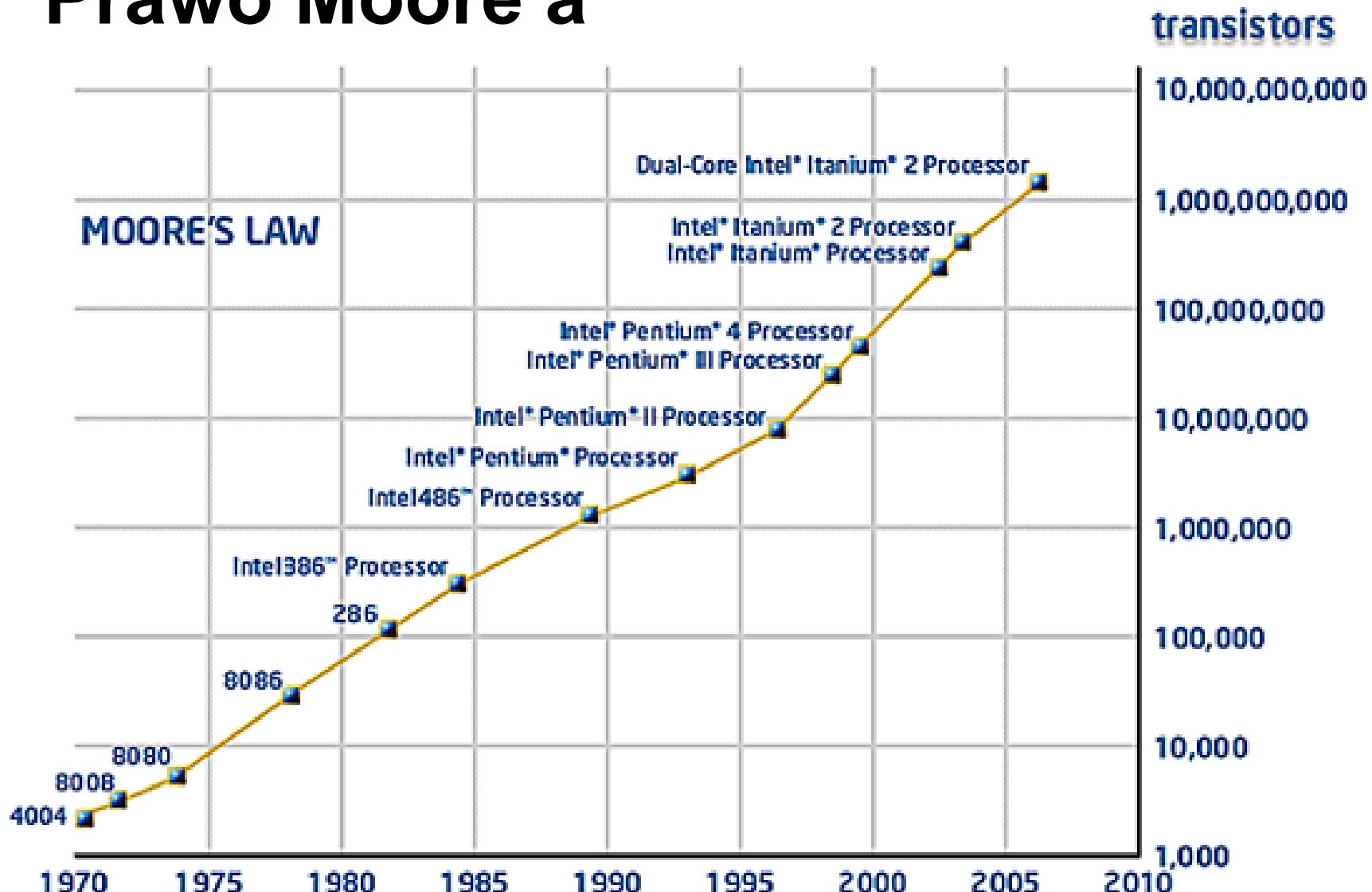
- lampa – 1946 -1957
- tranzystor – 1958 -1964
- układy scalone małej skali integracji (SSI) – od 1965
 - do 100 elementów w układzie
- średnia skala integracji (MSI) - od 1971
 - 100 - 3,000 elementów w układzie
- wielka skala integracji (LSI) – 1971 - 1977
 - 3,000 - 100,000 elementów w układzie
- bardzo wielka skala integracji (VLSI) - 1978 do dziś
 - 100,000 - 100,000,000 elementów w układzie
- ultra wielka skala integracji (UVLSI) – koniec lat 90'
 - ponad 100,000,000 elementów w układzie

Generation	Example Machines	Hardware	Software	Performance
1	ENIAC, UNIVAC I, IBM 700	Vacuum tubes, magnetic drums	Machine code, stored programs	2 Kb memory, 10 KIPS
2	IBM 7094	Transistors, core memory	High level languages	32 Kb memory, 200 KIPS
3	IBM 360 370, PDP 11	ICs, semiconductor memory, microprocesso rs	Timesharing, graphics, structured programming	2 Mb memory, 5 MIPS
4	IBM 3090, Cray XMP, IBM PC	VLSI, networkes, optical disks	Packaged programs, object-oriented languages, expert systems	8 Mb memory, 30 MIPS
5	Sun Sparc, Intel Paragon	ULSI, GaAs, parallel systems	Parallel languages symbolic processing, AI	64 Mb memory, 10 GFLOPS

Prawo Moore'a

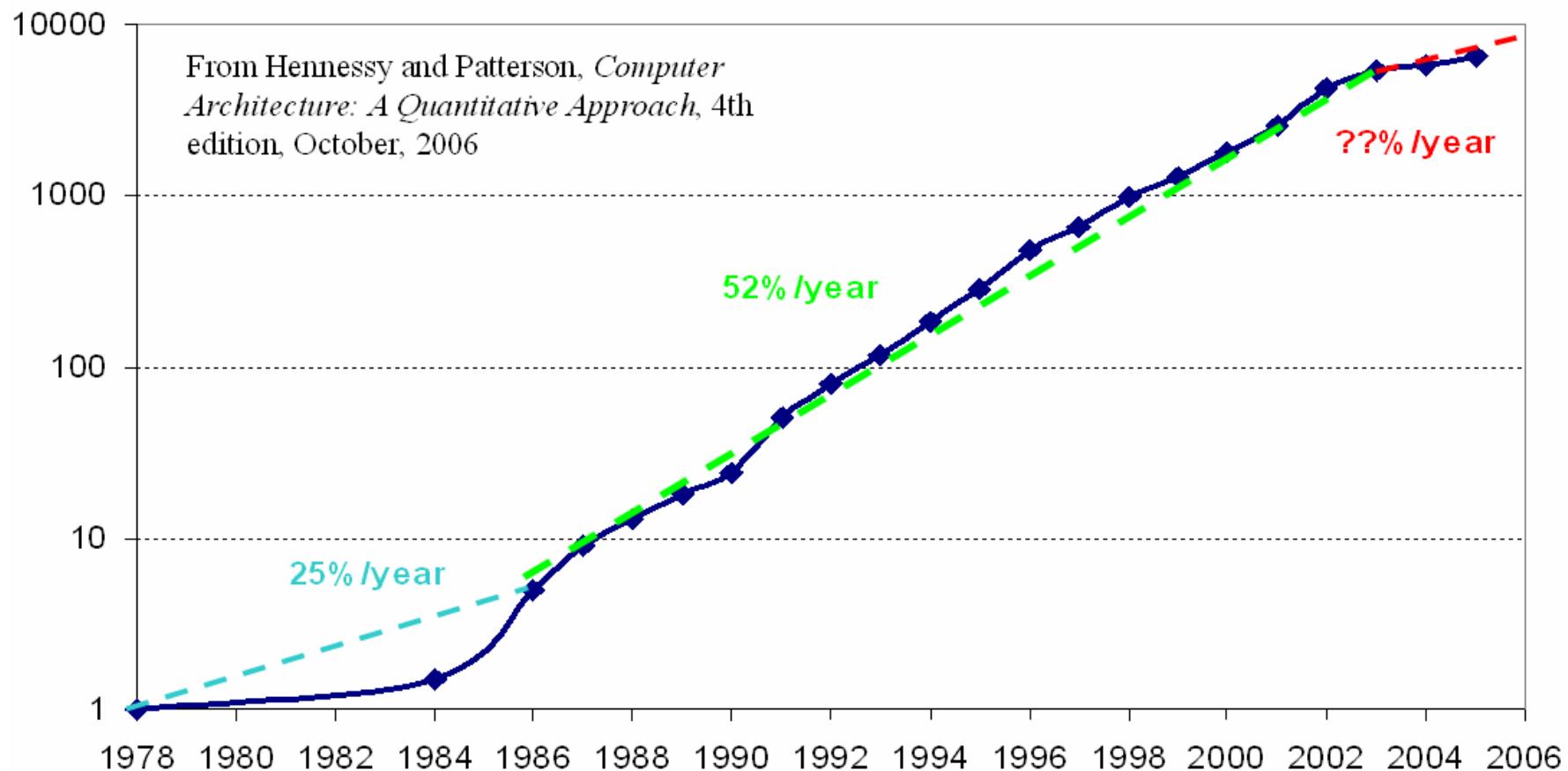
- opisuje tempo wzrostu liczby elementów w układzie
- Gordon Moore – współzałożyciel Intela
- **liczba tranzystorów w układzie podwaja się co roku**
- od roku 1970 wzrost został nieco zahamowany
 - liczba tranzystorów podwaja się co 18 miesięcy
- koszt układu pozostaje niemal bez zmian
- większa gęstość upakowania elementów skraca połączenia, przez co wzrasta szybkość układów
- zmniejsza się rozmiar układów
- zmniejszają się wymagania na moc zasilania i chłodzenie
- mniejsza liczba połączeń w systemie cyfrowym poprawia niezawodność

Prawo Moore'a



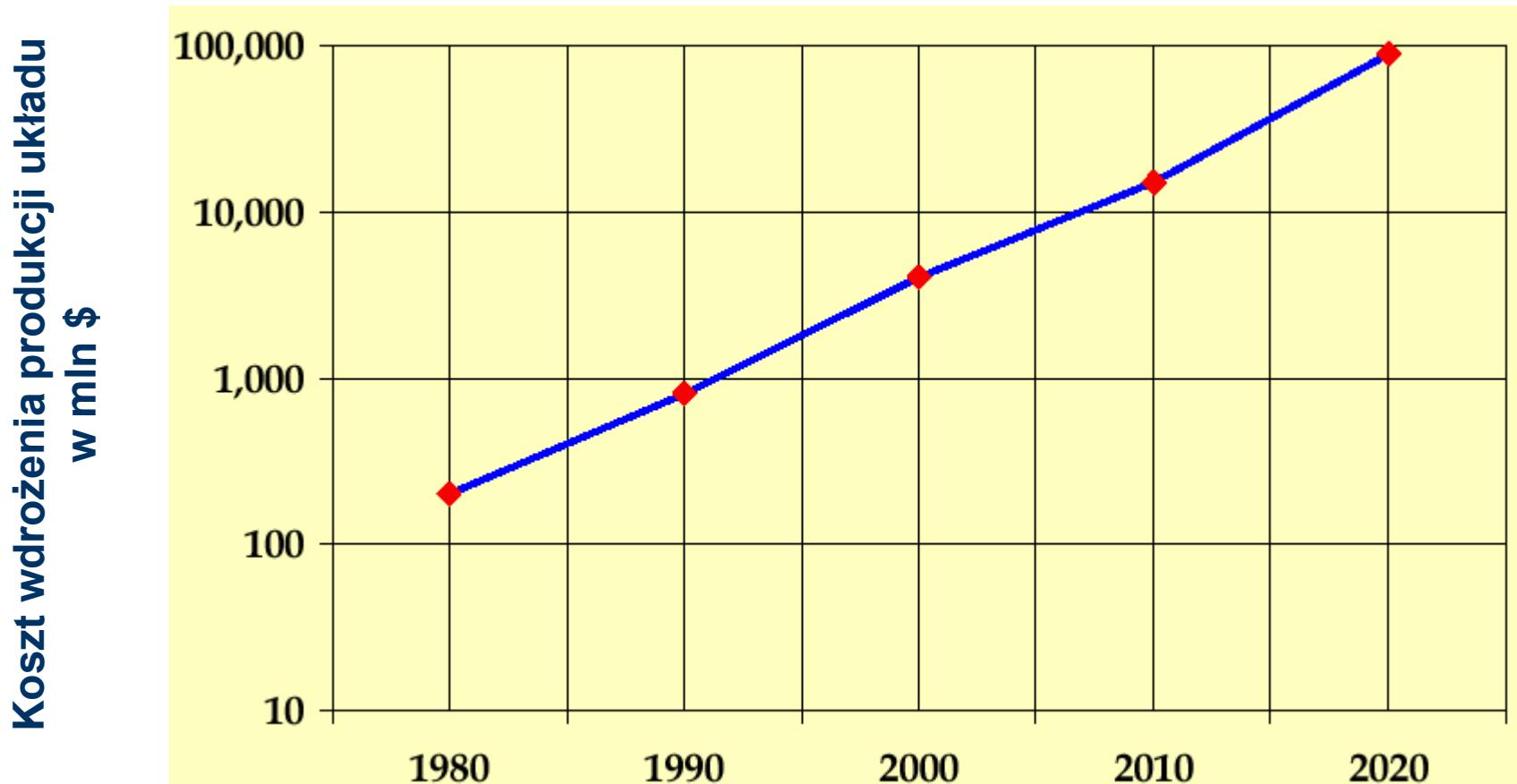
Prawo Moore'a

Uniprocessor Performance



2 prawo Moore'a

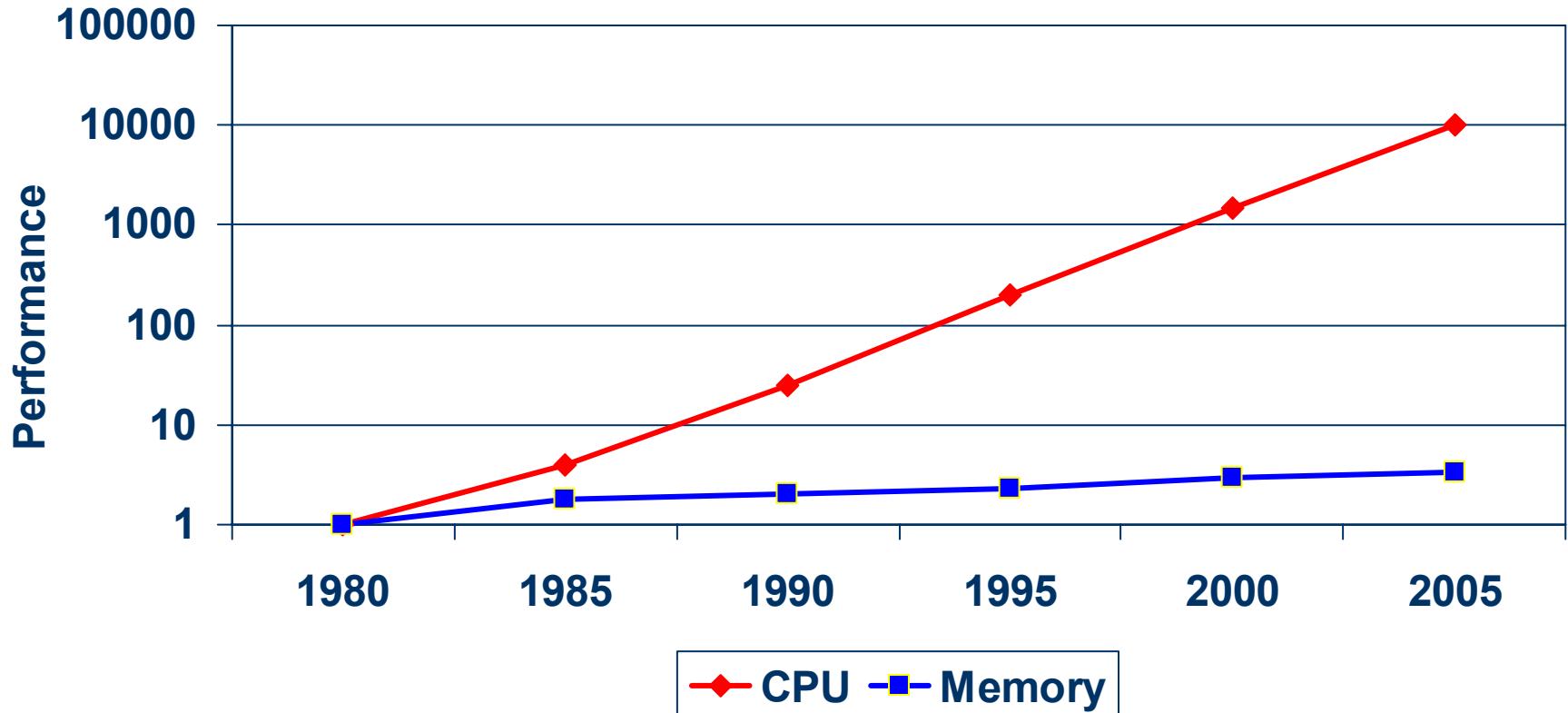
Koszt uruchomienia produkcji nowego układu CPU UVLSI wzrasta wykładniczo



Wzrost pojemności DRAM

<u>Rok</u>	<u>Rozmiar</u>
1980	64 Kb
1983	256 Kb
1986	1 Mb
1989	4 Mb
1992	16 Mb
1996	64 Mb
1999	256 Mb
2002	1 Gb
2010	?

Rozwój mikroprocesorów i DRAM



Szybkość DRAM i CPU

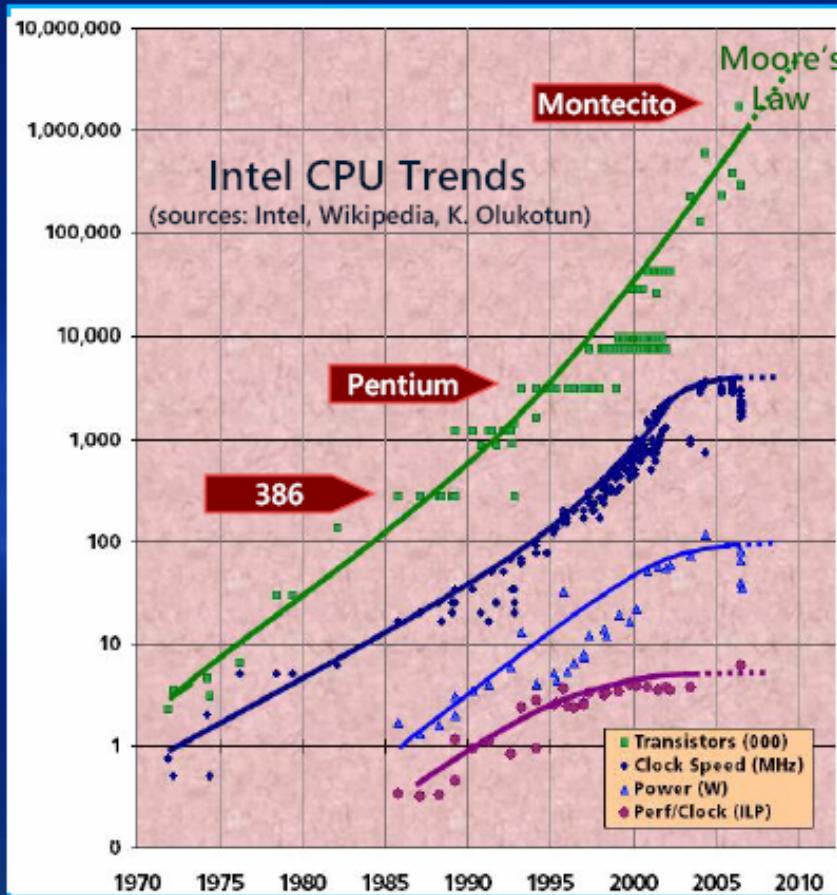
- Szybkość procesorów rosła przedżej niż szybkość pamięci RAM
- Jakie problemy wynikają stąd dla konstruktorów komputerów?

Rozwiązania (DRAM vs. CPU)

- zwiększyć liczbę bitów odczytywanych w jednym cyklu
 - DRAM powinna być „szersza” a nie „głębbsza”
- zmienić interfejs DRAM
 - pamięć cache
- zmniejszyćczęstość dostępów do pamięci
 - bardziej skomplikowana pamięć cache umieszczona w CPU
- zwiększyć przepływność szyn
 - szyny typu high-speed
 - szyny hierarchiczne

Wzrost złożoności procesorów

Each year we get faster ~~more~~ processors



- Historically: Boost single-stream performance via more complex chips, first via one big feature, then via lots of smaller features.
- Now: Deliver more cores per chip.
- The free lunch is over for today's sequential apps and many concurrent apps (expect some regressions). We need killer apps with lots of latent parallelism.
- A generational advance >OO is necessary to get above the "threads+locks" programming model.

Postęp technologii

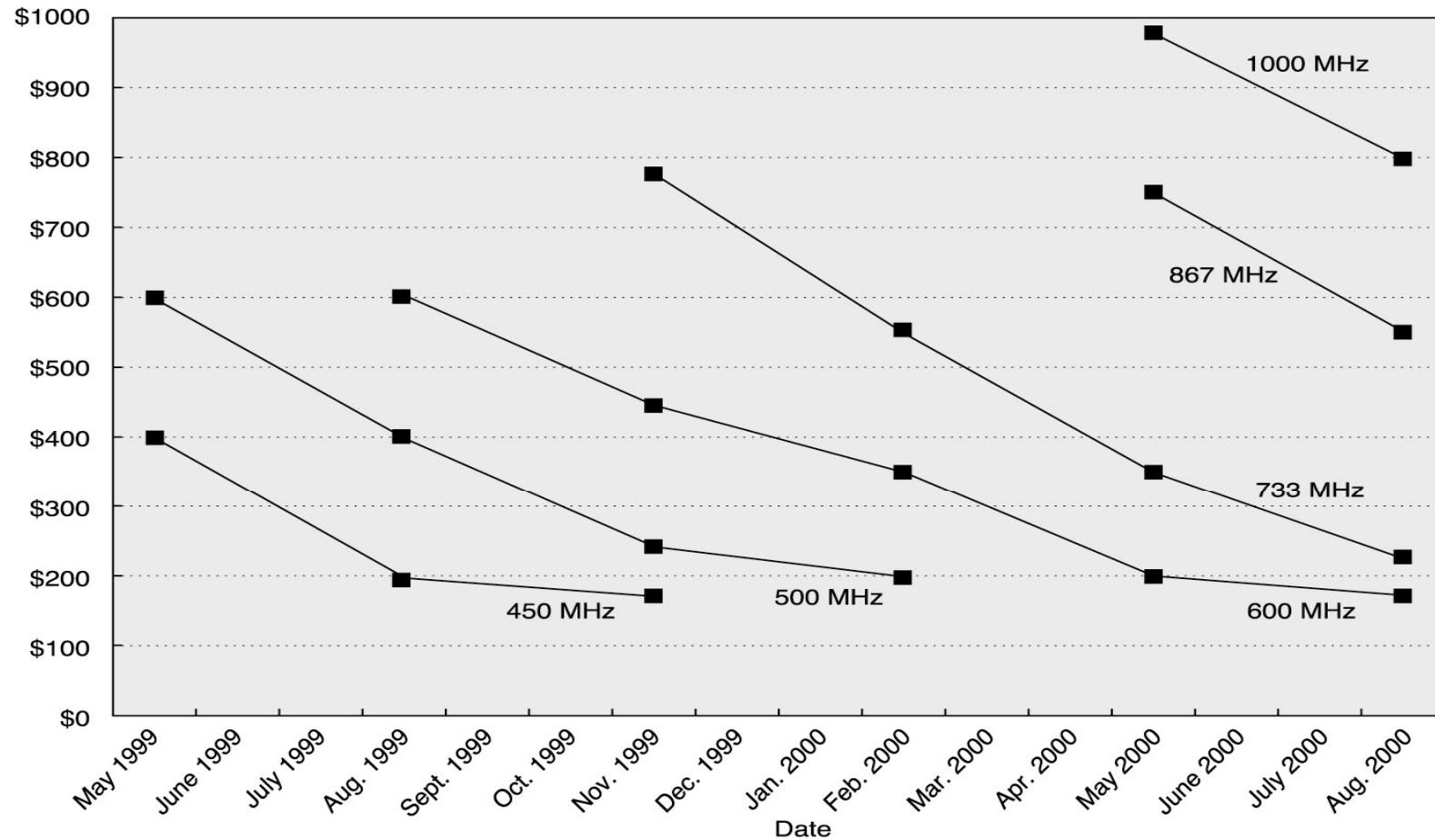
- Procesor
 - złożoność układowa: 30% na rok
 - częstotliwość zegara: 20% na rok
- Pamięć
 - DRAM pojemność: około 60% na rok (4x co 3 lata)
 - szybkość pamięci: około 10% na rok
 - koszt za 1 bit: maleje o 25% na rok
- Dysk
 - pojemność: 60% na rok
- Przepływność w sieciach
 - wzrost o więcej niż 100% na rok!

Postęp technologii

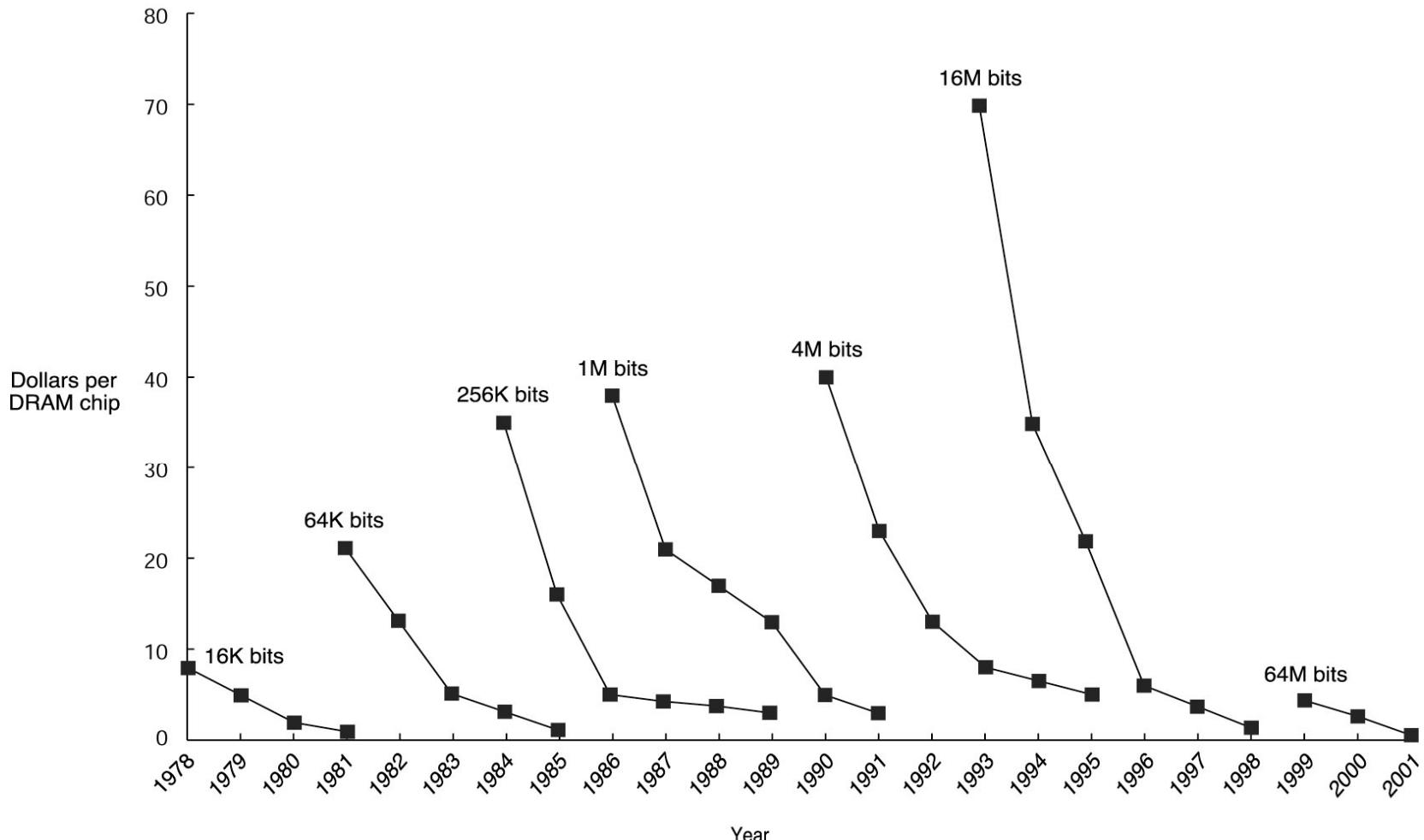
Inne ujęcie, według prof. Tyagi (2003)

	<u>Pojemność</u>	<u>Szybkość</u>
Układy cyfrowe	2x / 3 lata	2x / 3 lata
DRAM	4x / 3 lata	2x / 10 lat
Dyski	4x / 3 lata	2x / 10 lat

Koszt procesorów (Intel)



Koszt pamięci



Intel – pierwsze procesory

- 1971 – układ **4004**
 - uważany za pierwszy microprocessor na świecie
 - wszystkie elementy CPU w jednym układzie
 - 4 – bitowy format danych
- Zastąpiony w 1972 przez układ **8008**
 - 8 – bitowy format danych
 - obydwa układy zaprojektowano do specjalistycznych zastosowań
- 1974 – układ **8080**

pierwszy mikroprocesor Intel ogólnego przeznaczenia

Ewolucja Pentium

- 8080
 - pierwszy mikroprocesor ogólnego przeznaczenia
 - 8-bitowa szyna danych
 - zastosowany w pierwszym komputerze osobistym Altair
- 8086
 - znacznie większa moc obliczeniowa, użyty w IBM PC XT
 - 16-bitowa szyna danych, przestrzeń adresowa 1MB
 - pamięć cache programu, pobieranie kilku instrukcji na zasadzie
 - wersja 8088 (8-bitowa zewnętrzna szyna danych)
- 80286
 - przestrzeń adresowa 16 MB
- 80386
 - architektura 32-bitowa
 - wielozadaniowość (multitasking)

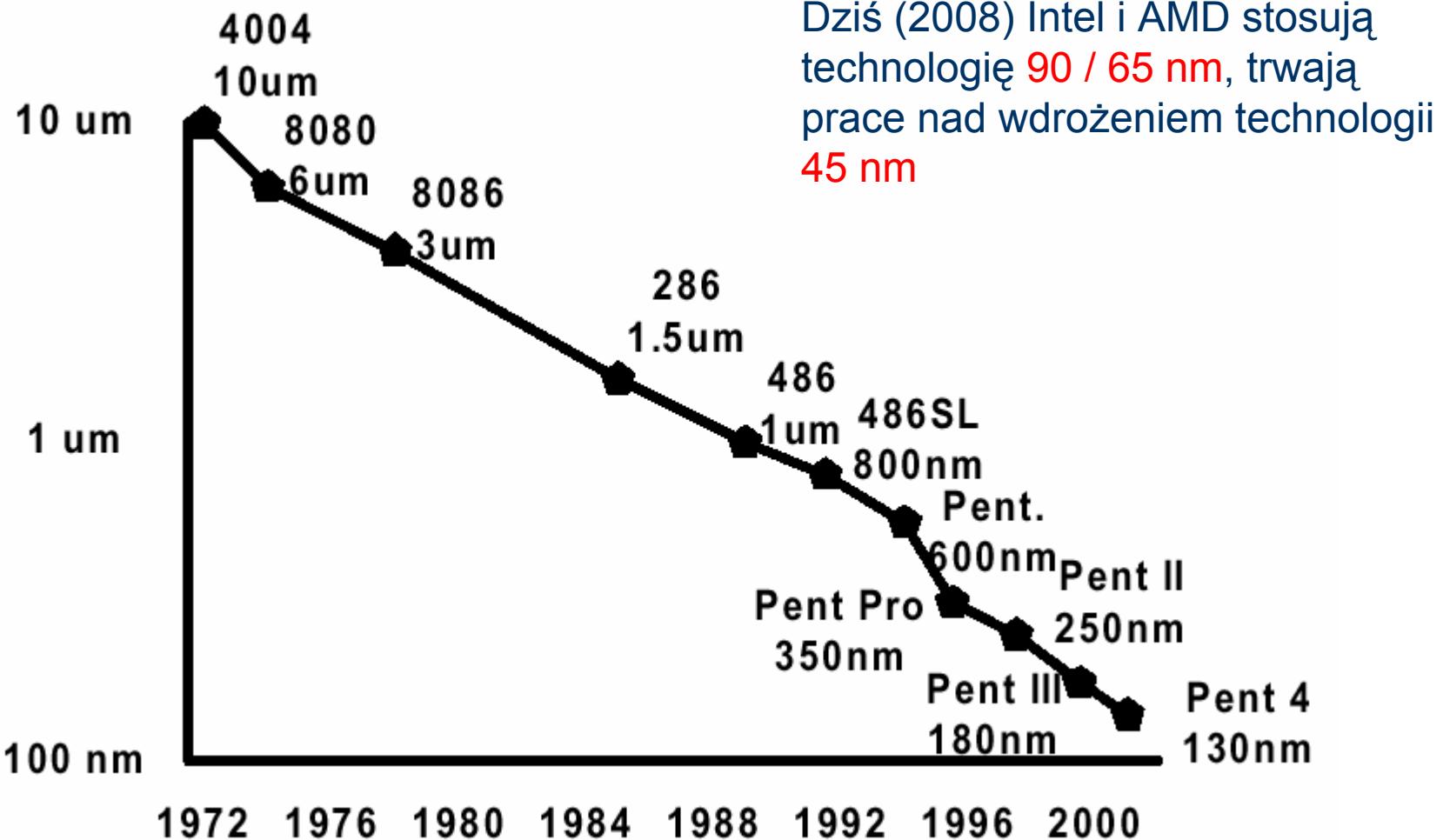
Ewolucja Pentium cd.

- 80486
 - zaawansowana pamięć cache i potok instrukcji
 - wbudowany koprocesor arytmetyczny
- Pentium
 - architektura superskalarna
 - równocześnie wykonuje się wiele instrukcji
- Pentium Pro
 - ulepszona architektura superskalarna
 - technika przemianowywania rejestrów
 - przewidywanie skoków
 - analiza przepływu danych
 - spekulacyjne wykonywanie instrukcji

Ewolucja Pentium cd.

- Pentium II
 - technologia MMX
 - przetwarzanie grafiki, audio & video
- Pentium III
 - dodatkowe instrukcje zmiennoprzecinkowe dla grafiki 3D
- Pentium 4
 - uwaga – pierwsze oznaczenie cyfrą arabską, a nie rzymską !
 - dalsze ulepszenia FPU i operacji multimedialnych
- Itanium / Itanium 2
 - architektura 64-bitowa
 - EPIC – Explicit Parallel Instruction Computing
 - rejesty 128-bitowe
 - Instrukcje 41-bitowe w pakietach 128-bitowych (3 instr.
+ 5 bitów pokazujących typ i współzależności)

Granice technologii ?



Granice technologii ?

**Moore's Law will provide
transistors**

Intel process technology capabilities

High Volume Manufacturing	2004	2006	2008	2010	2012	2014	2016	2018
Feature Size	90nm	65nm	45nm	32nm	22nm	16nm	11nm	8nm
Integration Capacity (Billions of Transistors)	2	4	8	16	32	64	128	256

Use transistors for

- Multiple cores
- On-core memory (caches)
- New features (*Ts)

**Multiple cores and caches address power and
memory latency issues**



Roadmap

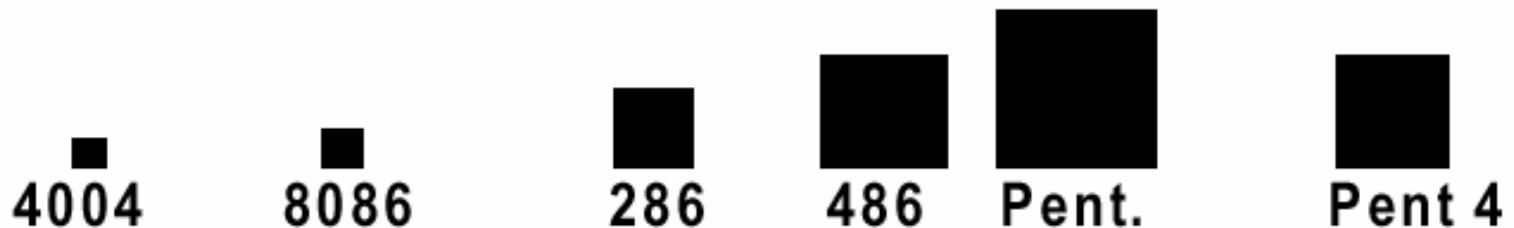
	2004	2007	2010	2013	2016
Technology Node [nm]	90	65	45	32	22
Transistor count [Mtr]			1500	3092	6184
Transistor Density [Mtr/cm ²]	77	154	309	617	1235
Chip Size	140				280
Clock freq [GHz]	3		15		53
Vdd	1.2	1.1	1.0	0.9	0.7
DRAM half pitch	90	65	54	32	22
Signal IO Pads	512	1024	1024	1024	1024
Power Pads	1024				2048

11

Trend rozwoju technologii

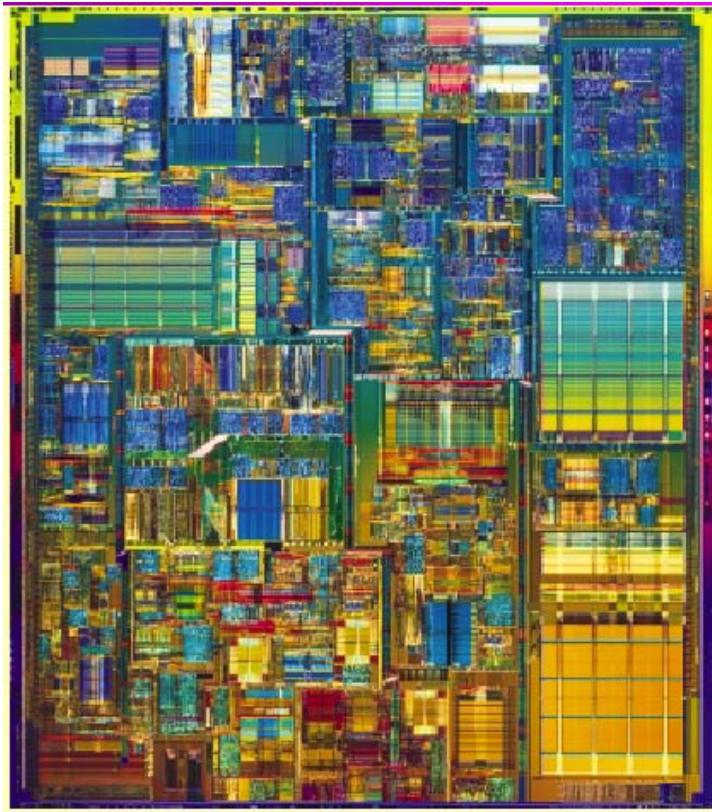
❑ Larger chips

- Trend is toward more RAM, less logic per chip
- Historically 2x per generation; leveling off? **McKinley**
- McKinley has large on-chip caches **IA64**

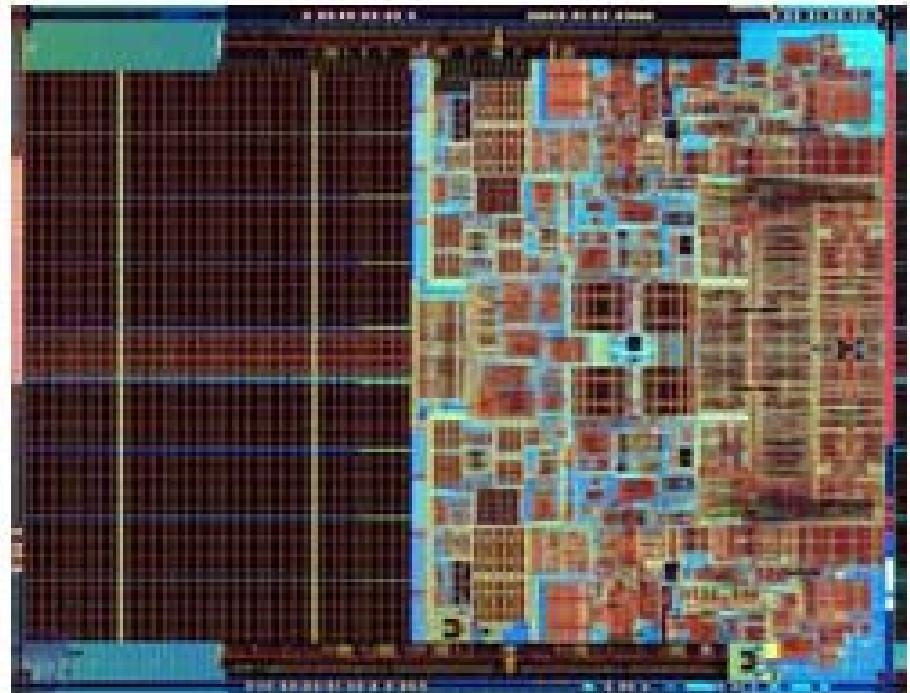


=> larger wafers to reduce fab costs

Współczesny procesor



P4HT: Courtesy: Intel Corp.

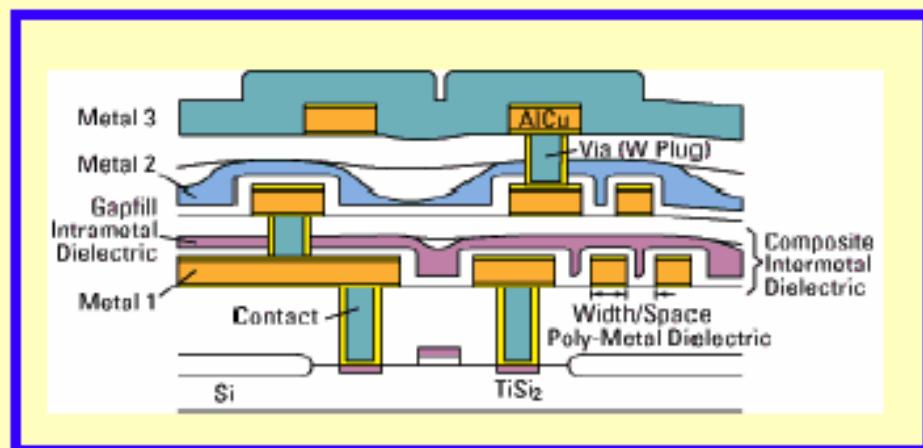


Intel Core 2 Duo: Courtesy: Intel Corp.

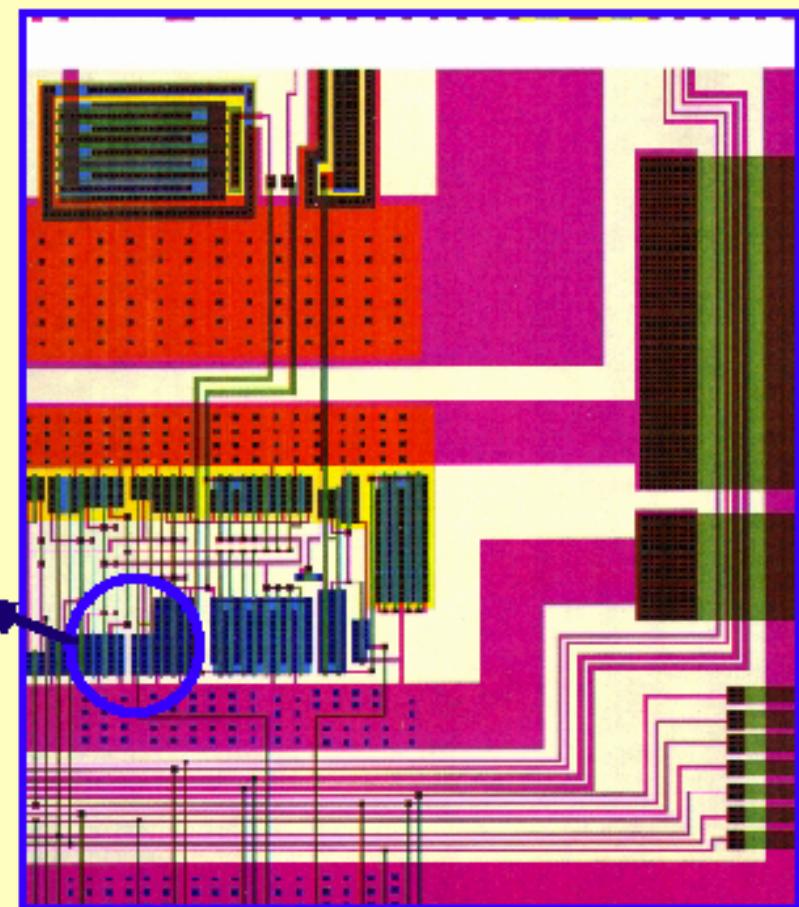
Technologia wielowarstwowa

More detail

Multilevel structure



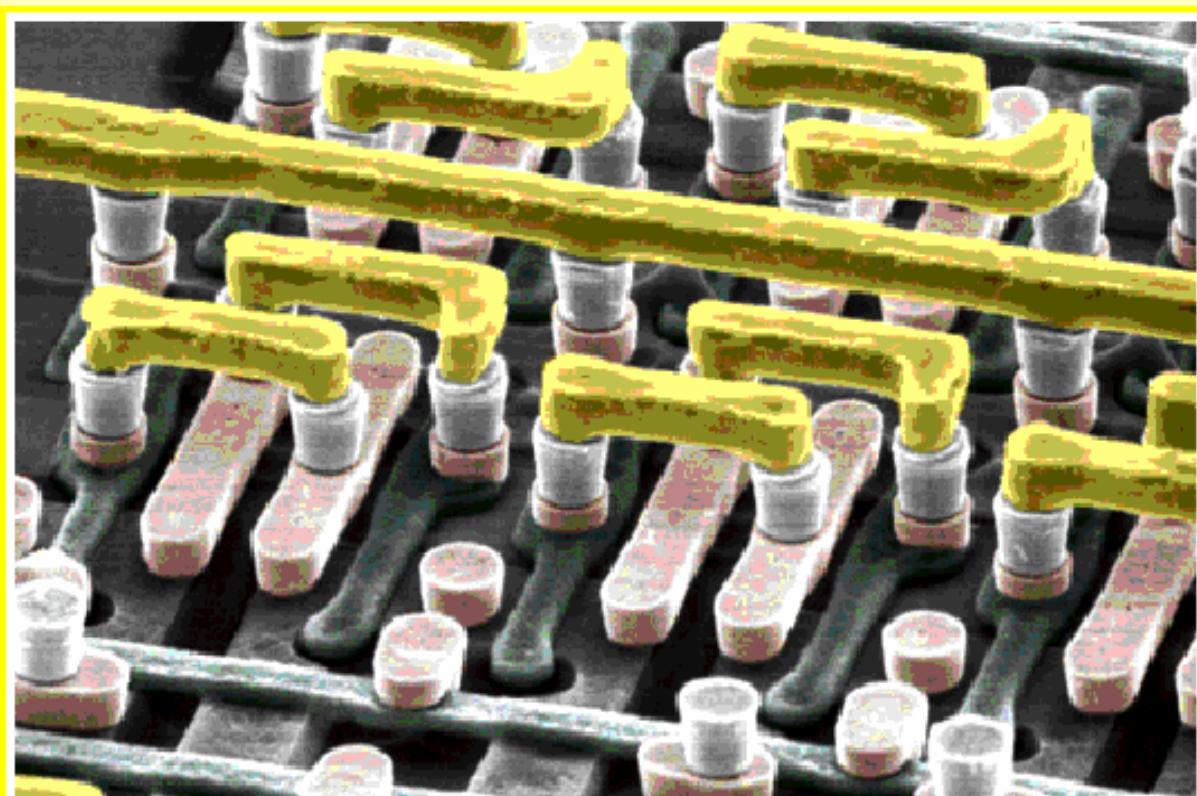
CROSS-SECTIONAL VIEW (IBM)



LAYOUT: TOP VIEW

Technologia wielowarstwowa cd..

Fragment procesora UVLSI, zdjęcie z mikroskopu elektronowego



Courtesy IBM Corp.

Prognozy bywają niedokładne

Twój PC w 2005 roku według J. Breechera, *Clark University, 2002*

- zegar procesora **8 GHz**
- pojemność pamięci RAM 1 GB
- pojemność dysku 240 GB (0,24 TB)

Potrzebujemy kolejnych jednostek miar:

Mega \Rightarrow Giga \Rightarrow Tera \Rightarrow ???

Prognozy bywają niedokładne cd..

YEAR	2002	2005	2008	2011	2014
TECHNOLOGY	130 nm	100 nm	70 nm	50 nm	35 nm
CHIP SIZE	400 mm ²	600 mm ²	750 mm ²	800 mm ²	900 mm ²
NUMBER OF TRANSISTORS (LOGIC)	400 M	1 Billion	3 Billion	6 Billion	16 Billion
DRAM CAPACITY	2 Gbits	10 Gbits	25 Gbits	70 Gbits	200 Gbits
MAXIMUM CLOCK FREQUENCY	1.6 GHz	2.0 GHz	2.5 GHz	3.0 GHz	3.5 GHz
MINIMUM SUPPLY VOLTAGE	1.5 V	1.2 V	0.9 V	0.6 V	0.6 V
MAXIMUM POWER DISSIPATION	130 W	160 W	170 W	175 W	180 W
MAXIMUM NUMBER OF I/O PINS	2500	4000	4500	5500	6000

**ITRS -
International
Technology
Roadmap for
Semiconductors**

**prognoza
z 1999 roku**

50 lat rozwoju komputerów

Year	Name	Size (cu. ft.)	Power (watts)	Performance (adds/sec)	Memory (KB)	Price	Price- performance vs. UNIVAC	Adjusted price (2003 \$)	Adjusted price- performance vs. UNIVAC
1951	UNIVAC I	1,000	125,000	2,000	48	\$1,000,000	1	\$6,107,600	1
1964	IBM S/360 model 50	60	10,000	500,000	64	\$1,000,000	263	\$4,792,300	318
1965	PDP-8	8	500	330,000	4	\$16,000	10,855	\$75,390	13,135
1976	Cray-1	58	60,000	166,000,000	32,000	\$4,000,000	21,842	\$10,756,800	51,604
1981	IBM PC	1	150	240,000	256	\$3,000	42,105	\$5,461	154,673
1991	HP 9000/ model 750	2	500	50,000,000	16,384	\$7,400	3,556,188	\$9,401	16,122,356
1996	Intel PPro PC (200 MHz)	2	500	400,000,000	16,384	\$4,400	47,846,890	\$4,945	239,078,908
2003	Intel Pentium 4 PC (3.0 GHz)	2	500	6,000,000,000	262,144	\$1,600	1,875,000,000	\$1,600	11,452,000,000

Systemy „desktop” i wbudowane

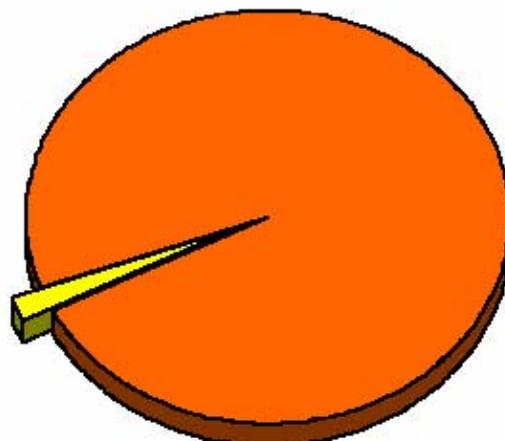


Dominant Species

- Embedded = most processors!
 - 300 million PC and server
 - 9000 million embedded



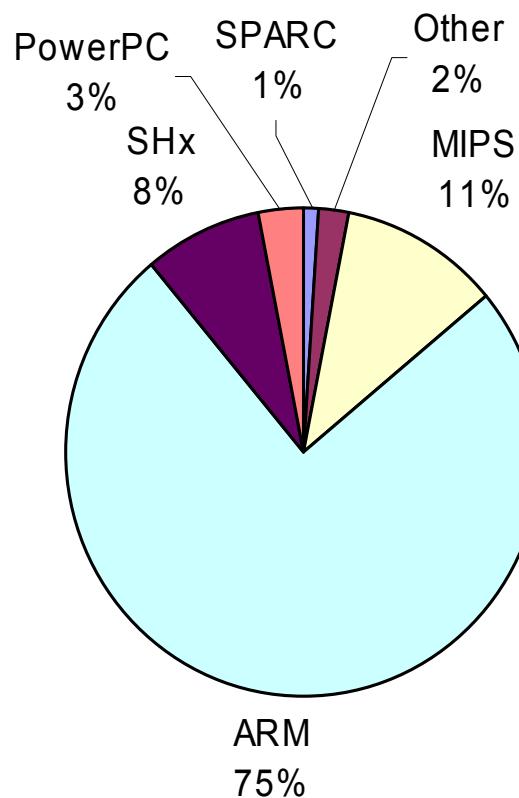
"Desktop"
2%



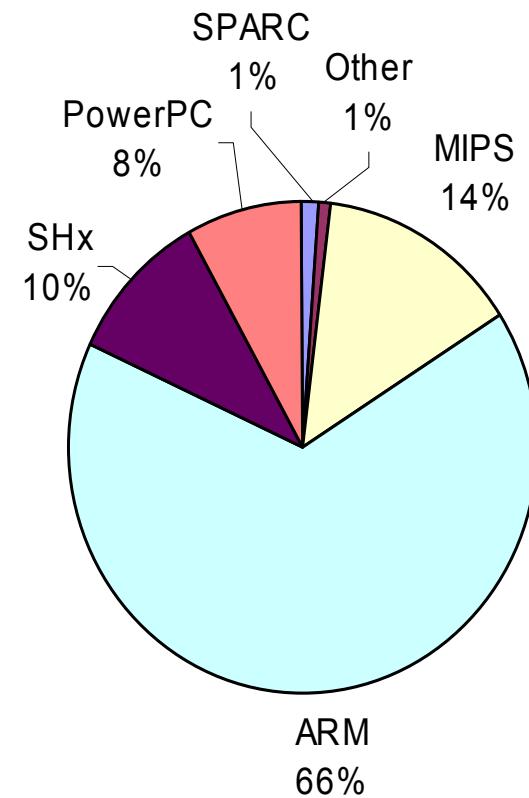
"Embedded"
98%

Rynek procesorów RISC

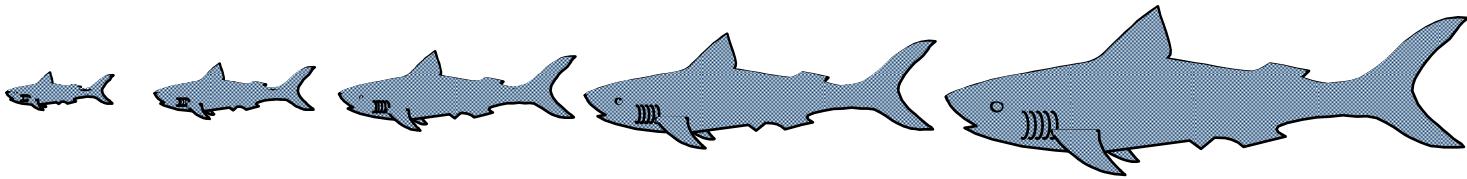
2001



2007

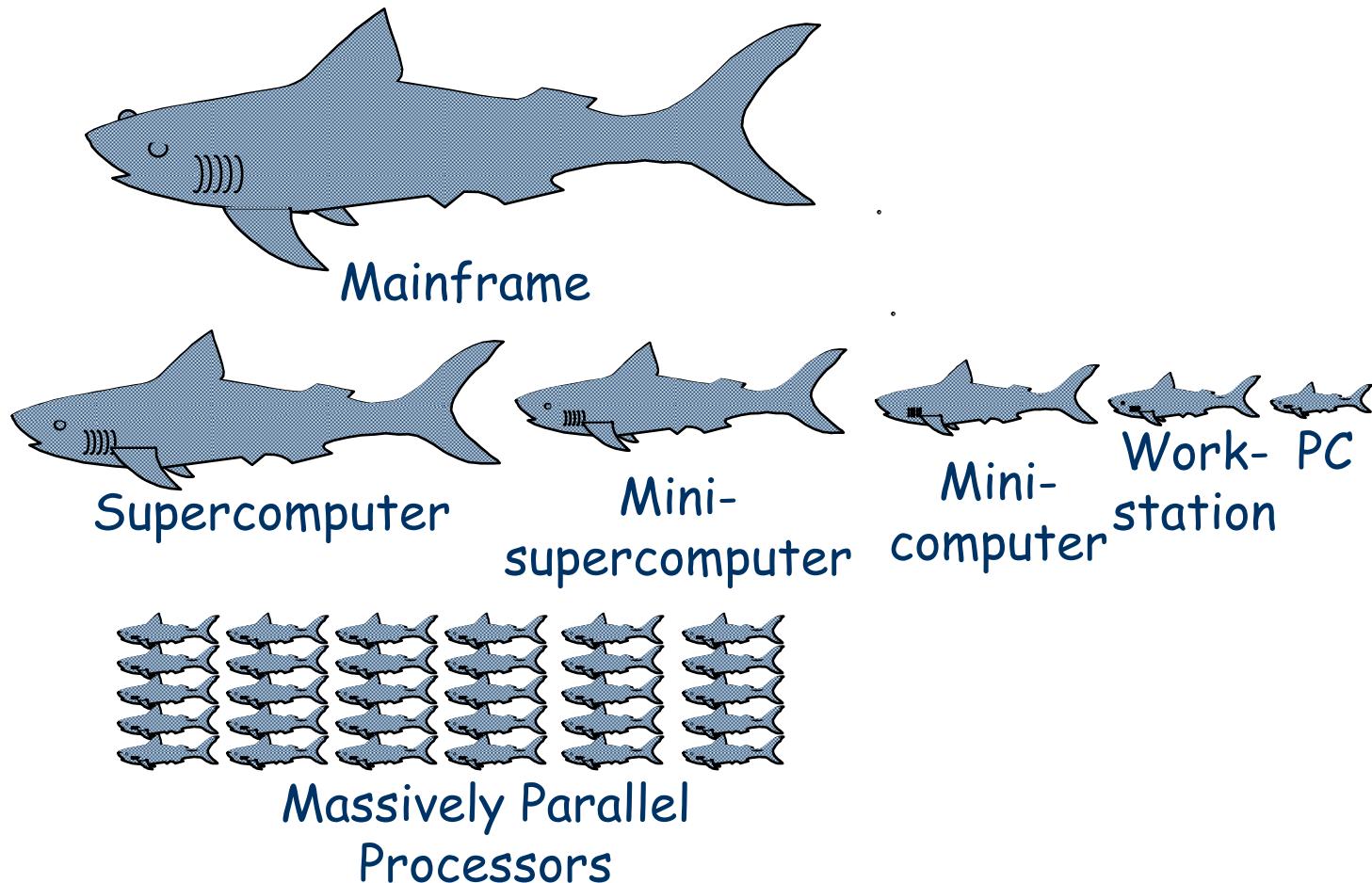


Łańcuch pokarmowy

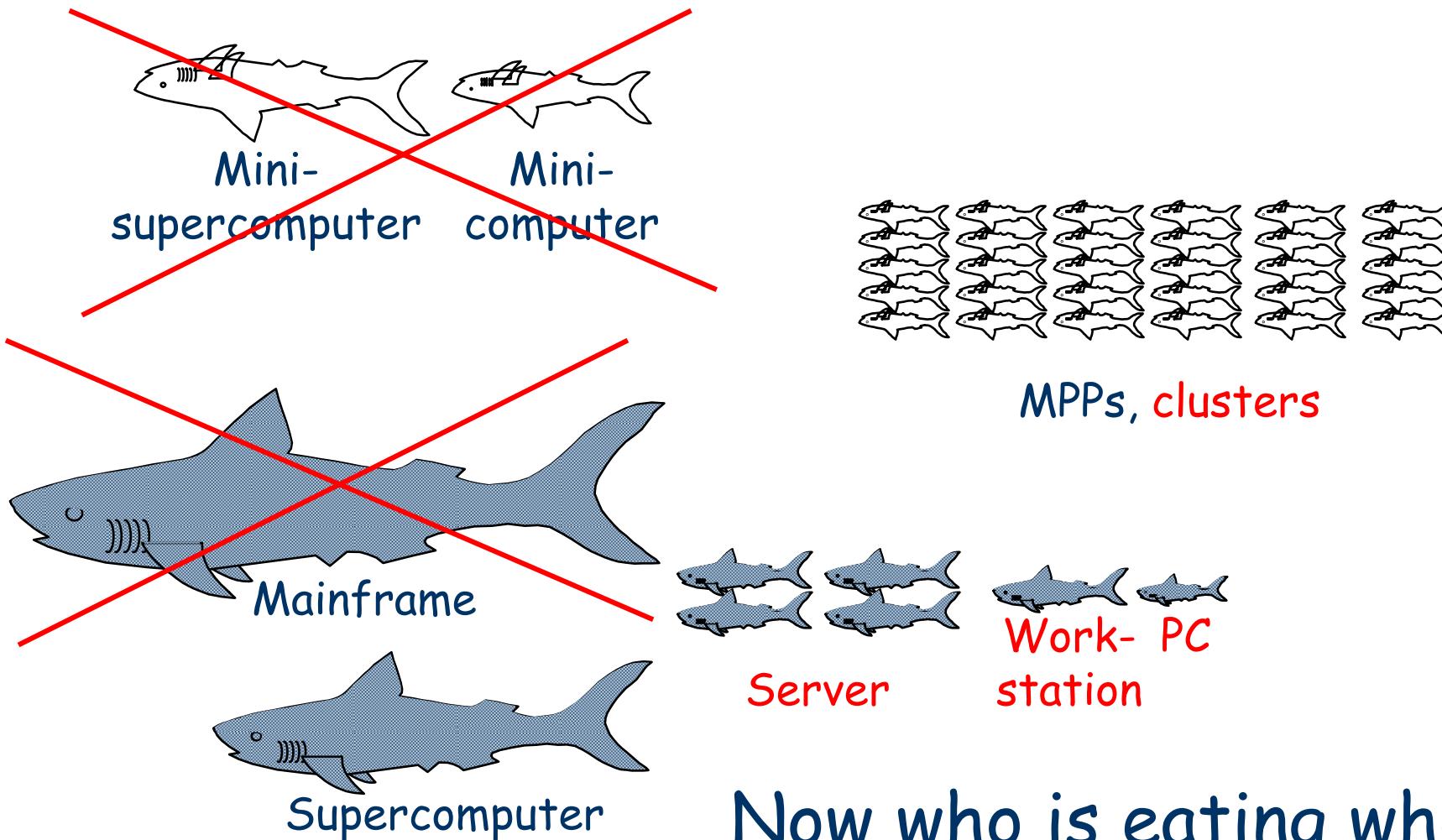


Big Fishes Eating Little Fishes

1988 Computer Food Chain



2008 Computer Food Chain



Komputery przyszłości

- Molecular Computing
 - <http://www.nytimes.com/2002/10/25/technology/25COMP.html?ex=1036558800&en=10e57e62ea60c115&ei=5062>
 - <http://pcquest.ciol.com/content/technology/10004402.asp>
 - <http://www.ddj.com/articles/2000/0013/0013b/0013b.htm>
- DNA Computing
 - <http://arstechnica.com/reviews/2q00/dna/dna-1.html>
 - <http://www.englib.cornell.edu/scitech/w96/DNA.html>
- Quantum Computing
 - <http://www.qtc.ecs.soton.ac.uk/lecture1/lecture1a.html>
 - http://www.mills.edu/ACAD_INFO/MCS/CS/S01MCS125/QuantumComputing/concepts.html
 - <http://www.cs.caltech.edu/~westside/quantum-intro.html>

Podsumowanie

- pierwszy komputer – ENIAC
- architektura von Neumanna – IAS
- prawo Moore'a
- generacje układów cyfrowych i komputerów
- szybkość procesorów rośnie w większym tempie niż szybkość pamięci – konsekwencje i rozwiązania
- ewolucja procesorów x86
- granice technologii
- komputery przyszłości

Organizacja i Architektura Komputerów

Wydajność komputerów

Metryki marketingowe

- MIPS
 - *Millions Instructions Per Second*
 - $(\text{liczba instrukcji} / 10^6) / \text{czas CPU}$
- MFLOPS
 - *Millions Float Point Operations Per Second*
 - $(\text{liczba operacji FP} / 10^6) / \text{czas CPU}$
- Zalety MIPS, MFLOPS i innych MOPS
 - Prosta i zrozumiała dla laików definicja, łatwość określenia
- Wady
 - Metryki MIPS i MFLOPS zależą od architektury ISA toteż porównanie różnych architektur jest niemożliwe
 - Dla różnych programów wykonywanych przez ten sam komputer otrzymuje się różne wartości metryk
 - W skrajnych przypadkach metryki MIPS i MFLOPS mogą dawać odwrotne wyniki niż metryki oparte na pomiarach CPU time

Przykłady metryki MIPS (x86)

Procesor	Data	Liczba tranzystorów	MIPS
4004	1971	2300	0.06
8080	1974	6000	0.1
8086	1978	29000	0.3
286	1982	134000	0.9
386	1985	275000	5
486	1989	1.2M	20
Pentium	1993	3.1M	100
Pentium 4 HT	2002	108M	>1000
Core 2 Extreme QX	2008		59 000
AMD Phenom II X6	2010		68 000
Core i7 EE i980 EE	2010		148 000

Czas w systemie komputerowym

- polecenie **time** w systemie UNIX

Przykład:

90.7u 12.9s 2:39 65%

user CPU time
system CPU time
elapsed time
user CPU/elapsed

= 90.7 s
= 12.9 s
= 2 min 39 s (159 s)
= 65 % = $(90.7 + 12.9)/159$

Wydajność komputera

- Wydajność - Computer Performance
 - czas oczekiwania na wynik
 - *Response time (latency)*
 - przepustowość systemu
 - *Throughput*
- Przedmiotem uwagi OAK jest szczególnie
 - czas CPU wykorzystany przez użytkownika
 - *User CPU time*
czas zużyty przez CPU na wykonanie programu

Wydajność cd.

- Metryki wydajności procesora

$$CPU \text{ execution time} = CPU \text{ clock cycles}/pgm \times \text{clock cycle time}$$
$$CPU \text{ clock cycles}/pgm = Instructions/pgm \times \text{avg. clock cycles per instruction}(CPI)$$

- CPI (*cycles per instruction*) – średnia liczba cykli procesora potrzebna do wykonania instrukcji wiąże się z:
 - architekturą listy instrukcji (ISA)
 - implementacją ISA

Wydajność cd.

- Definicja wydajności

$$\text{Performance}_x = \frac{1}{\text{Execution time}_x}$$

- Maszyna X jest n razy szybsza od maszyny Y jeśli

$$\frac{\text{Performance}_x}{\text{Performance}_y} = n$$

Wydajność cd.

- Aby określić wydajność komputera należy:
 - zmierzyć czas wykonania programu testowego (benchmark)
- Aby obliczyć czas wykonania należy znać:
 - czas trwania cyklu CPU (z danych katalogowych CPU)
 - łączną liczbę cykli CPU dla danego programu
- Aby oszacować liczbę cykli CPU w programie należy znać:
 - liczbę instrukcji w programie
 - przeciętną wartość CPI (average CPI)

Wydajność cd.

- Częstotliwość zegara CPU (*CPU clock rate*)
 - liczba cykli na sekundę (1 Hz = 1 cykl / s)
 - CPU cycle time = 1/CPU clock rate
- Przykład: Pentium4 2.4 GHz CPU

CPU cycle time

$$\begin{aligned} &= \frac{1}{2.4 \times 10^9} \\ &= 0.4167 \times 10^{-9} \text{ s} \\ &= 0.4167 \text{ ns} \end{aligned}$$

Wydajność cd.

- Liczba cykli zegara CPU w programie
 - liczba cykli = liczba instrukcji ?
 - błąd !
 - różne **instrukcje** wymagają różnej **liczby cykli** dla różnych komputerów

- Ważna uwaga:
 - zmiana czasu cyklu CPU (częstotliwości zegara) często powoduje zmianę liczby cykli potrzebnych do wykonania różnych instrukcji

Definicja CPI

- Średnia liczba cykli na instrukcję CPI (*average cycles per instruction*)

$$CPI = \frac{\sum_{i=1}^n CPI_i \times IC_i}{IC} = \sum_{i=1}^n CPI_i \times \left(\frac{IC_i}{IC} \right) = \sum_{i=1}^n CPI_i \times F_i$$

gdzie:

- CPI_i – liczba cykli potrzebnych do wykonania instrukcji typu i
- IC_i – liczba instrukcji typu i w programie
- IC – ogólna liczba instrukcji w programie
- F_i - względna częstość występowania instrukcji typu i w programie

Przykład wykorzystania CPI

- Założmy, że typowy program wykonywany przez CPU analizowanego systemu komputerowego ma następującą charakterystykę:
 - częstość występowania operacji FP = 25%
 - CPI operacji FP = 4.0
 - CPI innych instrukcji = 1.33
 - częstość występowania operacji FPSQR = 2%
 - CPI operacji FPSQR = 20

Przykład wykorzystania CPI cd.

- Konstruktorzy chcą zwiększyć wydajność systemu przez przeprojektowanie CPU. Mają dwa alternatywne rozwiązania:
 1. Zmniejszyć CPI operacji FPSQR z 20 do 2
 2. Zmniejszyć CPI wszystkich operacji FP do 2

Które z tych rozwiązań jest lepsze w sensie metryki (CPU time) ?

Przykład wykorzystania CPI cd.

Rozwiązanie:

Należy zauważyć, że zmienia się tylko CPI, natomiast liczba instrukcji częstotliwość zegara pozostaje bez zmiany.

Obliczamy CPI_{org} przed dokonaniem zmian w procesorze:

$$CPI_{org} = (4 \times 25\%) + (1.33 \times 75\%) = 2.0$$

Obliczamy $CPI_{new\,FPSQR}$ dla procesora z ulepszonym układem obliczania pierwiastka odejmując od CPI_{org} zysk spowodowany ulepszeniem:

$$\begin{aligned} CPI_{newFPSQR} &= CPI_{org} - 2\% \times (CPI_{oldFPSQR} - CPI_{of\,new\,FPSQR}) \\ &= 2.0 - 2\% \times (20 - 2) = 1.64 \end{aligned}$$

Przykład wykorzystania CPI cd.

Obliczamy CPI_{new FP} dla procesora z ulepszonym całym blokiem FPU

$$\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.0) = 1.5$$

Odpowiedź: ulepszenie całego bloku FP jest lepszym rozwiązaniem niż ulepszenie samego układu obliczania FPSQR, ponieważ

$$\text{CPI}_{\text{new FP}} = 1.5 < \text{CPI}_{\text{new FPSQR}} = 1.64$$

Przykład wykorzystania CPI dokończenie

Obliczmy wzrost wydajności systemu po ulepszeniu układow FPU

$$\begin{aligned} Speedup_{new FP} &= \frac{CPU\ time_{org}}{CPU\ time_{new FP}} = \frac{IC \times Clock\ cycle \times CPI_{org}}{IC \times Clock\ cycle \times CPI_{new FP}} = \\ &= \frac{CPI_{org}}{CPI_{new FP}} = 2.0 / 1.5 = 1.33 \end{aligned}$$

Problem

- Jeśli dwa komputery mają taką samą architekturę ISA, to które z wymienionych wielkości będą w obu przypadkach identyczne?
 - częstotliwość zegara
 - CPI
 - czas wykonania programu (CPU time)
 - liczba instrukcji (IC)
 - MIPS

Odpowiedź: CPI i liczba instrukcji (IC)

Wydajność - współzależności

	Liczba instrukcji	CPI	Cykl zegara
Program	X		
Kompilator	X	X	
Lista instrukcji	X	X	
Organizacja		X	X
Technologia			X

Prawo Amdahla

Określa współczynnik poprawy wydajności *Speed up* w rezultacie ulepszenia systemu komputerowego

$$CPUtime_{new} = CPUtime_{old} \times \left[(1 - Frac_{enh}) + \frac{Frac_{enh}}{Speedup_{enh}} \right]$$

$$Speedup_{overall} = \frac{CPUtime_{old}}{CPUtime_{new}} = \frac{1}{(1 - Frac_{enh}) + \frac{Frac_{enh}}{Speedup_{enh}}}$$

Gdzie:

Frac_{enh} – współczynnik wykorzystania ulepszonej części systemu (w sensie czasu), np. $Frac_{enh} = 0.3$ oznacza, że ulepszona część jest wykorzystywana przez 30% czasu wykonania programu

Speedup_{enh} – współczynnik poprawy wydajności ulepszonej części systemu

Prawo Amdahla – przykład

- Dla wcześniejszego analizowanego przykładu dotyczącego ulepszenia bloków realizujących obliczenia FP i FPSQR otrzymujemy następujące wyniki:

$$Speedup_{FPSQR} = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$Speedup_{FP} = \frac{1}{(1 - 0.5) + \frac{0.5}{2.0}} = \frac{1}{0.75} = 1.33$$

ten sam wynik co uzyskany wcześniej przy użyciu analizy CPI

Jeszcze jeden przykład

- Założmy, że dostęp do pamięci cache jest 10 razy szybszy niż dostęp do pamięci operacyjnej. Założmy ponadto, że pamięć cache jest wykorzystywana przez 90% czasu. Obliczyć współczynnik wzrostu wydajności systemu spowodowany dodaniem pamięci cache.
 - Proste zastosowanie prawa Amdahla

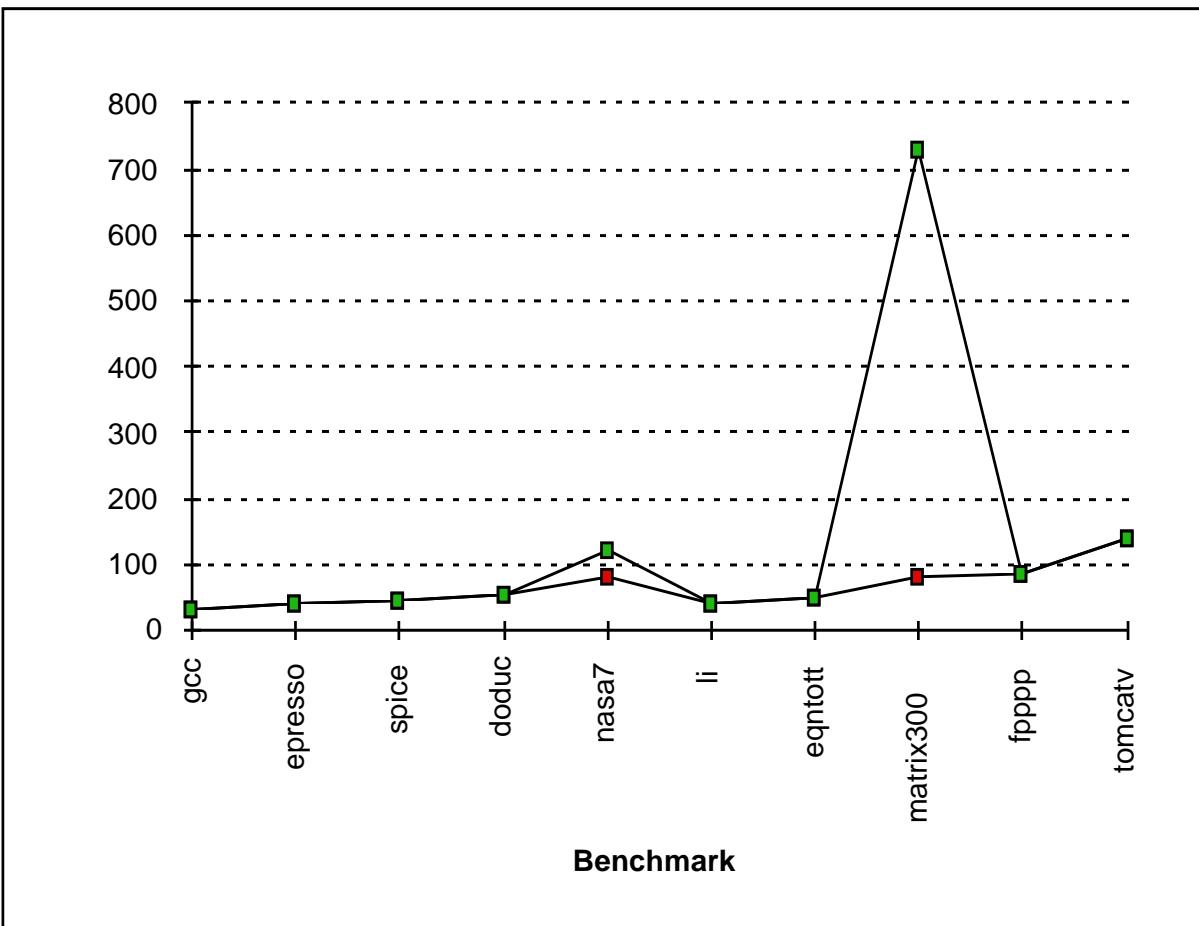
$$\begin{aligned} Speedup &= \frac{1}{(1 - \% \text{ czasu pracy cache}) + \frac{\% \text{ czasu pracy cache}}{Speedup z zastosowania cache}} \\ &= \frac{1}{(1 - 0.9) + \frac{0.9}{10}} = \frac{1}{0.19} = 5.3 \end{aligned}$$

Pomiary wydajności

- W profesjonalnych badaniach wydajności komputerów wykorzystuje się pomiar czasu wykonania rzeczywistych programów
 - najlepiej jest użyć typowych aplikacji, które będą wykonywane przez system podczas normalnej eksploatacji
 - lub wykorzystać klasy aplikacji, np. kompilatory, edytory, programy obliczeń numerycznych, programy graficzne itp.
- Krótkie programy testowe (benchmarks)
 - wygodne dla architektów i projektantów
 - łatwe w standaryzacji testów
- SPEC (System Performance Evaluation Cooperative)
 - <http://www.spec.org>
 - producenci sprzętu uzgodnili zestawy programów testowych (benchmarków)
 - wartościowe wyniki do oceny wydajności komputerów i jakości kompilatorów

SPEC benchmarks

- SPEC89 – 1989



- 10 krótkich testów
- wynik w postaci jednej liczby, średniej ważonej

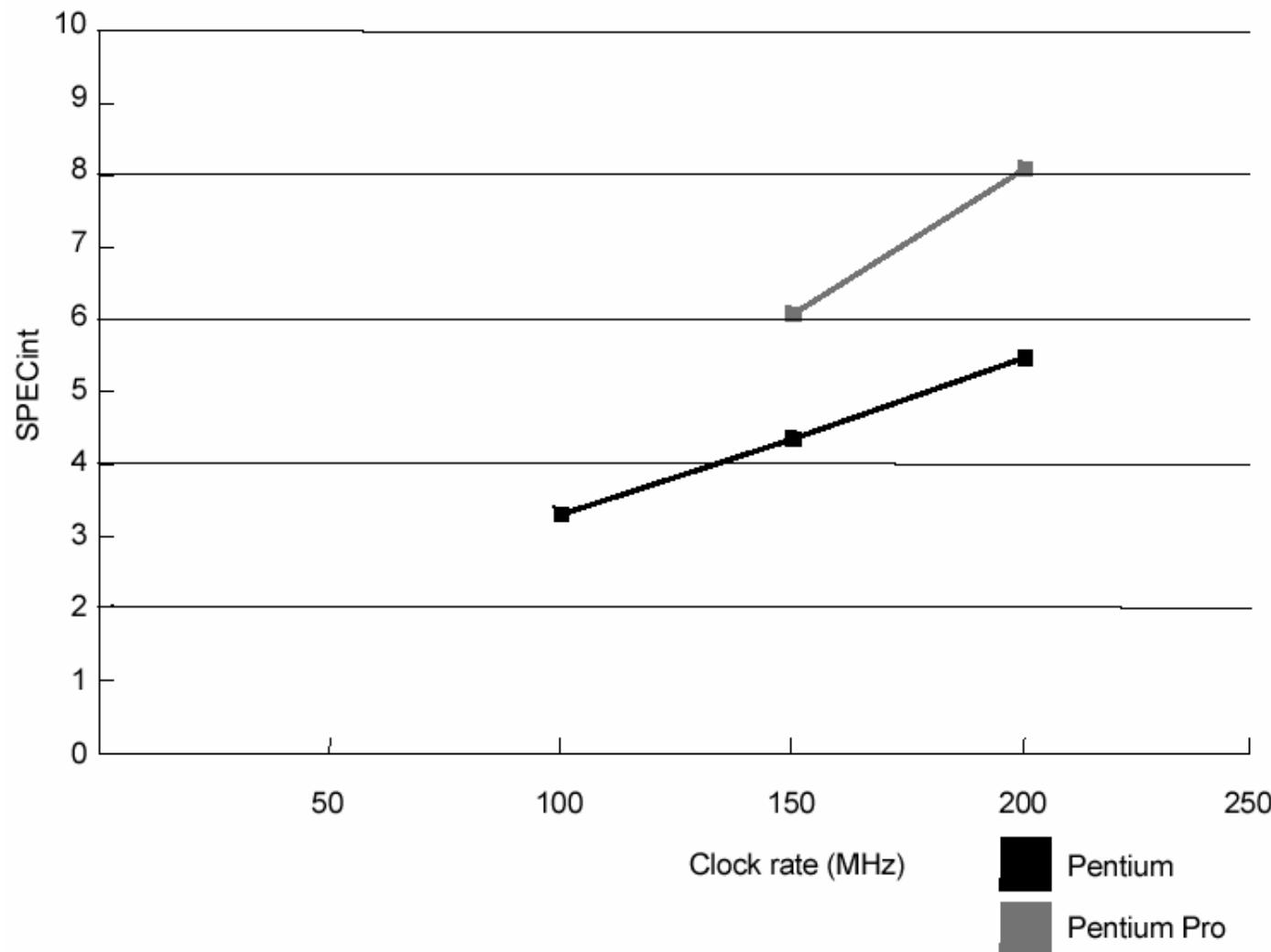
SPEC benchmarks cd.

- Druga runda – 1992
 - SPECint92 – zestaw 6 programów testowych operujących na liczbach całkowitych (integer)
 - SPECfp92 – zestaw 14 programów testowych operujących na liczbach zmiennoprzecinkowych
 - SPECbase
- Trzecia runda – 1995
 - SPECint95 – zestaw 8 programów (integer)
 - SPECfp95 – zestaw 10 programów (FP)
 - SPECint_base95 i SPECfp_base95

SPEC'95 – zestaw testów

Benchmark	Description
go	Artificial intelligence; plays the game of Go
m88ksim	Motorola 88k chip simulator; runs test program
gcc	The Gnu C compiler generating SPARC code
compress	Compresses and decompresses file in memory
li	Lisp interpreter
ijpeg	Graphic compression and decompression
perl	Manipulates strings and prime numbers in the special-purpose programming language Perl
vortex	A database program
tomcatv	A mesh generation program
swim	Shallow water model with 513 x 513 grid
su2cor	quantum physics; Monte Carlo simulation
hydro2d	Astrophysics; Hydrodynamic Naiver Stokes equations
mgrid	Multigrid solver in 3-D potential field
applu	Parabolic/elliptic partial differential equations
trub3d	Simulates isotropic, homogeneous turbulence in a cube
apsi	Solves problems regarding temperature, wind velocity, and distribution of pollutant
fpppp	Quantum chemistry
wave5	Plasma physics; electromagnetic particle simulation

SPECint95 - przykład



SPEC'2000

Table 2. CPU2000 integer and floating-point benchmark suite.

Benchmark	Language	KLOC	Resident size (Mbytes)	Virtual size (Mbytes)	Description
SPECint2000					
164.gzip	C	7.6	181	200	Compression
175.vpr	C	13.6	50	55.2	FPGA circuit placement and routing
176.gcc	C	193.0	155	158	C programming language compiler
181.mcf	C	1.9	190	192	Combinatorial optimization
186.crafty	C	20.7	2.1	4.2	Game playing: Chess
197.parser	C	10.3	37	62.5	Word processing
252.eon	C++	34.2	0.7	3.3	Computer visualization
253.perlbmk	C	79.2	146	159	Perl programming language
254.gap	C	62.5	193	196	Group theory, interpreter
255.vortex	C	54.3	72	81	Object-oriented database
256.bzip2	C	3.9	185	200	Compression
300.twolf	C	19.2	1.9	4.1	Place and route simulator
SPECfp2000					
168.wupwise	F77	1.8	176	177	Physics: Quantum chromodynamics
171.swim	F77	0.4	191	192	Shallow water modeling
172.mgrid	F77	0.5	56	56.7	Multigrid solver: 3D potential field
173.applu	F77	7.9	181	191	Partial differential equations
177.mesa	C	81.8	9.5	24.7	3D graphics library
178.galgel	F90	14.1	63	155	Computational fluid dynamics
179.art	C	1.2	3.7	5.9	Image recognition/neural networks
183.equake	C	1.2	49	51.1	Seismic wave propagation simulation
187.facerec	F90	2.4	16	18.5	Image processing: Face recognition
188.ammp	C	12.9	26	30	Computational chemistry
189.lucas	F90	2.8	142	143	Number theory/primality testing
191.fma3d	F90	59.8	103	105	Finite-element crash simulation
200.sixtrack	F77	47.1	26	59.8	Nuclear physics accelerator design
301.apsi	F77	6.4	191	192	Meteorology: Pollutant distribution

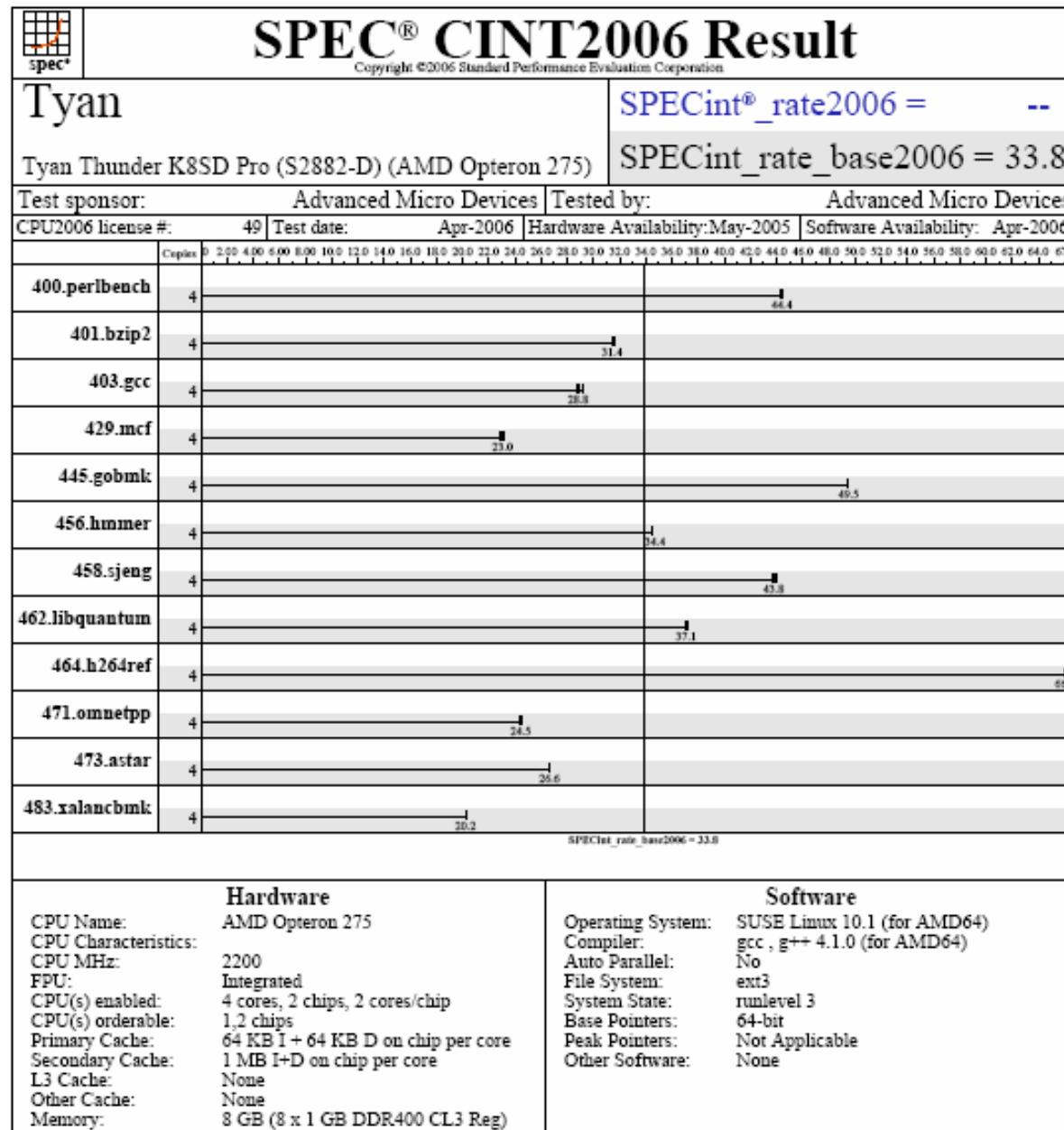
SPEC_2006 zestaw testów

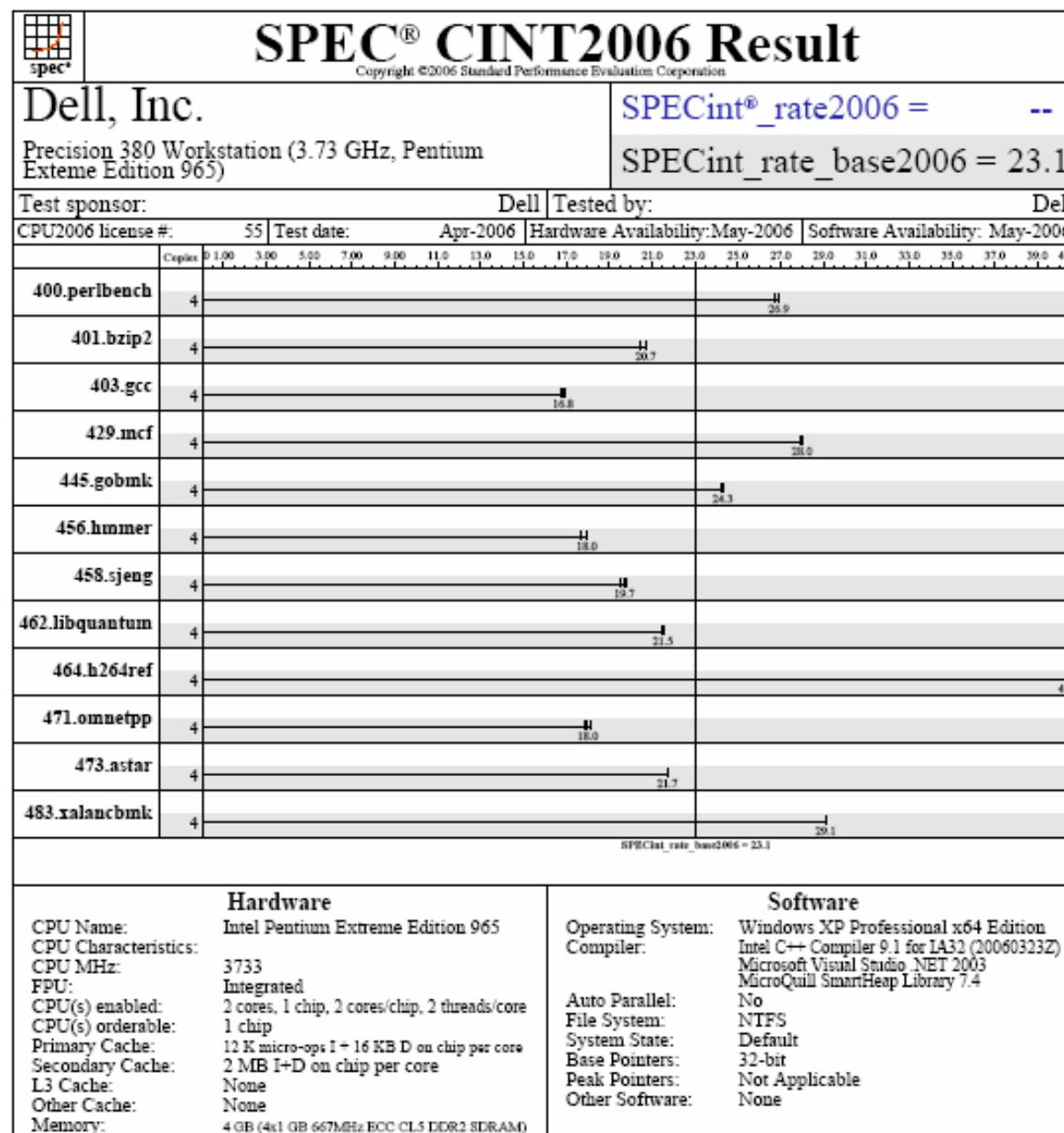
CINT 2006

400.perlbench	C	PERL Programming Language
401.bzip2	C	Compression
403.gcc	C	C Compiler
429.mcf	C	Combinatorial Optimization
445.gobmk	C	Artificial Intelligence: go
456.hmmr	C	Search Gene Sequence
458.sjeng	C	Artificial Intelligence: chess
462.libquantum	C	Physics: Quantum Computing
464.h264ref	C	Video Compression
471.omnetpp	C++	Discrete Event Simulation
473.astar	C++	Path-finding Algorithms
483.xalancbmk	C++	XML Processing

CFP 2006

410.bwaves	Fortran	Fluid Dynamics
416.gamess	Fortran	Quantum Chemistry
433.milc	C	Physics: Quantum Chromodynamics
434.zeusmp	Fortran	Physics/CFD
435.gromacs	C/Fortran	Biochemistry/Molecular Dynamics
436.cactusADM	C/Fortran	Physics/General Relativity
437.leslie3d	Fortran	Fluid Dynamics
444.namd	C++	Biology/Molecular Dynamics
447.dealII	C++	Finite Element Analysis
450.soplex	C++	Linear Programming, Optimization
453.povray	C++	Image Ray-tracing
454.calculix	C/Fortran	Structural Mechanics
459.GemsFDTD	Fortran	Computational Electromagnetics
465.tonto	Fortran	Quantum Chemistry
470.lbm	C	Fluid Dynamics
481.wrf	C/Fortran	Weather Prediction
482.sphinx3	C	Speech recognition





Raporty CFP - przykłady

		SPEC® CFP2006 Result																							
		Copyright ©2006 Standard Performance Evaluation Corporation																							
Intel Corporation		SPECfp®_rate2006 = Not Run																							
Intel DG965WH motherboard (2.93 GHz, Intel Core 2 Extreme processor X6800)		SPECfp_rate_base2006 = 26.8																							
Test sponsor:		Intel Corporation				Tested by:		Intel Corporation																	
CPU2006 license #:		13	Test date:		Aug-2006	Hardware Availability:		Jul-2006	Software Availability:		Aug-2006														
		Copies	0	1.00	3.00	5.00	7.00	9.00	11.0	13.0	15.0	17.0	19.0	21.0	23.0	25.0	27.0	29.0	31.0	33.0	35.0	37.0	39.0	41.0	

		SPEC® CFP2006 Result																																		
		Copyright ©2006 Standard Performance Evaluation Corporation																																		
ASUS Computer International		SPECfp_rate_base2006 = 18.8																																		
Asus M2N32-SLI Deluxe (AMD Athlon 64 FX-62)		SPECfp®_rate2006 = Not Run																																		
Test sponsor:		Intel Corporation				Tested by:		Intel Corporation																												
CPU2006 license #:		13	Test date:		Aug-2006	Hardware Availability:		Jul-2006	Software Availability:		Aug-2006																									
		Copies	0	1.00	2.00	3.00	4.00	5.00	6.00	7.00	8.00	9.00	10.0	11.0	12.0	13.0	14.0	15.0	16.0	17.0	18.0	19.0	20.0	21.0	22.0	23.0	24.0	25.0	26.0	27.0	28.0	29.0	30.0	31.0	32.0	34.0

SYSmark 2007 Preview

Adobe® After Effects® 7

Adobe® Illustrator® CS2

Adobe® Photoshop® CS2

AutoDesk® 3ds Max® 8

Macromedia® Flash 8

Microsoft® Excel 2003

Microsoft® Outlook 2003

Microsoft® PowerPoint 2003

Microsoft® Word 2003

Microsoft® Project 2003

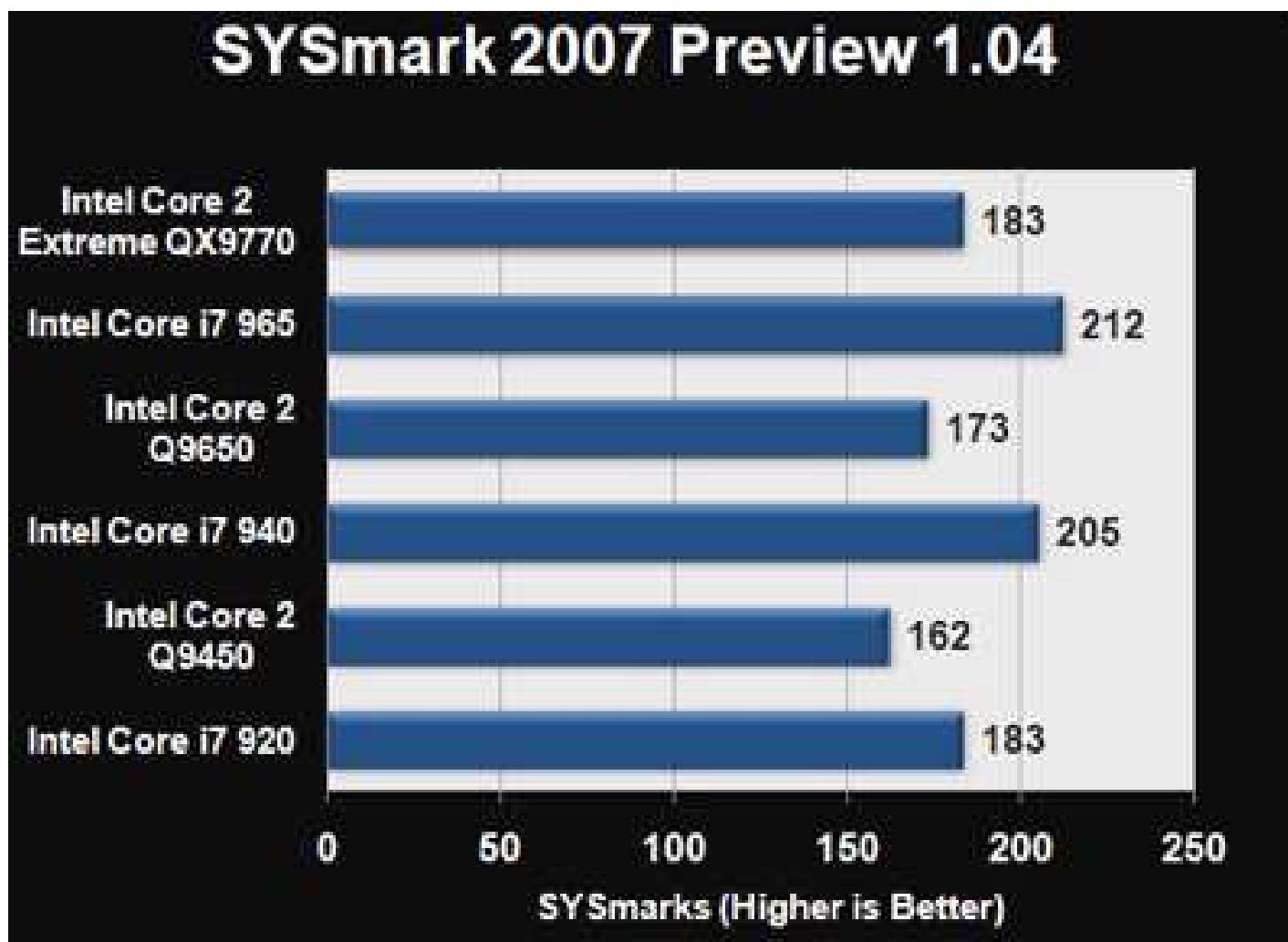
Microsoft® Windows Media™ Encoder 9 series

Sony® Vegas 7

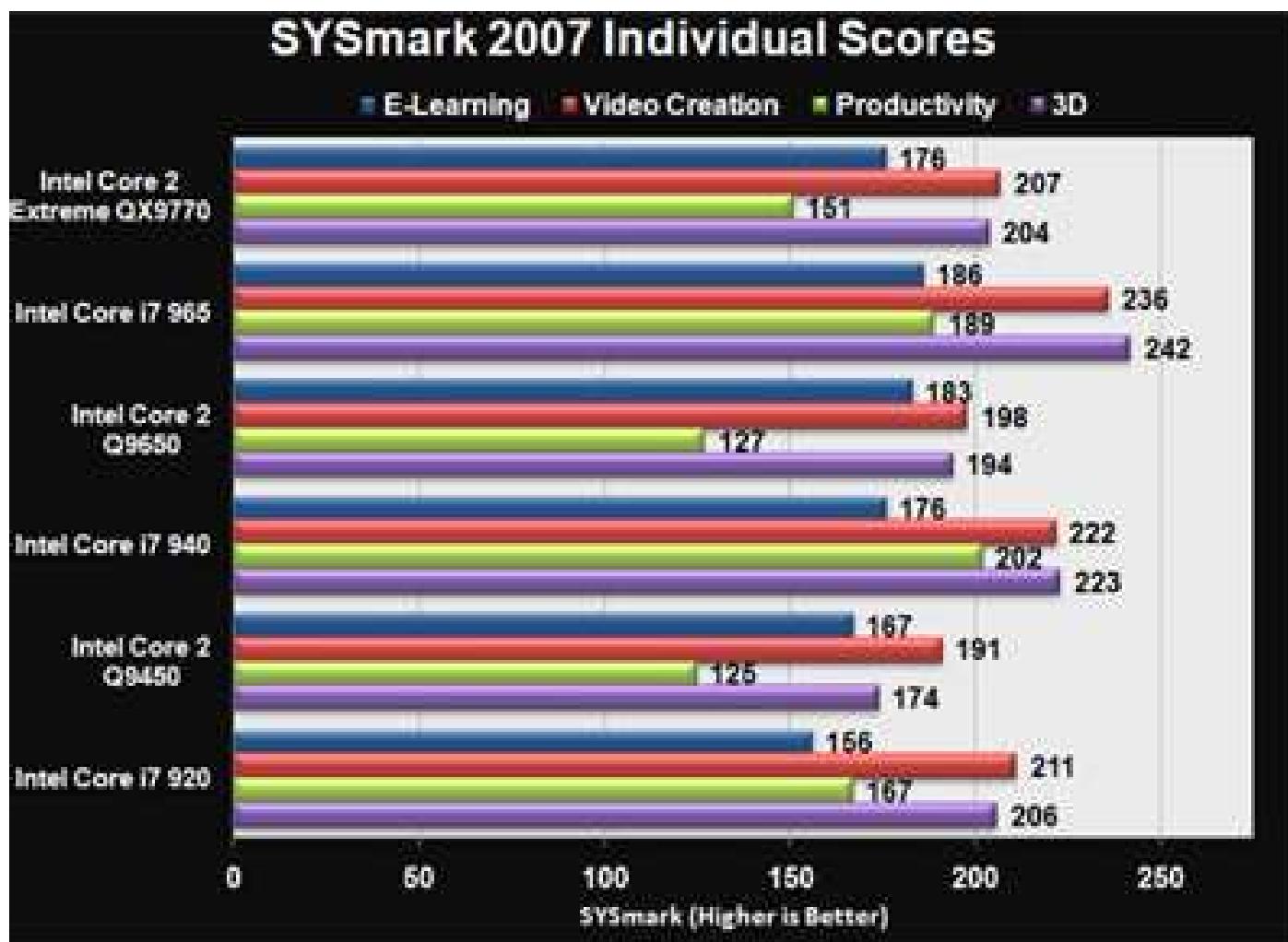
SketchUp 5

WinZip® 10.0

SYSmark 2007



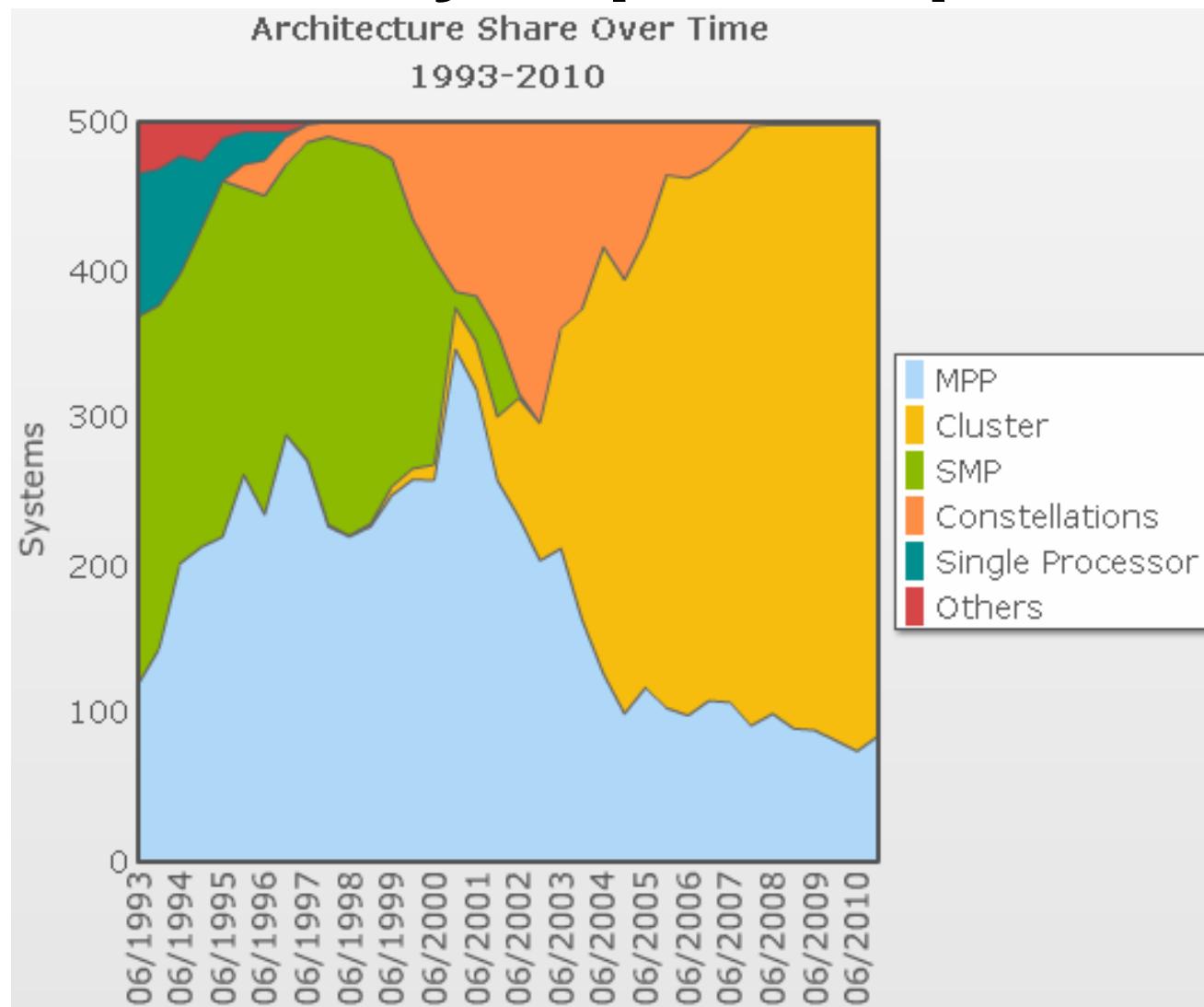
SYSmark 2007 cd.



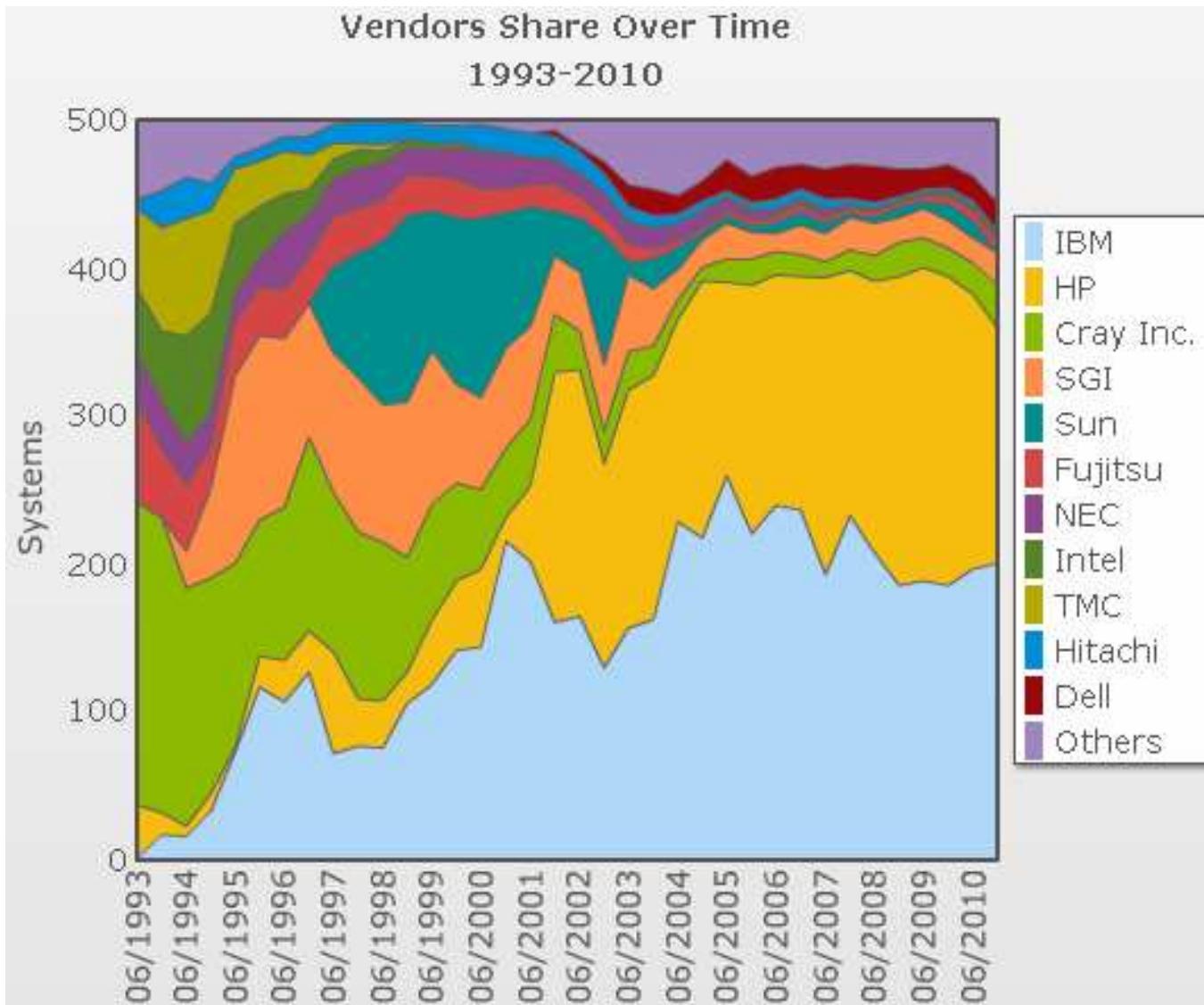
Benchmarki dla amatorów

- Karty graficzne
 - 3DMark SE
- Dyski
 - HDTach
 - WinBench
- Procesory
 - SPECcpu2006
 - SYSmark 2007
- Pamięci
 - ScienceMark
- Kompletny system komputerowy
 - PCMark

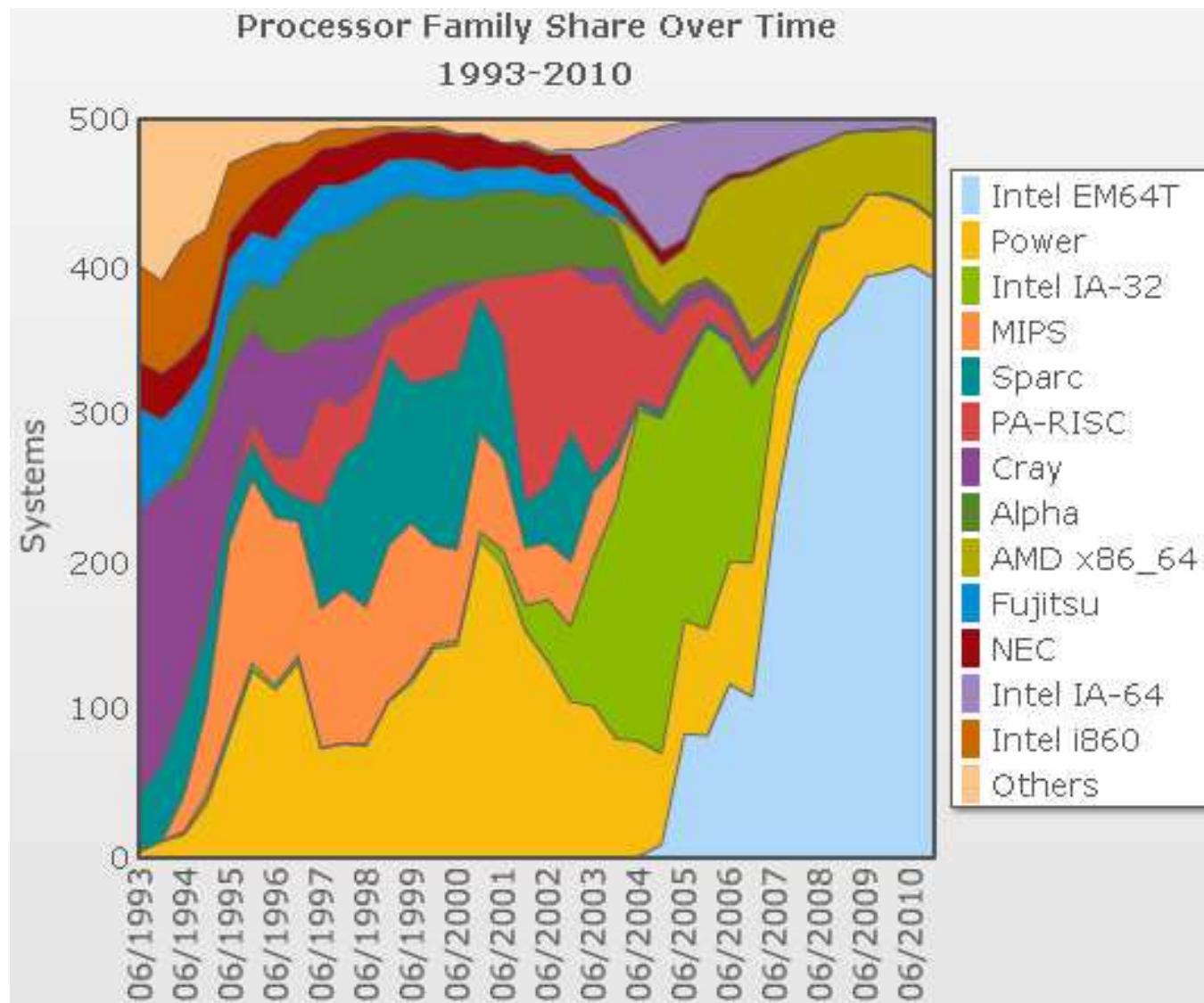
Architektury superkomputerów



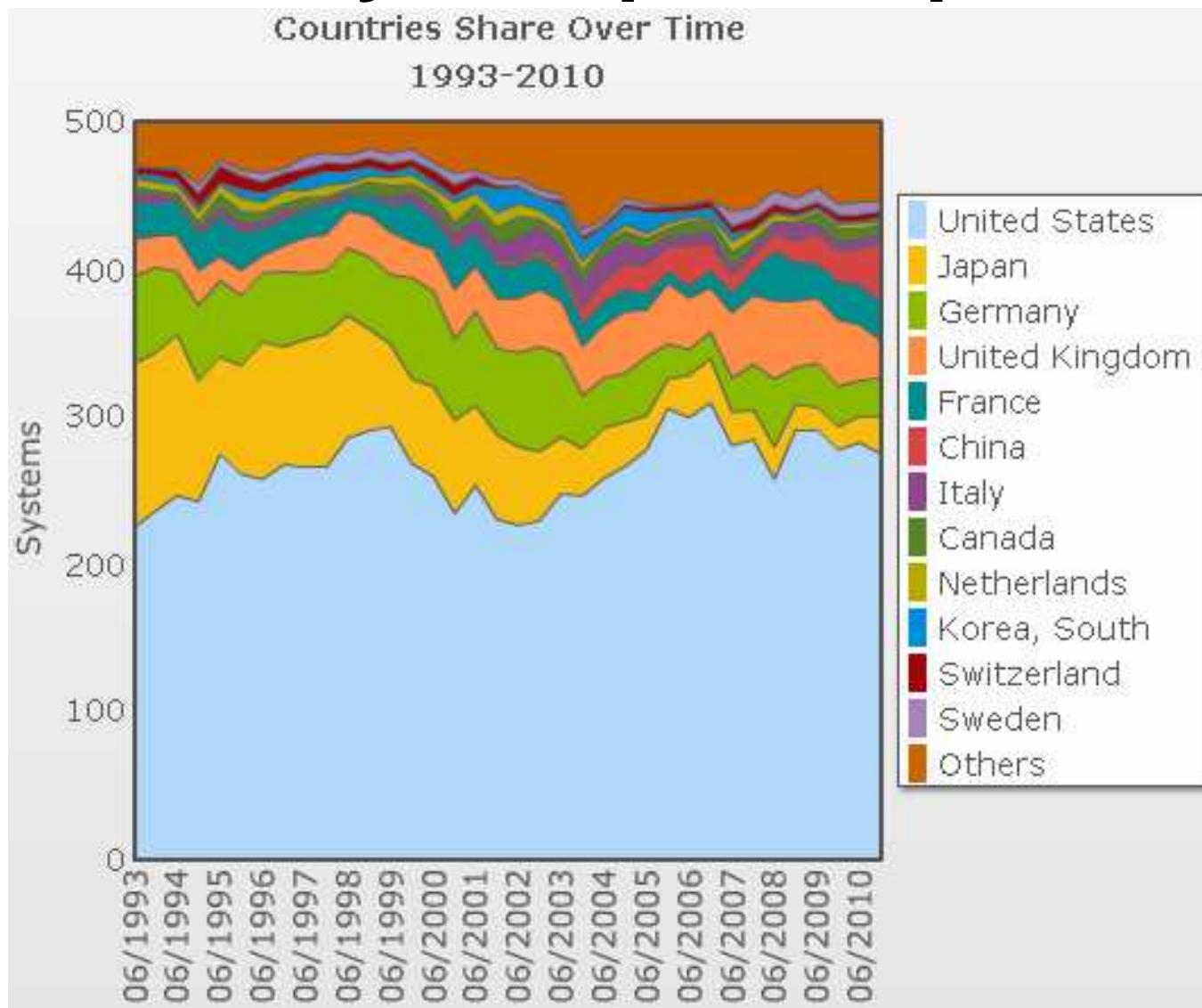
Producenci superkomputerów



Procesory w superkomputerach



Procesory w superkomputerach

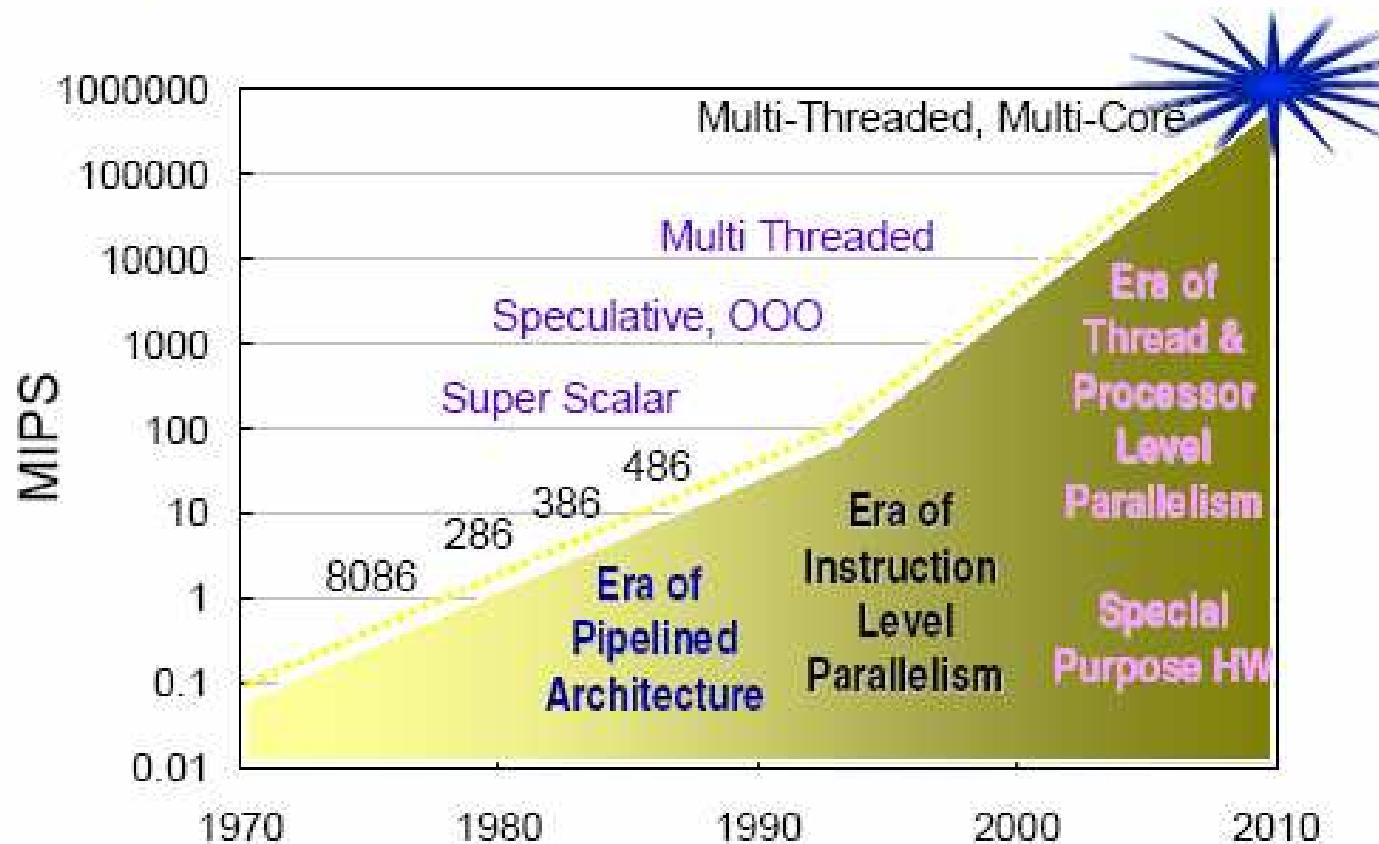


Moc obliczeniowa - przewidywania



Wydajność i architektura

The Exponential Reward



Podsumowanie

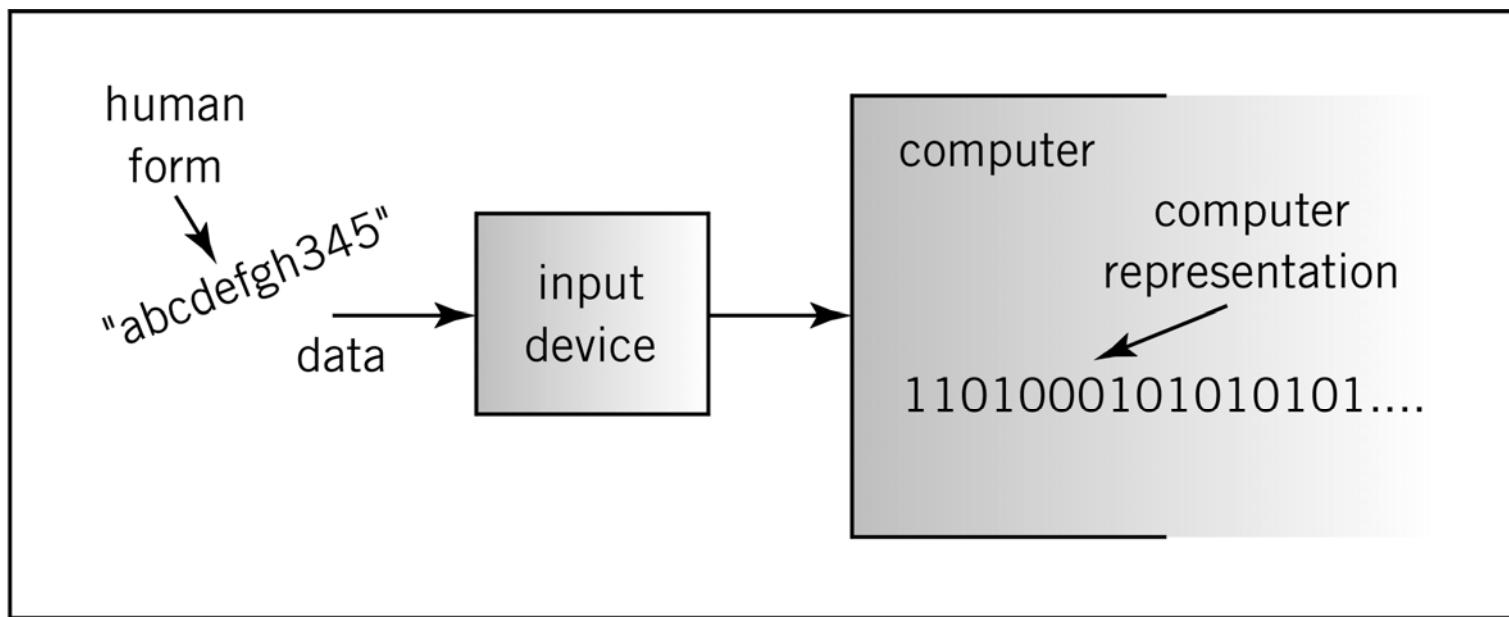
- Metryki wydajności komputera
 - CPU time, CPI
 - metryki komercyjne MIPS, MFLOPS
- Prawo Amdahla – wpływ ulepszeń systemu na wydajność
- Benchmarki
 - SPEC'89
 - SPEC'92
 - SPEC'95
 - SPEC'2000 i 2006
 - SYSmark 2007

Organizacja i Architektura Komputerów

Arytmetyka komputerów

Alfabet: 0, 1

- Dane w komputerach są reprezentowane wyłącznie przy użyciu alfabetu dwójkowego (binarnego)



Englander: The Architecture of Computer
Hardware and Systems Software, 2nd edition
Chapter 3, Figure 03-01

Bity, bajty, słowa

- Bity są grupowane w większe zespoły o długości będącej potągią liczby 2:
 - tetrada: 4 bity (*nibble*)
 - bajt: 8 bitów
 - słowo: 16, 32, 64 lub 128 bitów
 - podwójne słowo (*doubleword*)
 - półsłowo (*halfword*)
- Długość słowa zależy od **organizacji** komputera

Kody liczbowe

baza (radix)

2 10100111100

dwójkowy (binarny)

3 1211122

4 110330

5 20330

6 10112

7 3623

8 2474

Podstawy używane w systemach cyfrowych

oktalny

9 1748

dziesiętny

10 1340

11 1009

12 938

13 7C1

14 6BA

15 5E5

16 53C

szesnastkowy, heksadecymalny,
'hex'

Kod szesnastkowy HEX

- Powszechnie używany przez programistów programujących w języku asemblera
- Skraca długość notacji liczby
- Łatwa konwersja na kod NKB i odwrotnie
- Każda tetrada reprezentuje cyfrę szesnastkową

	Cyfra HEX	kod binarny
	0	0000
	1	0001
	2	0010
	3	0011
	4	0100
	5	0101
	6	0110
	7	0111
	8	1000
	9	1001
	A	1010
	B	1011
	C	1100
	D	1101
	E	1110
	F	1111

Konwersja HEX – radix 10

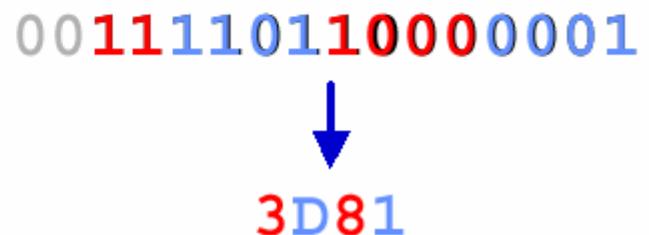
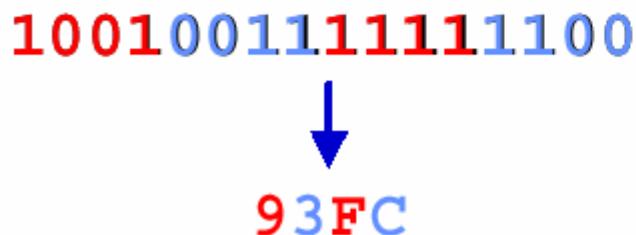
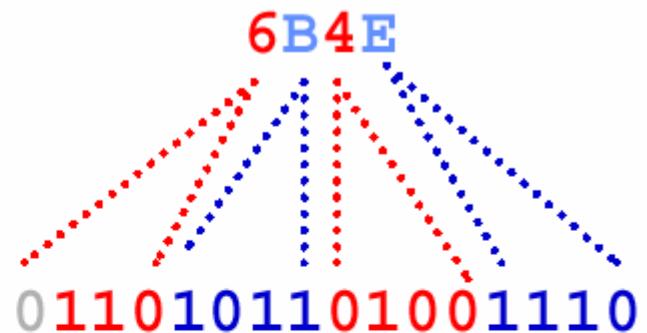
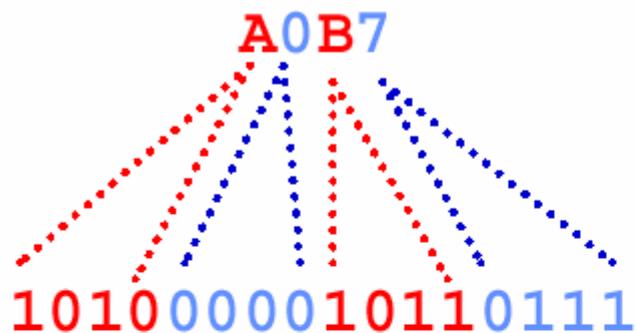
$$\begin{array}{rcl} \textcolor{red}{9647} & = & 9 \cdot 16^3 + 6 \cdot 16^2 + 4 \cdot 16^1 + 7 \cdot 16^0 \\ & = & 9 \cdot 4096 + 6 \cdot 256 + 4 \cdot 16 + 7 \cdot 1 \\ & & = \textcolor{blue}{38471} \end{array}$$

$$\begin{array}{rcl} \textcolor{red}{A3} & = & (10) \cdot 16^1 + 3 \cdot 16^0 \\ & = & 160 + 3 \\ & & = \textcolor{blue}{163} \end{array}$$

$$\begin{array}{rcl} \textcolor{red}{B2D} & = & (11) \cdot 16^2 + 2 \cdot 16^1 + (13) \cdot 16^0 \\ & = & 2816 + 32 + 13 \\ & & = \textcolor{blue}{2861} \end{array}$$

$$\begin{array}{rcl} \textcolor{red}{FFFF} & = & (15) \cdot 16^3 + (15) \cdot 16^2 + (15) \cdot 16^1 + (15) \cdot 16^0 \\ & = & 61440 + 3840 + 240 + 15 \\ & & = \textcolor{blue}{65535} \end{array}$$

Konwersja binarna – HEX



Kod BCD

- BCD (*binary coded decimal*)
 - każda cyfra dziesiętna jest reprezentowana jako 4 bity
 - kod opracowany dla wczesnych kalkulatorów
 - przykład: $359_{10} = 0011\ 0101\ 1001_{bcd}$
 - kod łatwy do zrozumienia przez człowieka, niewygodny dla komputerów

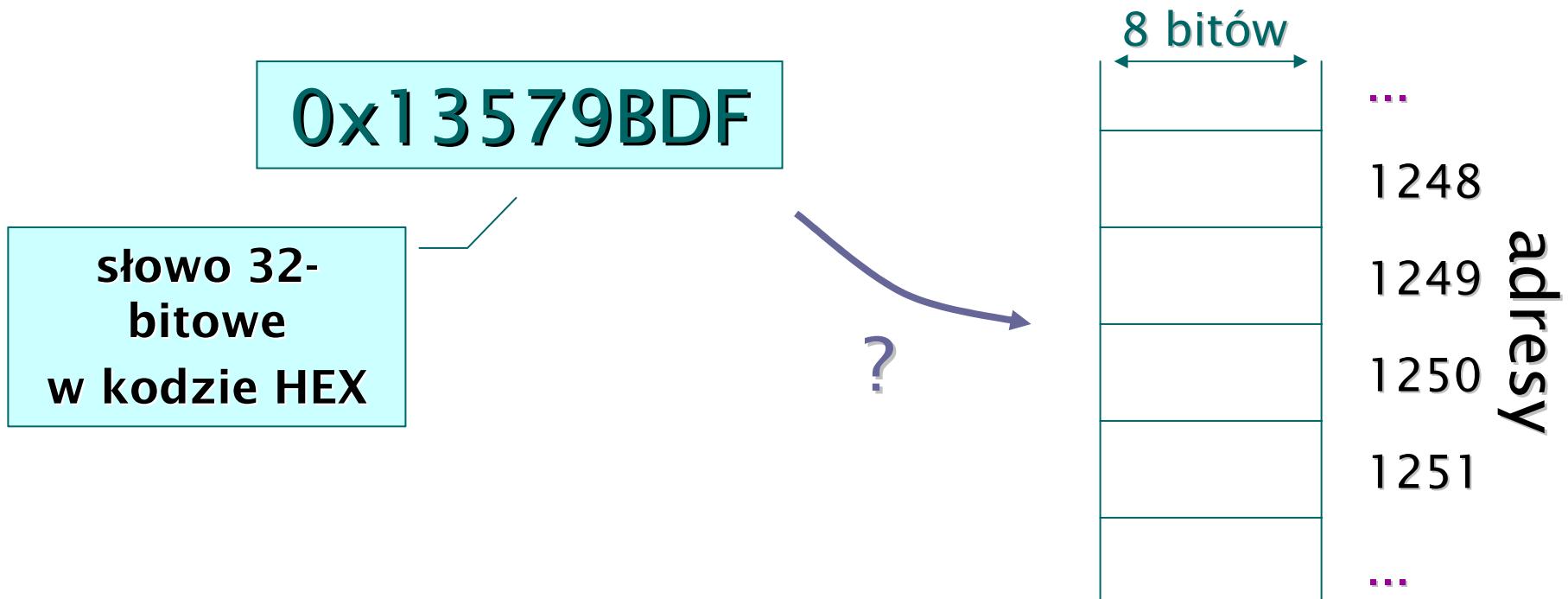
cyfra dziesiętna	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Kody od 1010 do 1111 nie są używane

Porządek bajtów w pamięci

- Problem

- Pamięć jest zwykle adresowana bajtami
- Jak zapisać w pamięci słowo 32-bitowe ?



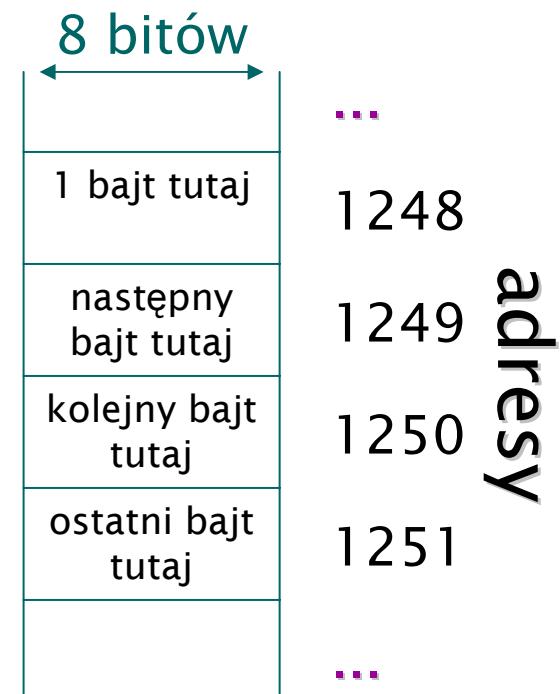
Kolejność bajtów w pamięci

- Rozwiążanie

- podziel słowo na bajty
- zapisz kolejne bajty w kolejnych komórkach

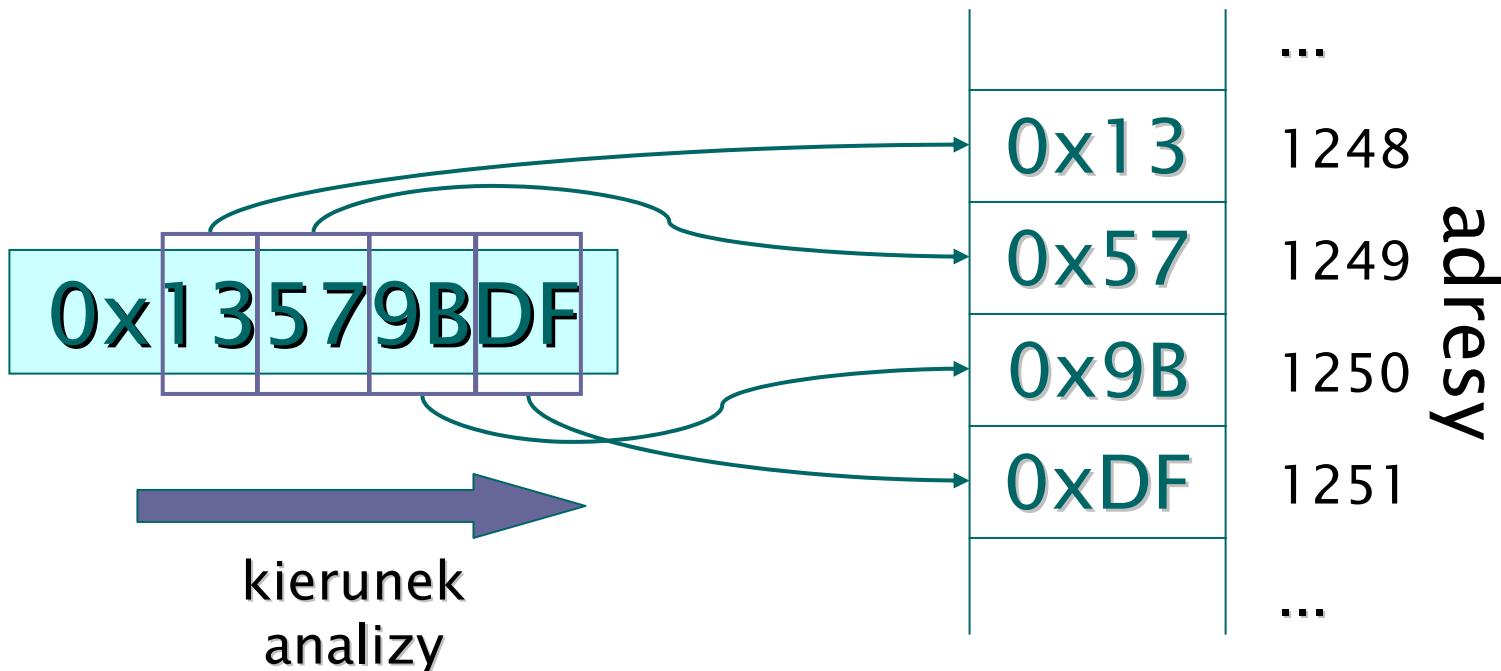


każda cyfra
HEX
reprezentuje
tertadę, zatem
dwie cyfry
tworzą bajt



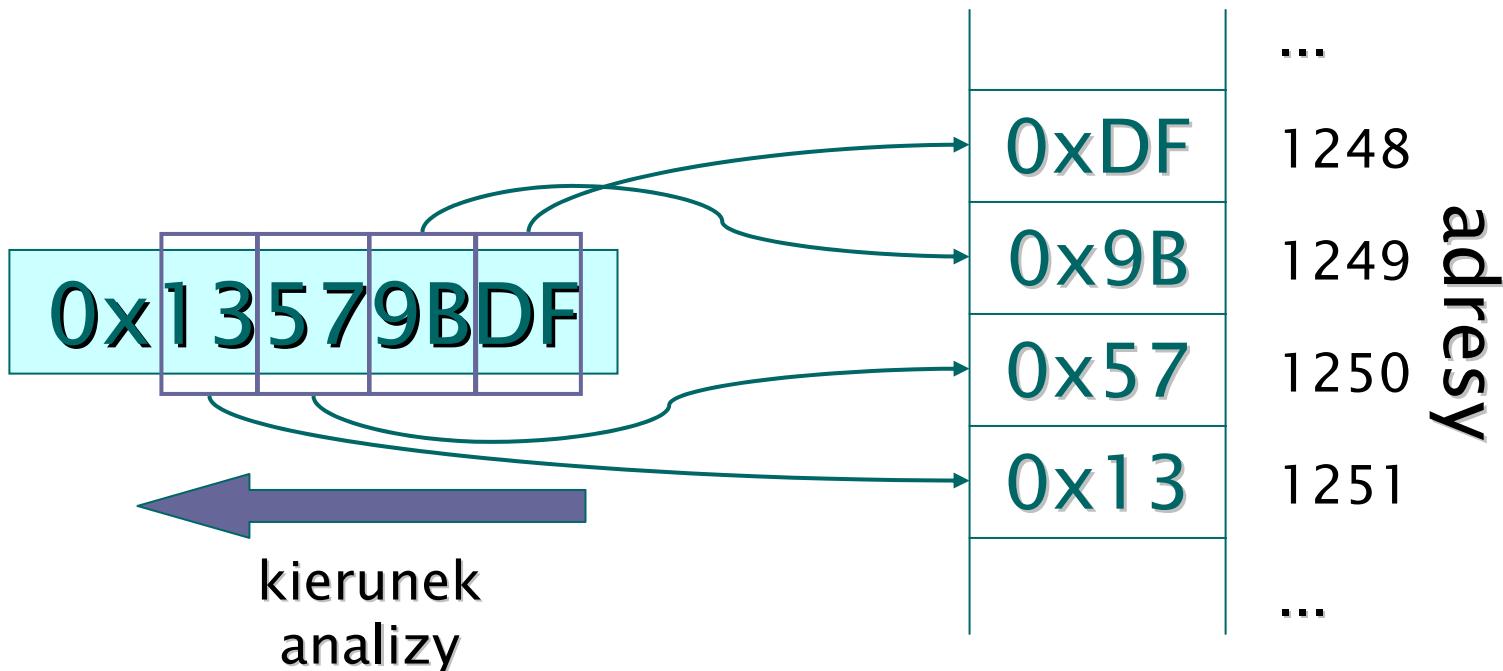
Kolejność bajtów cd.

- W jakiej kolejności zapisywać bajty?
 - Zaczynając od najbardziej znaczącego (“big” end)?



Kolejność bajtów cd.

- W jakiej kolejności zapisywać bajty?
 - Zaczynając od najmniej znaczącego (“little” end)?



Kolejność bajtów cd.

- Stosuje się dwa sposoby zapisu słów w pamięci
- BigEndian
 - most significant byte in lowest address
 - store least significant byte in highest address
- LittleEndian
 - store least significant byte in lowest address
 - store most significant byte in highest address

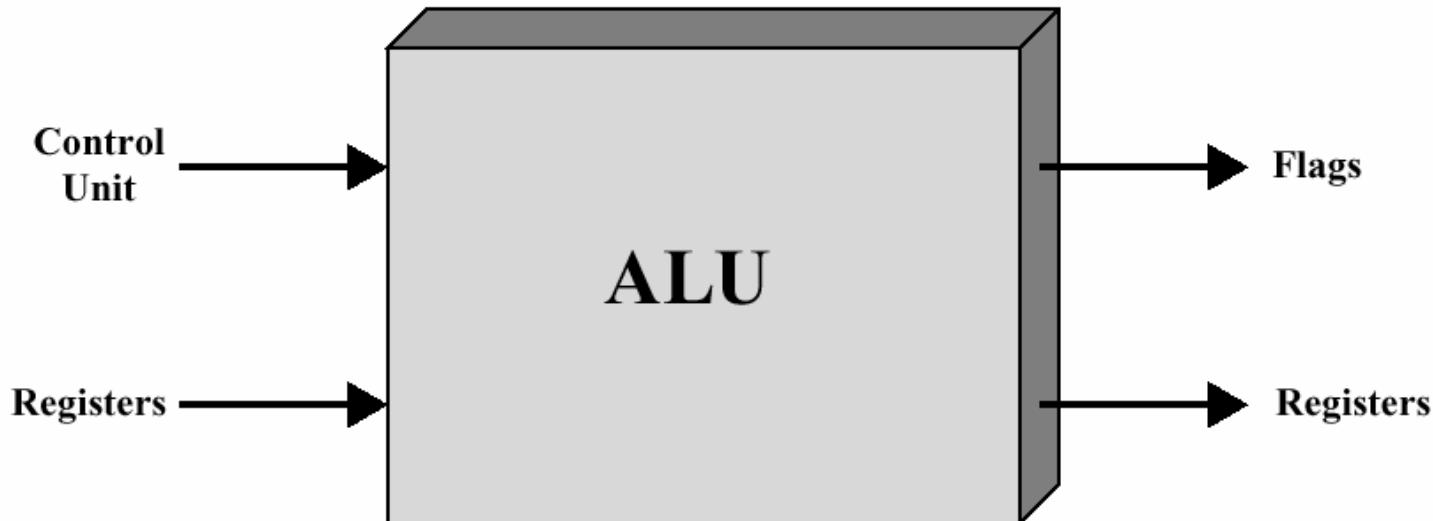
Kolejność bajtów cd.

- Wybór wersji kolejności zapisu bajtów zależy od konstruktorów procesora
 - Big Endian
 - Motorola 680x0 (Amiga/Atari ST/Mac)
 - IBM/Motorola PowerPC (Macintosh)
 - MIPS (SGI Indy/Nintendo 64)
 - Motorola 6800
 - Little Endian
 - Intel 80x86/Pentium (IBM PC)
 - Rockwell 6502 (Commodore 64)
 - MIPS (Sony Playstation/Digital DECstation)
 - Niektóre procesory (np. MIPS) można konfigurować zarówno w trybie Big Endian jak i Little Endian

ALU

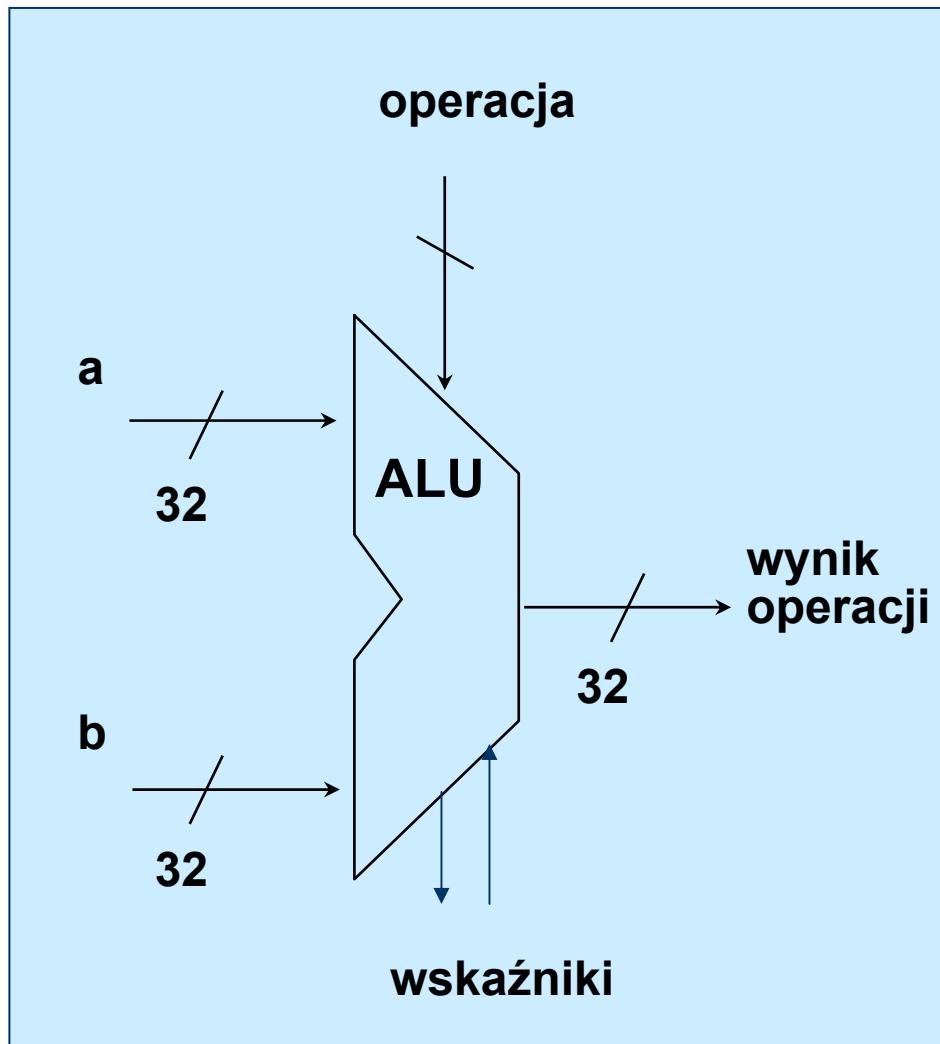
- Jednostka arytmetyczno-logiczna (**ALU – arithmetic-logic unit**) wykonuje operacje arytmetyczne na liczbach w kodzie dwójkowym
- ALU jest centralnym blokiem komputera; wszystkie inne bloki funkcjonalne służą do właściwej obsługi ALU
- Proste ALU wykonuje operacje na liczbach całkowitych (**integer**)
- Bardziej złożone ALU może wykonywać operacje zmiennoprzecinkowe FP na liczbach rzeczywistych
- Najczęściej ALU_INT i ALU_FP (**FPU – Floating Point Unit**) są wykonane jako dwa osobne bloki cyfrowe
 - FPU może być wykonane jako osobny układ scalony (koprocesor)
 - FPU może być wbudowane do procesora

ALU – wejścia i wyjścia



- CU – układ sterowania (Control Unit)
- Registers – rejesty wbudowane do CPU
- Flags – wskaźniki, znaczniki stanu, flagi, warunki: zespół wskaźników bitowych określających specyficzne właściwości wyniku operacji (zero, znak, przeniesienie itp.)

ALU – symbol logiczny



- Kod operacji pochodzi z układu sterowania (CU)
- Liczba bitów wyniku jest taka sama jak liczba bitów argumentów (operandów) – w tym przykładzie wynosi 32
- ALU nie tylko generuje bity wskaźników, ale także uwzględnia poprzedni stan wskaźników

Wieloznaczność informacji

- Co oznacza poniższy zapis?

10010111

- Liczbę całkowitą bez znaku: 151
- Liczbę całkowitą w kodzie znak-moduł: - 23
- Liczbę całkowitą w kodzie uzupełnień do dwóch: - 105
- Znak w rozszerzonym kodzie ASCII (IBM Latin 2): Ś
- Kod operacji procesora x86: **XCHG AX,DI**

To zależy od kontekstu

Liczby całkowite bez znaku

- używane są tylko cyfry 0 i 1

$$A = a_{n-1}a_{n-2} \dots a_1 \ a_0 \quad a_i \in \{0, 1\}$$

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

- kod ten bywa nazywany NKB
(naturalny kod binarny)
- zakres reprezentacji liczb dla słowa n -bitowego wynosi:
 $\langle 0, 2^{i-1} \rangle$
dla $n = 8$: $\langle 0, 255 \rangle$
dla $n = 16$: $\langle 0, 65535 \rangle$

Przykłady:

00000000 = 0

00000001 = 1

00101001 = 41

10000000 = 128

11111111 = 255

Kod znak-moduł (ZM)

- Najbardziej znaczący bit oznacza znak
 - 0 oznacza liczbę dodatnią
 - 1 oznacza liczbę ujemną
- Przykład
 - +18 = 00010010
 - 18 = 10010010
- Problemy

Układy arytmetyczne muszą osobno rozpatrywać bit znaku i moduł

Występują dwie reprezentacje zera:

$$\begin{aligned}+0 &= 00000000 \\-0 &= 10000000\end{aligned}$$

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{gdy } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{gdy } a_{n-1} = 1 \end{cases}$$

Kod uzupełnień do 2 (U2)

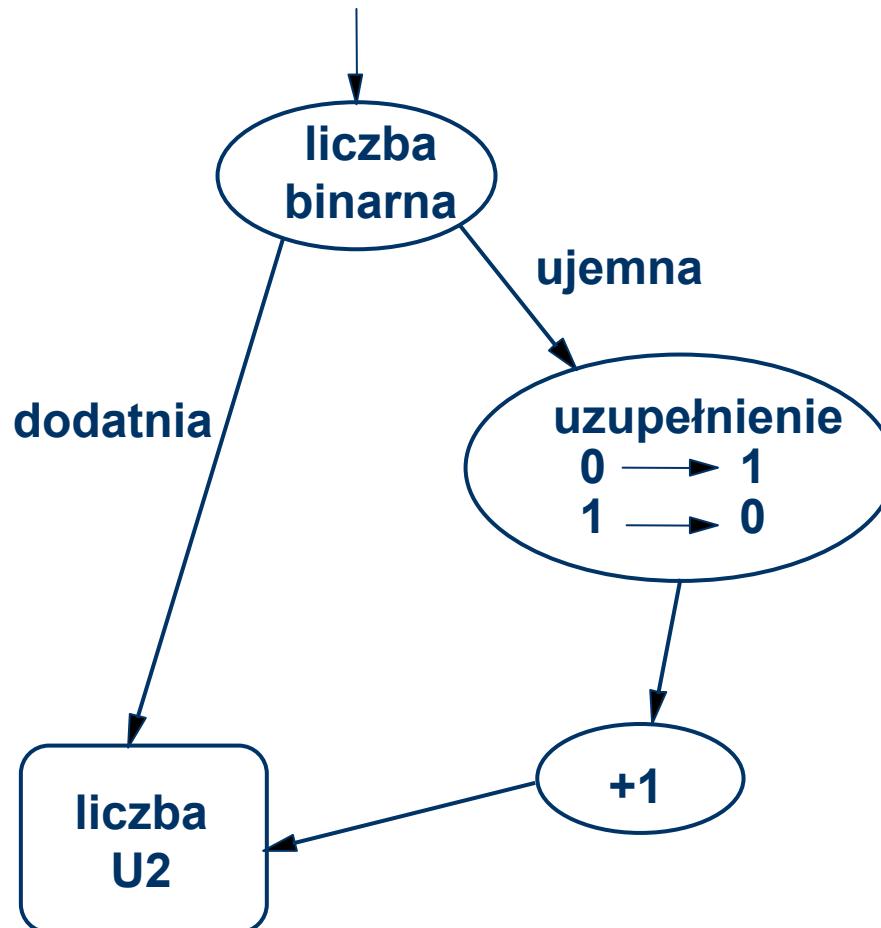
- Najbardziej znaczący bit oznacza znak liczby
 - 0 – liczba dodatnia
 - 1 – liczba ujemna
- Tylko jedna reprezentacja zera:
 $+0 = 00000000$
- Ogólna postać U2:

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

Przykłady:

+3	=	00000011
+2	=	00000010
+1	=	00000001
+0	=	00000000
-1	=	11111111
-2	=	11111110
-3	=	11111101

Konwersja NKB – U2



Zalety kodu U2

- Jedna reprezentacja zera
- Układy arytmetyczne (ALU) mają prostszą budowę (wykażemy to później)
- Negacja (zmiana znaku liczby) jest bardzo prosta:

3 = 00000011

uzupełnienie do 1(negacja boolowska)

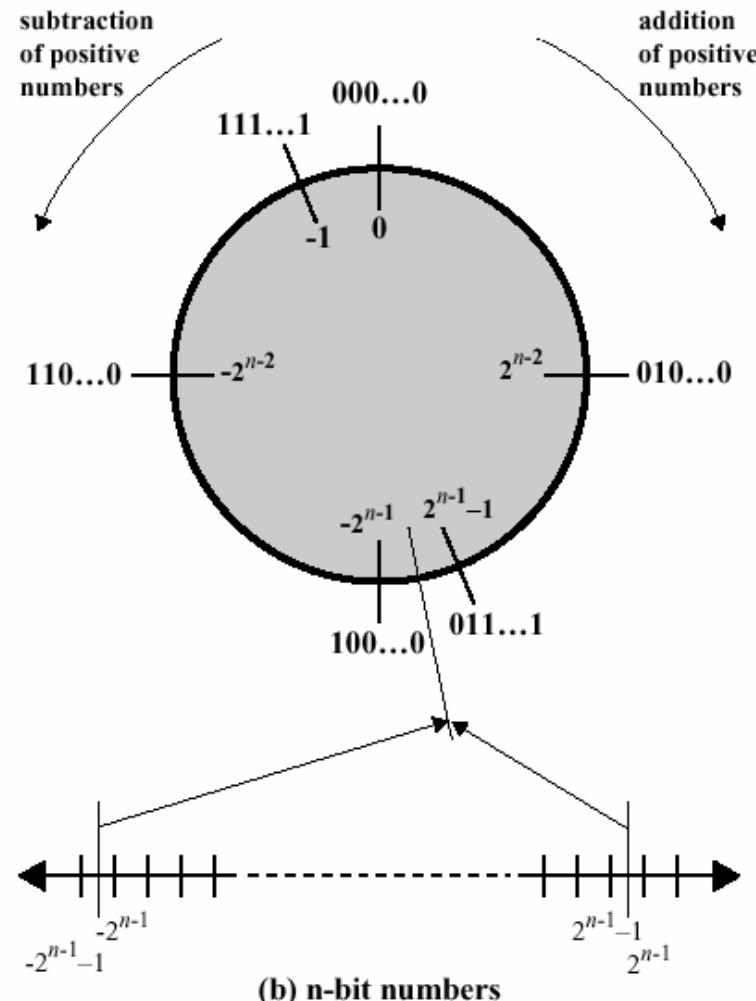
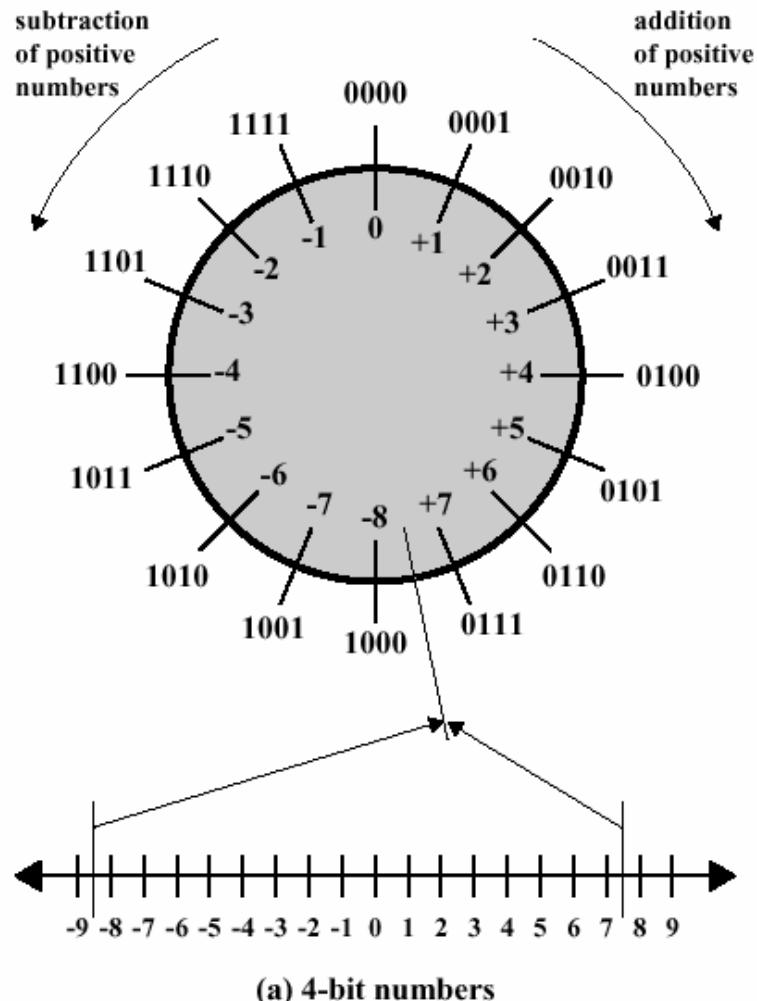
dodanie 1

11111100

11111101

$-3 = 11111101$

Kod U2 – ilustracja geometryczna



Wada kodu U2

- Zakres reprezentowanych liczb jest niesymetryczny
najmniejsza liczba: -2^{n-1} (*minint*)
największa liczba: $2^{n-1}-1$ (*maxint*)
- dla $n=8$
najmniejsza liczba: -128
największa liczba: +127
- dla $n=16$
najmniejsza liczba: -32 768
największa liczba: +32 767

Negacja w U2 – przypadki specjalne

0 =	00000000
negacja bitowa	11111111
dodaj 1 do LSB	+1
wynik	1 00000000
przepełnienie jest ignorowane	
więc	–0 = 0 ✓

–128 =	10000000
negacja bitowa	01111111
dodaj 1 do LSB	+1
wynik	10000000
więc	–(–128) = –128 ✗

Wniosek:

należy sprawdzać bit znaku
po negacji
(powinien się zmienić)

Konwersja długości słowa w U2

- Liczby dodatnie uzupełnia się wiodącymi zerami

+18 = 00010010

+18 = 00000000 00010010

- Liczby ujemne uzupełnia się wiodącymi jedynkami

-18 = 10010010

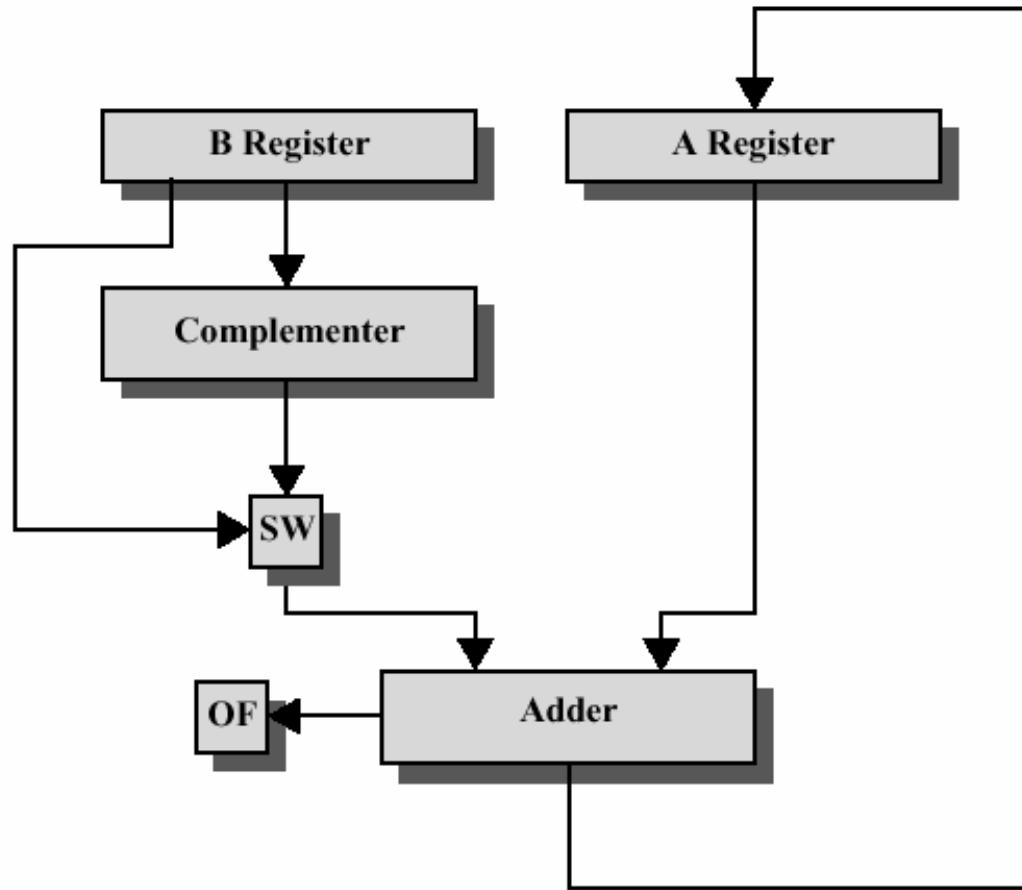
-18 = 11111111 10010010

- Ogólnie biorąc, powiela się bit MSB (bit znaku)

Dodawanie i odejmowanie w U2

- Obydwie operacje wykonuje się używając zwykłego dodawania liczb dwójkowych
- Należy sprawdzać bit znaku, aby wykryć nadmiar (*overflow*)
- Odejmowanie wykonuje się przez negowanie odjemnej i dodawanie:
 $a - b = a + (-b)$
- Do realizacji dodawania i odejmowania potrzebny jest zatem tylko **układ negacji bitowej i sumator**

Układ dodawania i odejmowania U2



OF = overflow bit

SW = Switch (select addition or subtraction)

Mnożenie

- Mnożenie liczb dwójkowych jest znacznie bardziej złożone od dodawania i odejmowania
- Podstawowy algorytm jest taki sam jak przy piśmieńskim mnożeniu liczb:
 - określa się iloczyny cząstkowe dla każdej cyfry mnoźnika
 - kolejne iloczyny cząstkowe należy umieszczać z przesunięciem o jedną pozycję (kolumnę) w lewo
 - należy dodać do siebie iloczyny cząstkowe

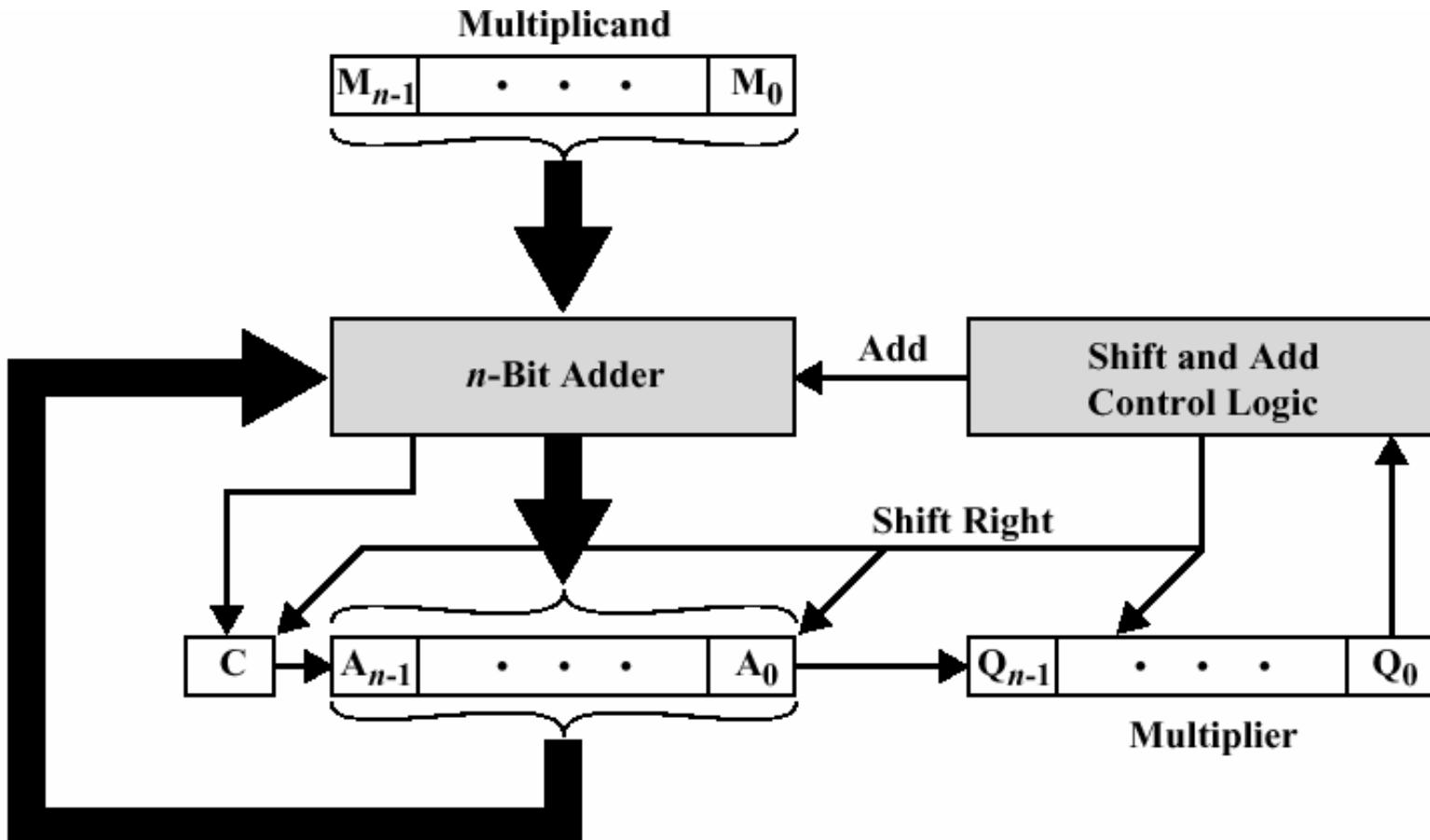
Przykład mnożenia (liczby bez znaku)

1011	mnożna (11 w kodzie dziesiętnym)
x 1101	mnożnik (13 w kodzie dziesiętnym)
<hr/>	iloczyny cząstkowe
0000	Uwaga: jeśli bit mnożnika jest równy 1, iloczyn cząstkowy jest równy mnożnej, w przeciwnym przypadku jest równy 0
1011	
<hr/>	
1011	
<hr/>	
10001111	wynik mnożenia (143 w kodzie dziesiętnym)

Uwaga #1: wynik ma podwójną długość; potrzebujemy słowa o podwójnej precyzji

Uwaga #2: powyższy algorytm funkcjonuje tylko dla liczb bez znaku; jeśli przyjąć kod U2 mnożna = -5, mnożnik = -3, natomiast iloraz wynosiłby -113, co oczywiście jest nieprawdą

Układ mnożenia liczb bez znaku



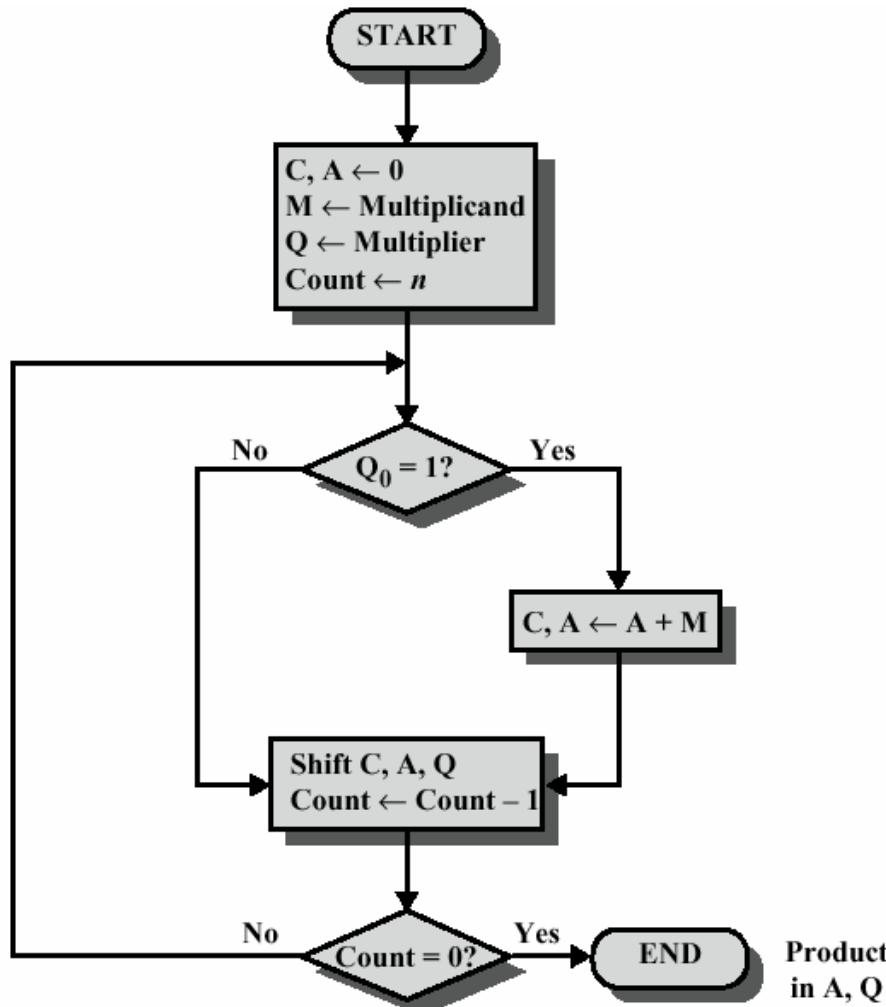
(a) Block Diagram

Przykład mnożenia liczb bez znaku

C	A	Q	M			
0	0000	1101	1011	Initial	Values	
0	1011	1101	1011	Add	}	First
0	0101	1110	1011	Shift		Cycle
0	0010	1111	1011	Shift	}	Second
0	1101	1111	1011	Add		Cycle
0	0110	1111	1011	Shift	}	Third
1	0001	1111	1011	Add		Cycle
0	1000	1111	1011	Shift	}	Fourth
						Cycle

wynik mnożenia

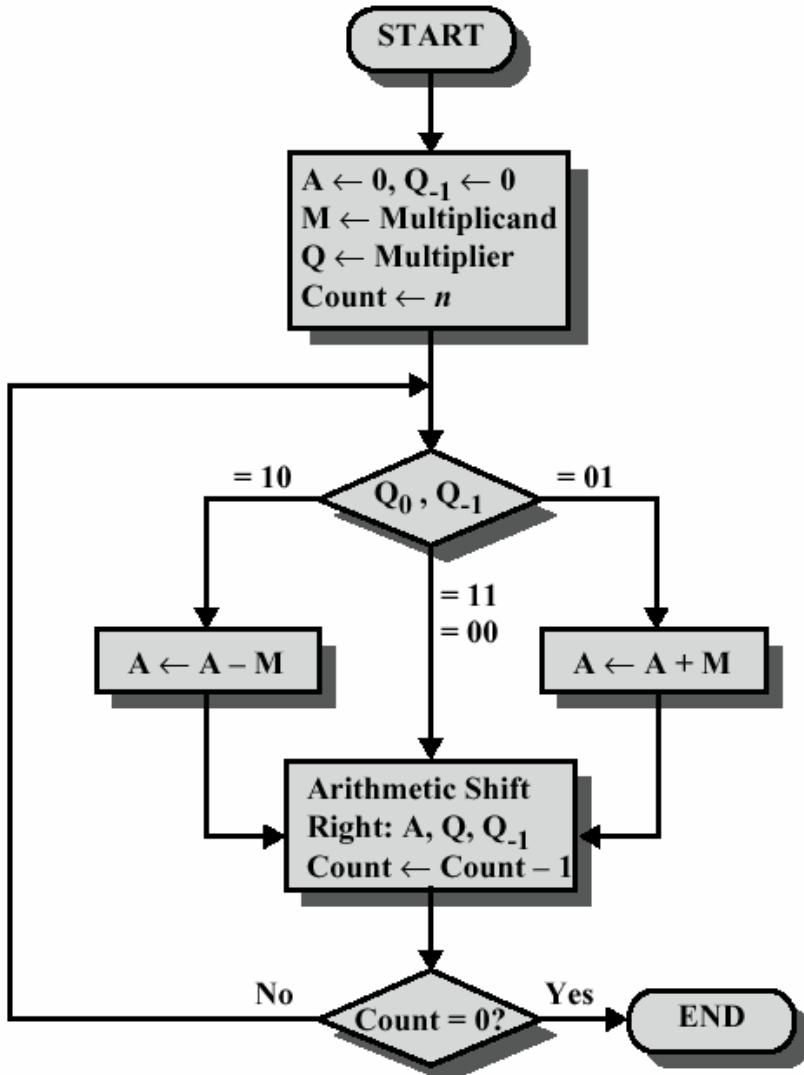
Mnożenie liczb bez znaku cd.



Mnożenie liczb ze znakiem

- Prosty algorytm podany wcześniej nie działa
- Rozwiązanie 1
 - zamienić czynniki ujemne na dodatnie
 - pomnożyć liczby korzystając z podanego wcześniej algorytmu
 - jeśli znaki czynników (przed wykonaniem negacji) były różne, zanegować iloczyn
- Rozwiązanie 2
 - algorytm Booth'sa

Algorytm Booth'sa



Q-1 dodatkowy bit (przerzutnik) pamiętający najmniej znaczący bit rejestru Q opuszczający ten rejestr przy przesunięciu w prawo

Przesunięcie arytmetyczne w prawo – przesunięcie z powieleniem bitu znaku

Przykład działania metody Booth'sa

A	Q	Q_{-1}	M	Initial Values
0000	0011	0	0111	
1001	0011	0	0111	$A \leftarrow A - M$
1100	1001	1	0111	Shift } First Cycle
1110	0100	1	0111	Shift } Second Cycle
0101	0100	1	0111	$A \leftarrow A + M$ } Third Cycle
0010	1010	0	0111	Shift }
0001	0101	0	0111	Shift } Fourth Cycle

iloczyn: $3 \times 7 = 21$

testowane bity: 10 $A := A - M$
 01 $A := A + M$

Algorytm Booth'sa (znak dowolny)

$$\begin{array}{r}
 0111 \\
 \times 0011 \\
 \hline
 11111001 \\
 00000000 \\
 \hline
 000111 \\
 \hline
 00010101
 \end{array}
 \quad (0) \quad (21)$$

$$(a) (7) \times (3) = (21)$$

$$\begin{array}{r}
 0111 \\
 \times 1101 \\
 \hline
 11111001 \\
 0000111 \\
 \hline
 111001 \\
 \hline
 11101011
 \end{array}
 \quad (0) \quad (-21)$$

$$(b) (7) \times (-3) = (-21)$$

$$\begin{array}{r}
 1001 \\
 \times 0011 \\
 \hline
 00000111 \\
 00000000 \\
 \hline
 111001 \\
 \hline
 11101011
 \end{array}
 \quad (0) \quad (-21)$$

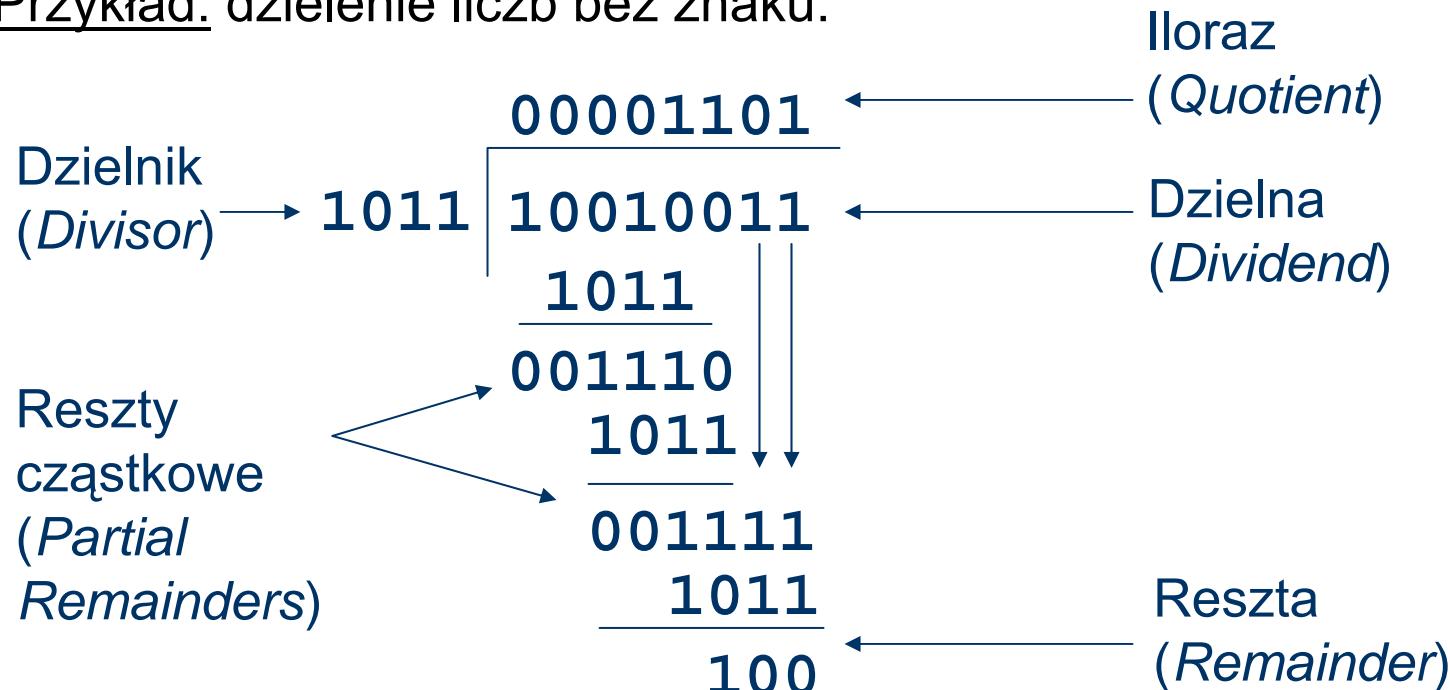
$$(c) (-7) \times (3) = (-21)$$

$$\begin{array}{r}
 1001 \\
 \times 1101 \\
 \hline
 00000111 \\
 1111001 \\
 000111 \\
 \hline
 00010101
 \end{array}
 \quad (0) \quad (21)$$

$$(d) (-7) \times (-3) = (21)$$

Dzielenie

- Dzielenie liczb dwójkowych jest znacznie bardziej skomplikowane od mnożenia
- Liczby ujemne sprawiają spore kłopoty (*Stallings*, s. 341–342)
- Podstawowy algorytm podobny do dzielenia liczb na papierze: kolejne operacje przesuwania, dodawania lub odejmowania
- Przykład: dzielenie liczb bez znaku:



Liczby rzeczywiste

- Liczby z częścią ułamkową
- Można je zapisać korzystając z kodu NKB i dodatkowo znaków ‘–’ oraz ‘.’
$$1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9.625$$
- Problem: gdzie znajduje się kropka dziesiętna?
- W stałym miejscu?
 - rozwiązanie niedobre z punktu widzenia metod numerycznych
- Na zmiennej pozycji?
 - jak podać informację o miejscu położenia kropki?

Liczby zmiennoprzecinkowe (FP)

bit znaku	przesunięty wykładnik <i>(exponent)</i>	mantysa (<i>significand or mantissa</i>)
-----------	--	---

- Wartość liczby: $+/- 1.$ mantysa $\times 2^{\text{wykładnik}}$
- Podstawa 2 jest ustalona i nie musi być przechowywana
- Położeniu punktu dziesiętnego jest ustalone: na lewo od najbardziej znaczącego bitu mantisy

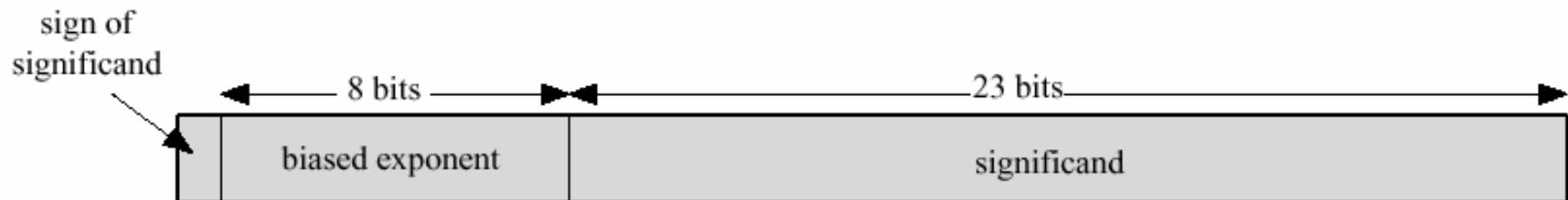
FP – mantysa

- Mantysa jest zapisana w kodzie U2
- Wykładnik jest zapisany z przesunięciem (*excess or biased notation*)
 - np. przesunięcie równe 127 oznacza:
 - 8-bitowe pole wykładnika
 - zakres liczb 0-255
 - od przesuniętego wykładnika należy odjąć 127 abytrzymać prawdziwą wartość
 - zakres wartości wykładnika -127 to +128

FP – normalizacja

- Liczby FP są zwykle normalizowane, tzn. wykładnik jest tak dobierany, aby najbardziej znaczący bit (MSB) mantisy był równy 1
- Ponieważ bit ten jest zawsze równy 1, nie musi być przechowywany. Dlatego 23-bitowe pole mantisy w rzeczywistości odpowiada mantysie 24-bitowej, z cyfrą 1 na najbardziej znaczącej pozycji (mantysa mieści się zatem w przedziale od 1 do 2)
- Uwaga: w notacji naukowej FP (*Scientific notation*) liczby są normalizowane inaczej, tak aby mantysa miała jedną znającą cyfrę przed kropką dziesiętną,
np. 3.123×10^3

FP – przykład



(a) Format

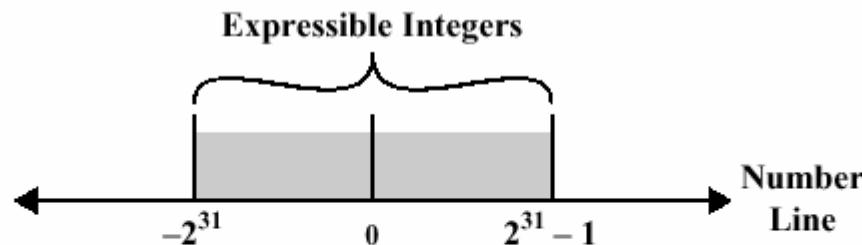
$$\begin{aligned} 1,1010001 \times 2^{10100} &= 0\ 10010011\ 101000100000000000000000 \\ -1,1010001 \times 2^{10100} &= 1\ 10010011\ 101000100000000000000000 \\ 1,1010001 \times 2^{-10100} &= 0\ 01101011\ 101000100000000000000000 \\ -1,1010001 \times 2^{-10100} &= 1\ 01101011\ 101000100000000000000000 \end{aligned}$$

**pozycje mantisy
uzupełniane są zerami**

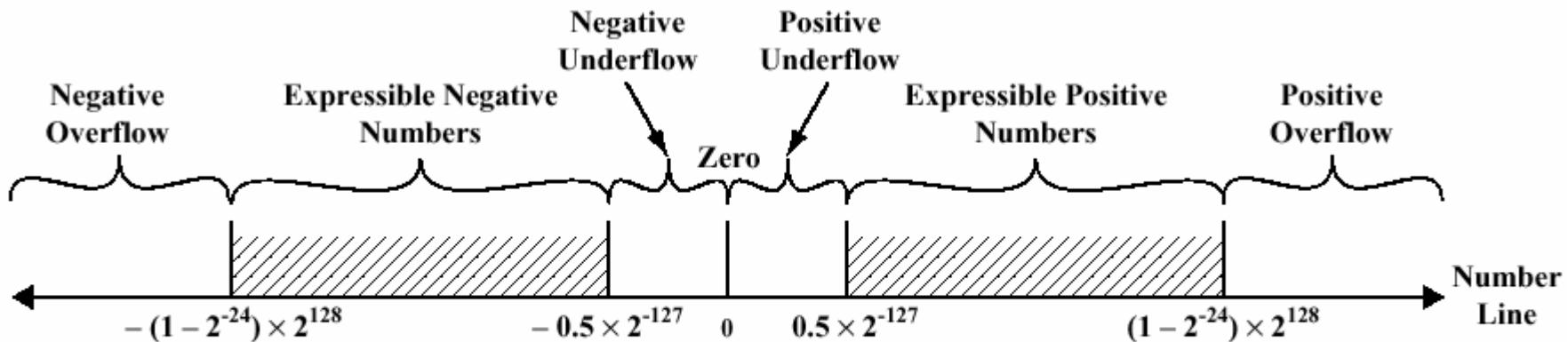
Zakres reprezentacji liczb FP

- Dla formatu 32-bitowego
 - 8-bitowy wykładnik
 - $+/- 2^{256} \approx 1.5 \times 10^{77}$
- Dokładność
 - efekt nierównomiernego pokrycia osi liczb rzeczywistych (zmienna wartość LSB mantisy)
 - 23-bitowa mantysa: $2^{-23} \approx 1.2 \times 10^{-7}$
 - około 6 pozycji dziesiętnych

Zakresy liczb FP i U2 (format 32-bitowy)



(a) Twos Complement Integers



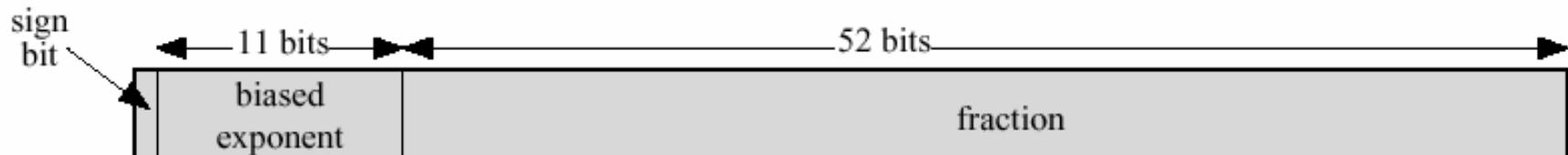
(b) Floating-Point Numbers

Standard IEEE 754

- Dwa warianty: 32- i 64-bitowy format liczb
- 8- lub 11-bitowy wykładnik ($bias = 127$ lub 1023)
- IEEE 754 dopuszcza ponadto tzw. formaty rozszerzone (*extended formats*) dla obliczeń pośrednich w systemach cyfrowych



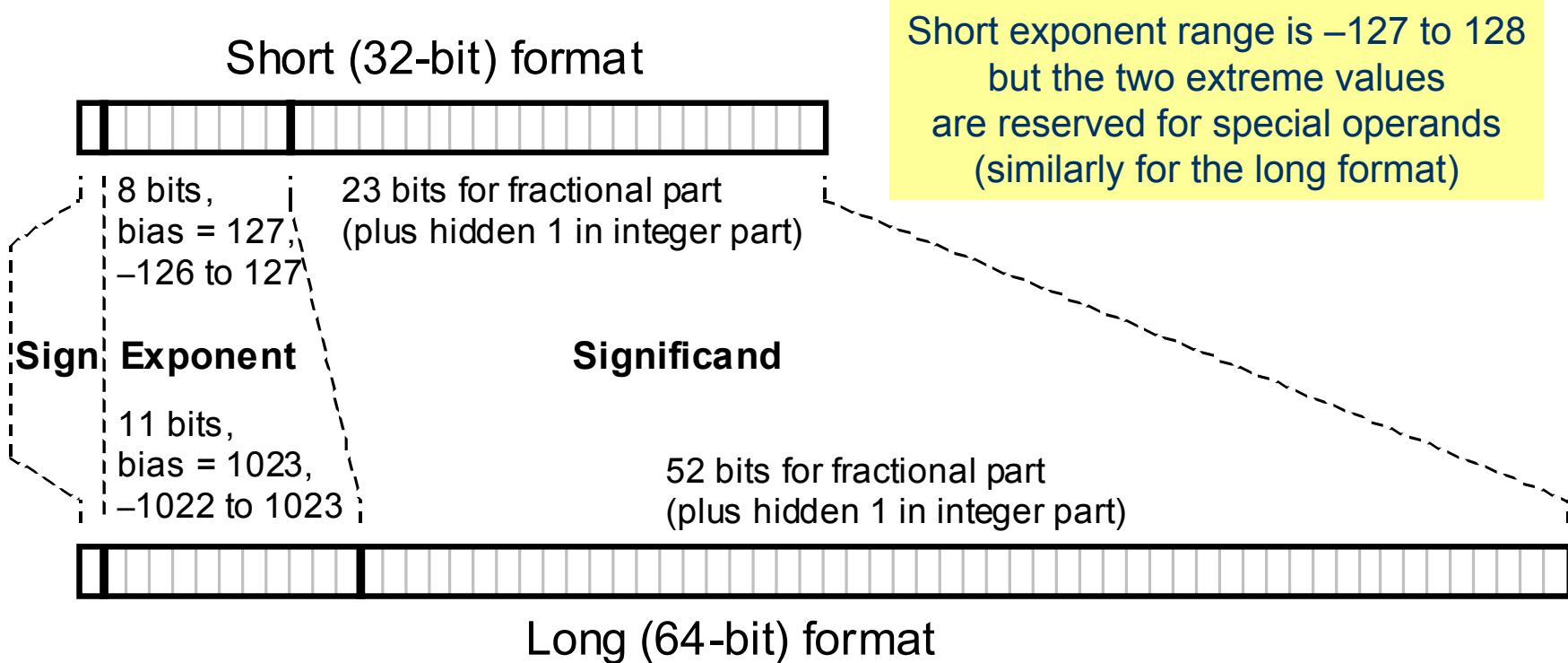
(a) Single format



(b) Double format

ANSI/IEEE Standard Floating-Point Format (IEEE 754)

Revision (IEEE 754R) is being considered by a committee



The two ANSI/IEEE standard floating-point formats.

Standard IEEE 754

- W rzeczywistości standard IEEE 754 jest bardziej skomplikowany:
 - dodatkowe dwa bity: *guard* i *round*
 - cztery warianty zaokrąglania
 - liczba dodatnia podzielona przez 0 daje nieskończoność
 - nieskończoność dzielona przez nieskończoność daje NaN (*not a number*)
 - niektóre procesory nie są w pełni zgodne z IEEE 754, skutki są na ogólnie niedobre (*Pentium bug*)

Some features of ANSI/IEEE standard floating-point formats

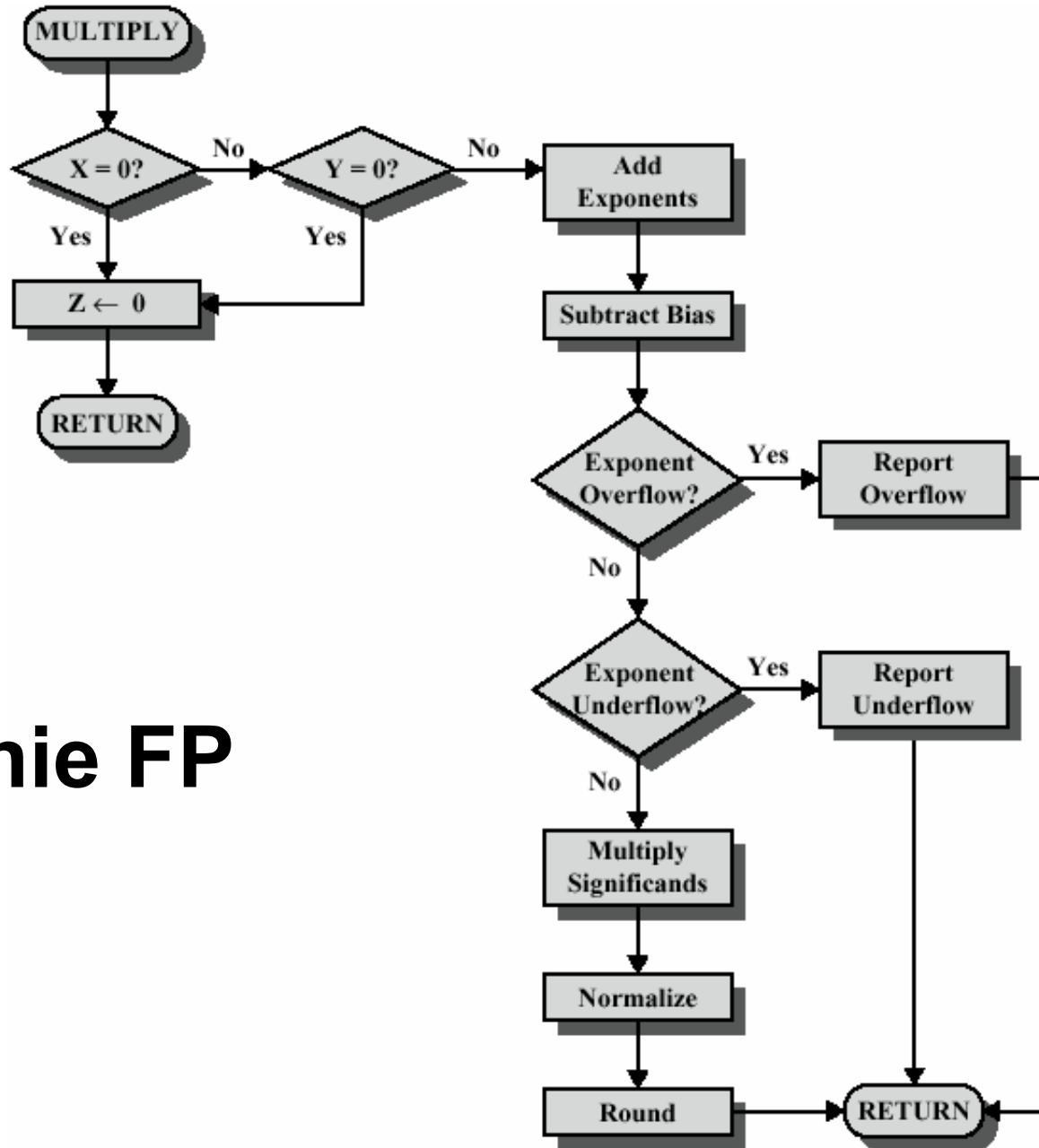
Feature	Single/Short	Double/Long
Word width in bits	32	64
Significand in bits	23 + 1 hidden	52 + 1 hidden
Significand range	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Exponent bits	8	11
Exponent bias	127	1023
Zero (± 0)	$e + \text{bias} = 0, f = 0$	$e + \text{bias} = 0, f = 0$
Denormal	$e + \text{bias} = 0, f \neq 0$ represents $\pm 0.f \times 2^{-126}$	$e + \text{bias} = 0, f \neq 0$ represents $\pm 0.f \times 2^{-1022}$
Infinity ($\pm\infty$)	$e + \text{bias} = 255, f = 0$	$e + \text{bias} = 2047, f = 0$
Not-a-number (NaN)	$e + \text{bias} = 255, f \neq 0$	$e + \text{bias} = 2047, f \neq 0$
Ordinary number	$e + \text{bias} \in [1, 254]$ $e \in [-126, 127]$ represents $1.f \times 2^e$	$e + \text{bias} \in [1, 2046]$ $e \in [-1022, 1023]$ represents $1.f \times 2^e$
<i>min</i>	$2^{-126} \cong 1.2 \times 10^{-38}$	$2^{-1022} \cong 2.2 \times 10^{-308}$
<i>max</i>	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

Arytmetyka FP +/-

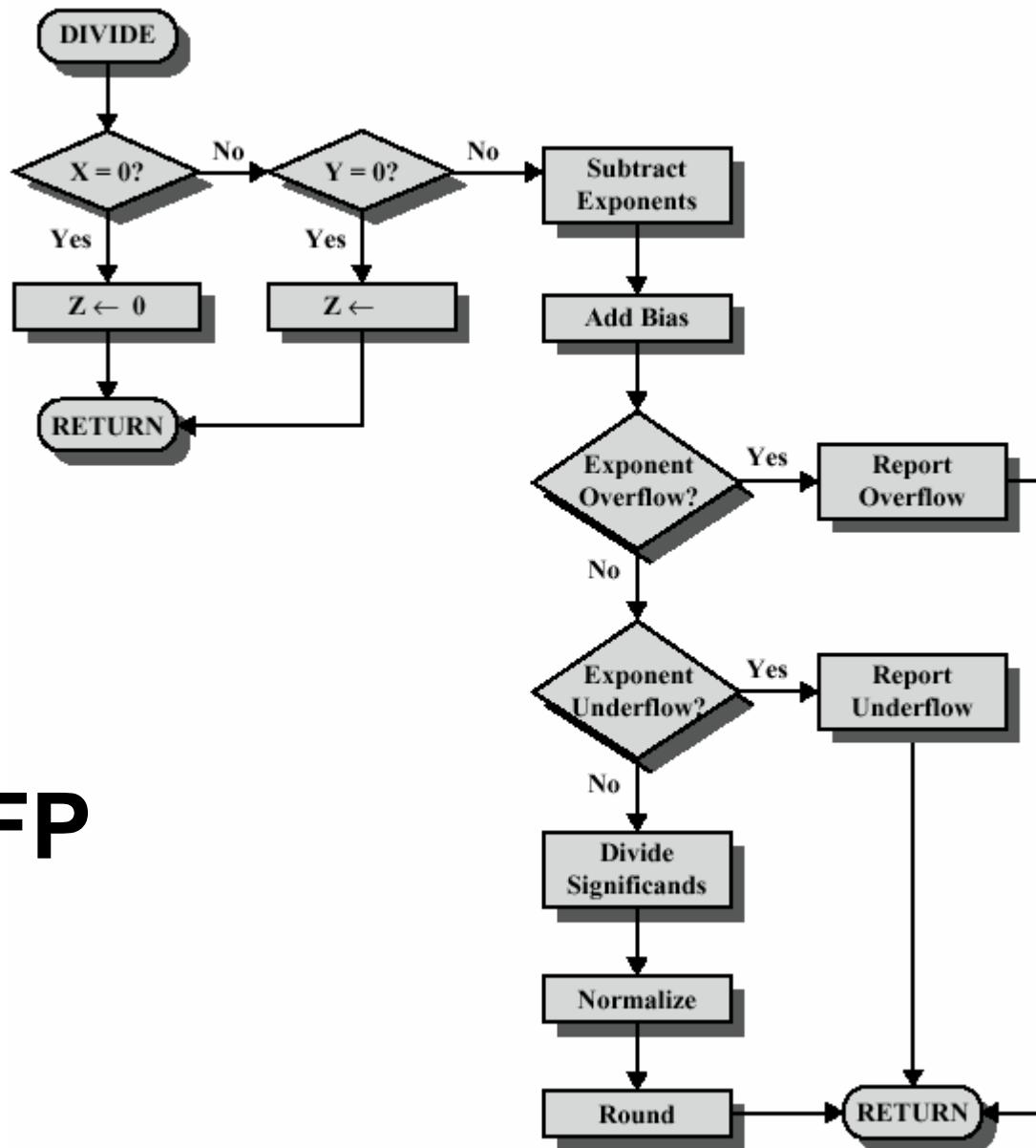
- W arytmetyce FP dodawanie i odejmowanie są bardziej złożonymi operacjami niż mnożenie i dzielenie (powodem jest konieczność tzw. wyrównywania składników)
- Etapy dodawania i odejmowania
 - Sprawdzenie zer
 - Wyrównanie mantys (i korekcja wykładników)
 - Dodawanie lub odjęcie mantys
 - Normalizowanie wyniku

Arytmetyka FP \times / \div

- Etapy mnożenia i dzielenia
 - sprawdzenie zer
 - dodawanie/odejmowanie wykładników
 - mnożenie/dzielenie mantys (uwaga na znak)
 - normalizacja
 - zaokrąglanie
 - Uwaga: wszystkie wyniki obliczeń pośrednich powinny być wykonywane w podwójnej precyzji



Mnożenie FP



Dzielenie FP

Podsumowanie

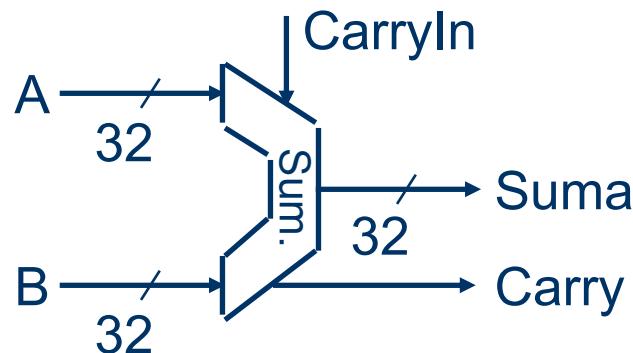
- Zapis słów w pamięci (Endian)
- Reprezentacje liczb: NKB, ZM, U2, BCD, HEX
- Ogólna charakterystyka ALU
- Operacje na liczbach całkowitych bez znaku
- Operacje na liczbach całkowitych ze znakiem
- Mnożenie – algorytm Booth'sa
- Liczby zmiennoprzecinkowe FP
- Formaty liczb FP, IEEE 754
- Operacje na liczbach FP

Organizacja i Architektura Komputerów

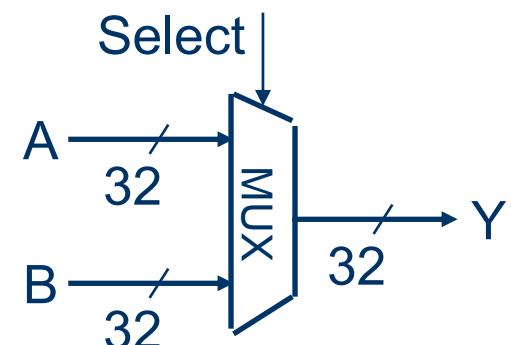
Elementy i bloki cyfrowe

Bloki kombinacyjne

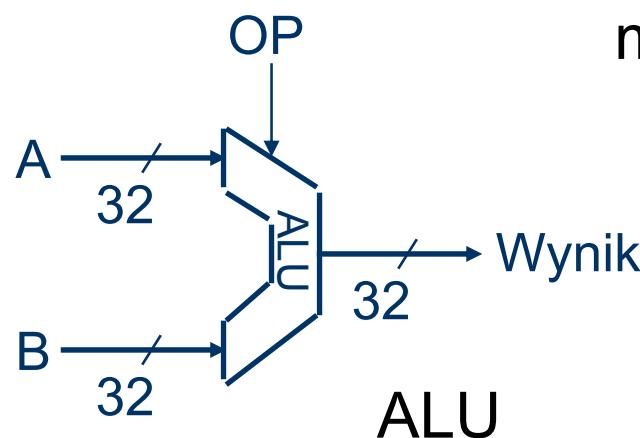
Przykłady: elementy 32-bitowego procesora



sumator



multiplekser

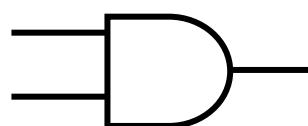


ALU

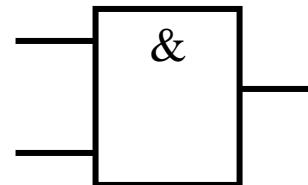
Bramki logiczne

Dowolny blok kombinacyjny można zbudować z bramek

Symbol tradycyjny
IEC



Symbol prostokątny
ANSI/IEEE



- iloczyn logiczny AND
- zanegowany iloczyn logiczny NAND
- suma logiczna OR
- zanegowana suma logiczna NOR
- negator (inwerter) NOT
- suma modulo 2 (exclusive OR) XOR

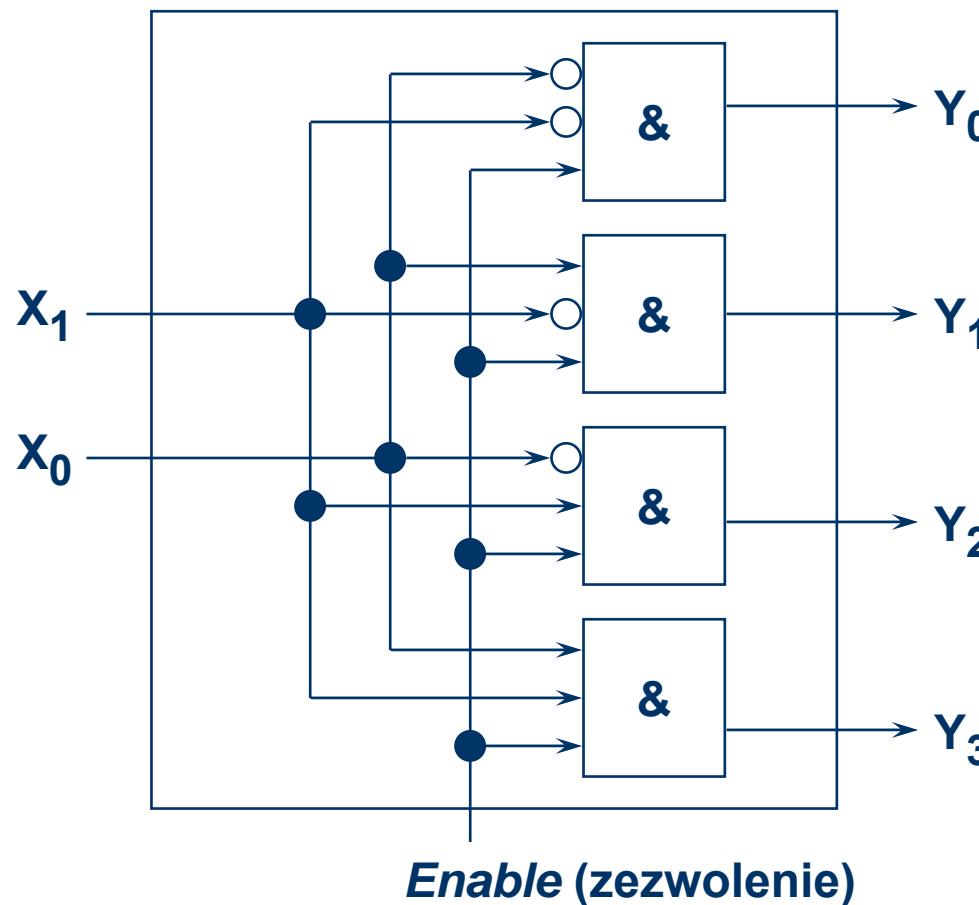
Dekoder binarny

- n wejść, 2^n wyjść
- tylko jedno wyjście jest w stanie ‘1’ dla danej kombinacji stanów na wejściu (funkcja ‘1 z n ’)

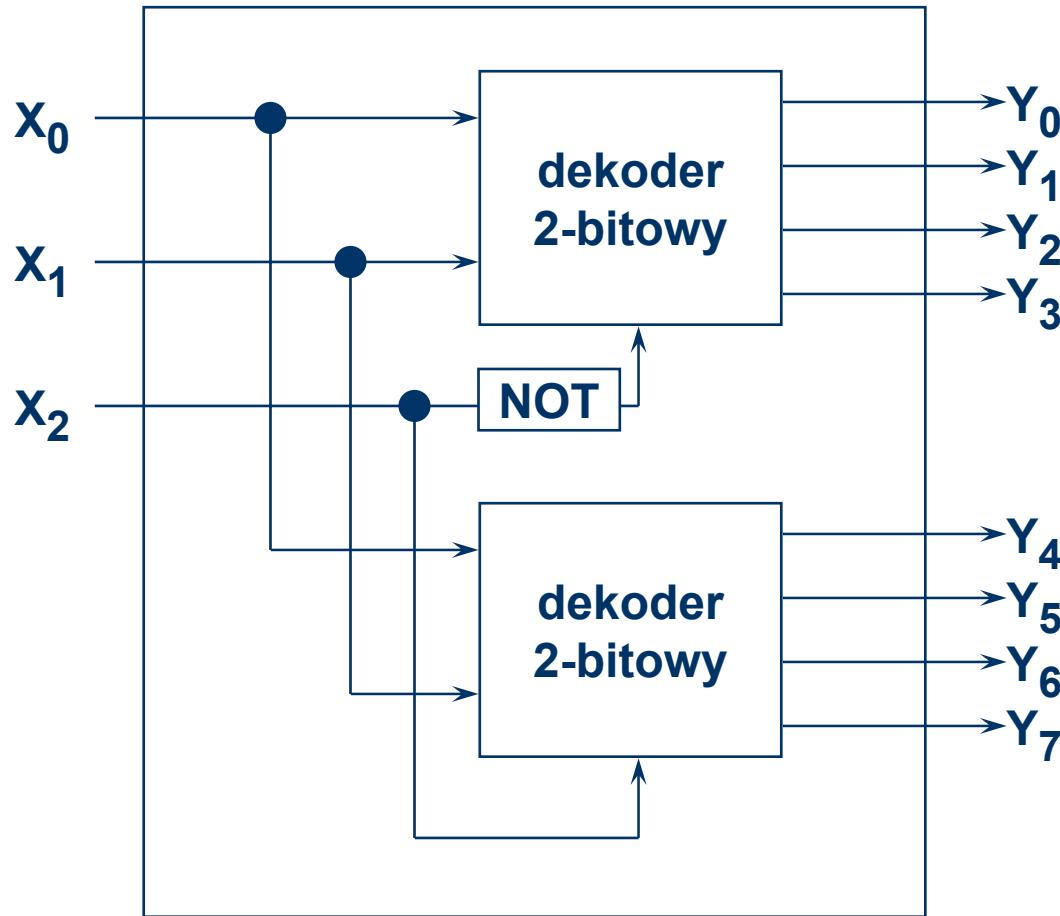


Dekoder binarny – implementacja

Prosty dekoder 2-bitowy:

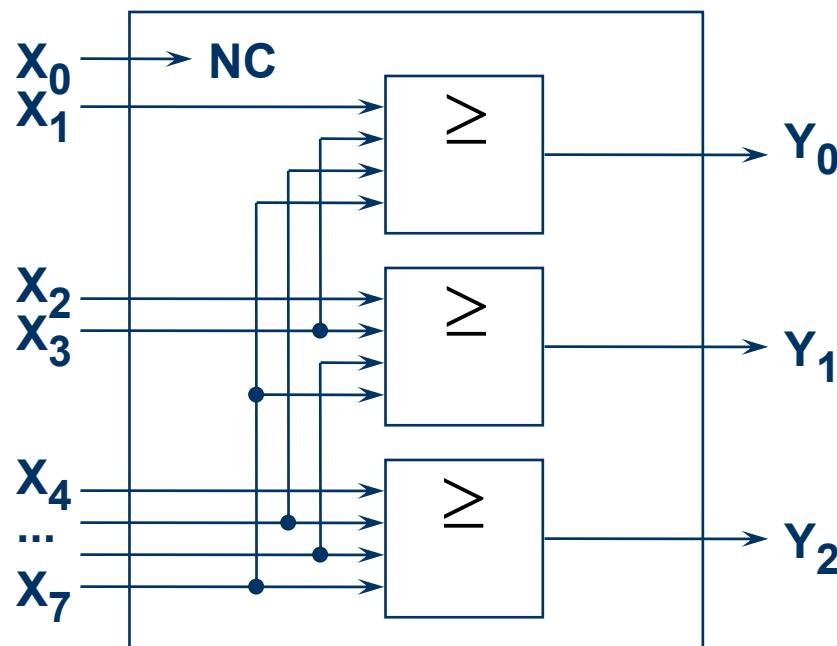


2 + 2 = 3 ?



Dwa dekodery 2-bitowe tworzą dekoder 3-bitowy

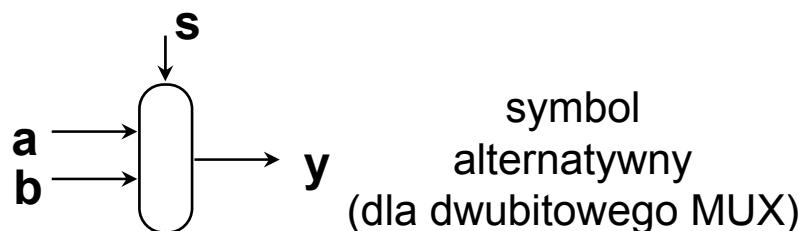
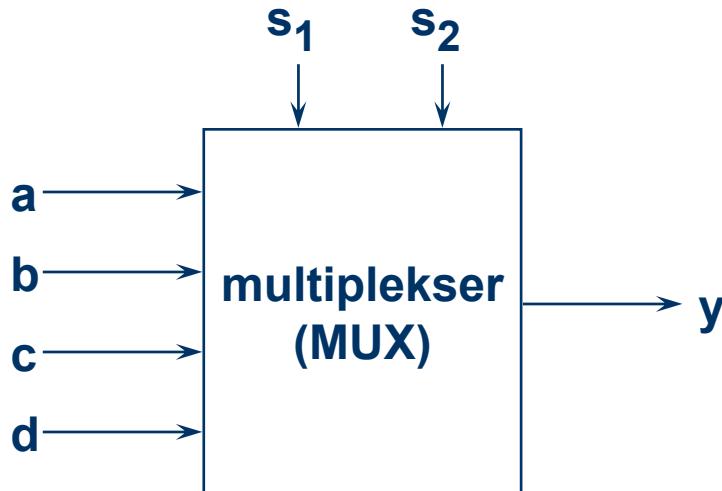
Koder kodu ‘1 z n’ na kod NKB



\geq suma logiczna

Multiplekser

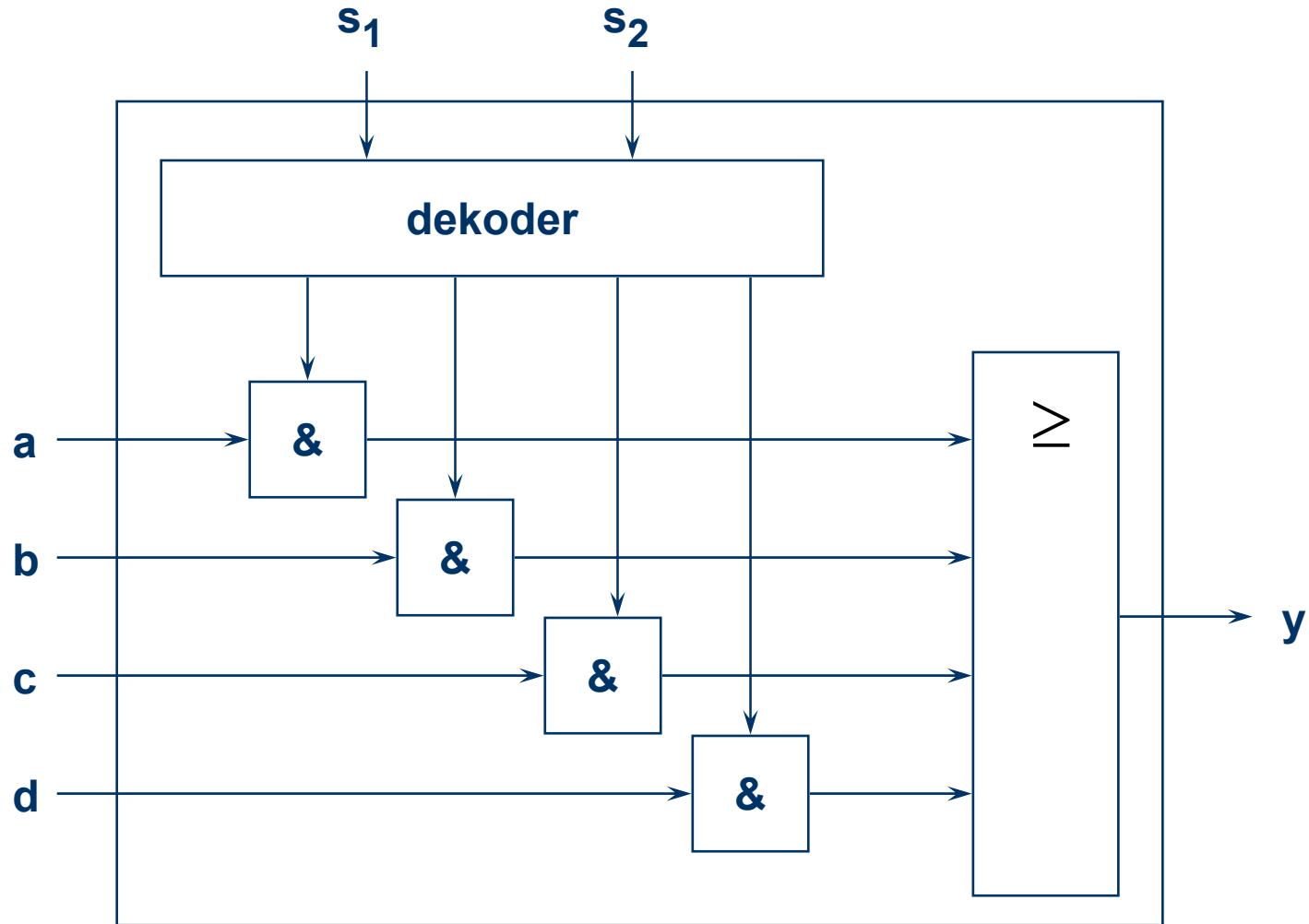
- Działa jak przełącznik; nazywany multiplekserem lub w skrócie MUX



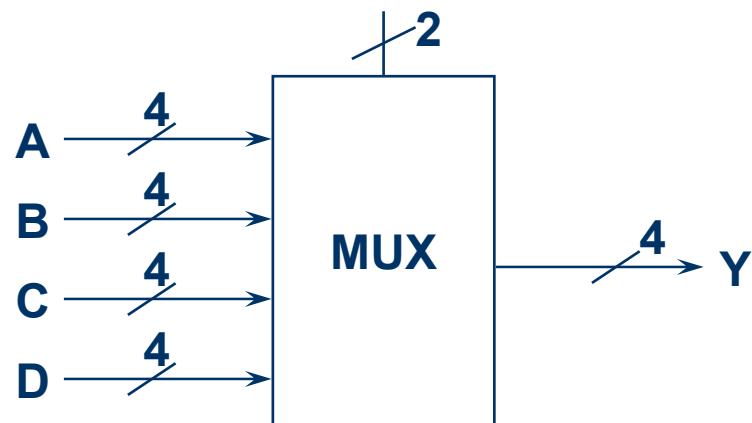
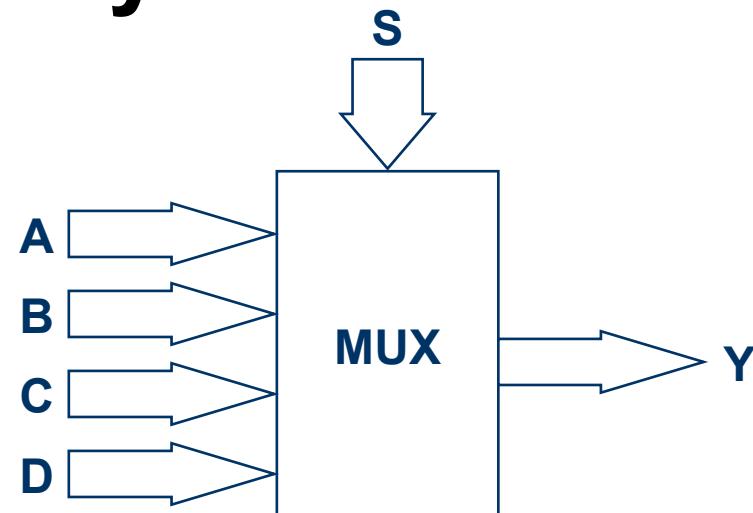
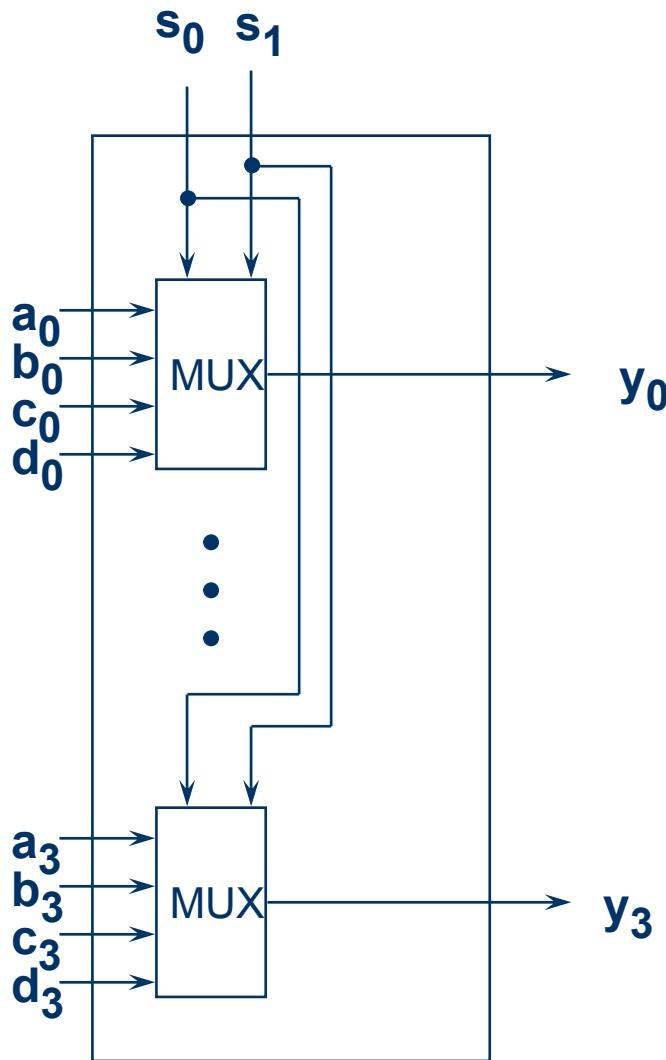
Tablica prawdy

s_2	s_1	y
0	0	a
0	1	b
1	0	c
1	1	d

Multiplekser – implementacja



Multiplekser grupowy



Język VHDL – przykład

```
entity MUX32X2 is
generic (output_delay : TIME := 4 ns);
port(A,B:           in  vlbit_1d(31 downto 0);
      DOUT:        out vlbit_1d(31 downto 0);
      SEL:         in  vlbit);
end MUX32X2;

architecture behavior of MUX32X2 is
begin
  mux32x2_process: process(A, B, SEL)
    begin
      if (vlb2int(SEL) = 0) then
        DOUT <= A after output_delay;
      else
        DOUT <= B after output_delay;
      end if;
    end process;
  end behavior;
```

- **VHDL** – *Very (High Speed Integrated Circuit) Hardware Description Language*
- Opracowany w latach 80-tych przez Departament Obrony USA
- Norma IEEE Standard 1076 (1987, 1993, 2001)

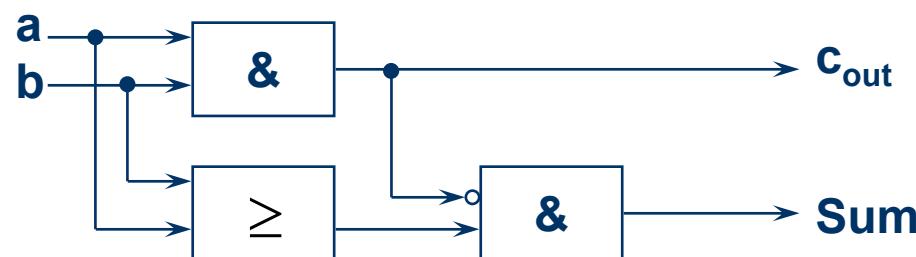
Półsumator – HA (*half adder*)

$$\text{Sum} = (\bar{a} \bullet b) + (a \bullet \bar{b}) = (\bar{a} + \bar{b}) \bullet (a + b)$$

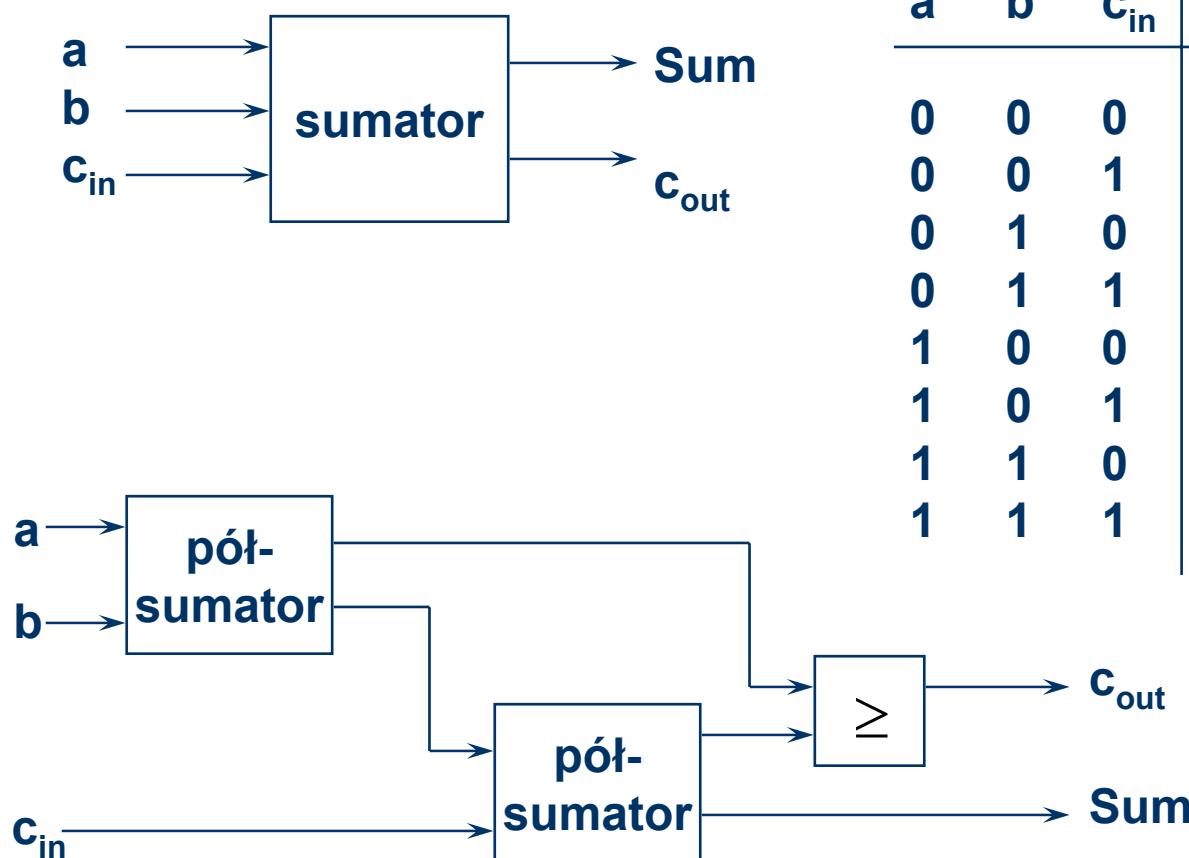
$$\text{Carry} = a \bullet b$$



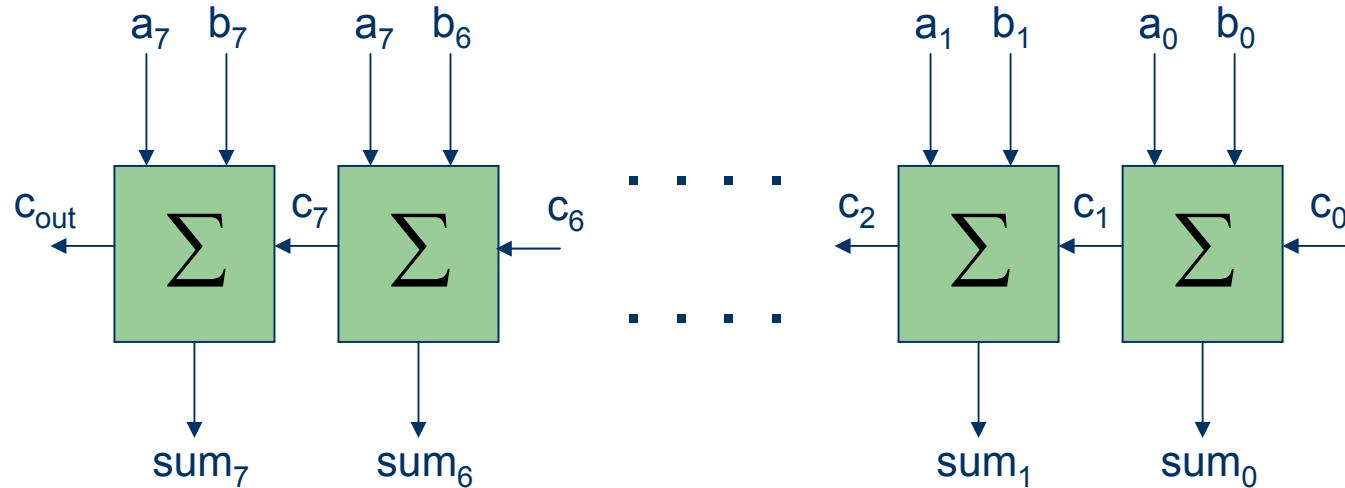
a	b	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Sumator: HA + HA = A (adder)



Sumator wielobitowy, szeregowy



- Szeregowa propagacja przeniesień (*ripple-carry*)
- Wada: długi czas ustalania wyniku; sygnał przeniesienia propaguje przez wszystkie stopnie sumatora

Sumator równoległy (4-bitowy)

Określamy funkcję g_i generującą przeniesienie i funkcję p_i , określającą propagację przeniesienia

$$g_i = a_i b_i$$

$$p_i = a_i \oplus b_i$$

Kolejne przeniesienia można teraz zapisać w postaci:

$$c_{i+1} = g_i + p_i c_i$$

$$c_{i+2} = g_{i+1} + p_{i+1} c_{i+1}$$

$$c_{i+3} = g_{i+2} + p_{i+2} c_{i+2}$$

$$c_{i+4} = g_{i+3} + p_{i+3} c_{i+3}$$

Sumator równoległy cd.

a następnie, po dokonaniu podstawień, w postaci:

$$c_{i+1} = g_i + p_i c_i$$

$$c_{i+2} = g_{i+1} + p_{i+1} g_i + p_{i+1} p_i c_i$$

$$c_{i+3} = g_{i+2} + p_{i+2} g_{i+1} + p_{i+2} p_{i+1} g_i + p_{i+2} p_{i+1} p_i c_i$$

$$c_{i+4} = g_{i+3} + p_{i+3} g_{i+2} + p_{i+3} p_{i+2} g_{i+1} + p_{i+3} p_{i+2} p_{i+1} g_i + p_{i+3} p_{i+2} p_{i+1} p_i c_i$$

wprowadzając oznaczenia:

$$g_{(i,i+3)} = g_{i+3} + p_{i+3} g_{i+2} + p_{i+3} p_{i+2} g_{i+1} + p_{i+3} p_{i+2} p_{i+1} g_i$$

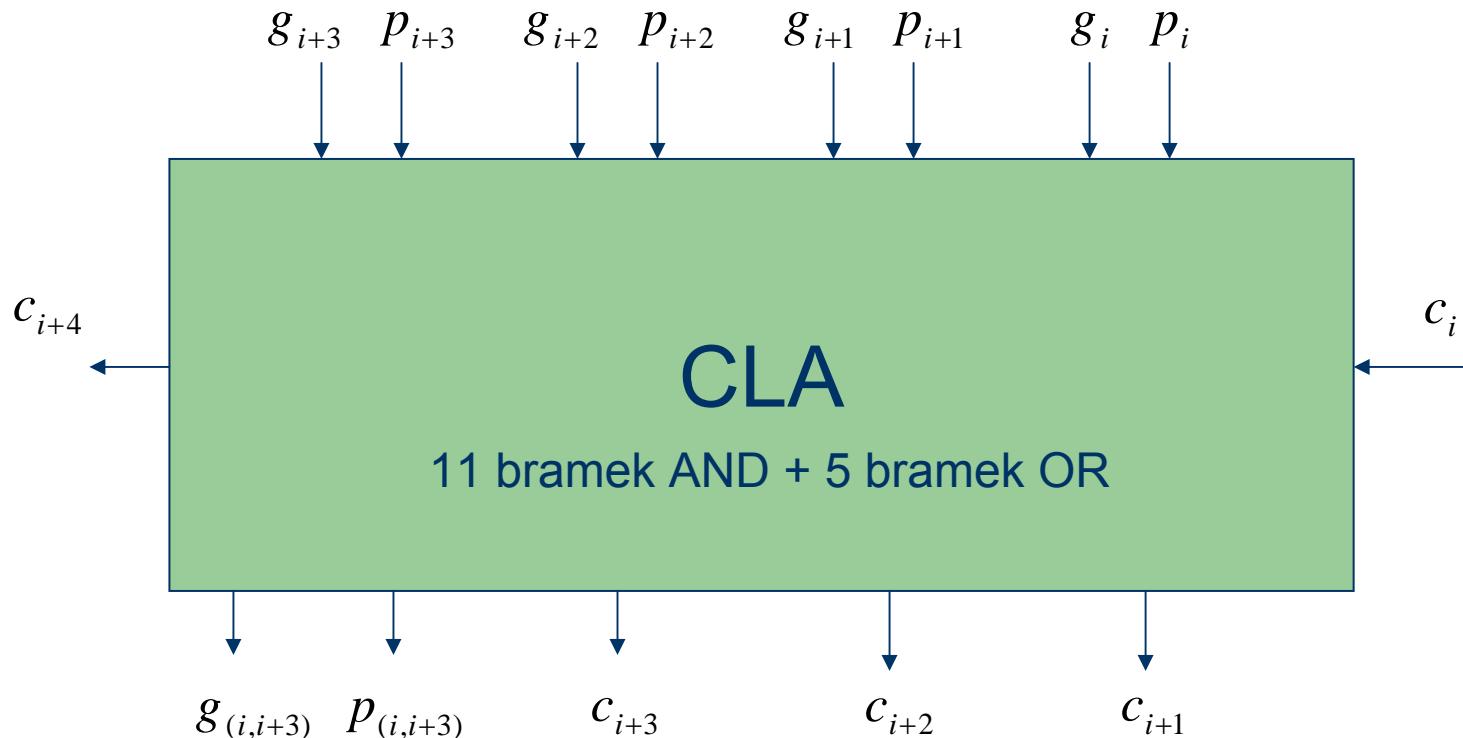
$$p_{(i,i+3)} = p_{i+3} p_{i+2} p_{i+1} p_i$$

otrzymujemy ostatecznie:

$$c_{i+4} = g_{(i,i+3)} + p_{(i,i+3)} c_i$$

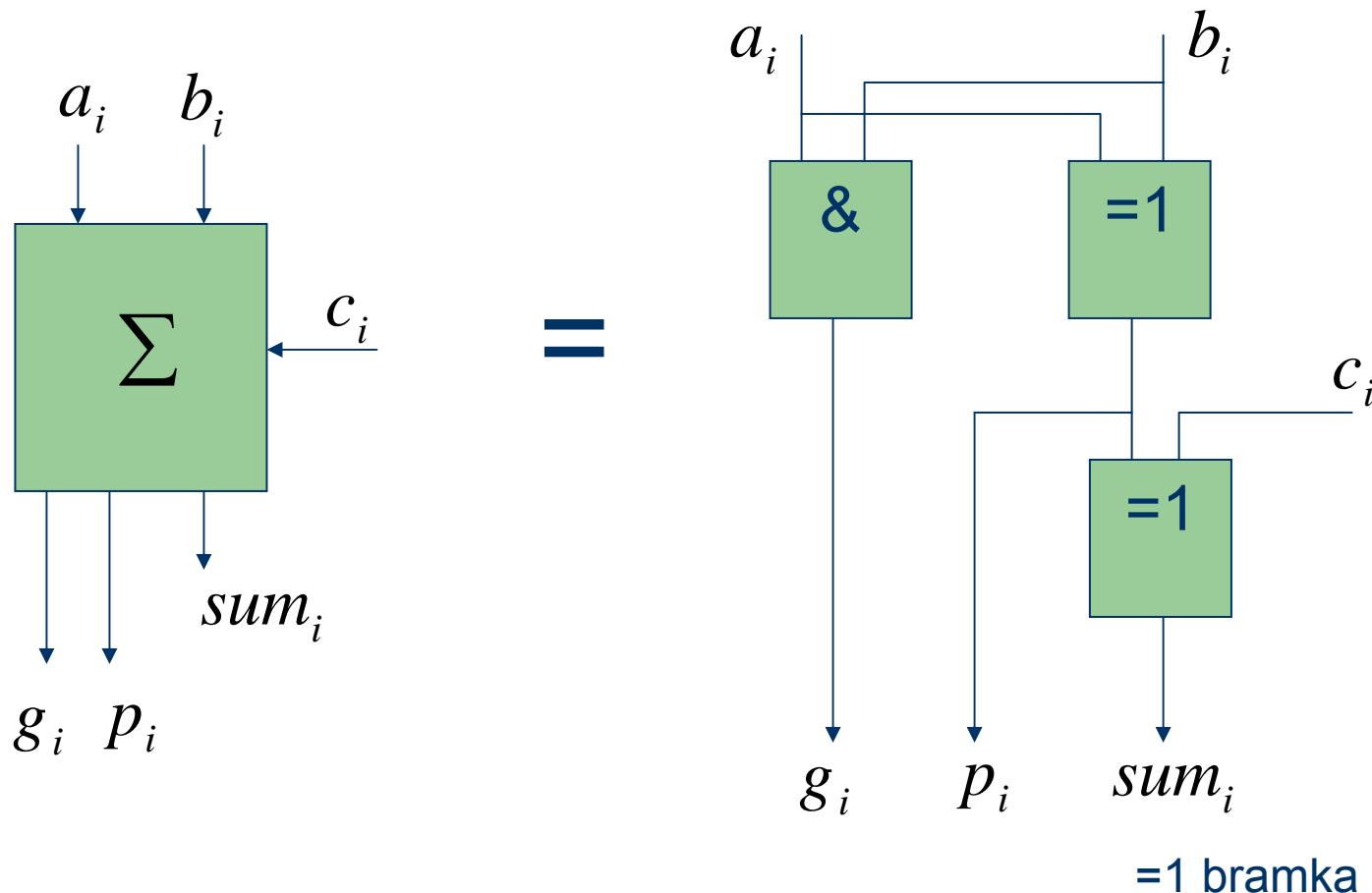
Sumator równoległy cd.

Równanie z poprzedniego slajdu określają metodę nazywaną **CLA** – *carry-look-ahead*, w której przeniesienia są generowane przez blok CLA:



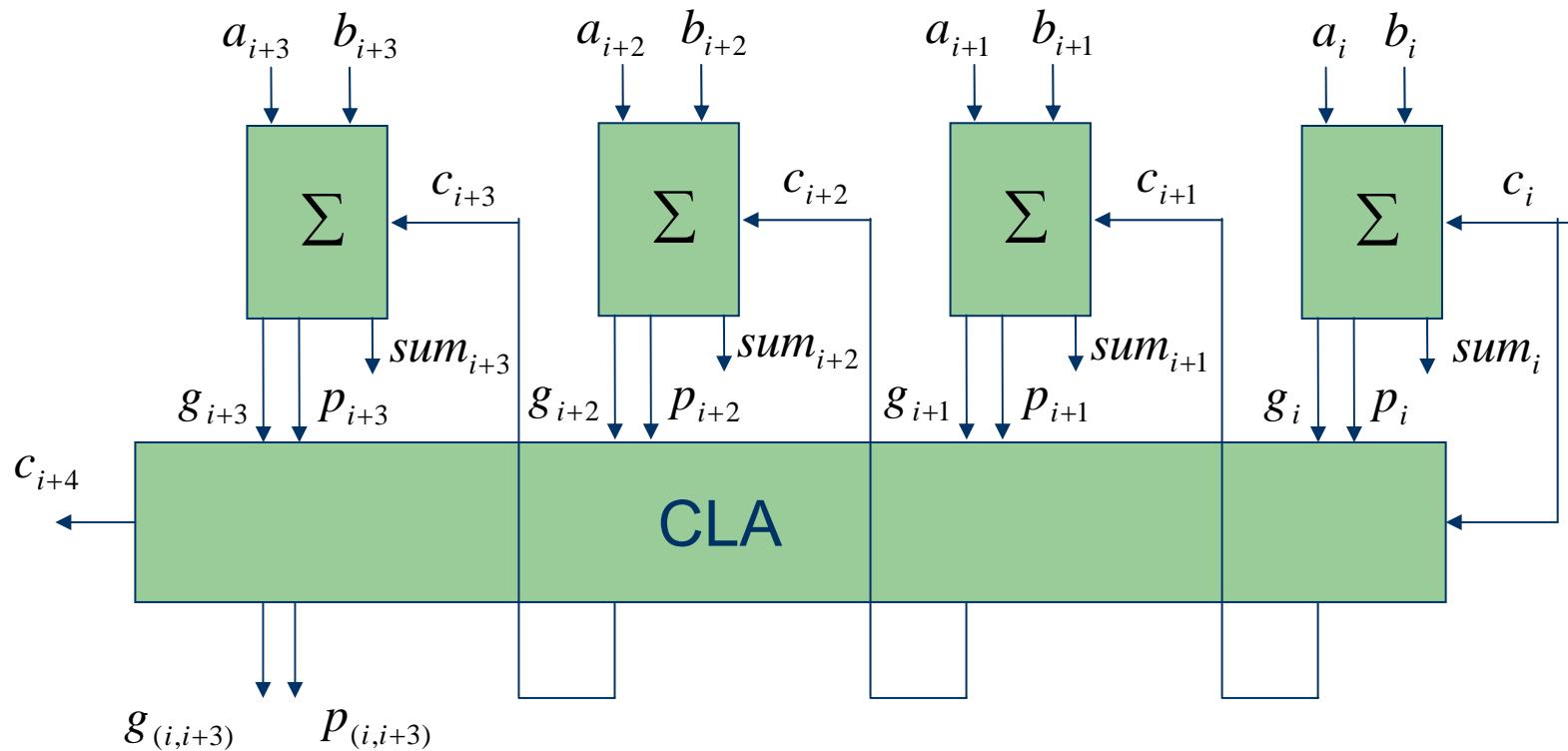
Sumator równoległy cd.

Budujemy z bramek zmodyfikowany sumator 1-bitowy:



Sumator równoległy cd.

4-bitowy sumator z równoległą generacją przeniesień



Sumator równoległy dokończenie

- Ustalanie wyniku oraz propagacja przeniesień w metodzie CLA są znacznie szybsze niż w metodzie *ripple-carry*:

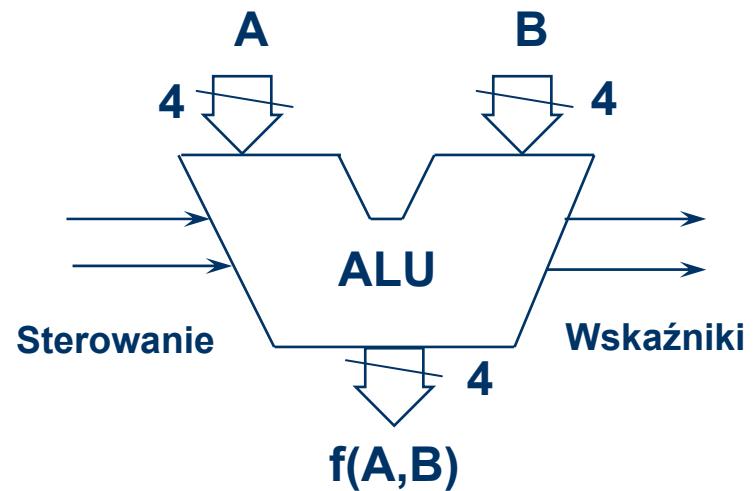
<u>propagacja</u>	ripple	CLA	
c_1 do c_{i+1}	4.8	4.8	czas propagacji w ns
c_1 do c_{i+2}	9.6	5.6	
c_1 do c_{i+3}	14.4	6.4	
c_1 do c_{i+4}	19.2	4.8	

- Używając 4-bitowego sumatora CLA z poprzedniego slajdu można budować sumatory 8-, 12-, 16-, bądź ogólnie $4 \times n$ bitowe
- Sumatory CLA można łączyć tak, by przeniesienia między nimi propagowały szeregowo (*ripple*), lub wprowadzić układ CLA drugiego poziomu, zbudowany na takiej samej zasadzie jak układ CLA pierwszego poziomu

Budujemy ALU

- Założenia projektowe

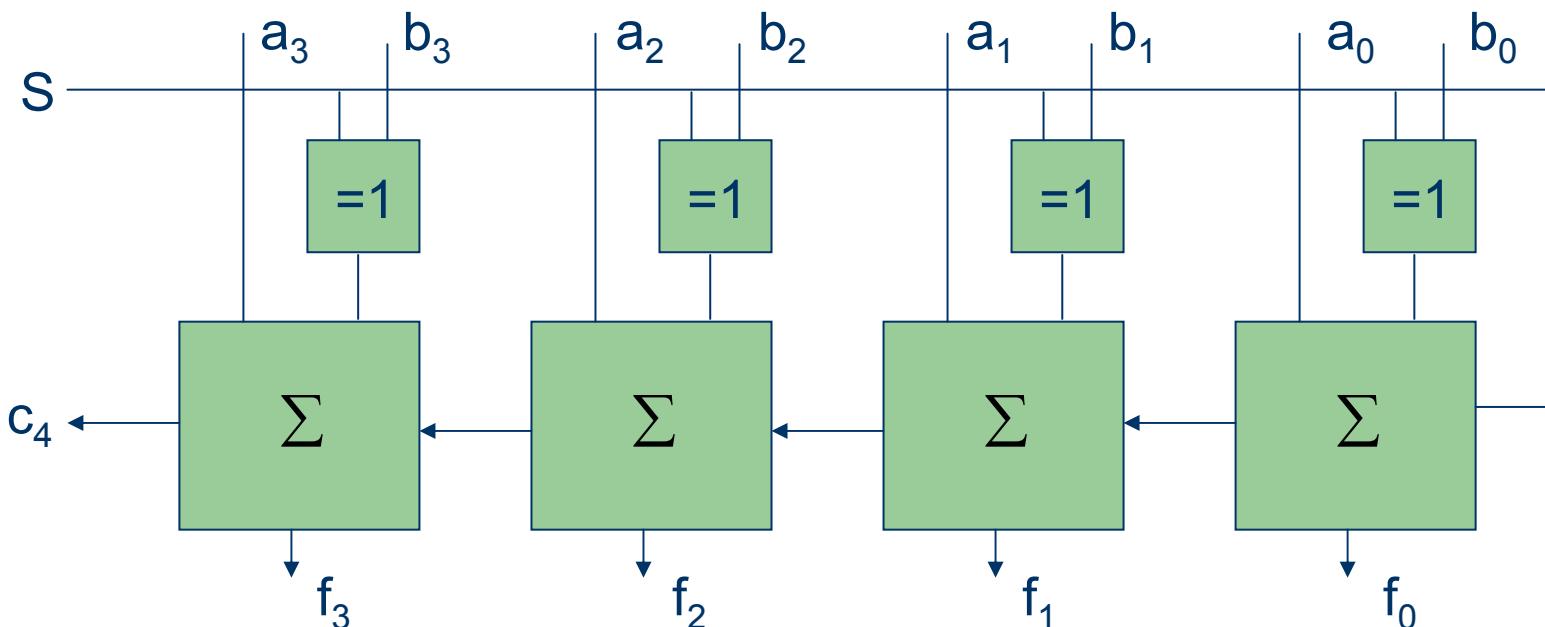
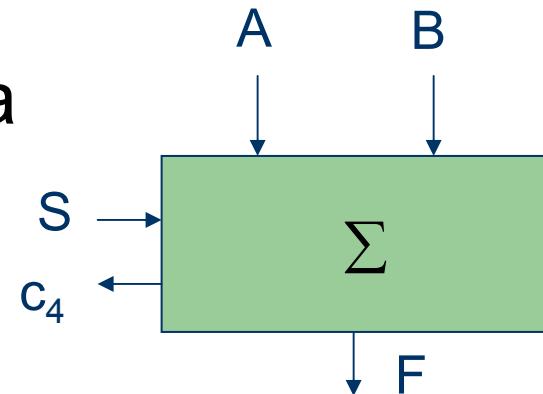
- 4-bitowe argumenty A i B
- 4-bitowy wynik operacji
- Operacje arytmetyczne
 - dekrementacja A
 - inkrementacja A
 - dodawanie $A+B$
 - odejmowanie $A-B$
- Operacje logiczne
 - negacja A (uzupełnienie do 1)
 - iloczyn logiczny A and B
 - tożsamość A
 - suma logiczna A or B
- Wskaźniki: przeniesienie i nadmiar



Krok 1

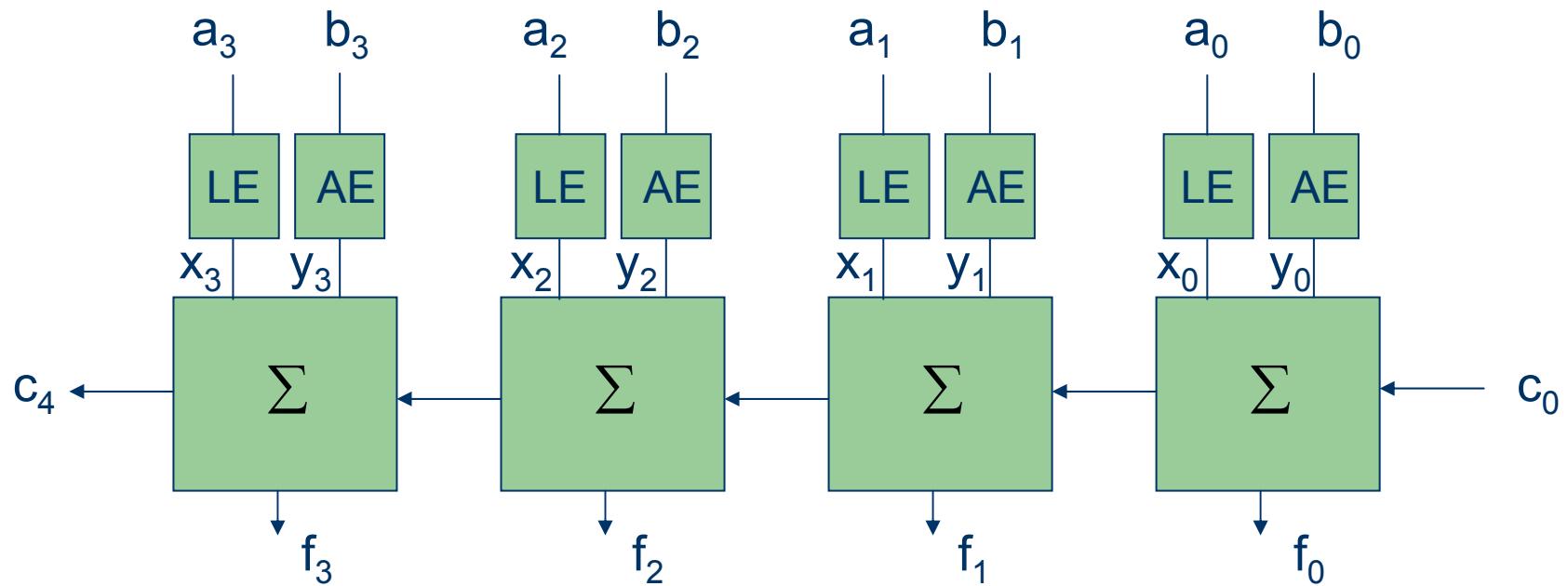
- Sumator z funkcją odejmowania

S	Funkcja	Opis
0	$A+B$	Dodawanie
1	$A+B'+1$	Odejmowanie



Krok 2

- Wniosek z kroku 1: sumator z ekstenderem może realizować więcej funkcji niż samo dodawanie
- Ogólna koncepcja ALU:



LE – *logic extender*, AE – *arithmetic extender*

Krok 3 – projekt AE

M	S ₁	S ₀	Funkcja	F	X	Y	c ₀
1	0	0	dekrementacja	A-1	A	1...11	0
1	0	1	dodawanie	A+B	A	B	0
1	1	0	odejmowanie	A+B'+1	A	B'	1
1	1	1	inkrementacja	A+1	A	0...00	1

M	S ₁	S ₀	b _i	y _i
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Tablica prawdy

M wybór funkcji: 1 – arytmetyczne,
0 – logiczne

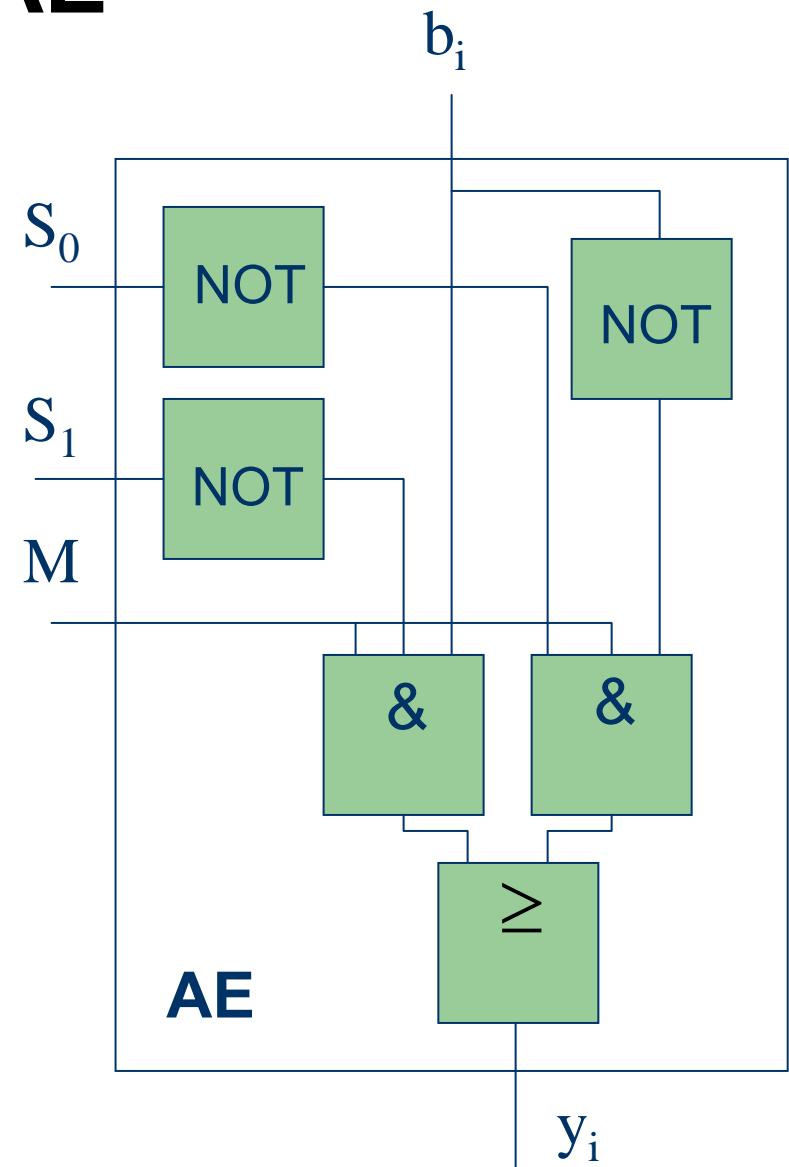
S₁, S₀ wybór operacji

y_i stan na wyjściu *i*-tego bloku AE

Krok 4 – realizacja AE

b_i	$S_1 S_0$	00	01	11	10
0		1			1
1		1	1		

$$y_i = M \bar{S}_1 b_i + M \bar{S}_0 \bar{b}_i$$



Krok 5 – projekt LE

M	S ₁	S ₀	Funkcja	F	X	Y	c ₀
0	0	0	negacja bitowa	A'	A'	0	0
0	0	1	iloczyn logiczny	A and B	A and B	0	0
0	1	0	tożsamość	A	A	0	0
0	1	1	suma logiczna	A or B	A or B	0	0

M	S ₁	S ₀	x _i
0	0	0	a' _i
0	0	1	a _i b _i
0	1	0	a _i
0	1	1	a _i +b _i
1	x	x	a _i

Tablica prawdy

M wybór funkcji: 1 – arytmetyczne,
 0 – logiczne

S₁, S₀ wybór operacji

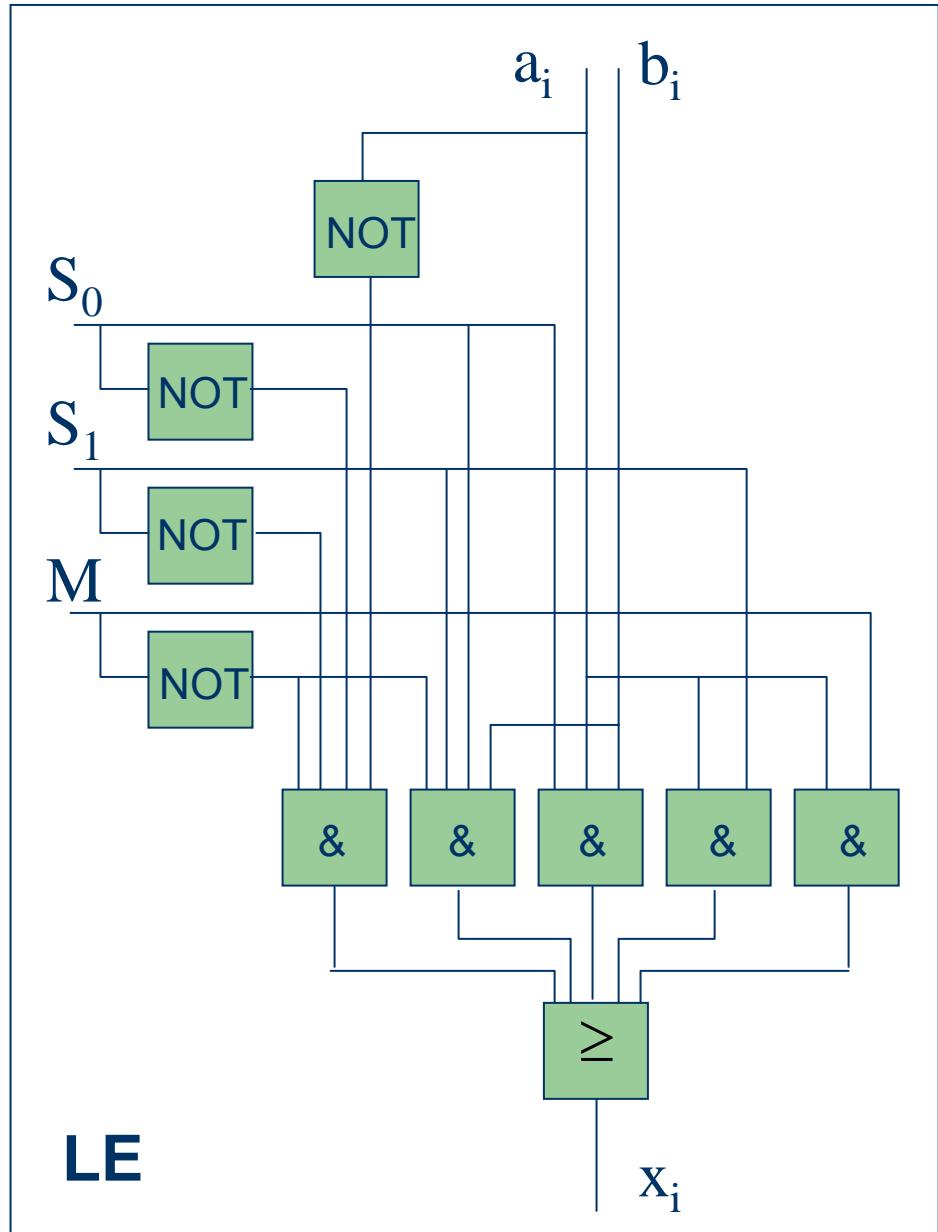
x_i stan na wyjściu i-tego bloku LE

Krok 6 – LE

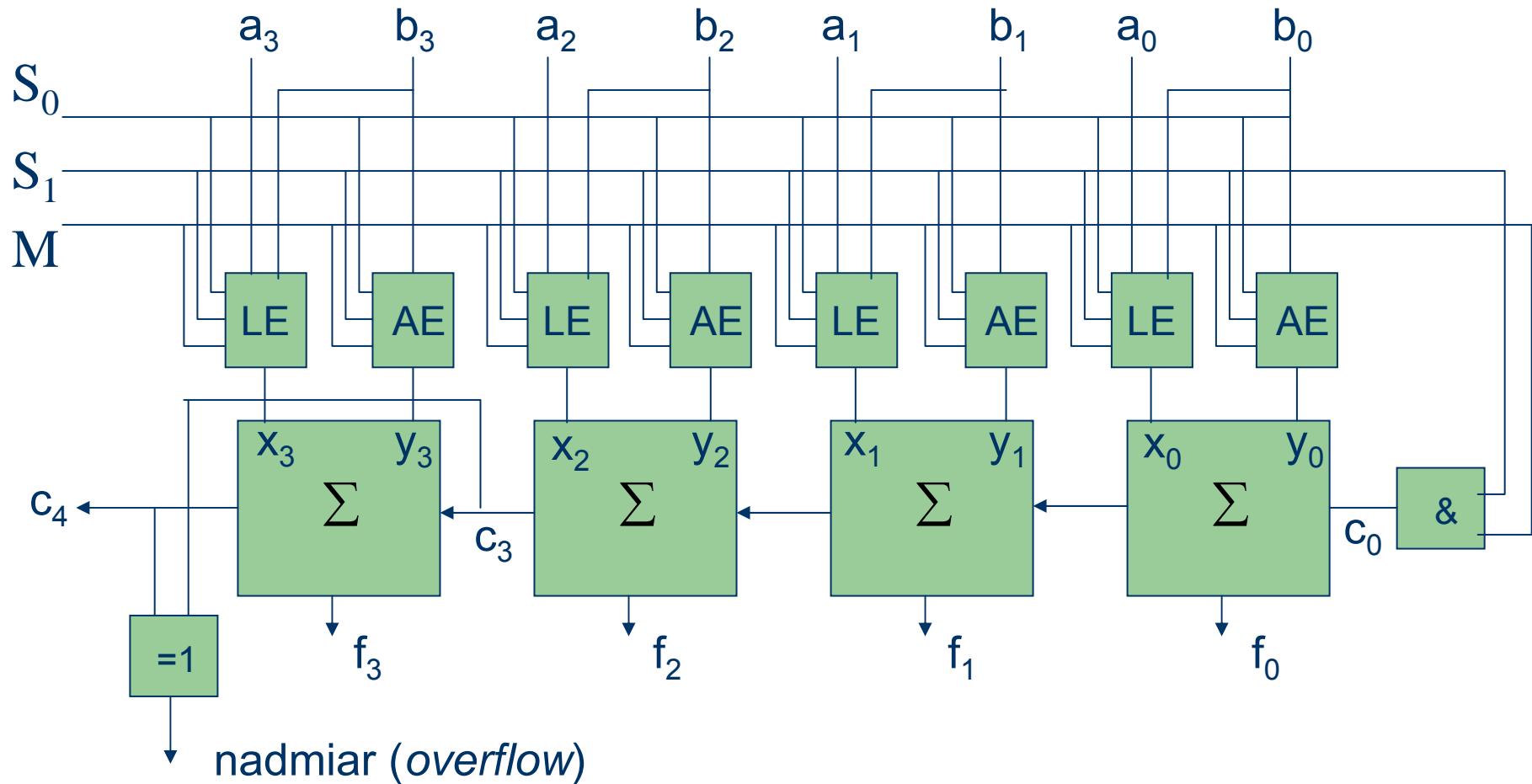
Ćwiczenie domowe:

zbudować mapę Karnaugh'a dla tablicy prawdy LE z poprzedniego slajdu i sprawdzić, że:

$$x_i = \overline{M} \overline{S_1} \overline{S_0} a_i + \overline{M} S_1 S_0 b_i + \\ + S_0 a_i b_i + S_1 a_1 + M a_i$$

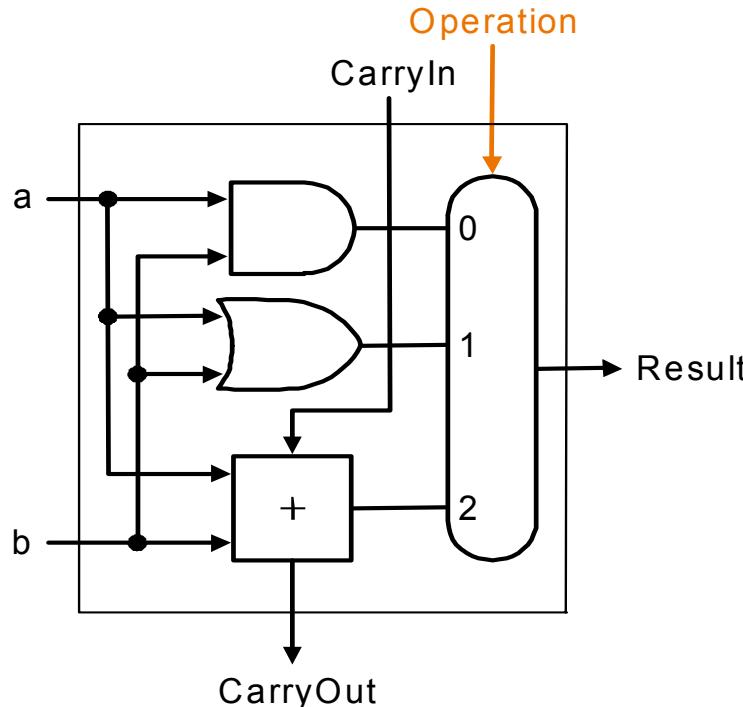


Krok 7 – projekt końcowy

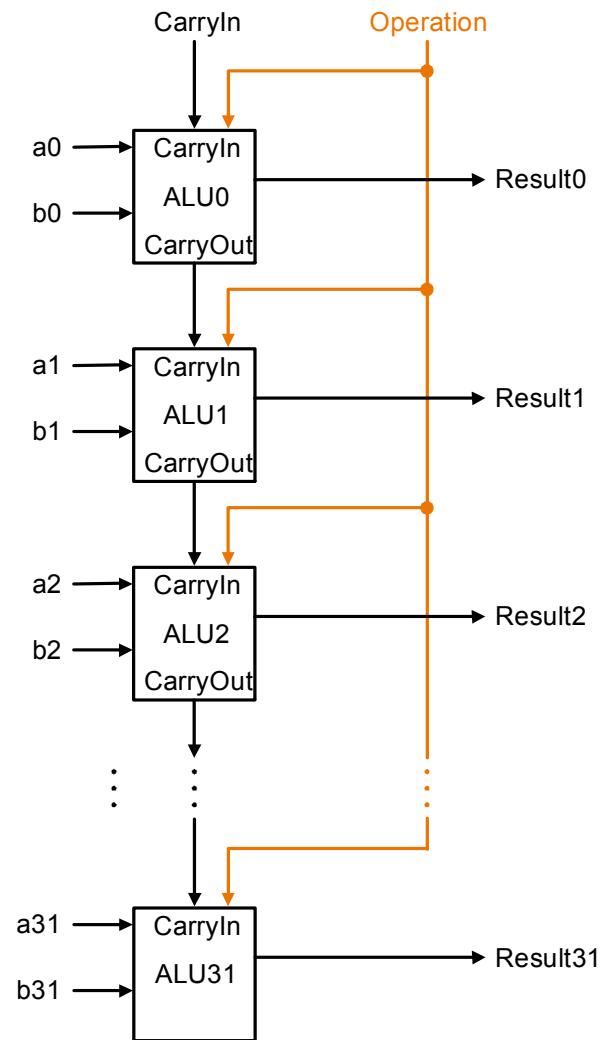


Rozwiążanie alternatywne ALU

Komórka ALU zbudowana przy użyciu multipleksera oraz schemat 32-bitowego ALU

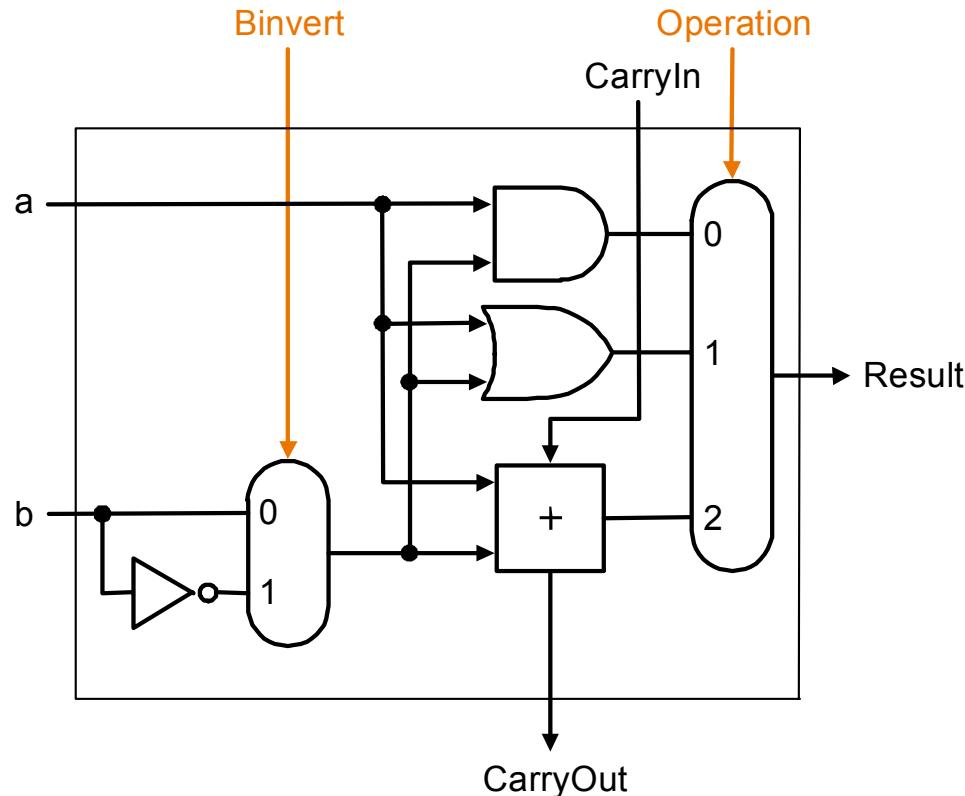


Operacje: +, and, or



Rozwiązańe alternatywne cd.

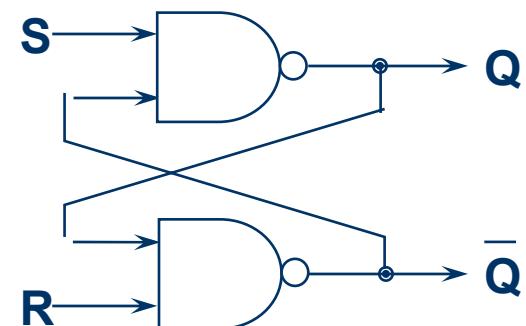
Komórka ALU wzbogacona o funkcję odejmowania



Elementy i bloki sekwencyjne

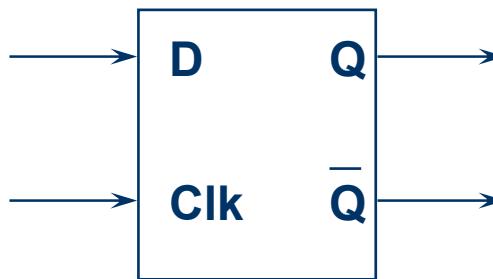
- Przerzutnik – podstawowy element sekwencyjnych bloków funkcjonalnych komputerów

S	R	Q	\bar{Q}
0	1	1	0
1	0	0	1
1	1	stan niedozwol.	
0	0	stan niedozwol.	

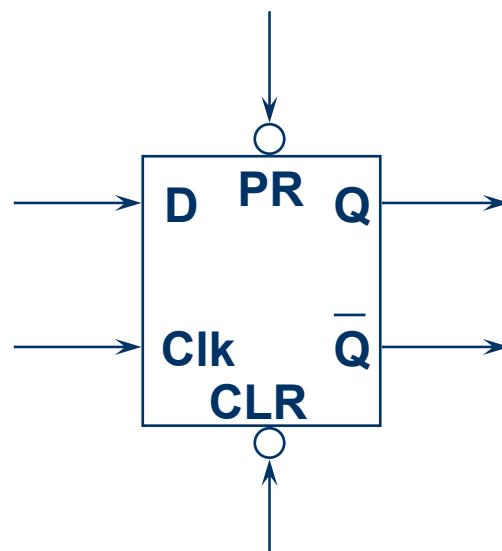


Przerzutniki cd.

Przerzutnik D – schemat logiczny i tablica stanów



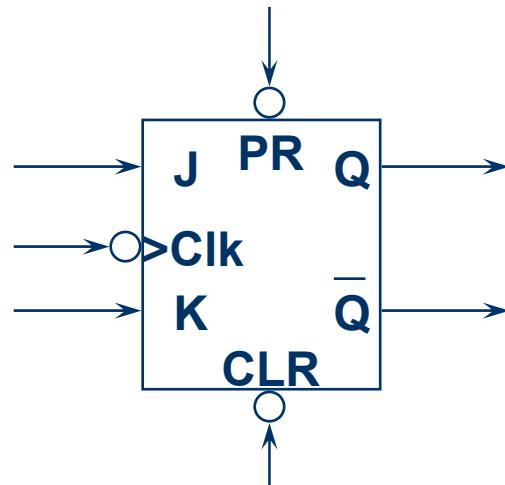
D	Clock	Q
0	0	0
1	1	1



Przerzutnik D
z asynchronicznymi
wejściami do ustawiania
(PR – preset) i zerowania
(CLR – clear)

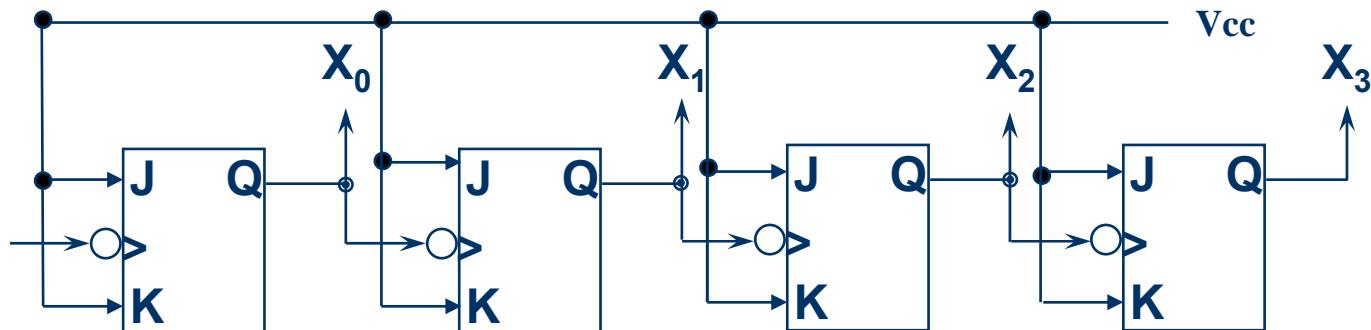
Przerzutniki cd.

Przerzutnik J-K – schemat logiczny

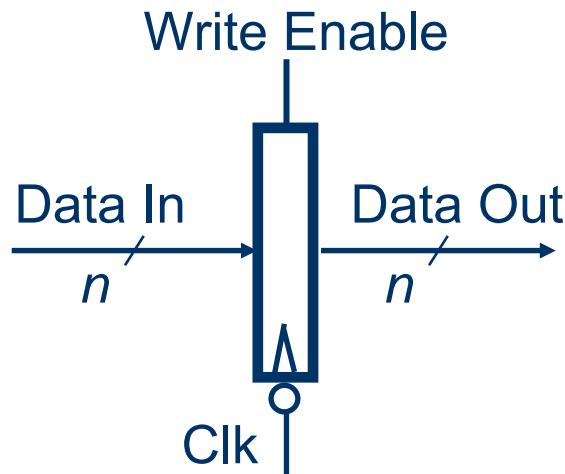


J	K	Clock	Q
0	0	0	bez zmian
0	1	1	0
1	0	1	1
1	1	1	stan przeciwny

4-bitowy licznik binarny z przeniesieniami szeregowymi



Rejestry



- Najczęściej zbudowany z zatrzaskowych przerzutników typu D (*latch*)
- Jeśli $WE = 1$, stan Data In zostaje wraz z opadającym zboczem zegara wpisany do rejestru i przeniesiony na wyjście Data Out

- W technice komputerowej najczęściej wykorzystuje się rejesty przesuwające, które oprócz zapamiętywania n -bitowego słowa mogą je przesuwać
- Najprostsze rejesty przesuwające przesuwają swoją zawartość tylko w jednym kierunku o 1 pozycję (1 bit)
- Bardziej złożone rejesty przesuwają dane w obu kierunkach (SL – *shift left*, SR – *shift right*)
- Na wyjściu ALU znajduje się zwykle szybki rejestr przesuwający (*barrel shifter*)

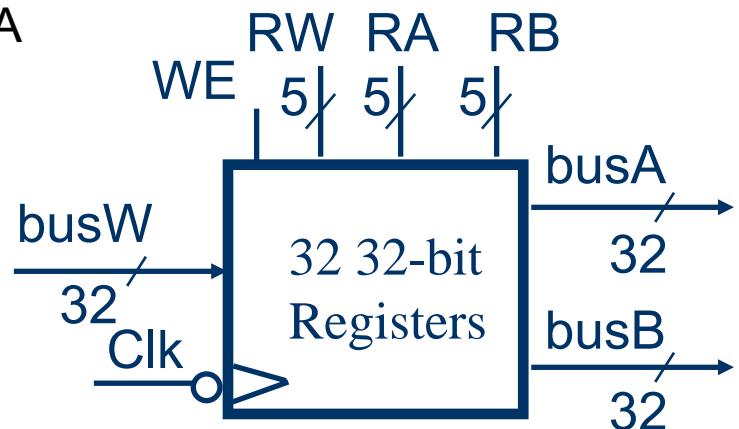
Plik (bank) rejestrów

- Przykładowy plik rejestrów pokazany obok składa się z 32 rejestrów:

- dwie 32-bitowe szyny wyjściowe busA i busB
- jedna 32-bitowa szyna wejściowa: busW
- sygnał zegara ma znaczenie tylko przy zapisie; przy odczycie układ działa jak układ kombinacyjny

- Rejestry są wybierane przez:

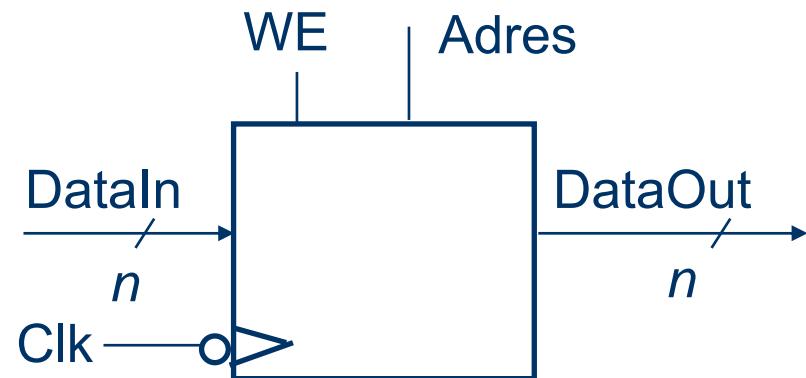
- RA – numer rejestru wyprowadzanego na szynę busA
- RB – numer rejestru wyprowadzanego na szynę busB
- RW – numer rejestru zapisywanygo z szyny busW jeśli WE=1



Przykład pliku rejestrów o organizacji
32 x 32 bity

Pamięć

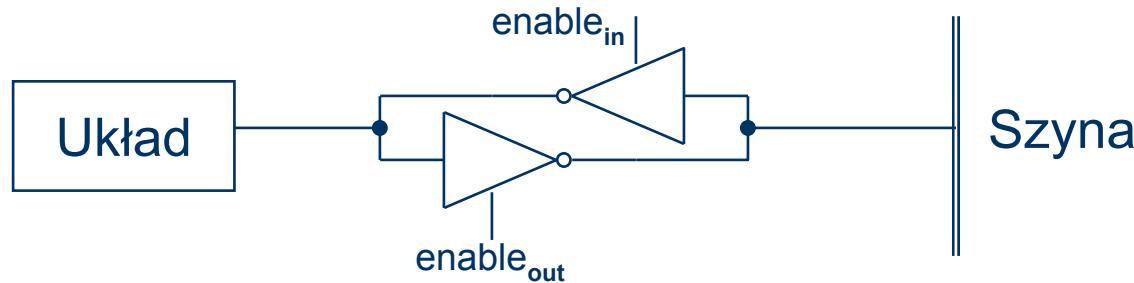
- Szyny
 - jedna szyna wejściowa: Data In
 - jedna szyna wyjściowa: Data Out
- Słowo n -bitowe zapisane w pamięci jest wybierane przez:
 - adres o długości zależnej od pojemności pamięci
 - sygnał WE = 1
- Sygnał zegara Clock (CLK)
 - sygnał CLK ma znaczenie tylko przy zapisie
 - przy odczycie układ zachowuje się jak kombinacyjny



Schemat logiczny bloku pamięci n -bitowej (wyidealizowany)

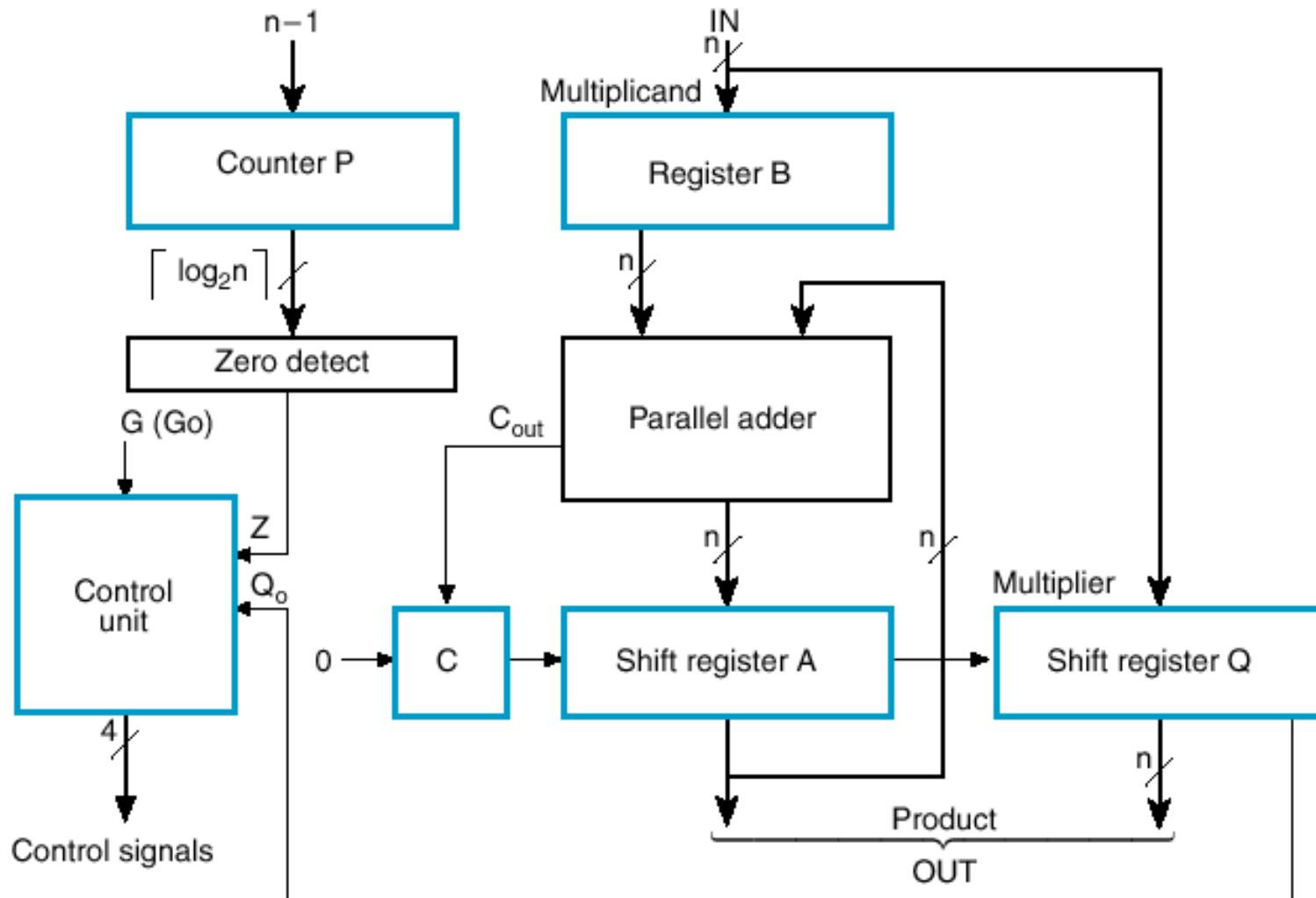
Elementy trzystanowe

- W systemach komputerowych często zachodzi potrzeba łączenia wyjść różnych układów cyfrowych na wspólnej szynie. Wygodnym rozwiązaniem są wyjścia trzystanowe. W trzecim stanie (wysokiej impedancji) wyjście stanowi wysoką impedancję umożliwiającą traktowanie połączenia jako rozwarcie.



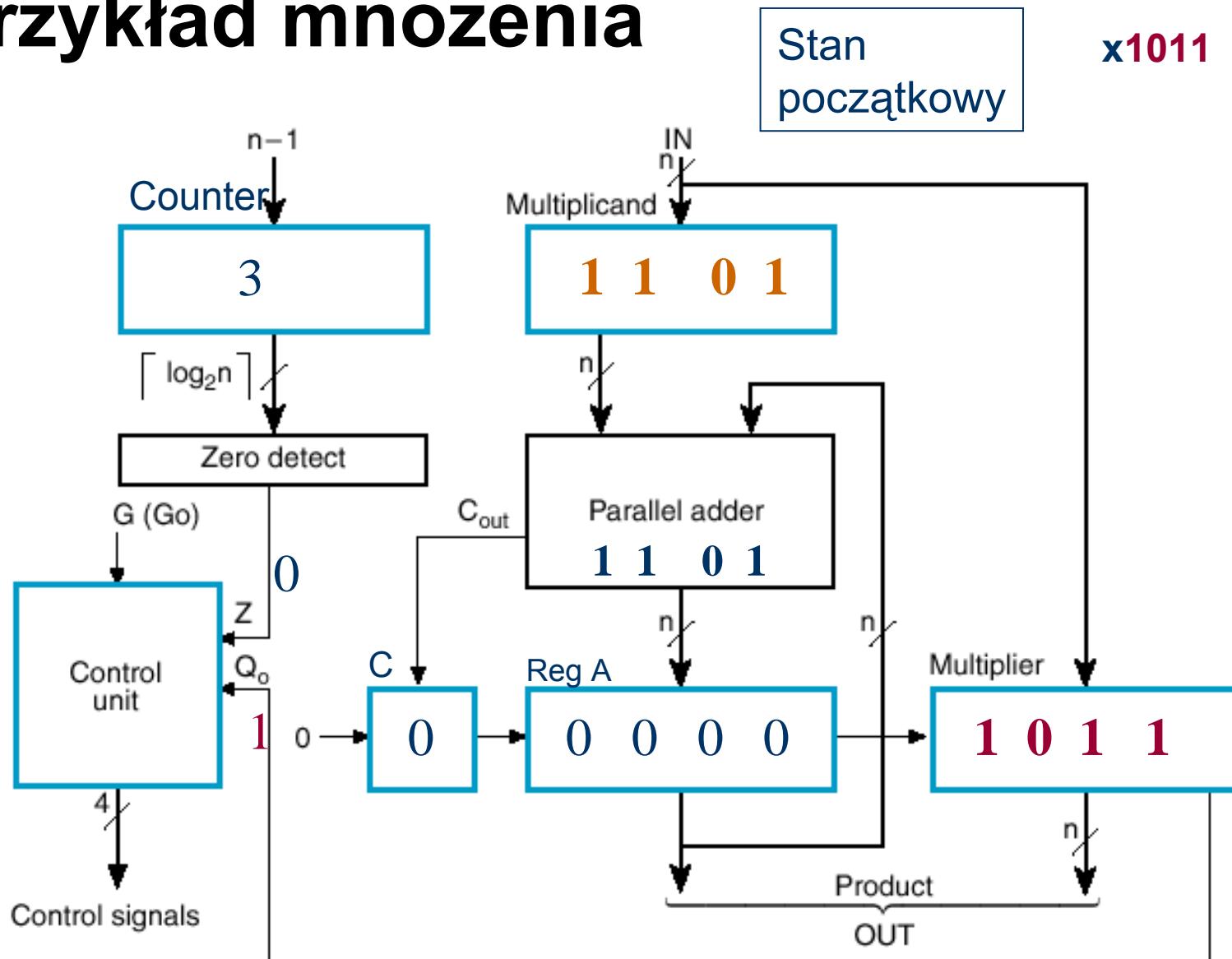
- Inwertery i bufory z wyjściem trzystanowym są często używane do połączenia układów z szynami (*bus drivers*)
 - dwa driverzy umożliwiają realizację połączenia dwukierunkowego
 - dodatkowym zadaniem driverów jest wzmacnianie sygnału, dzięki czemu szyny mogą być dłuższe

Multiplikator

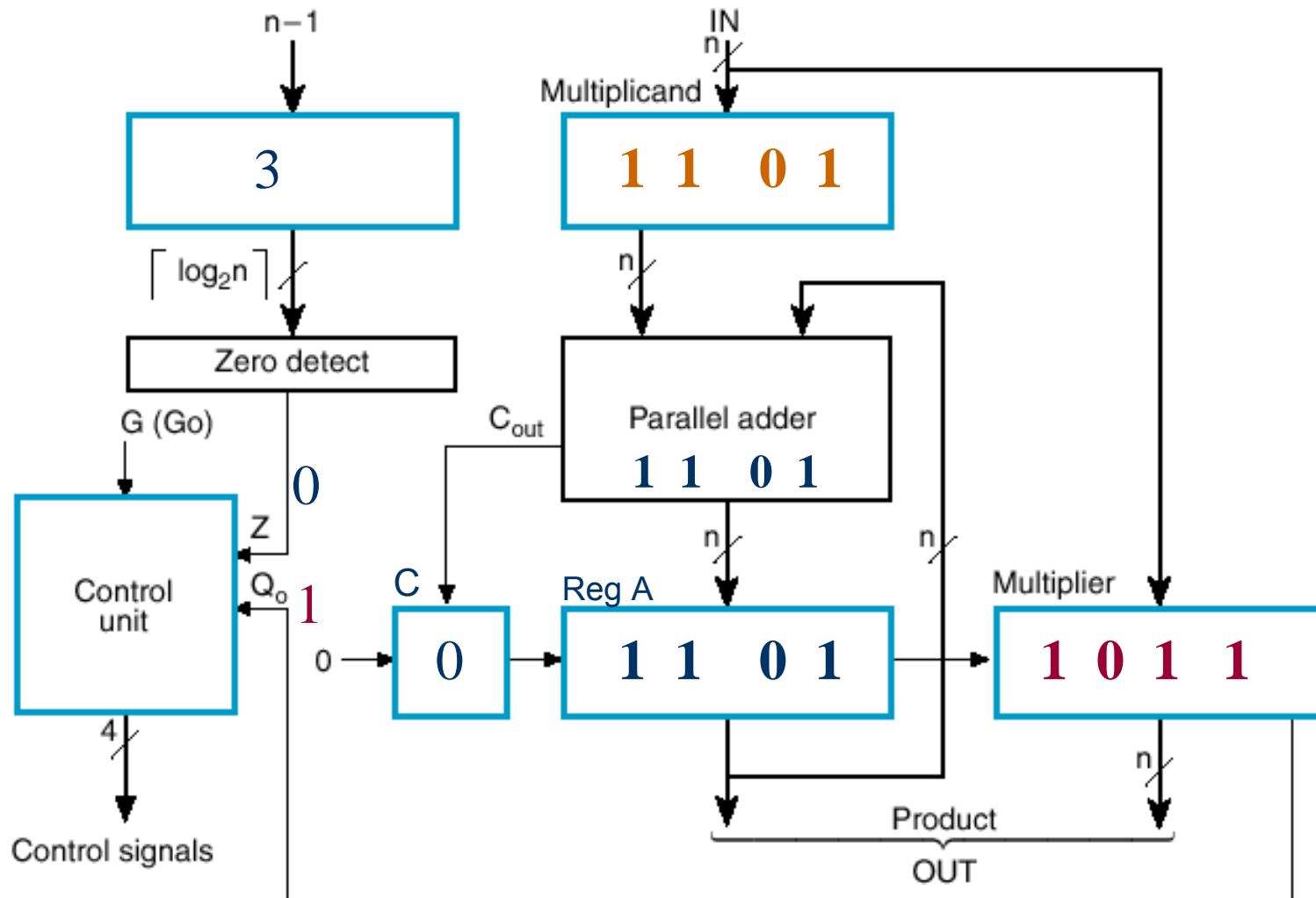


Przykład mnożenia

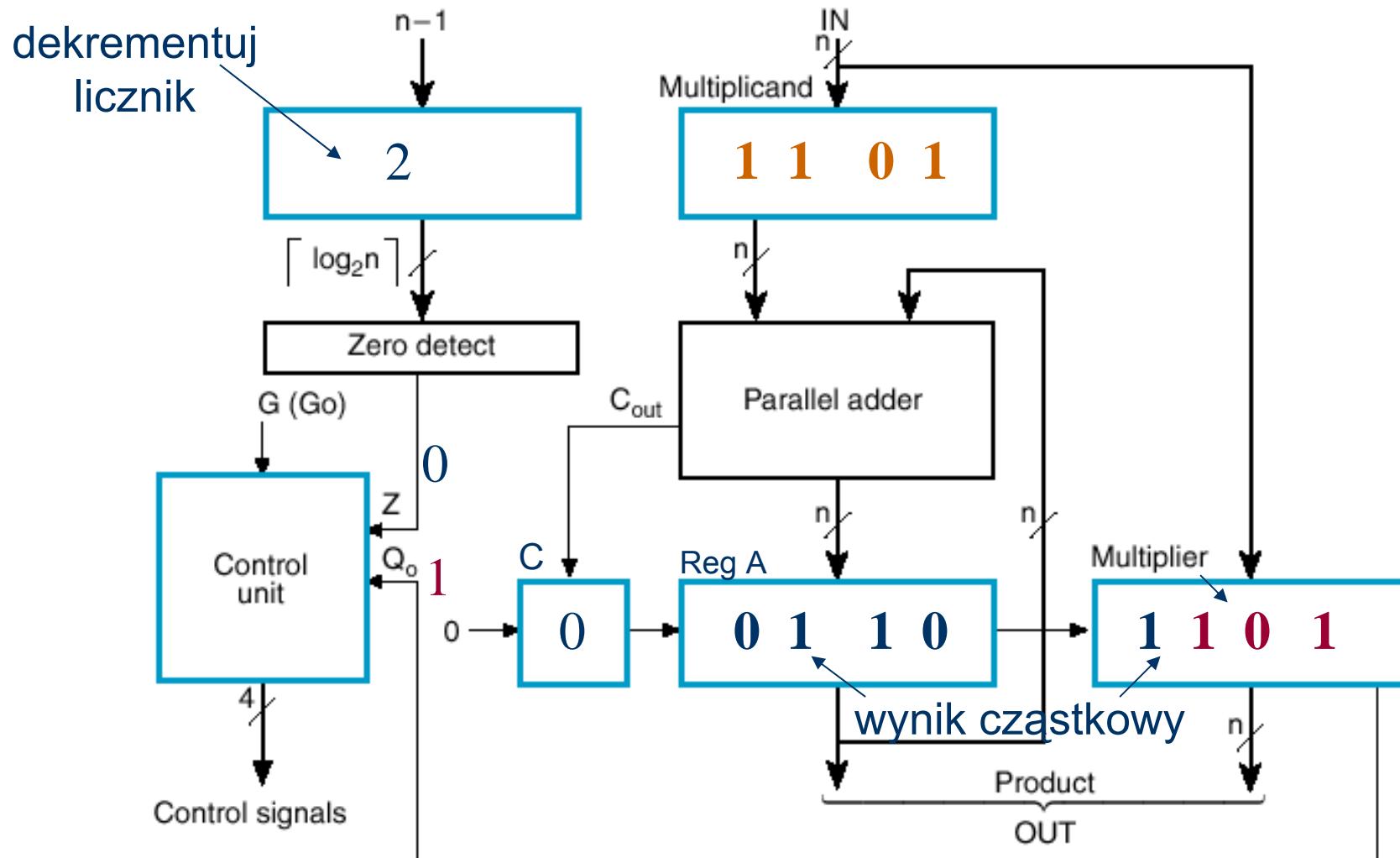
1101 (13)
x1011 (11)



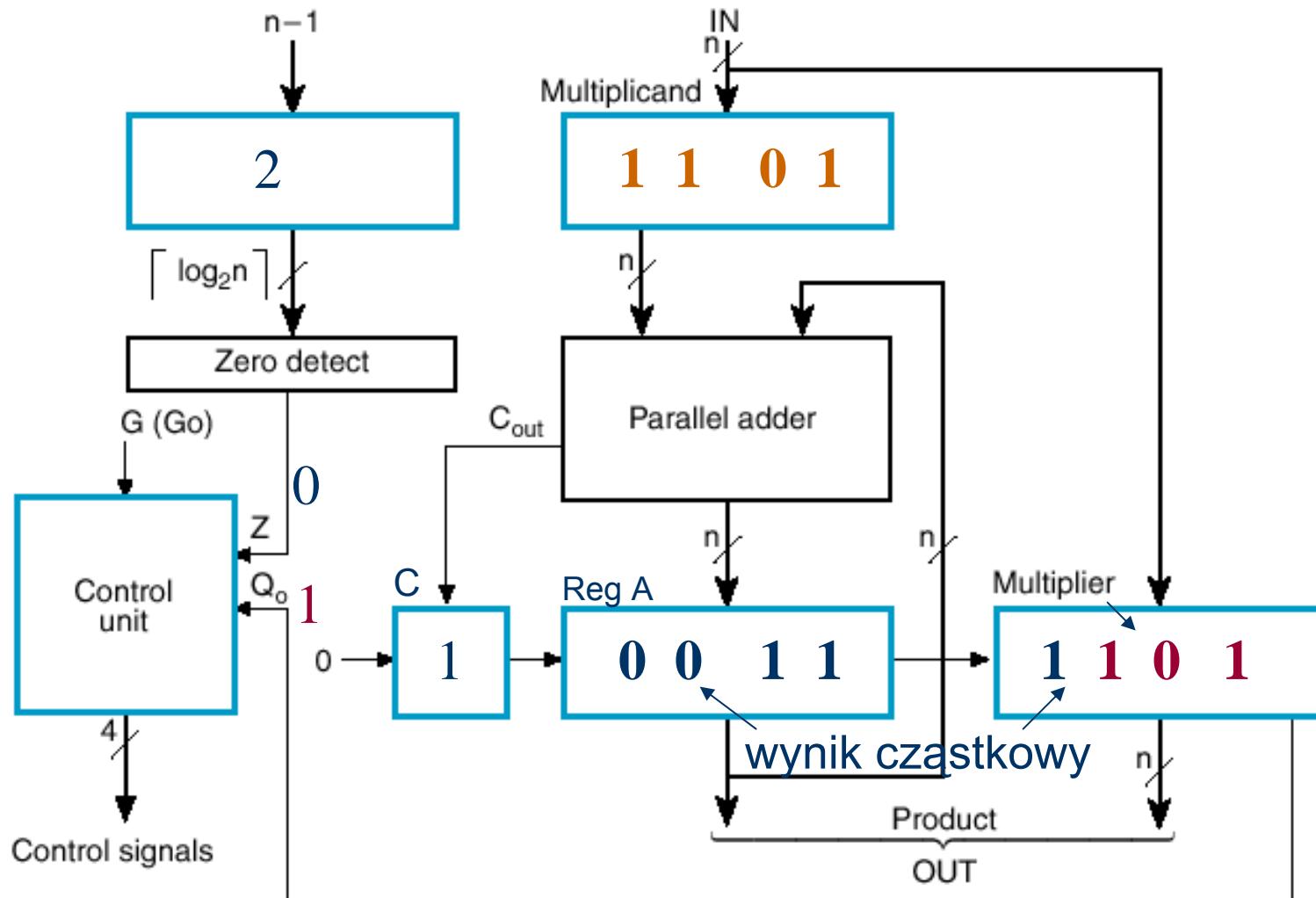
Qo =1: dodaj mnożną



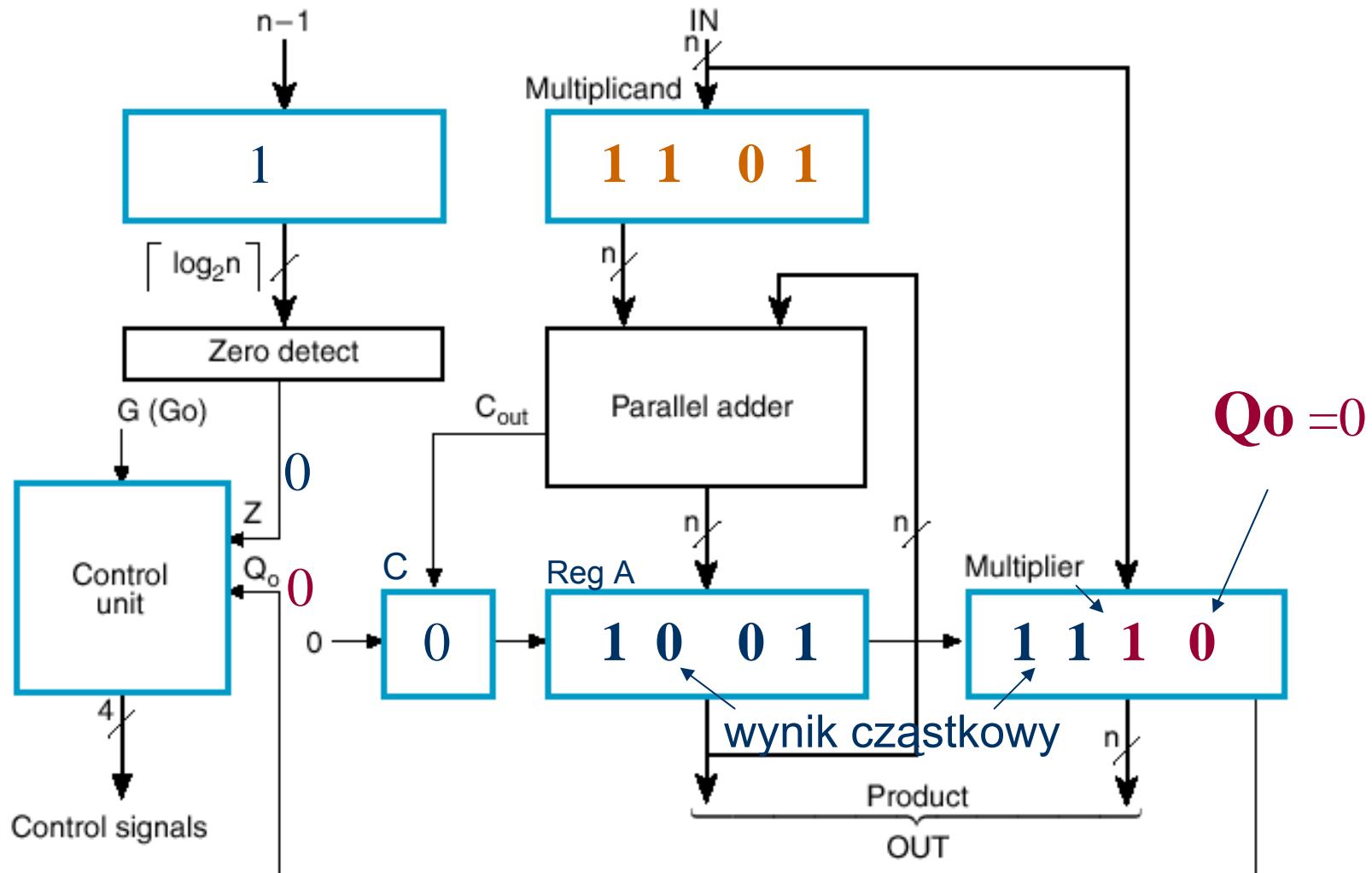
Przesuń wynik cząstkowy



Qo =1: dodaj mnożną

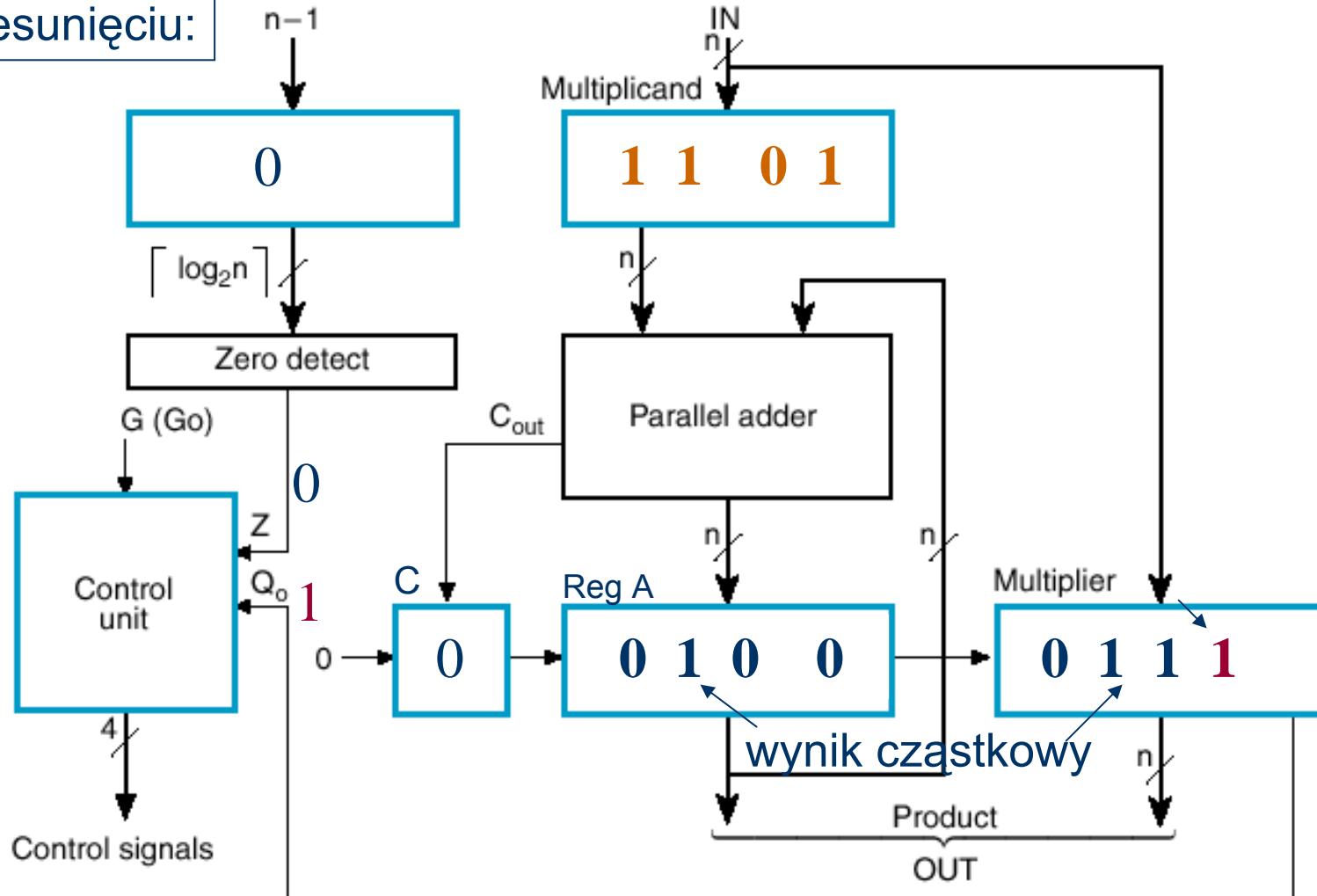


Przesuń wynik cząstkowy

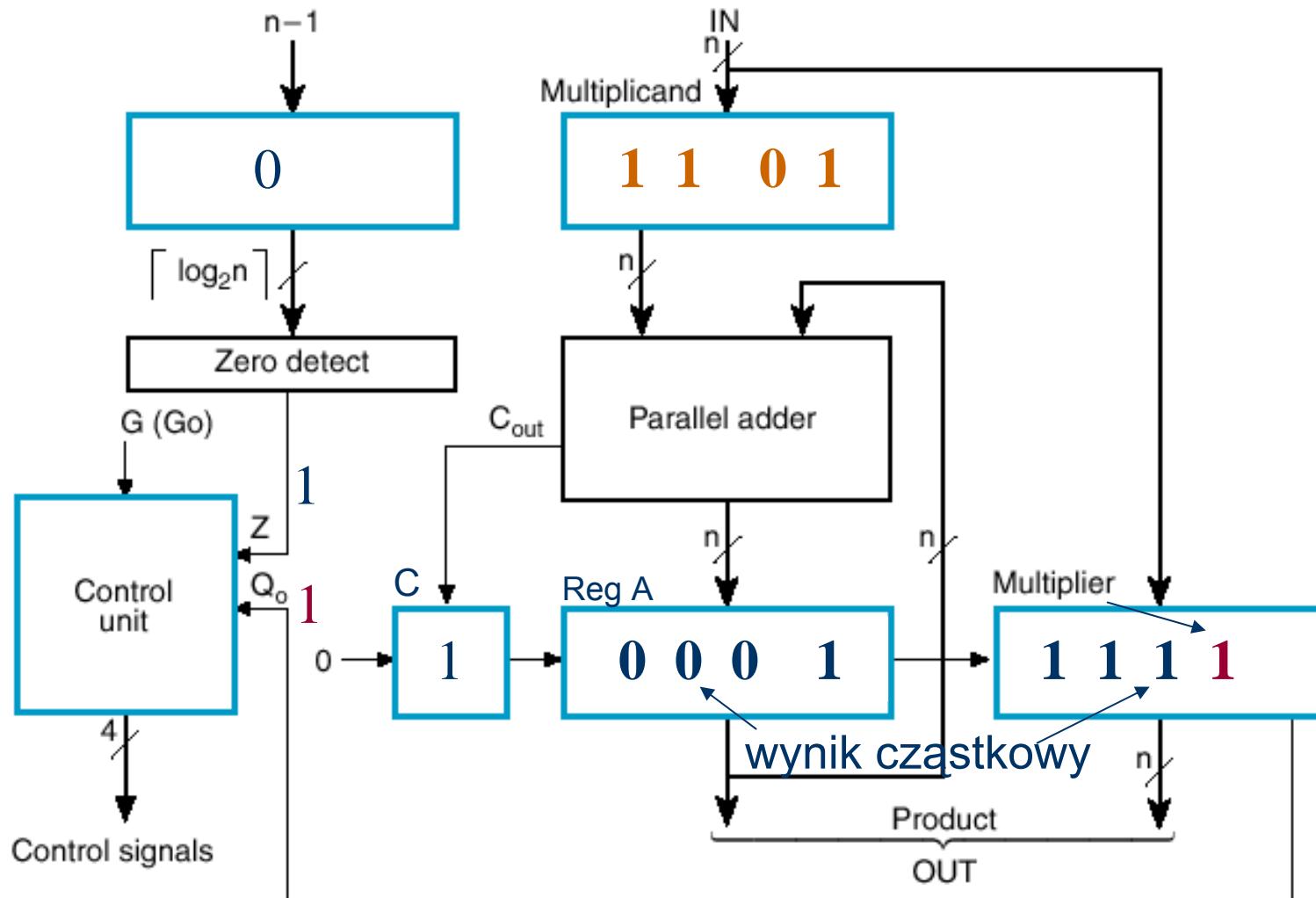


Qo=0: przesuń (nie dodawaj)

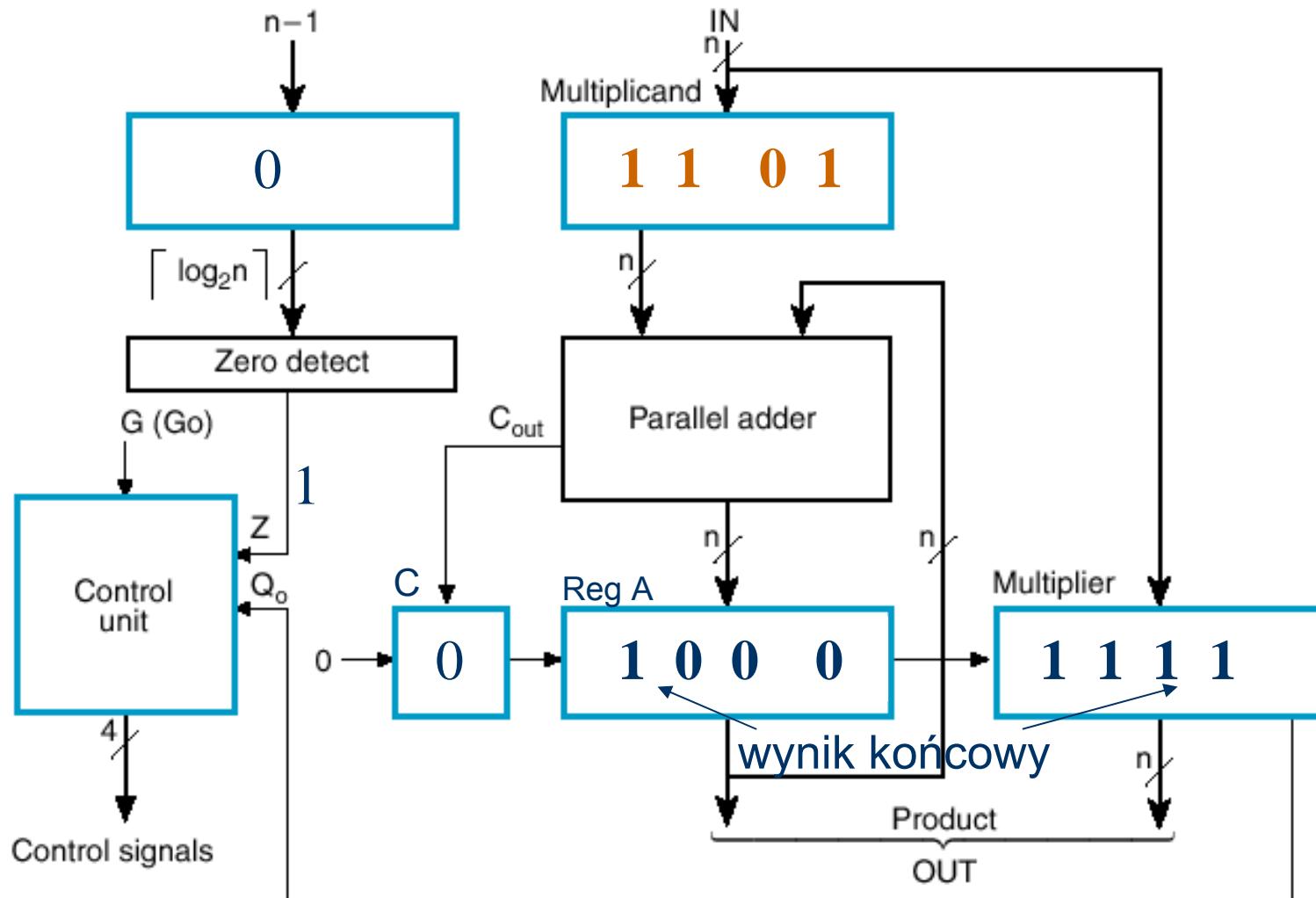
po przesunięciu:



Qo=1: dodaj mnożną



Przesuń (koniec, Z=1)



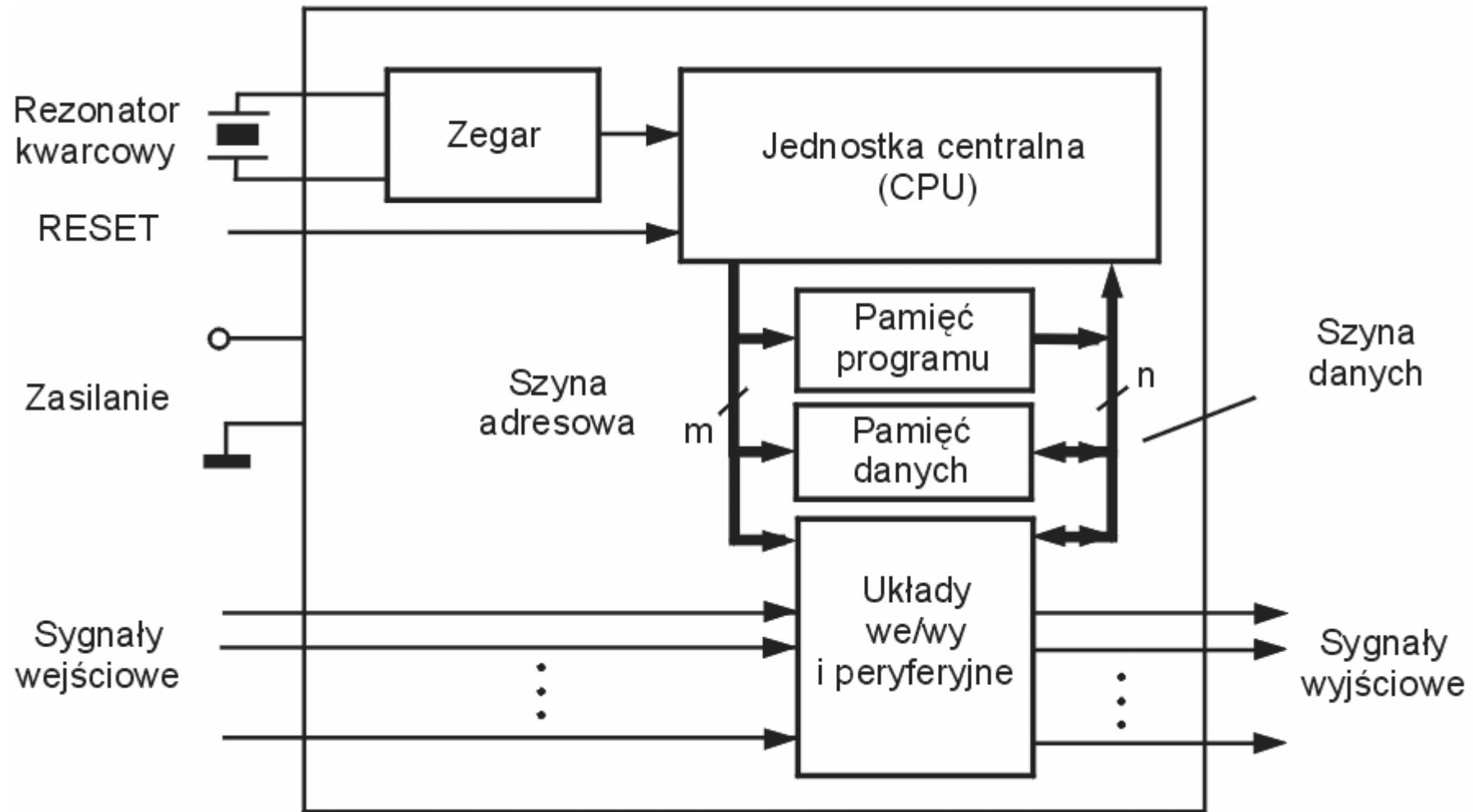
Podsumowanie

- Elementy i bloki kombinacyjne
 - bramki, bufory (w tym trzystanowe)
 - kodery i dekodery
 - multipleksery
- Synteza półsumatora i sumatora (*look-ahead*)
- Synteza ALU
- Elementy i bloki sekwencyjne
 - przerzutniki
 - liczniki
 - rejestrzy
 - pliki rejestrów
 - pamięci
- Metody opisu bloków cyfrowych (język VHDL)
- Synteza układu mnożenia

Organizacja i Architektura Komputerów

Struktura i działanie jednostki centralnej

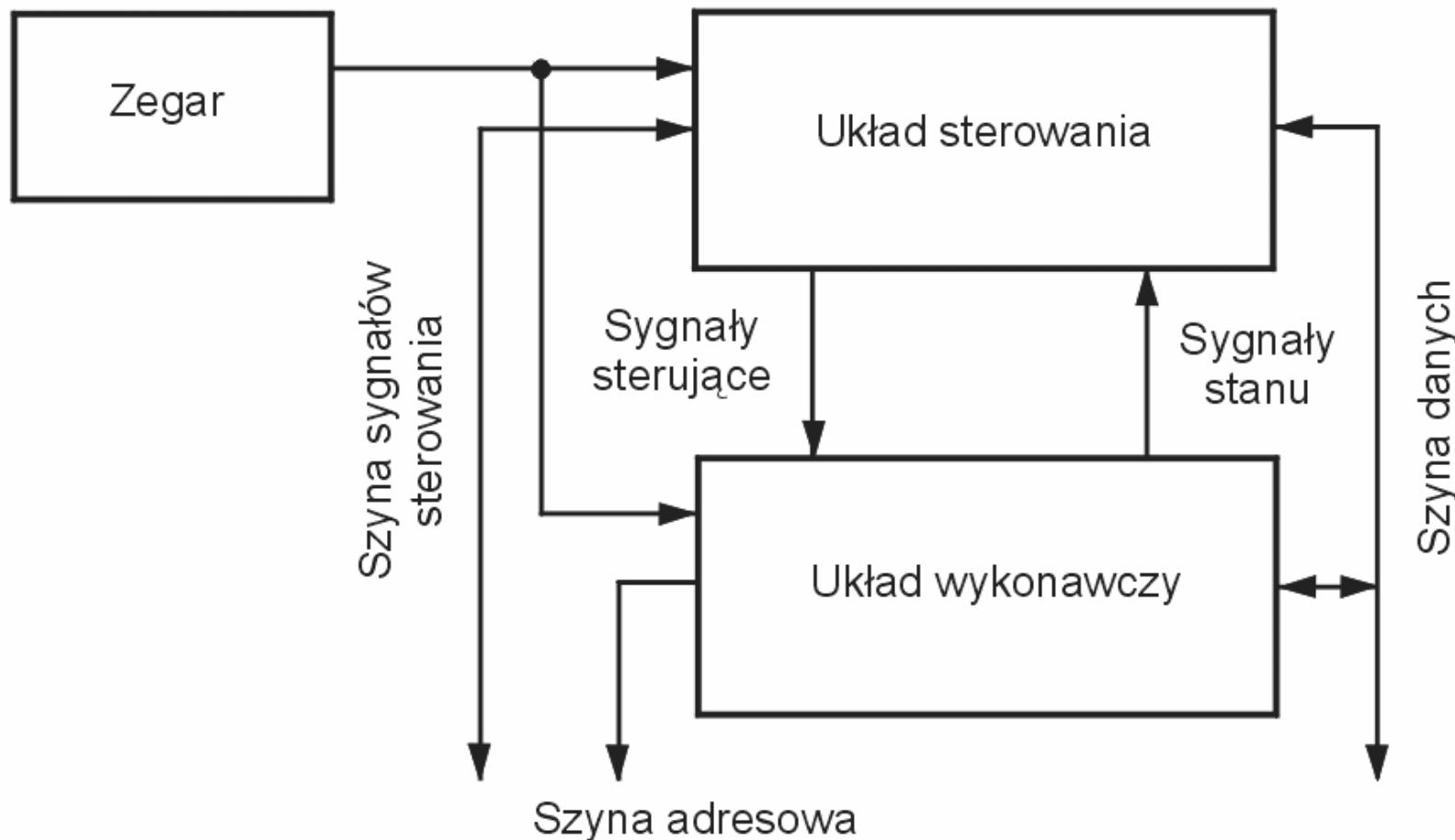
Budowa mikrokomputera



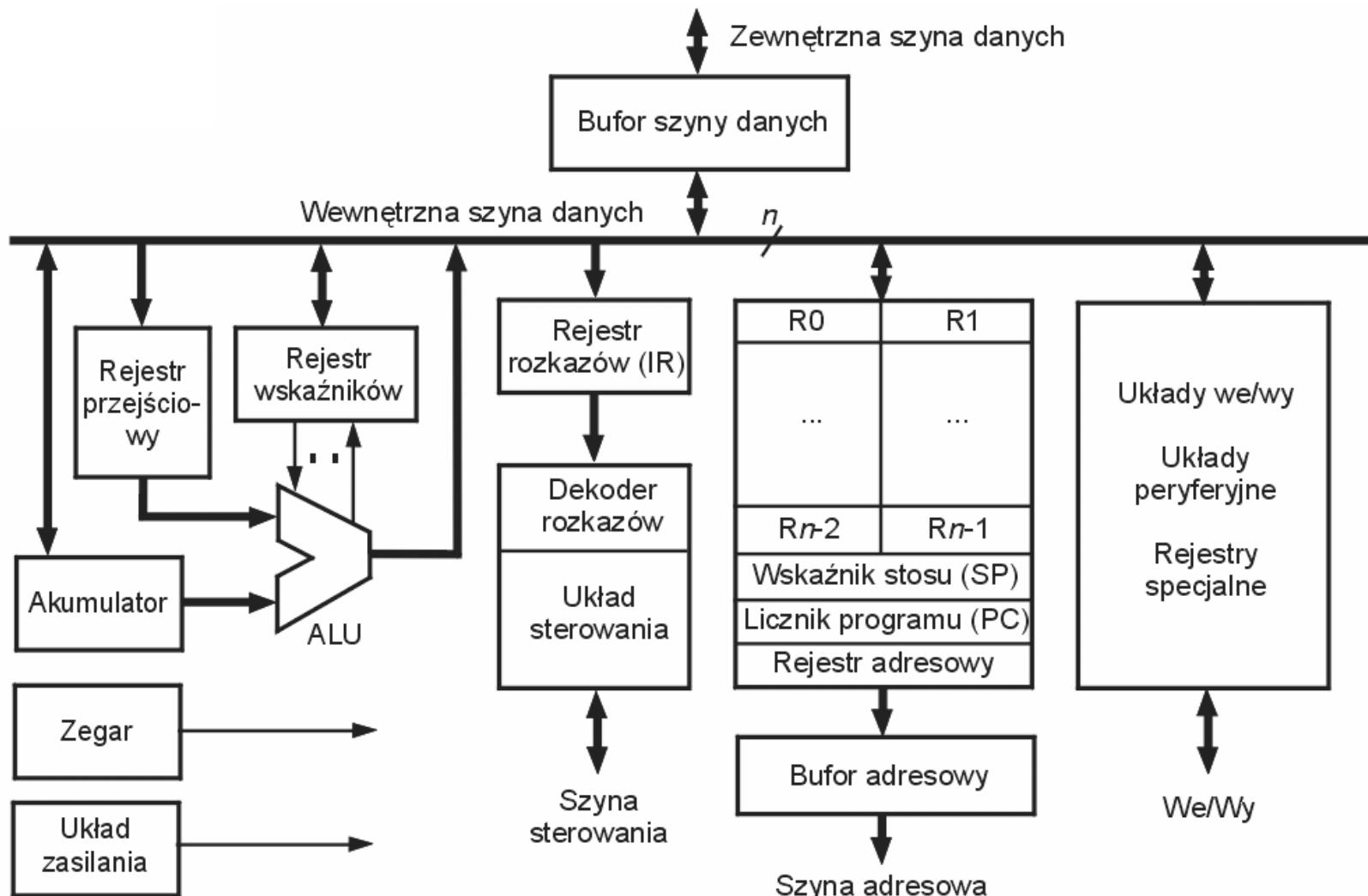
Zadania CPU

- CPU wykonuje następujące zadania:
 - pobiera instrukcje programu z pamięci (*instruction fetch*)
 - interpretuje (dekoduje) instrukcje
 - pobiera dane (*data fetch*)
 - przetwarza dane
 - zapisuje dane

Struktura CPU

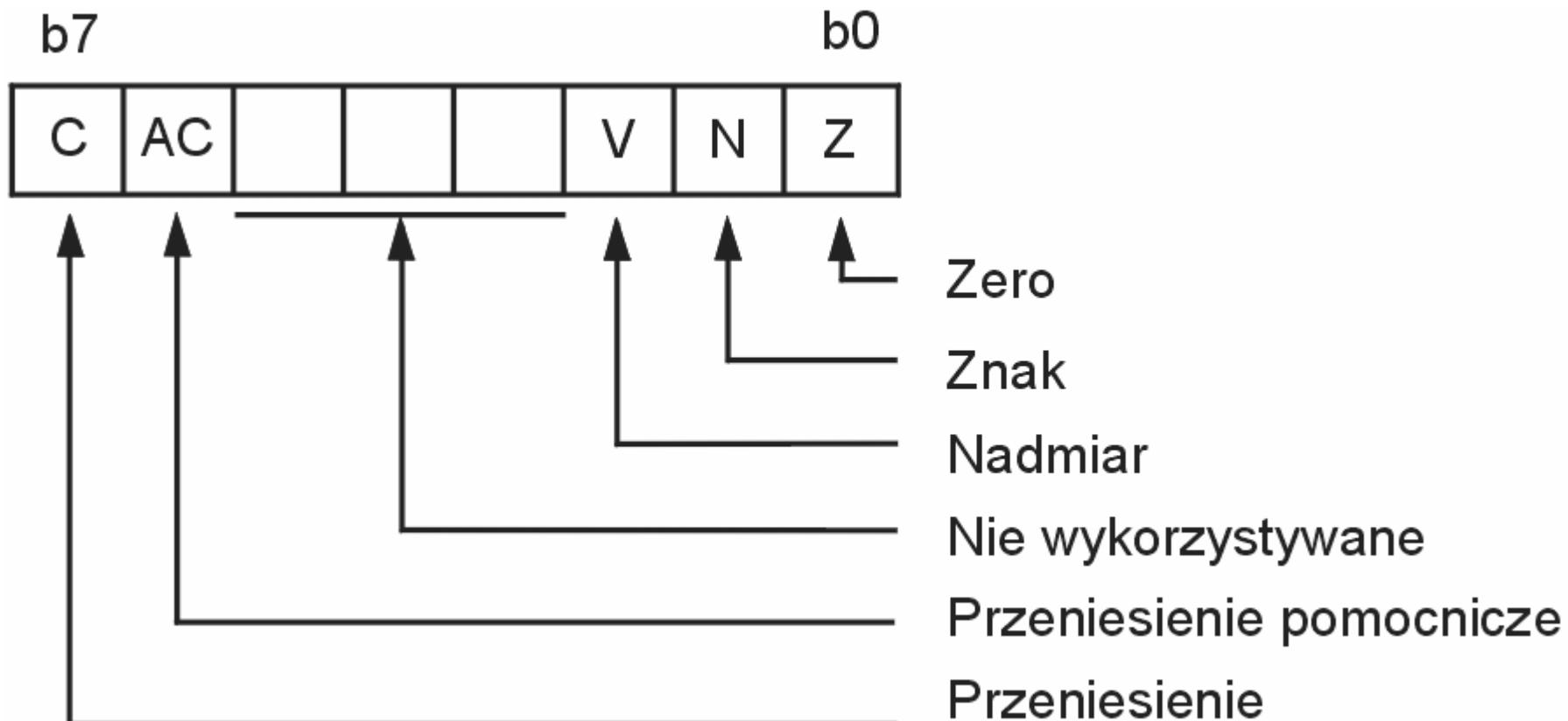


Standardowa architektura CPU

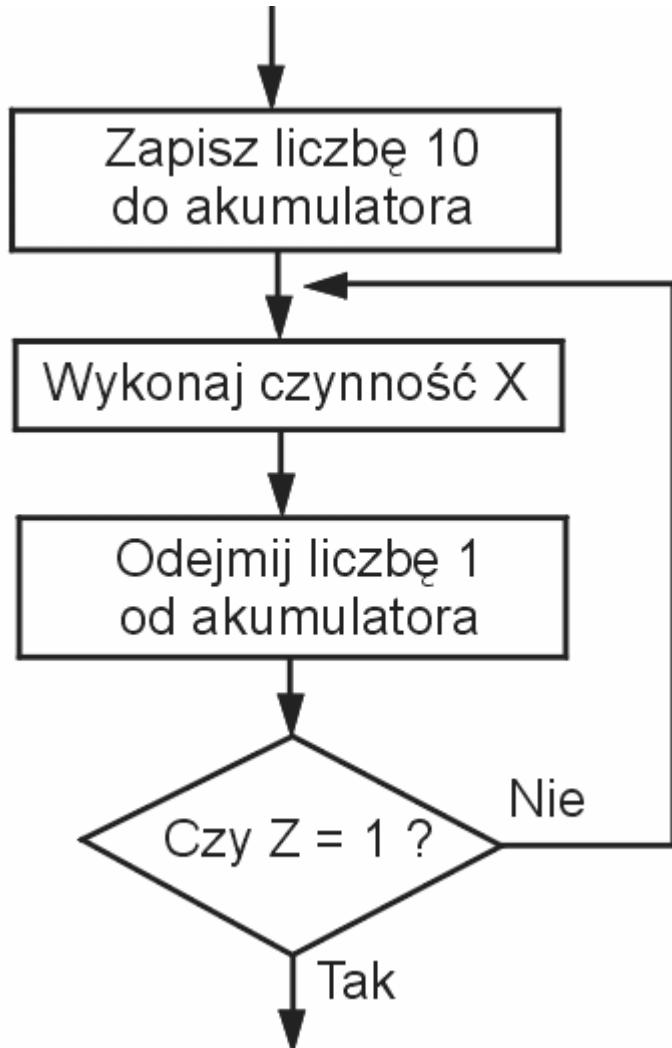


Rejestr wskaźników

Przykładowa struktura rejestru wskaźników: procesor 80C51 XA (*Philips*)



Bit Z - przykład



Przykład wykorzystania bitu Z do organizacji pętli w programie; pętla jest powtarzana 10 razy, aż zawartość akumulatora stanie się równa zeru

Bit C - przykład

Przykład operacji dodawania z uwzględnieniem bitu przeniesienia C

`adc ax, 71h`

Akumulator

1 0 1 1 1 0 0 1

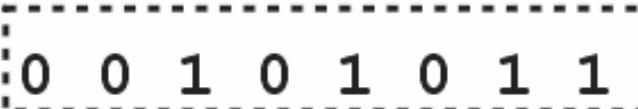
Rejestr przejściowy

0 1 1 1 0 0 0 1

Bit C

1

Wynik dodawania

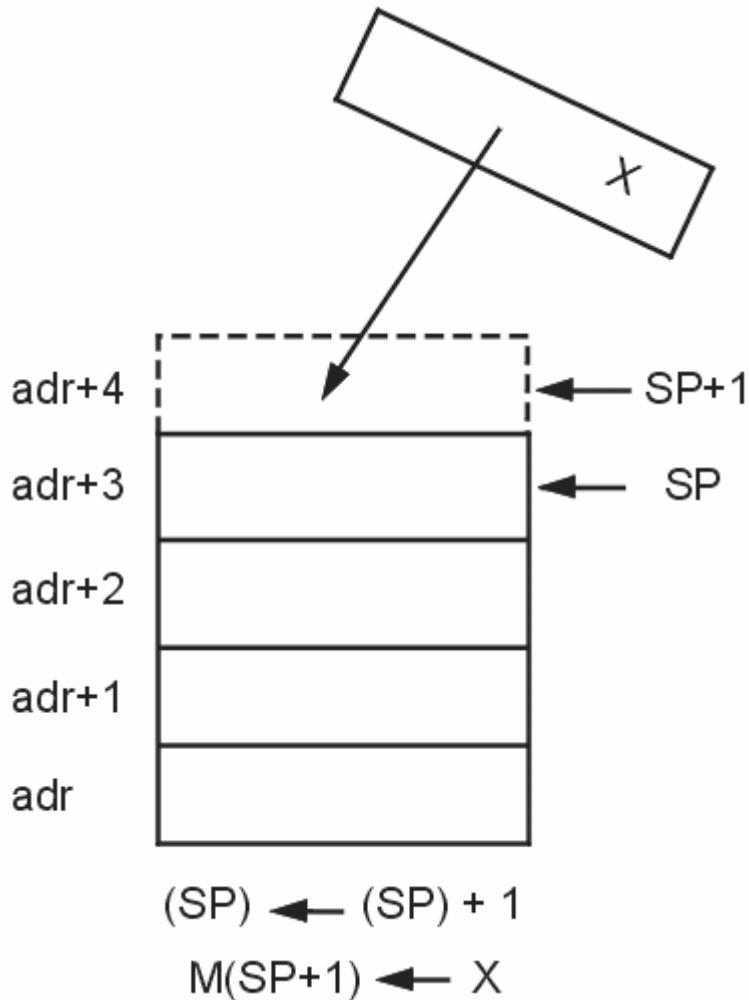
1  1



Wynik w akumulatorze

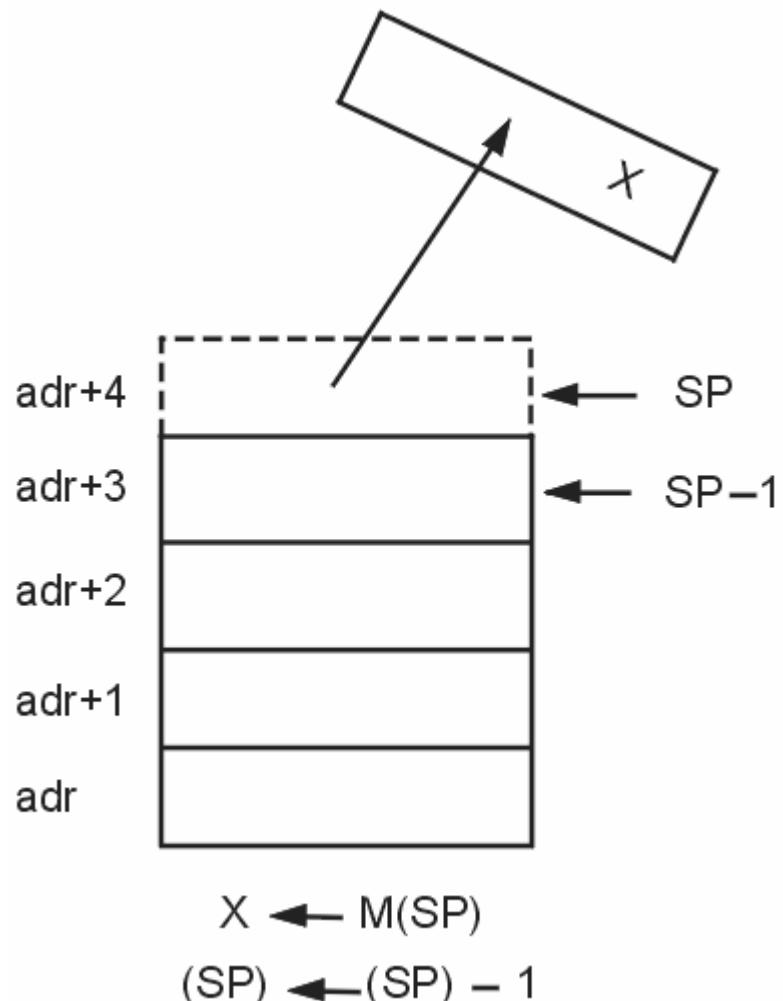
Nowa wartość bitu C

Stos – operacja zapisu



- typowa notacja w języku asemblera:
push x
- w podanym przykładzie stos rozbudowuje się w stronę wzrastających adresów pamięci (np. Intel x51)
- wiele innych procesorów wykorzystuje stos rozbudowywany w stronę malejących adresów, np. Intel Pentium, Motorola 68HC16

Stos – operacja odczytu



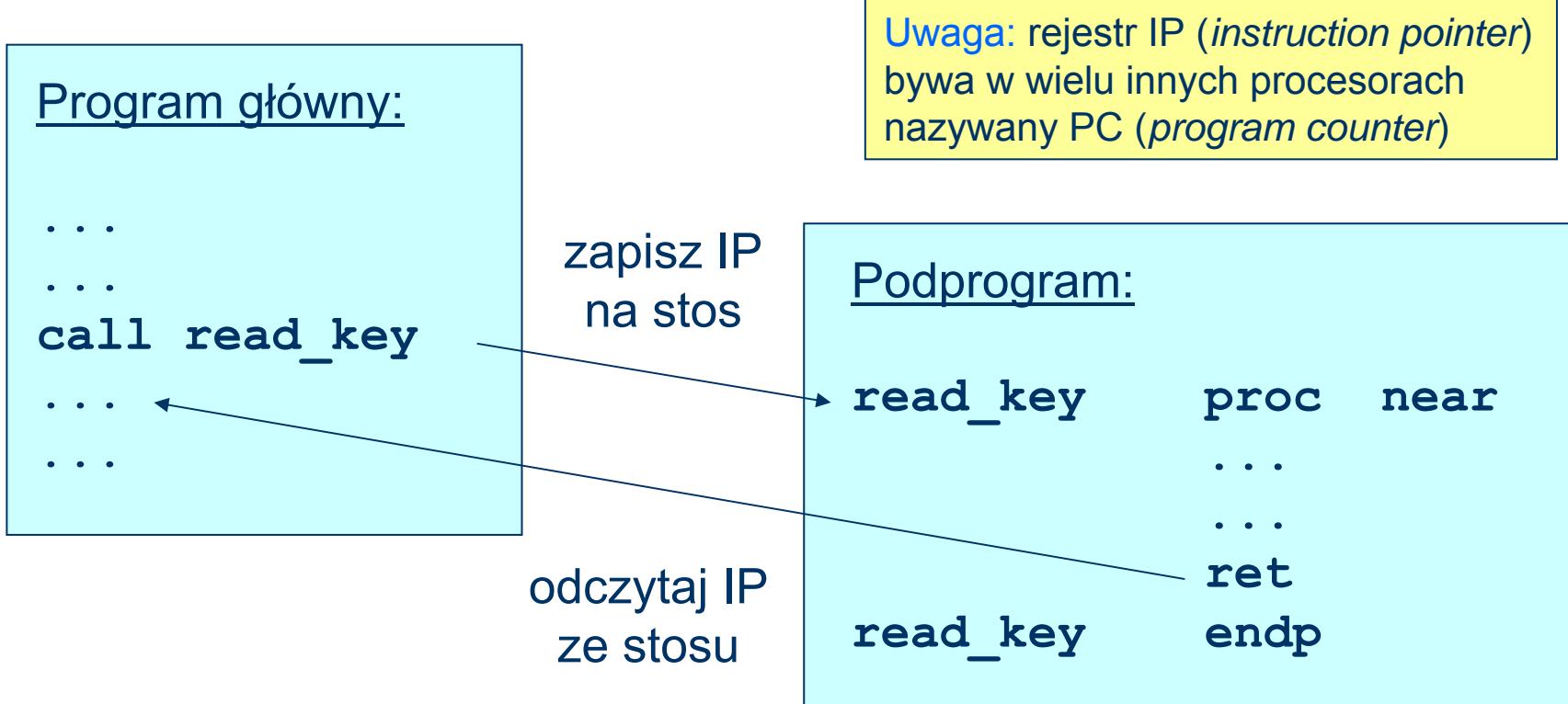
- typowa notacja w języku asemblera:
pop x
- w podanym przykładzie stos rozbudowuje się w stronę wzrastających adresów pamięci

Funkcje i właściwości stosu

- Stos jest pamięcią typu LIFO (*last-in-first-out*)
- Typowe zastosowania stosu
 - przy wywołaniu podprogramu zapamiętuje adres powrotu do programu głównego
 - przy przejściu do programu obsługi przerwania zapamiętuje adres powrotu do przerwanego programu
 - służy do chwilowego przechowywania zawartości rejestrów w celu uwolnienia ich do innych zadań
 - może być użyty do przekazywania parametrów do podprogramów
- Programista musi dbać o zbilansowanie liczby operacji zapisu i odczytu oraz o zachowanie właściwej kolejności zapisu i odczytu (odczyt wartości przebiega w odwrotnej kolejności niż przy zapisie)
- Programista musi dbać o to, by stos nie rozrosł się nadmiernie i nie przekroczył limitu pamięci (*stack overflow*) – częsty błąd w przypadku programów rekurencyjnych lub przy wystąpieniu nieskończonej pętli

Stos – wywołanie podprogramu

Ilustracja wykorzystania stosu do zapamiętania adresu powrotu (zawartości licznika rozkazów IP) przy wywołaniu podprogramu (assembler Pentium)



Rejestry

- Niewielka pamięć robocza CPU do przechowywania tymczasowych wyników obliczeń
- Liczba rejestrów i ich funkcje różnią się dla różnych procesorów
- Rejestry mogą pełnić rozmaite funkcje:
 - Rejestry ogólnego przeznaczenia (GP – *general purpose*)
 - Rejestry danych (np. akumulator)
 - Rejestry adresowe
 - Rejestr wskaźników (stanu, warunków, flag)
- Architektura oparta na rejestrach GP jest bardziej elastyczna przy programowaniu, ale procesor jest bardziej złożony a czas wykonania rozkazu dłuższy
- Architektura oparta na specjalizowanych rejestrach upraszcza budowę procesora i przyspiesza wykonanie rozkazu, ale ogranicza programistę

Rejestry cd.

- Typowa liczba rejestrów GP w CPU: 8 – 32
 - mniej rejestrów GP – częstsze odwołania do pamięci
 - znaczne zwiększenie liczby rejestrów GP nie wpływa znacząco na zmniejszenie liczby odwołań do pamięci
- Rozmiar rejestru
 - wystarczający do przechowywania adresu
 - wystarczający do przechowywania pełnego słowa danych
- Często można łączyć ze sobą dwa rejesty, dzięki czemu można łatwo reprezentować typowe formaty danych w językach wysokiego poziomu, np. w C:
 - **double int a**
 - **long int a**

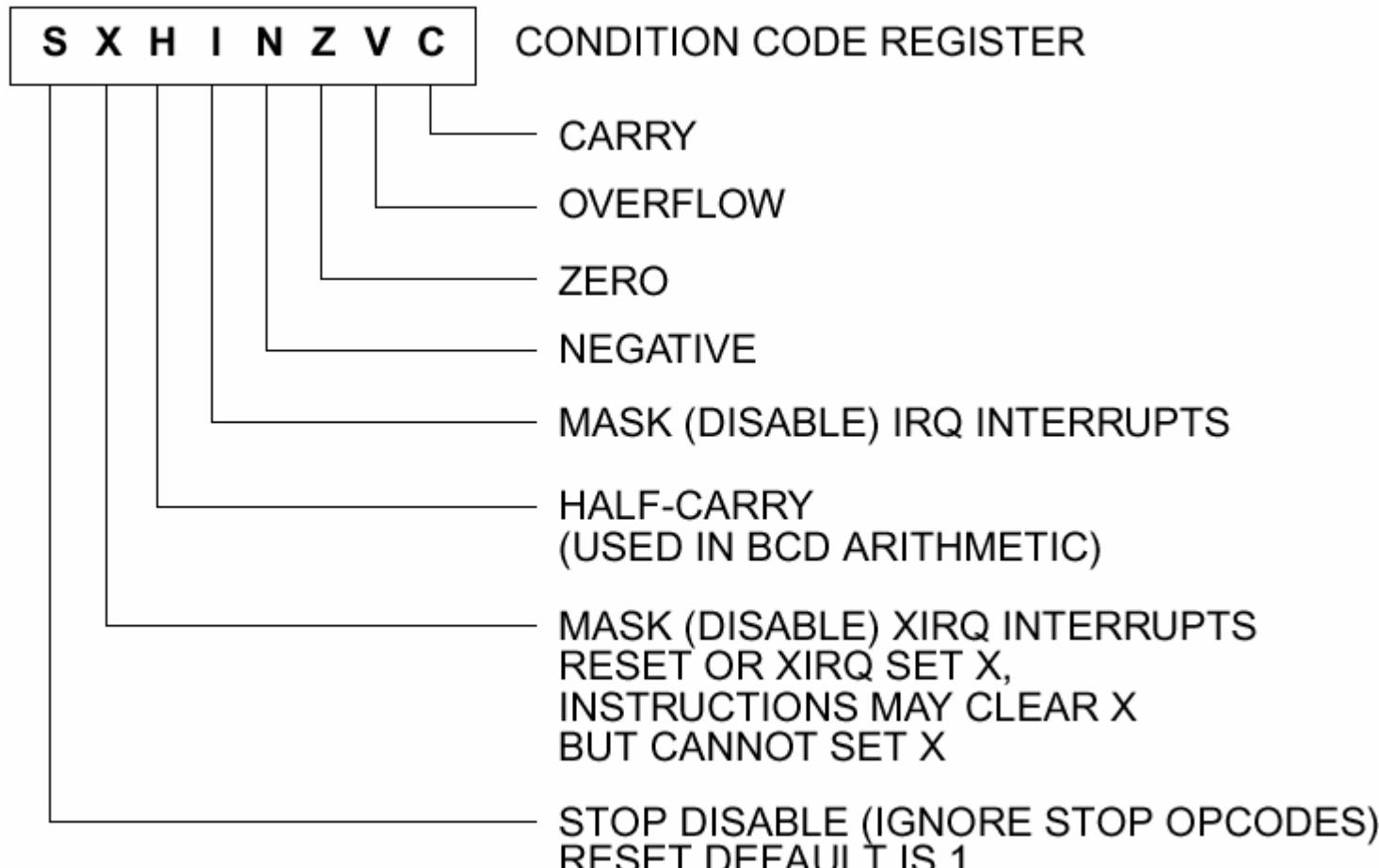
Rejestry cd.

Przykład architektury pliku rejestrów: procesor Motorola 68HC12

7	A	0	7	B	0	8-BIT ACCUMULATORS A AND B OR 16-BIT DOUBLE ACCUMULATOR D			
15			D		0				
15			IX		0	INDEX REGISTER X			
15			IY		0	INDEX REGISTER Y			
15			SP		0	STACK POINTER			
15			PC		0	PROGRAM COUNTER			
	S	X	H	I	N	Z	V	C	CONDITION CODE REGISTER

Rejestry cd.

Rejestr wskaźników w procesorze Motorola 68HC12



Rejestry cd.

Przykład architektury rejestrów GP:

Philips 80C51 XA

R15
R14
R13
R12
R11
R10
R09
R08

16-bitowe rejesty globalne

R7H	R7L
R6H	R6L
R5H	R5L
R4H	R4L

8- i 16-bitowe rejesty globalne

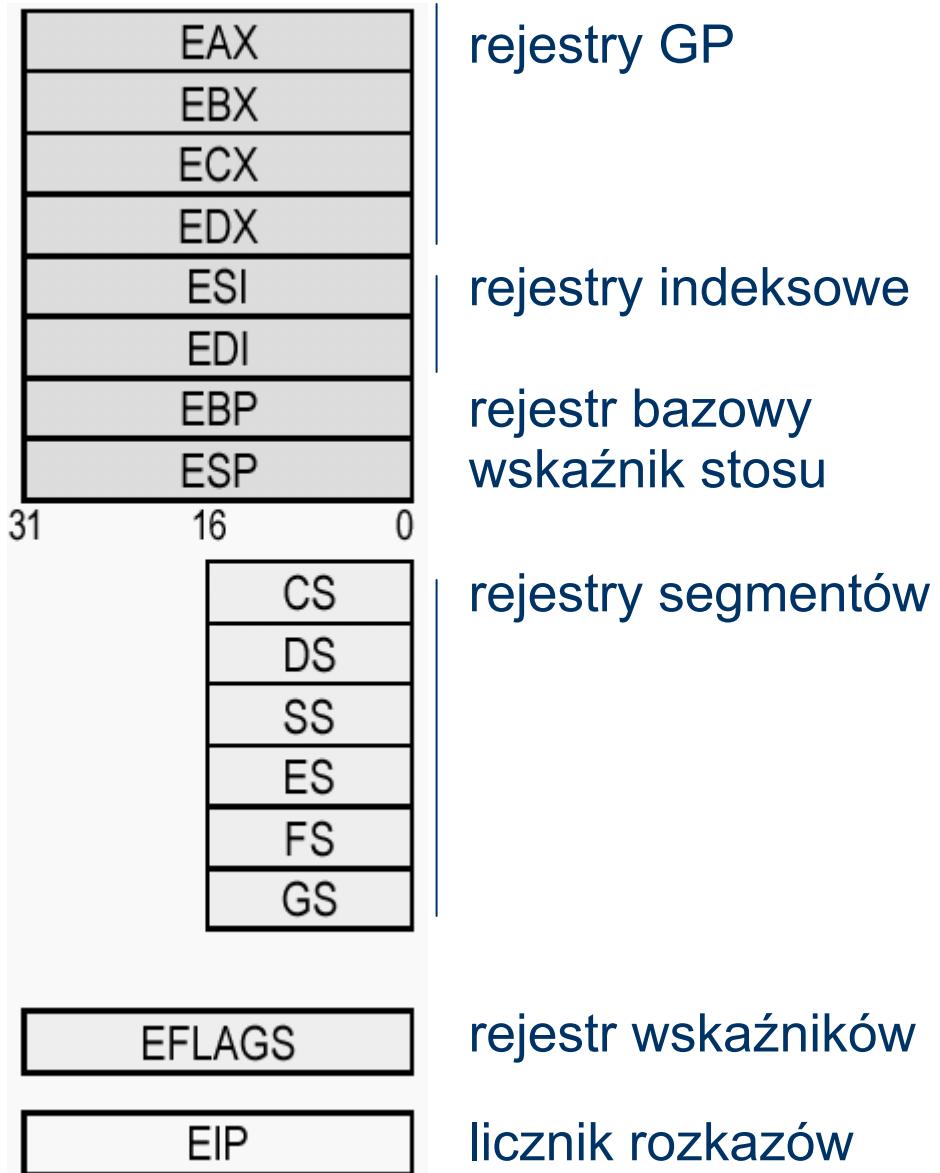
R3H	R3L
R2H	R2L
R1H	R1L
R0H	R0L

4 zestawy (banki) rejestrów 8- i 16-bitowych

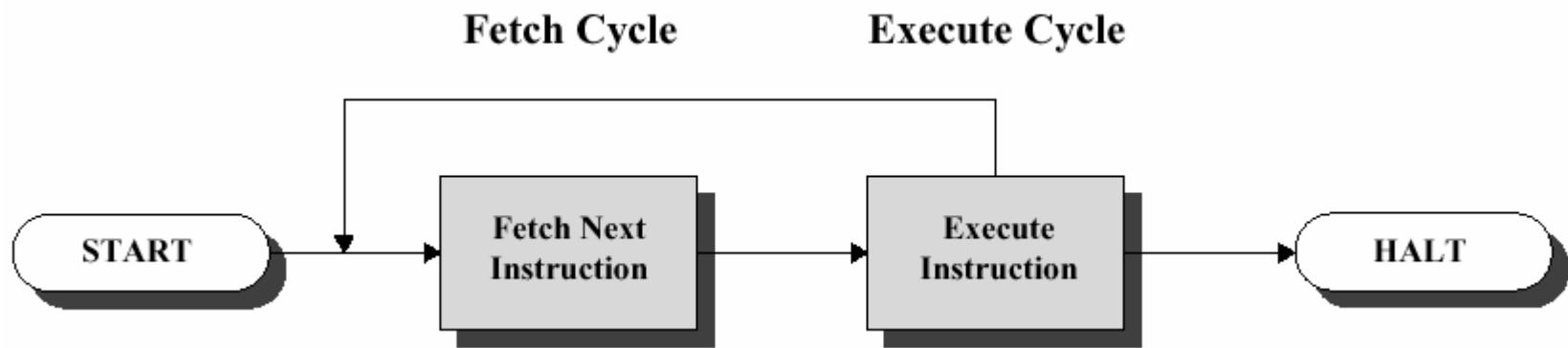
Rejestry cd.

Przykład architektury pliku rejestrów:

procesor Intel Pentium



Cykl wykonania rozkazu



fetch – pobranie binarnego kodu rozkazu z pamięci

W celu przyspieszenia wykonywania rozkazu stosuje się niekiedy cykl *prefetch* – pobieranie kodu rozkazu jeszcze przed zakończeniem wykonania poprzedniego rozkazu. Rozwój tej koncepcji doprowadził do *cachingu* i *pipeliningu*

Format rozkazu

W procesorach o architekturze CISC rozkazy mają niejednolitą postać (format):

- instrukcja zapisana binarnie może mieć różną długość
- stosuje się różne sposoby adresowania argumentów

Przykład (Pentium):

`mov ax, cx`

10001011
11000001

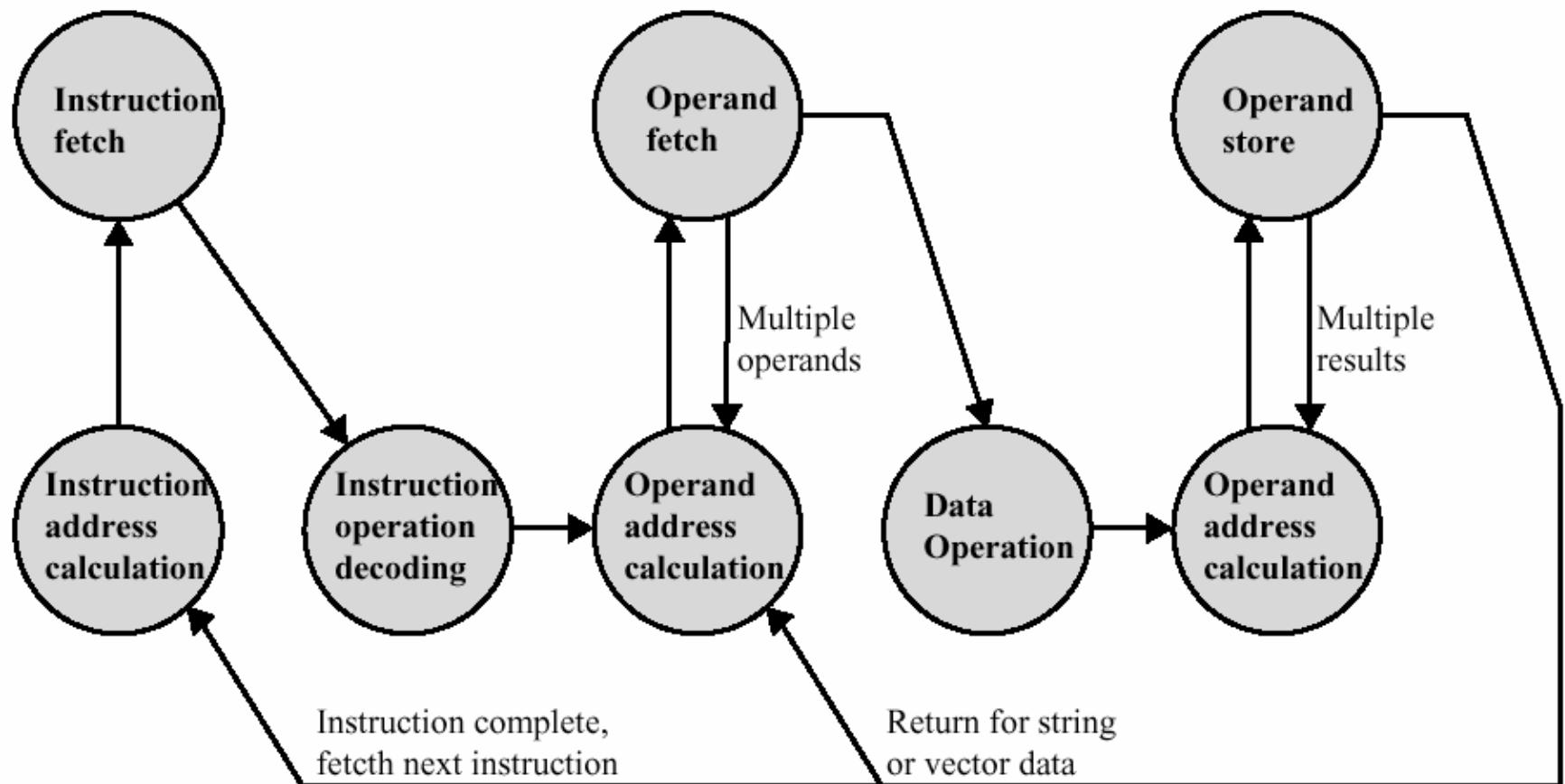
argument lub adres

`mov ax, [e42d]`

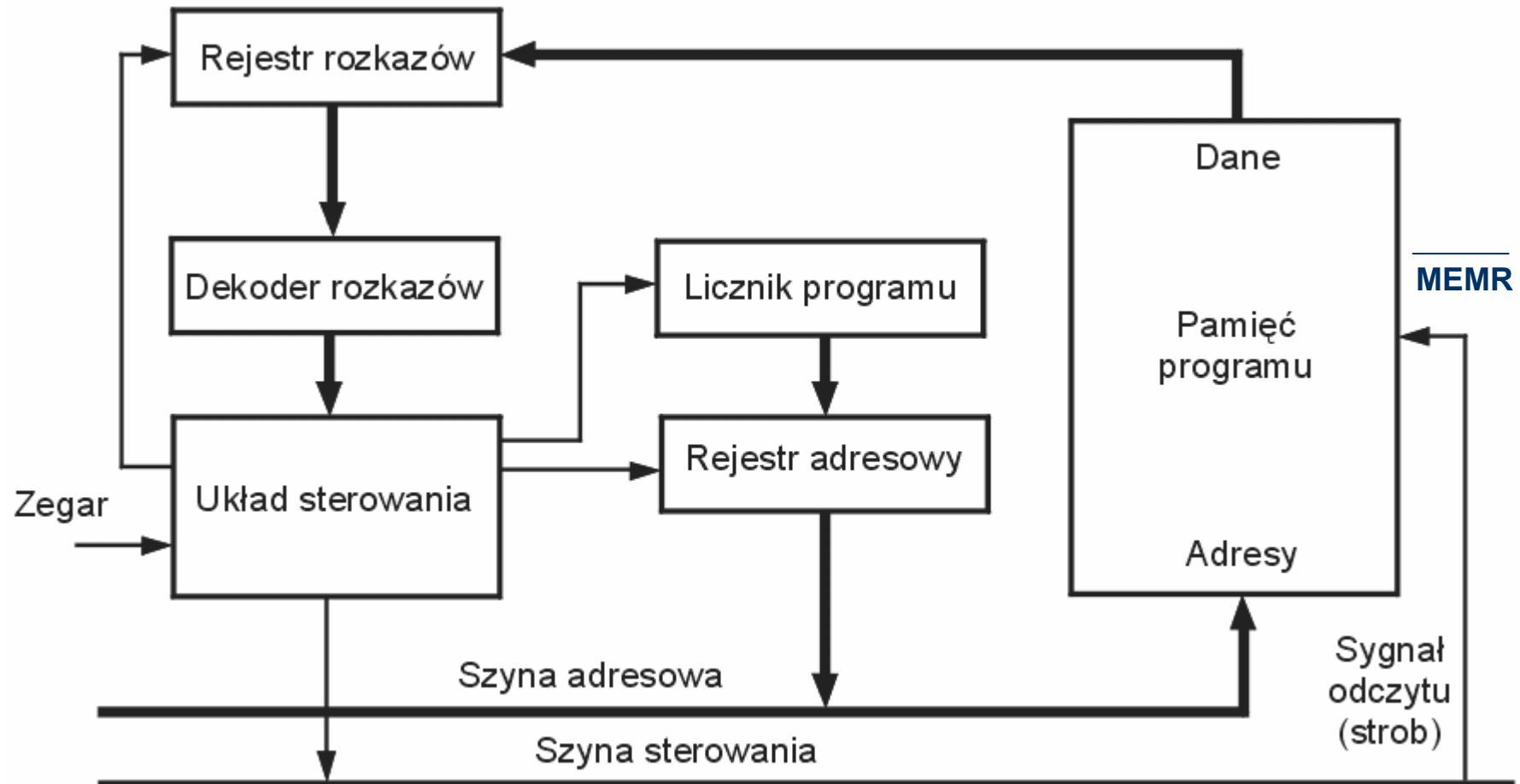
10100001
00101101
11100100

little endian!

Cykl wykonania rozkazu cd.



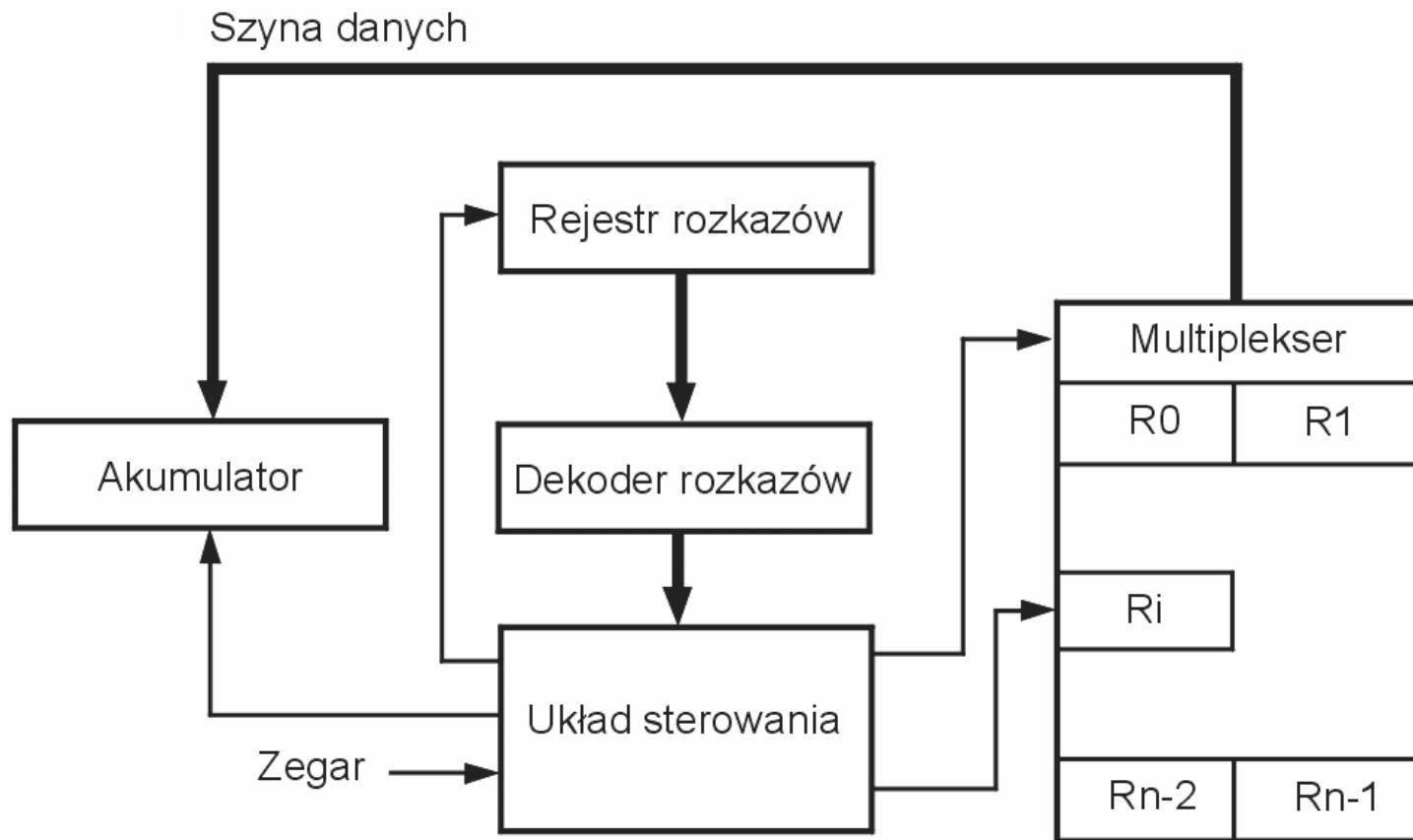
Pobranie rozkazu



Wykonanie rozkazu

$$Ri \rightarrow A$$

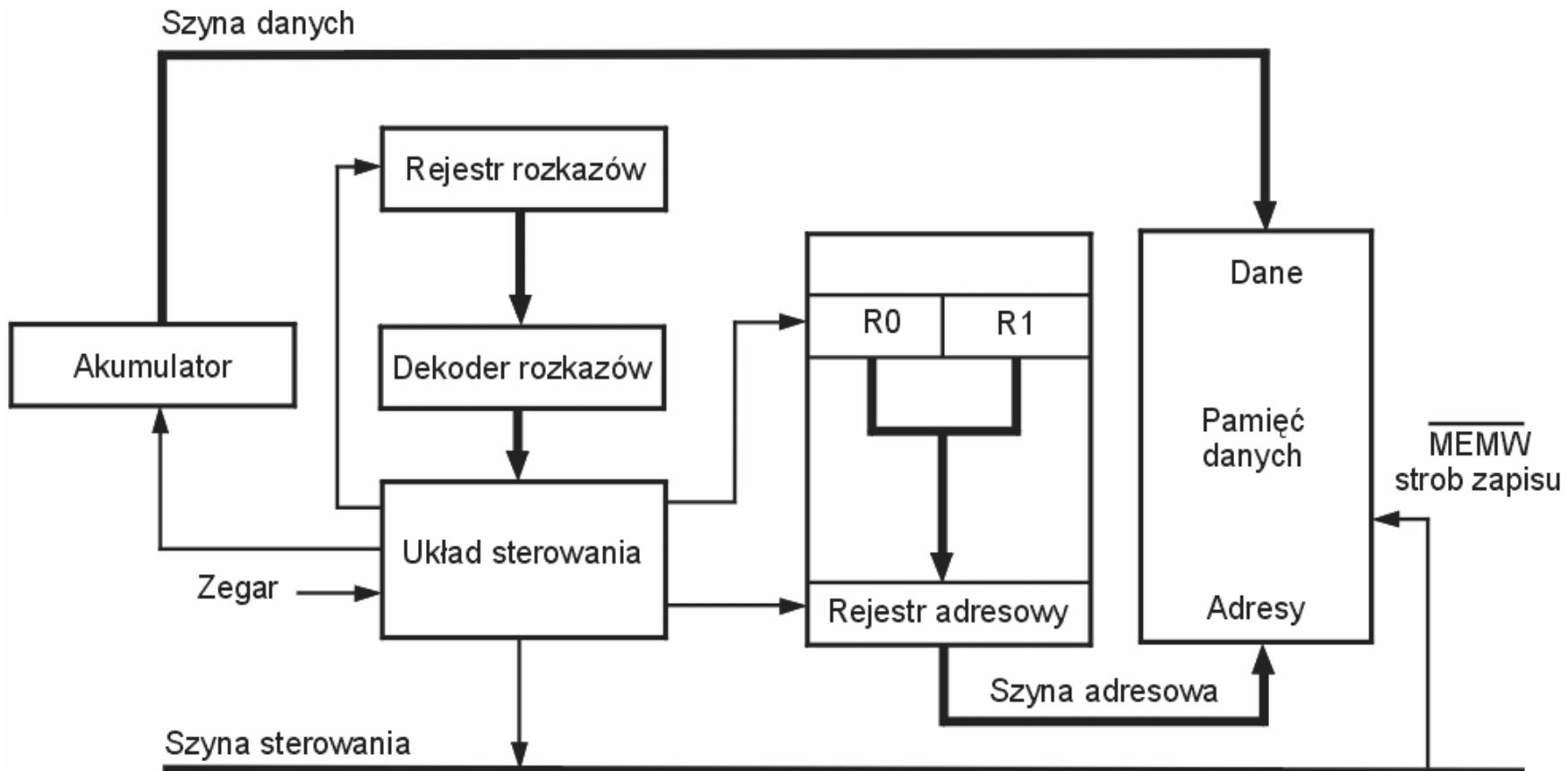
Przykład wykonania rozkazu przesłania zawartości rejestru Ri do akumulatora



Wykonanie rozkazu cd.

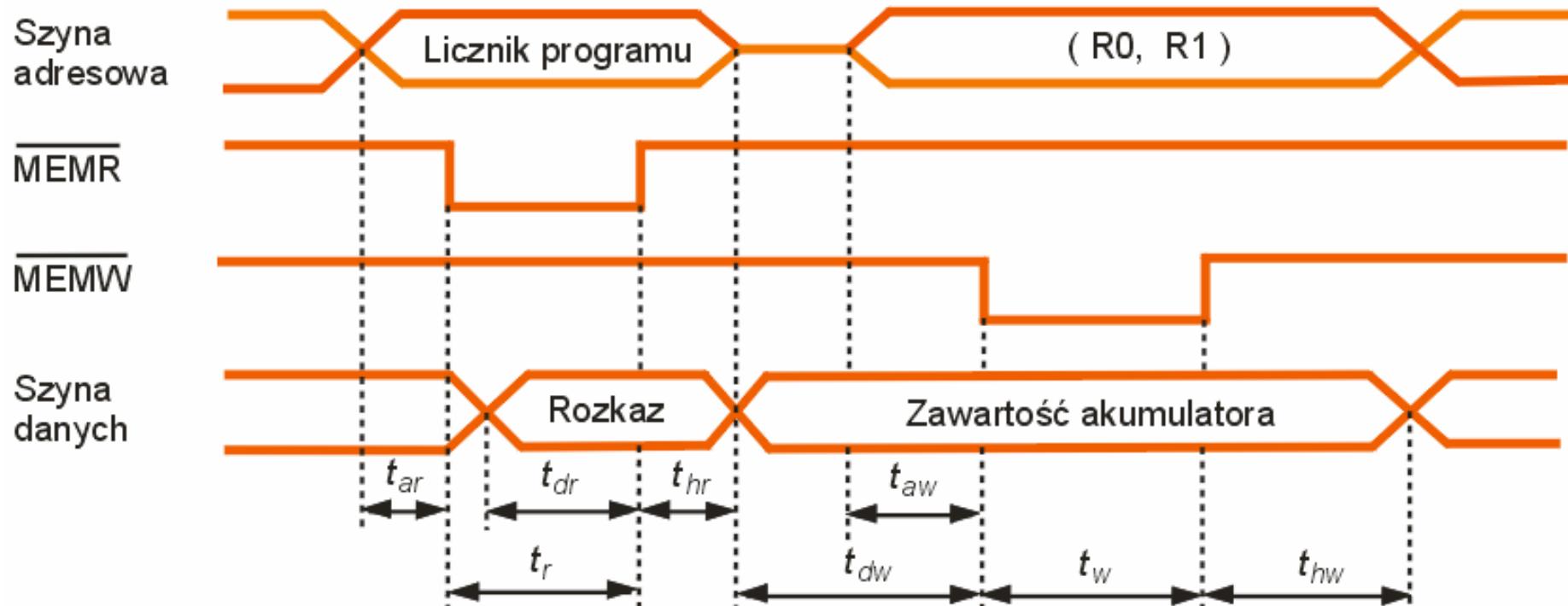
$$A \rightarrow M(R0, R1)$$

Przykład: wykonanie rozkazu przesłania zawartości akumulatora A do komórki pamięci o adresie zawartym w parze rejestrów R0 i R1



Wykonanie rozkazu cd.

Przebiegi w trakcie pobrania i wykonania rozkazu: $A \rightarrow M(R0, R1)$



Wykonanie rozkazu cd.

- Analizowane poprzednio rozkazy w prostych procesorach 8-bitowych zajmują tylko 1 bajt zawierający kod operacji
 - informacja o rejestrach biorących udział w operacji mieści się na kilku bitach w kodzie operacji
 - wymagany jest tylko jeden cykl dostępu do pamięci w fazie *fetch*
 - w przypadku rozkazu $Ri \rightarrow A$ nie trzeba dostępu do pamięci przy zapisie wyniku
 - w przypadku rozkazu $A \rightarrow M(R0, R1)$ potrzebny jest jeden cykl dostępu do pamięci przy zapisie wyniku

Wykonanie rozkazu cd.

- Przykład rozkazu wielobajtowego:

```
add a,7bh ;dodaj do akumulatora liczbę 7b  
;zapisaną w kodzie hex
```

- wykonanie

IR	\leftarrow	M(PC)	;pobierz kod operacji do IR
PC	\leftarrow	PC+1	;inkrementuj licznik programu PC
Temp	\leftarrow	M(PC)	;załaduj do Temp argument z pamięci
PC	\leftarrow	PC+1	;inkrementuj PC
A	\leftarrow	ALU+	;prześlij wynik dodawania A+Temp do A

Wykonanie rozkazu cd.

- Przykład rozkazu wielobajtowego

```
add a,[1c47h] ;dodaj do A zawartość komórki  
;o adresie 1c47 w kodzie hex
```

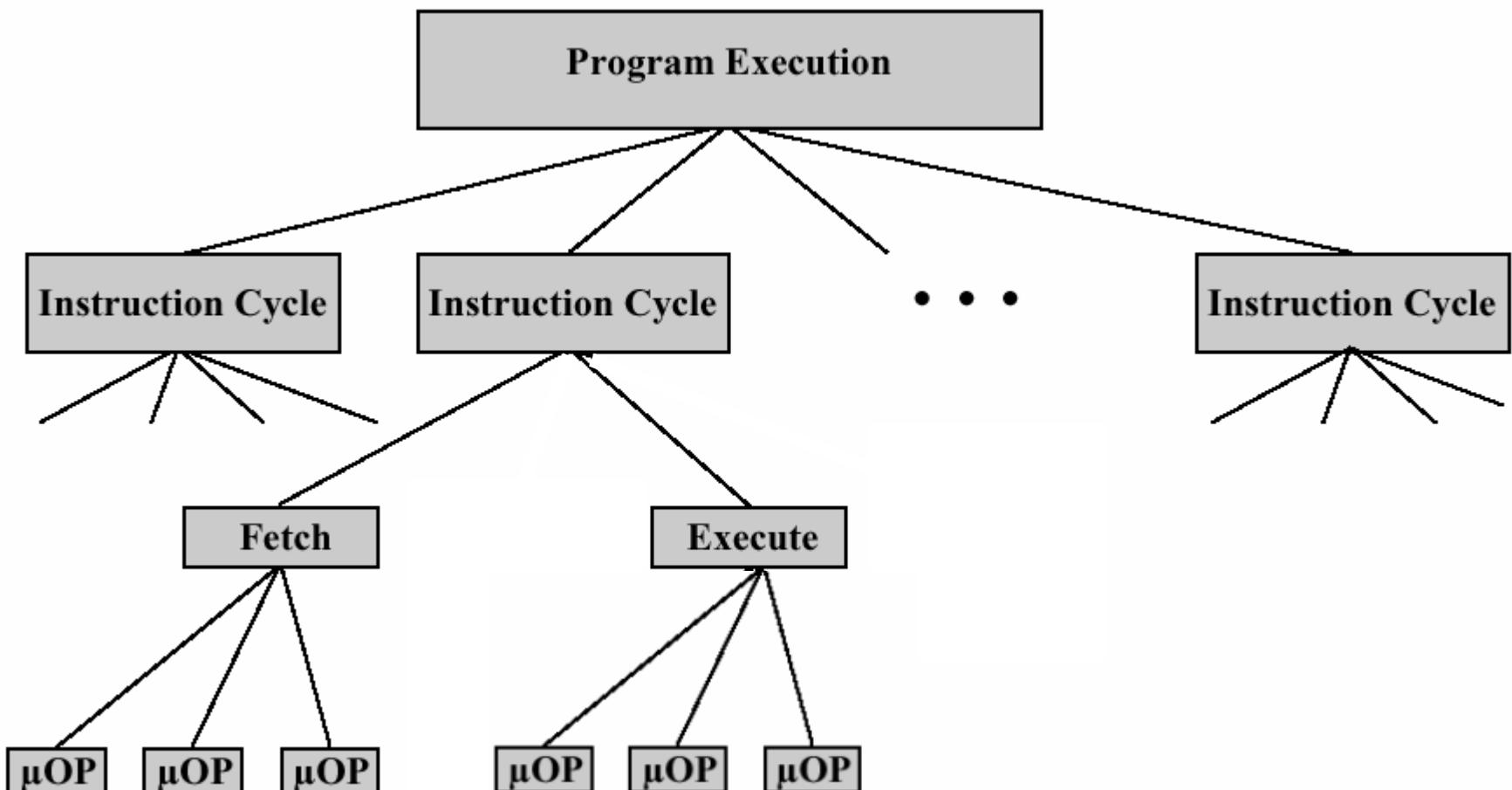
- wykonanie

IR	\leftarrow	M(PC)	;pobierz kod operacji do IR
PC	\leftarrow	PC+1	;inkrementuj licznik programu PC
RAH	\leftarrow	M(PC)	;ładuj bardziej znaczący bajt adresu
PC	\leftarrow	PC+1	;inkrementuj PC
RAL	\leftarrow	M(PC)	;ładuj mniej znaczący bajt adresu
PC	\leftarrow	PC+1	;inkrementuj PC
Temp	\leftarrow	M(RA)	;ładuj dodajnik do rejestru Temp
A	\leftarrow	ALU+	;prześlij wynik dodawania do A

Cykl rozkazowy – wnioski

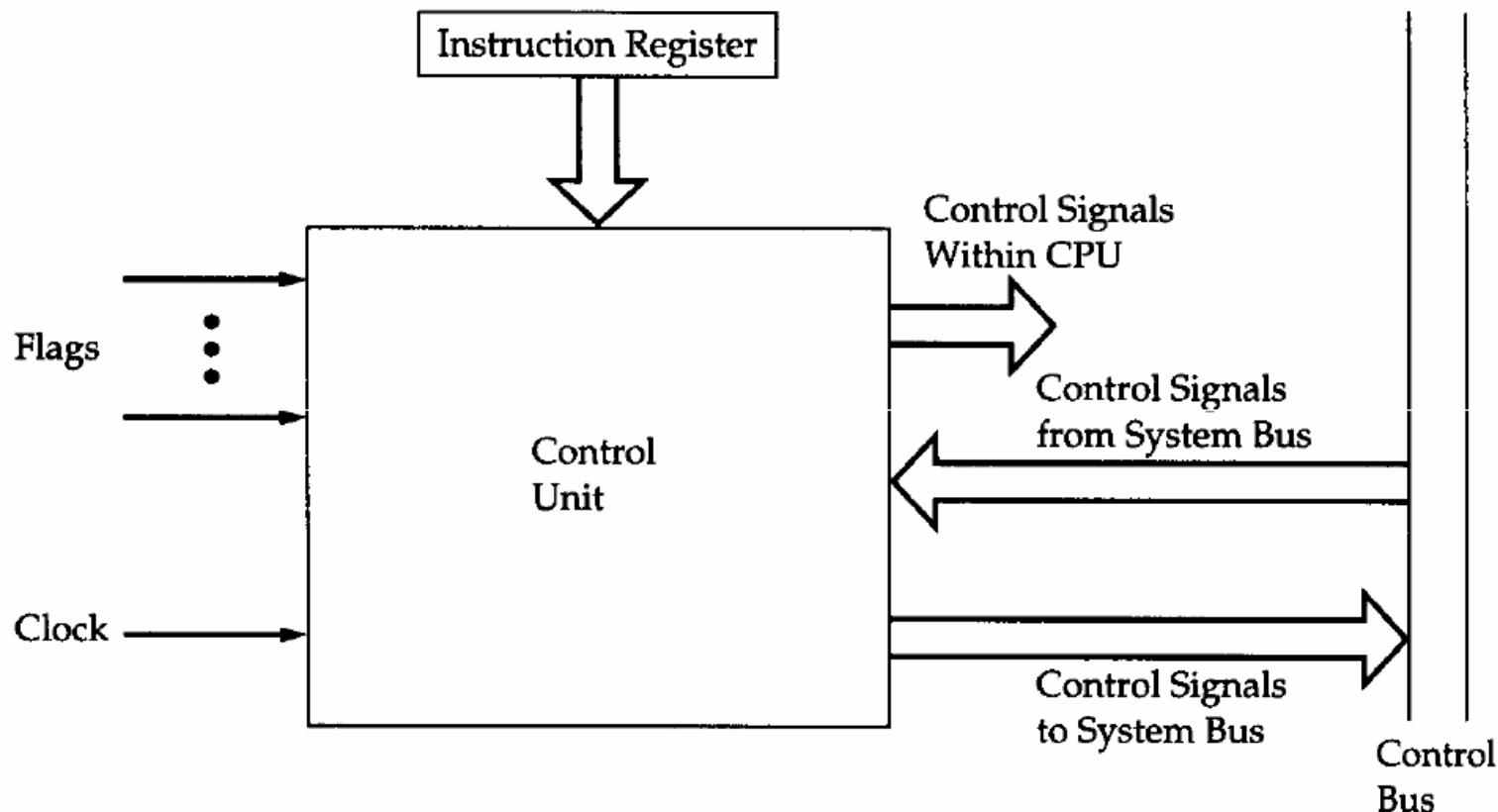
- Wykonanie programu polega na wykonaniu ciągu cykli rozkazowych
- Cykl rozkazowy składa się z dwóch podstawowych faz: cyklu pobrania rozkazu i wykonania rozkazu
- W zależności od długości kodu rozkazu i sposobu adresowania argumentów wykonanie rozkazu zajmuje różną liczbę elementarnych cykli maszynowych realizowanych przez jednostkę sterującą
- Elementarne operacje składające się na wykonanie rozkazu są nazywane **mikrooperacjami** (μ OP)

Instrukcje i mikrooperacje

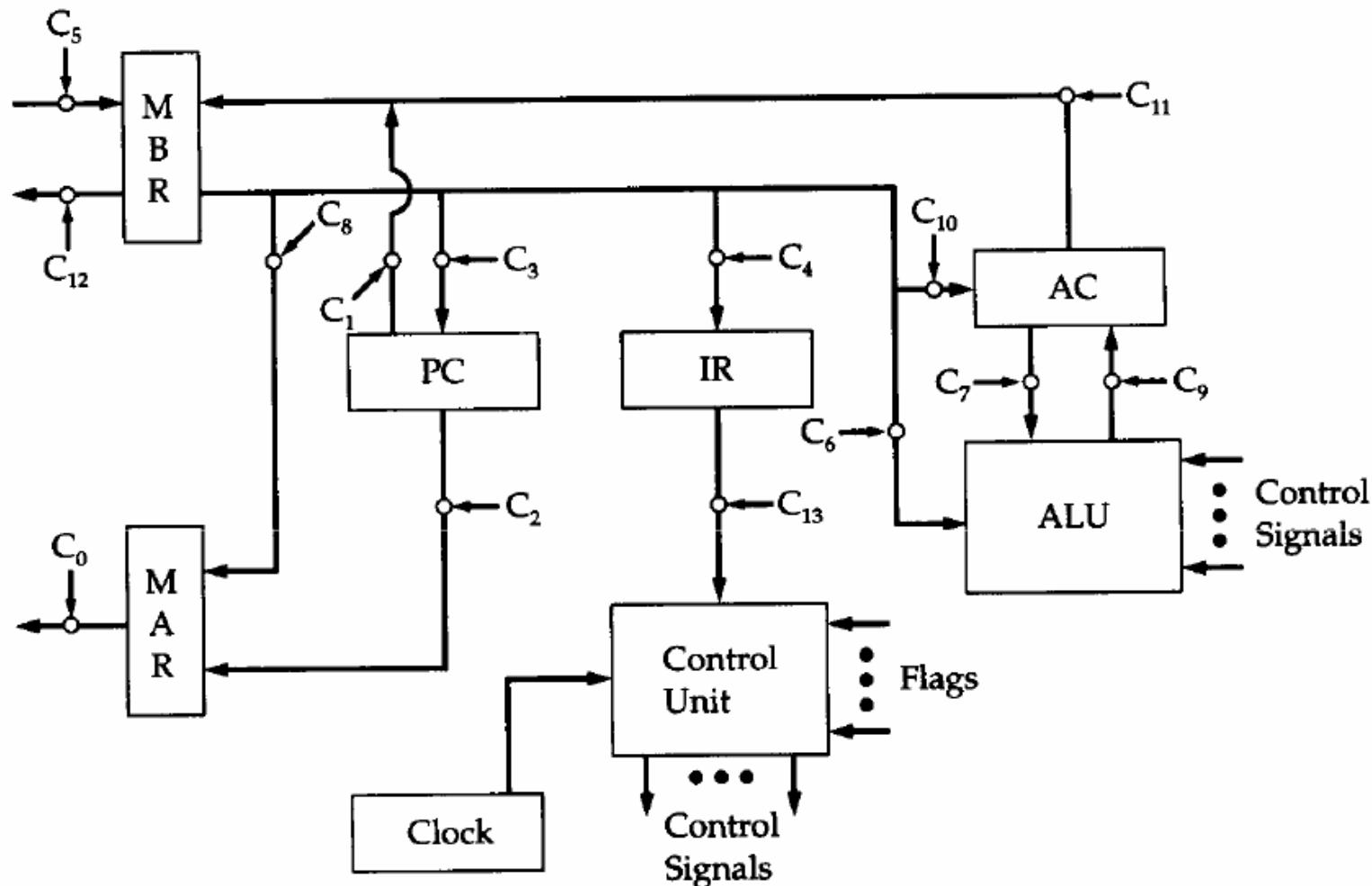


Układ sterowania

Zadaniem jednostki sterującej jest wygenerowanie sekwencji sygnałów sterujących, które spowodują wykonanie sekwencji mikrooperacji realizującej dany rozkaz



Sterowanie w prostym CPU



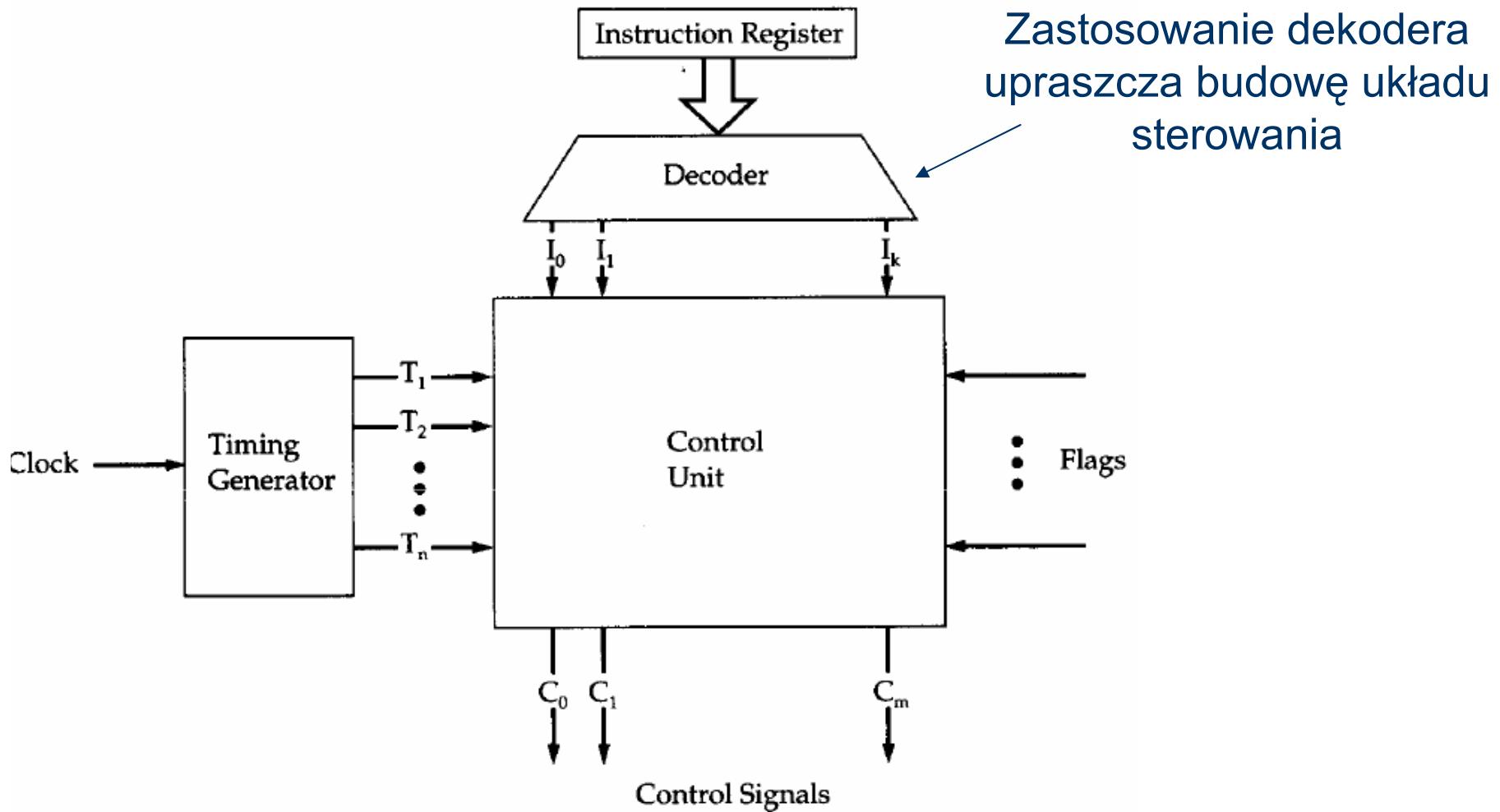
Mikrooperacje i sterowanie

Przykład realizacji prostej operacji dla CPU z poprzedniego rysunku:

add ac, [adres]

cykl	mikrooperacja	sygnały sterujące
t1	MAR	PC
t2	MBR	memory
	PC	PC+1
t3	IR	MBR
t4	MAR	IR (adres)
t5	MBR	memory
t6	AC	AC+MBR

Układ sterowania cd.



Warianty realizacji CU

Stosuje się dwa sposoby realizacji jednostki sterującej:

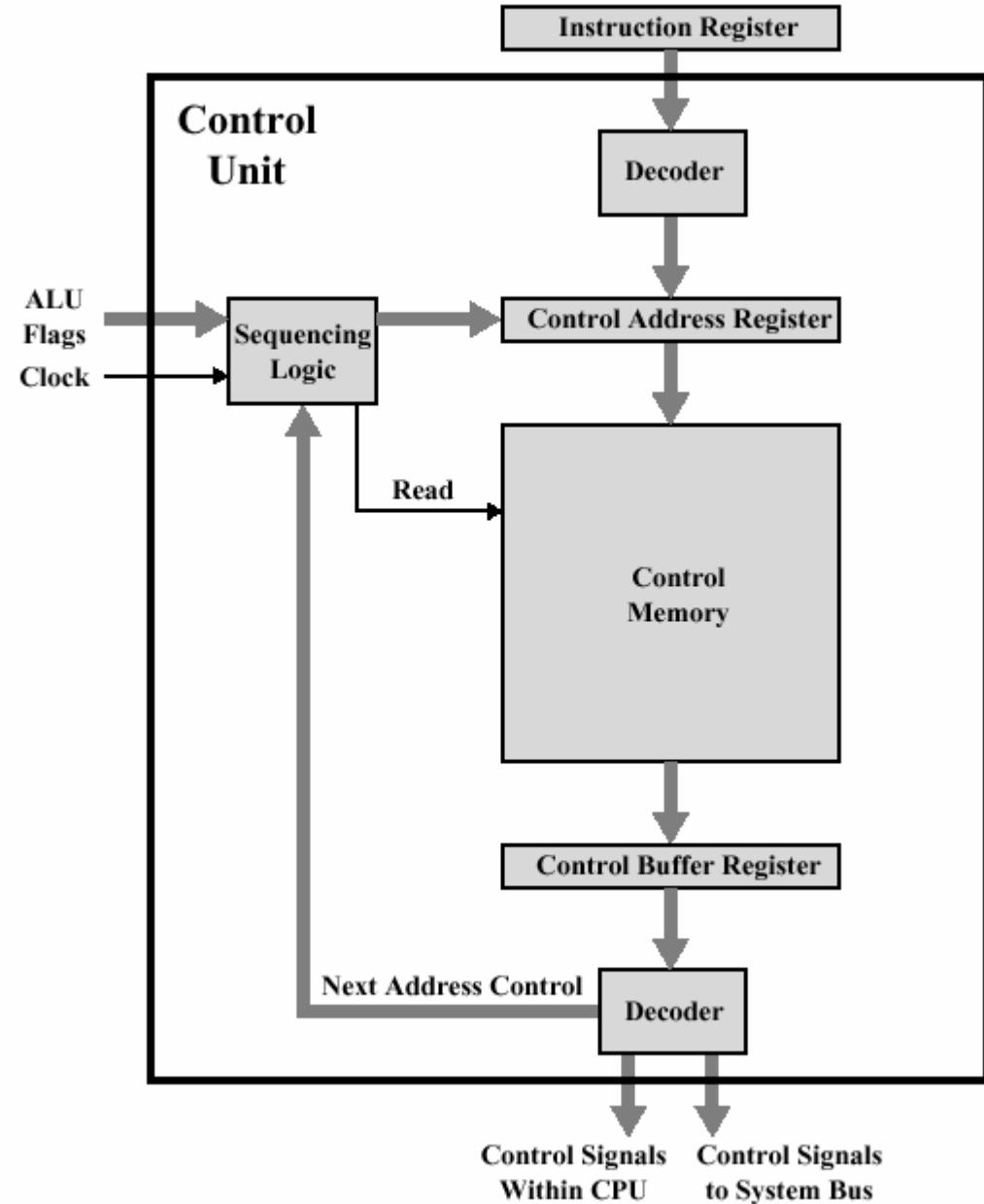
- w postaci typowego układu sekwencyjnego (*hardwired* CU)
- z wykorzystaniem mikroprogramowania (*microprogrammed* CU)
- Jednostka CU typu *hardwired* :
 - Zaprojektowana jako układ sekwencyjny – automat o skończonej liczbie stanów (FSM – *finite state machine*), z wykorzystaniem klasycznych metod syntezy układów sekwencyjnych
 - Zalety: duża szybkość działania, zoptymalizowana liczba elementów logicznych
 - Wada: trudna modyfikacja w razie konieczności zmiany projektu
 - Jednostki CU typu *hardwired* są preferowane w architekturze RISC

Mikroprogramowanie

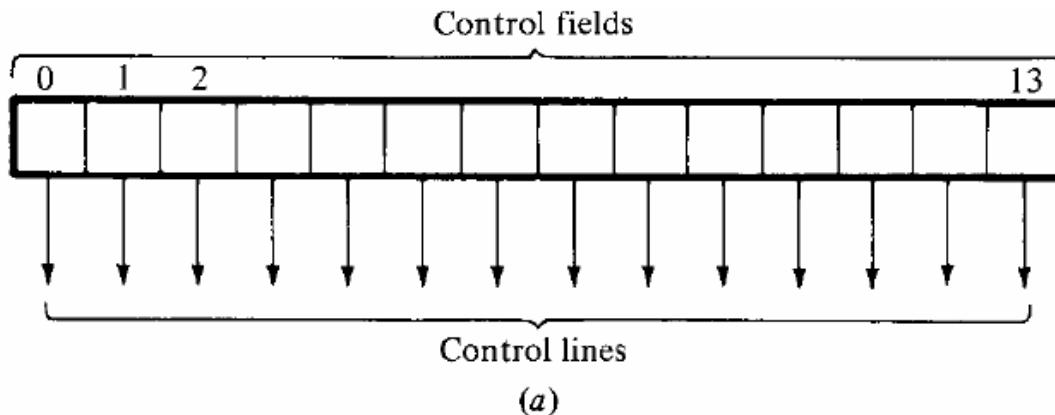
- Mikroprogramowana jednostka CU (M.V. Wilkes, 1951):
- Sekwencje sygnałów sterujących są przechowywane w wewnętrznej pamięci CU i tworzą mikroprogram; każdy rozkaz CPU ma własny kod mikroprogramu
- Praca CU polega na sekwencyjnym odczytywaniu kolejnych słów mikroprogramu
- **Zalety:**
 - przejrzysta, usystematyzowana budowa CU
 - łatwość modyfikacji pamięci mikroprogramu
- **Wady:**
 - mniejsza szybkość działania w porównaniu z jednostką CU typu *hardwired*
 - większa liczba elementów logicznych (w tym pamięciowych)

Mikroprogramowanie cd.

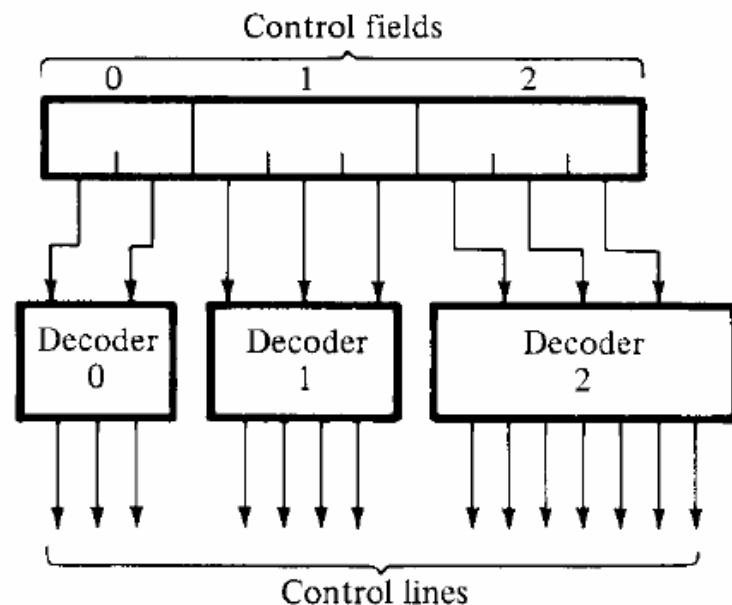
Schemat funkcjonalny mikroprogramowanej jednostki sterującej



Kodowanie mikrorozkazów

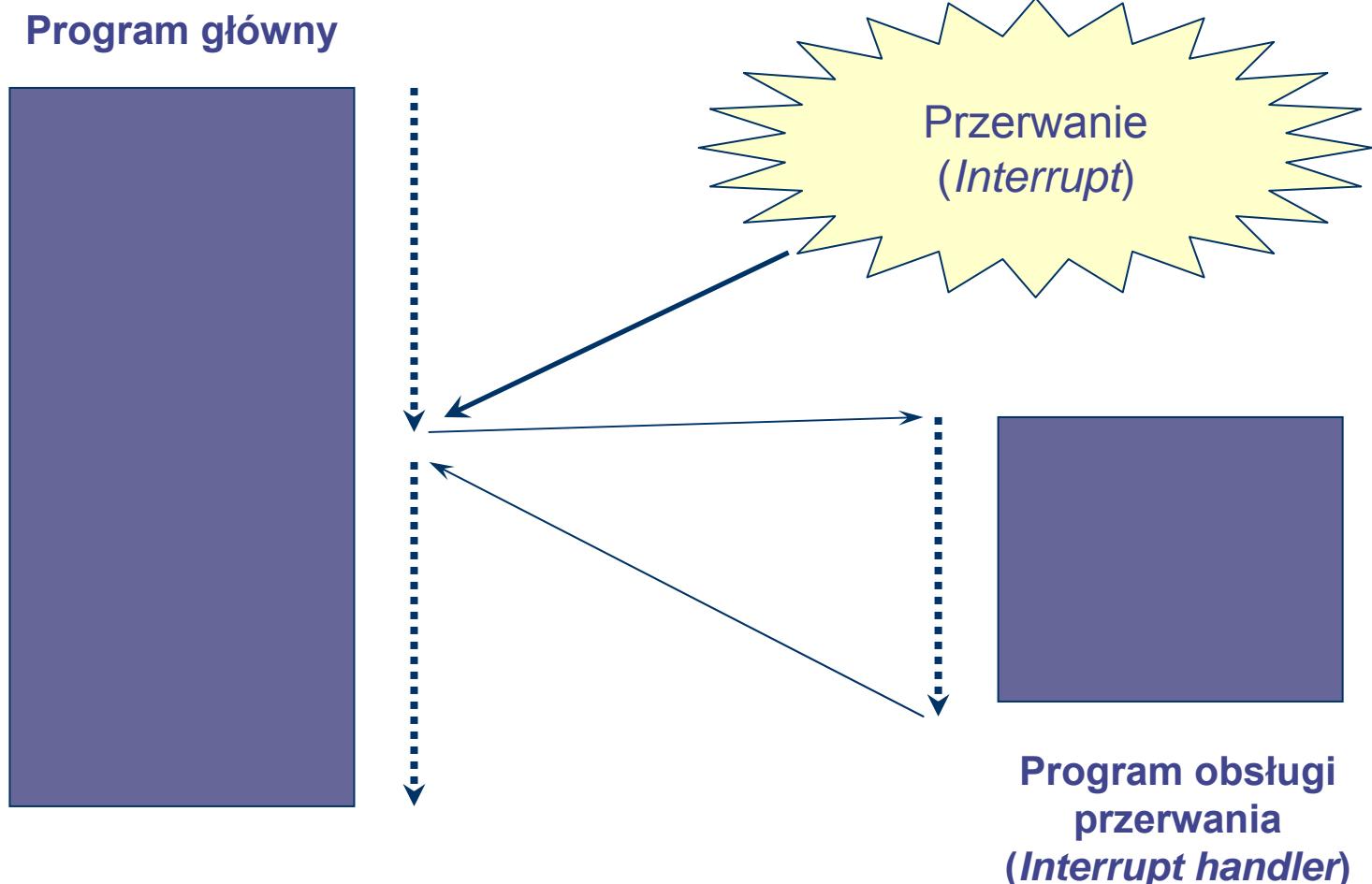


kodowanie poziome

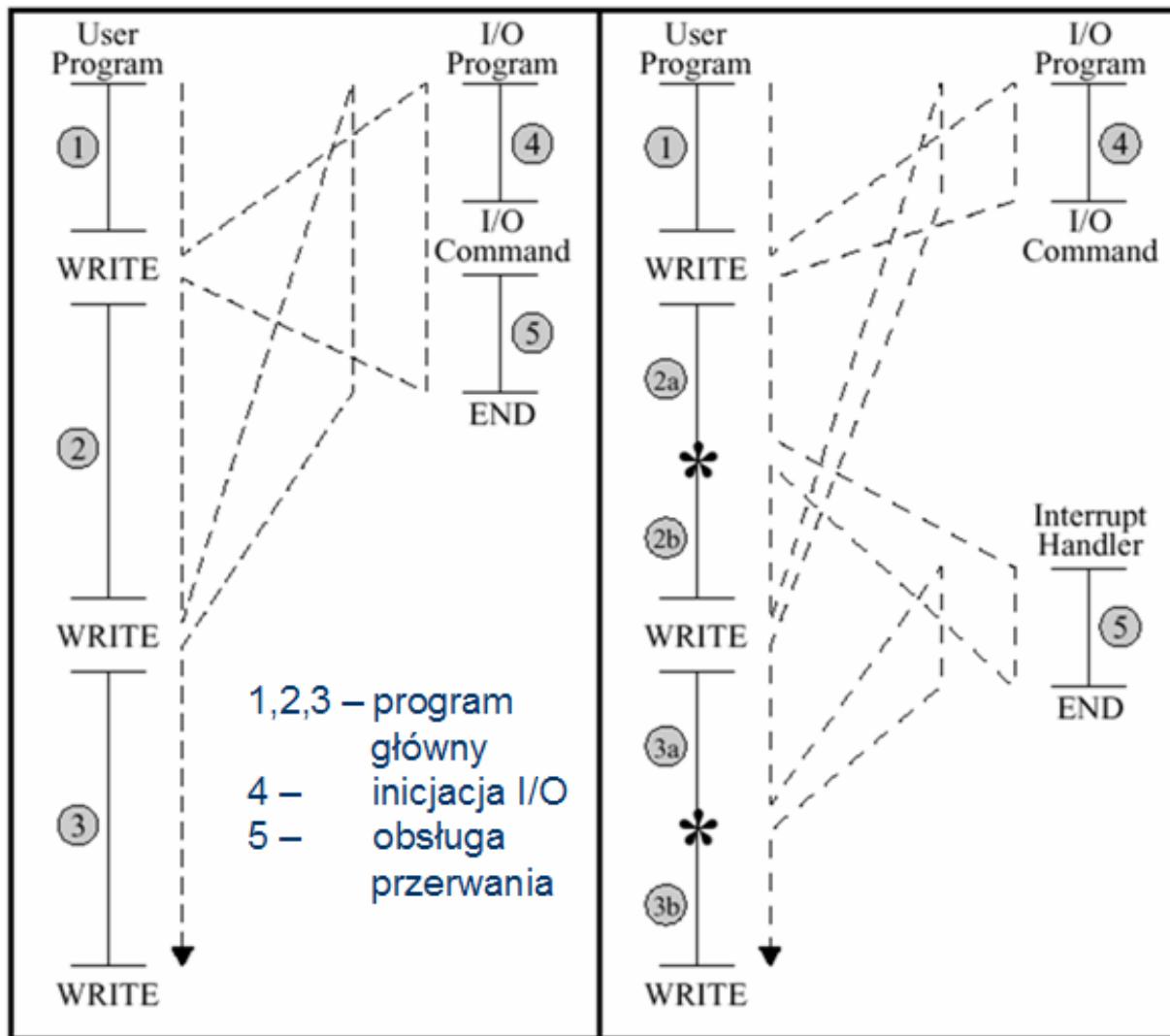


kodowanie pionowe

Przerwanie

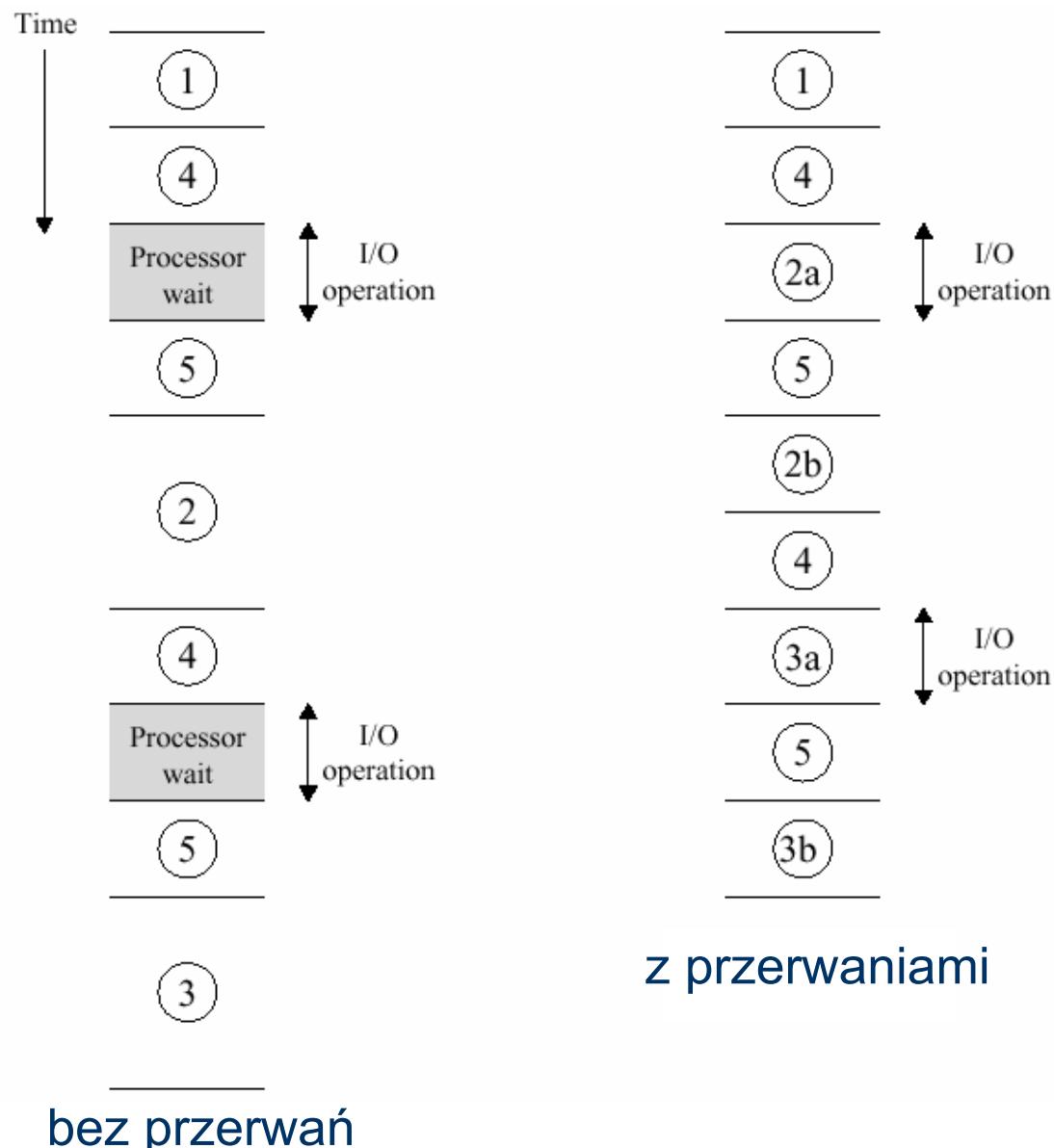


Koncepcja systemu przerwań



Przerwania – zysk czasu CPU

1,2,3 – program główny
4 – inicjacja operacji I/O
5 – obsługa przerwania



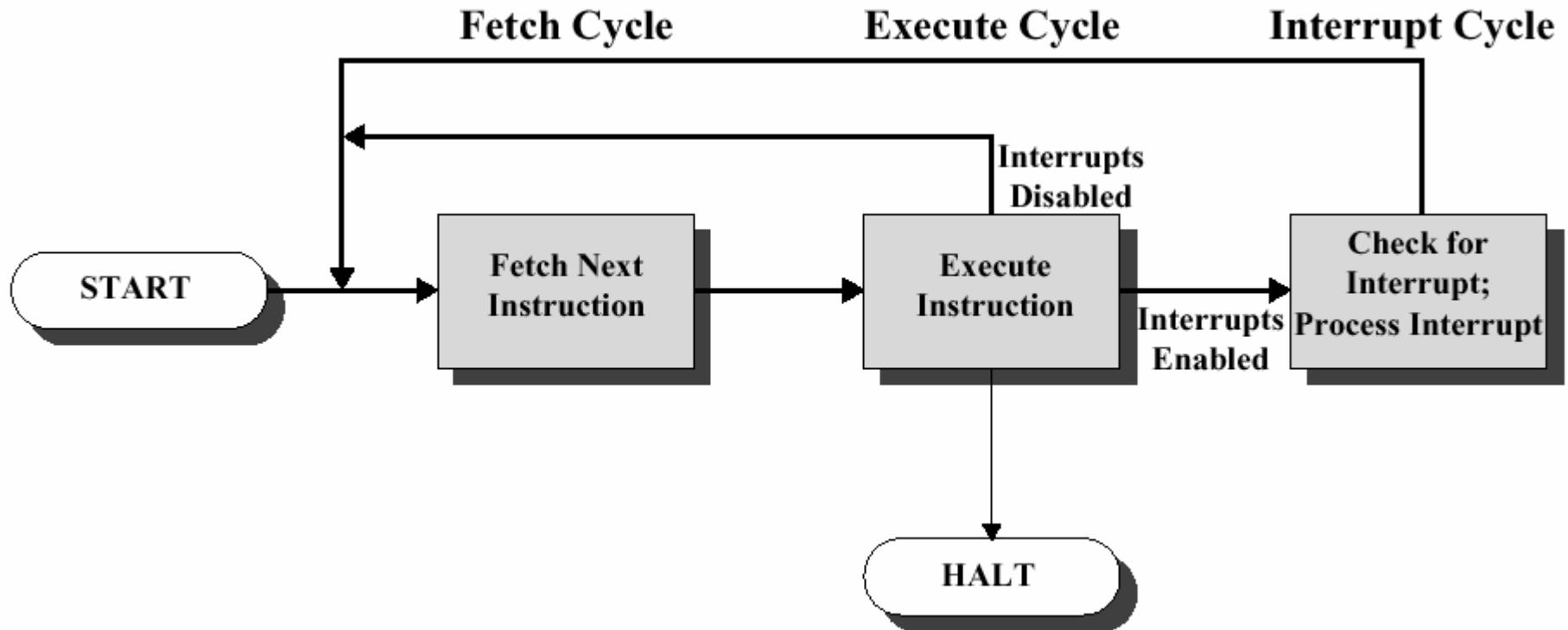
Przerwania zewnętrzne

- **Przerwania zewnętrzne** (*interrupts*) spowodowane zdarzeniem poza CPU, sygnalizowane sygnałem doprowadzonym do odpowiedniego wejścia procesora
 - **Maskowane** – sygnalizowane przez wejście INTR; CPU reaguje na przerwanie maskowane jeśli bit zezwolenia na przerwanie IF w rejestrze wskaźników (w Pentium EFLAGS) jest ustawiony
 - **Niemaskowane** – sygnalizowane przez wejście NMI; CPU zawsze reaguje na takie przerwanie

Przerwania wewnętrzne - wyjątki

- **Wyjątki (exceptions)** powodują takie same efekty jak przerwania zewnętrzne, ale są wynikiem realizacji programu
 - **Błędy** – w przypadku wykrycia błędu uniemożliwiającego wykonanie instrukcji CPU generuje wyjątek
 - **Wyjątki programowane** – generowane celowo przez programistę przy użyciu specjalnych instrukcji (INTO, INT3, INT, BOUND – Pentium)

Cykl rozkazu a przerwania

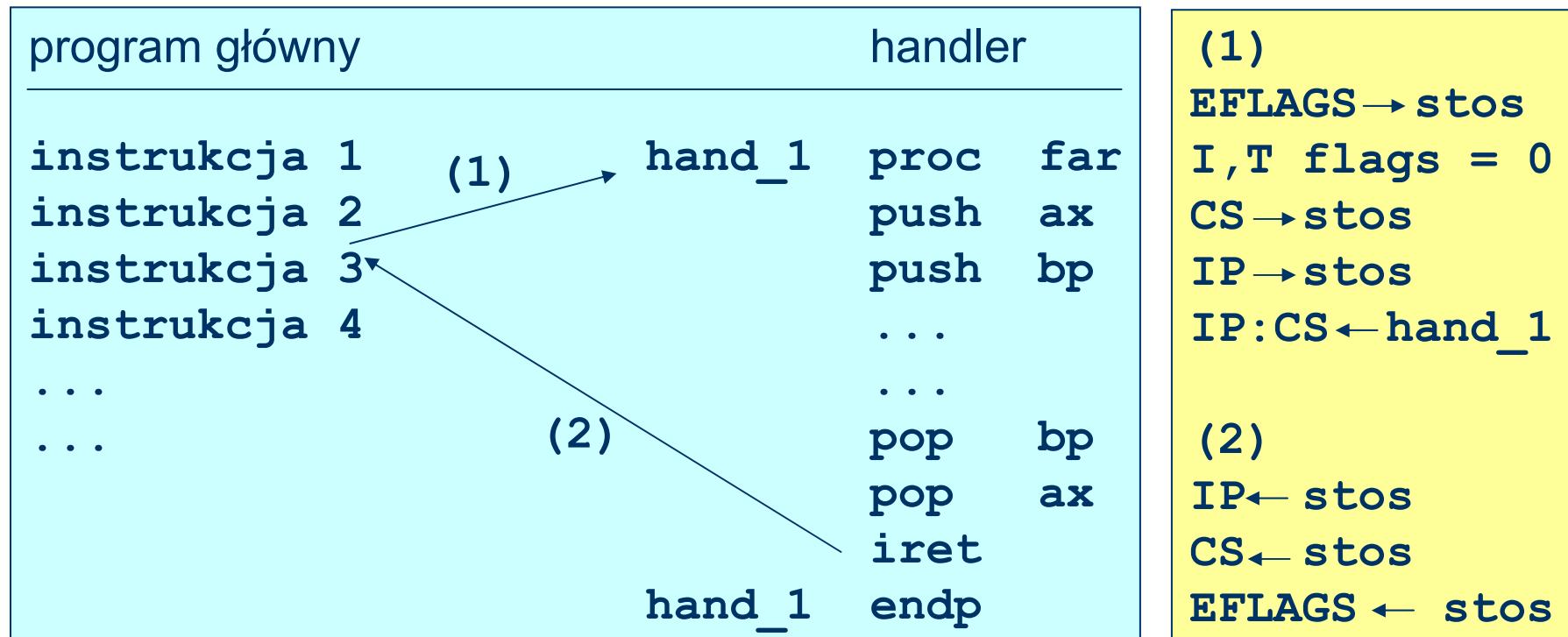


Przerwania – problemy projektowe

1. Jak zapobiec interferencji programu głównego i handlera (obydwa działają na tych samych zasobach CPU) ?
2. W jakim obszarze pamięci należy umieścić programy obsługi przerwań (handlery) ?
3. Jeśli w systemie jest wiele źródeł przerwań, jak rozpoznać, które z urządzeń zgłosiło przerwanie ?
4. Co zrobić, jeśli jednocześnie wystąpi więcej niż jedno żądanie przerwania ?

Przerwania – handler (problem 1)

Przykład realizacji współpracy programu głównego z handlerem (Pentium)



- czynności (1) i (2) są wykonywane automatycznie przez CPU
 - ochronę rejestrów w handlerze (w tym przypadku **ax** i **bp**) programista musi zorganizować samodzielnie

Adresy handlerów (problem 2)

- Sposób rozmieszczenia adresów handlerów przerwań i wyjątków jest w większości procesorów ustalony na stałe przez producenta CPU – użytkownik może definiować część adresów, reszta jest ustalona przez system
- Najczęściej adresy handlerów są umieszczone na samej górze (Motorola) lub na samym dole przestrzeni adresowej (Intel)

Przykład: adresy handlerów przerwań w MC68HC12

Vector Address	Source
\$FFFE-\$FFFF	System Reset
\$FFFC-\$FFFD	Clock Monitor Reset
\$FFFA-\$FFFB	COP Reset
\$FFF8-\$FFF9	Unimplemented Opcode Trap
\$FFF6-\$FFF7	Software Interrupt Instruction (SWI)
\$FFF4-\$FFF5	<u>XIRQ</u> Signal
\$FFF2-\$FFF3	<u>IRQ</u> Signal
\$FFC0-\$FFC1	Device-Specific Interrupt Sources

watchdog

NMI

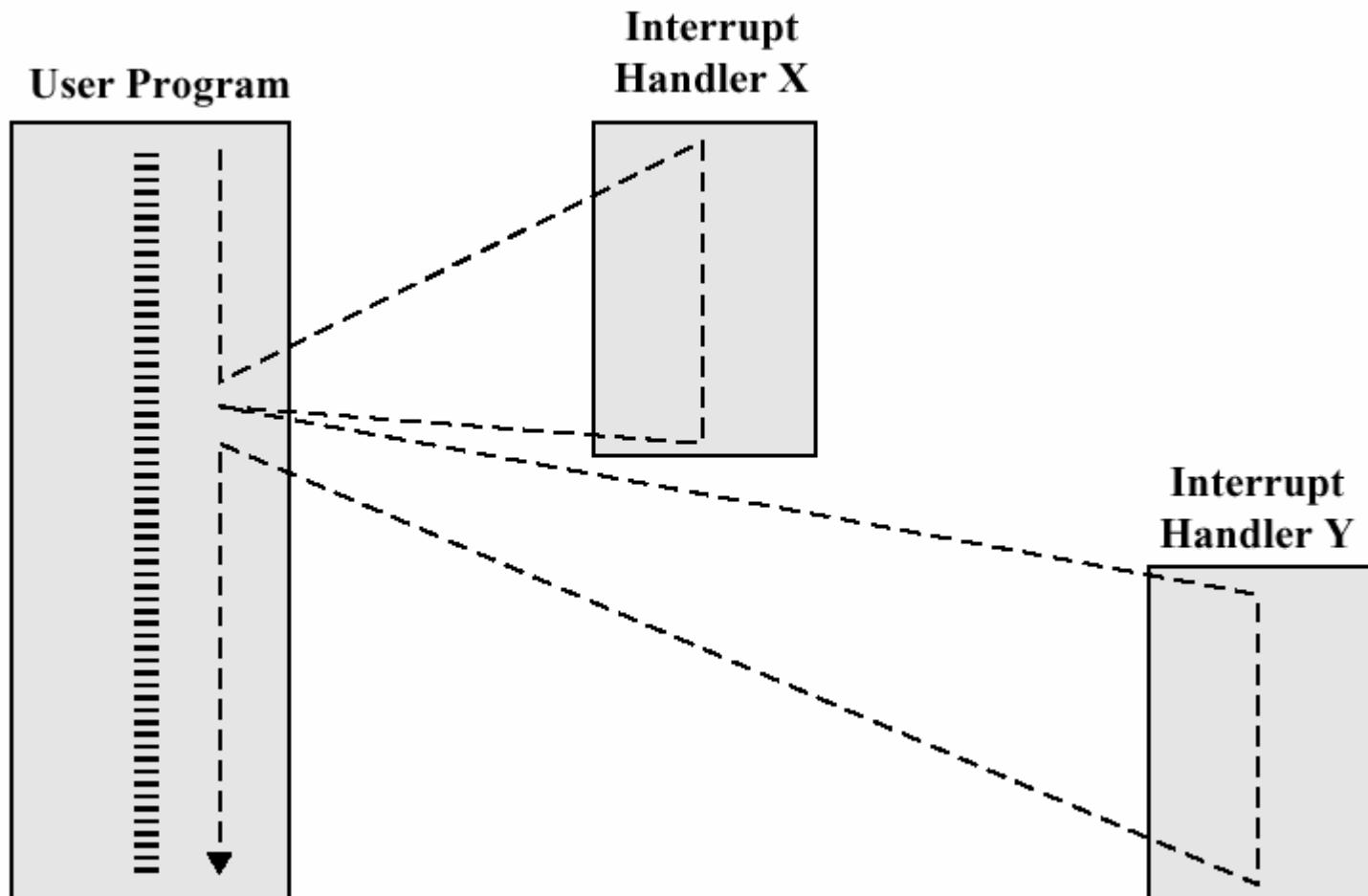
NMI

59 x NMI

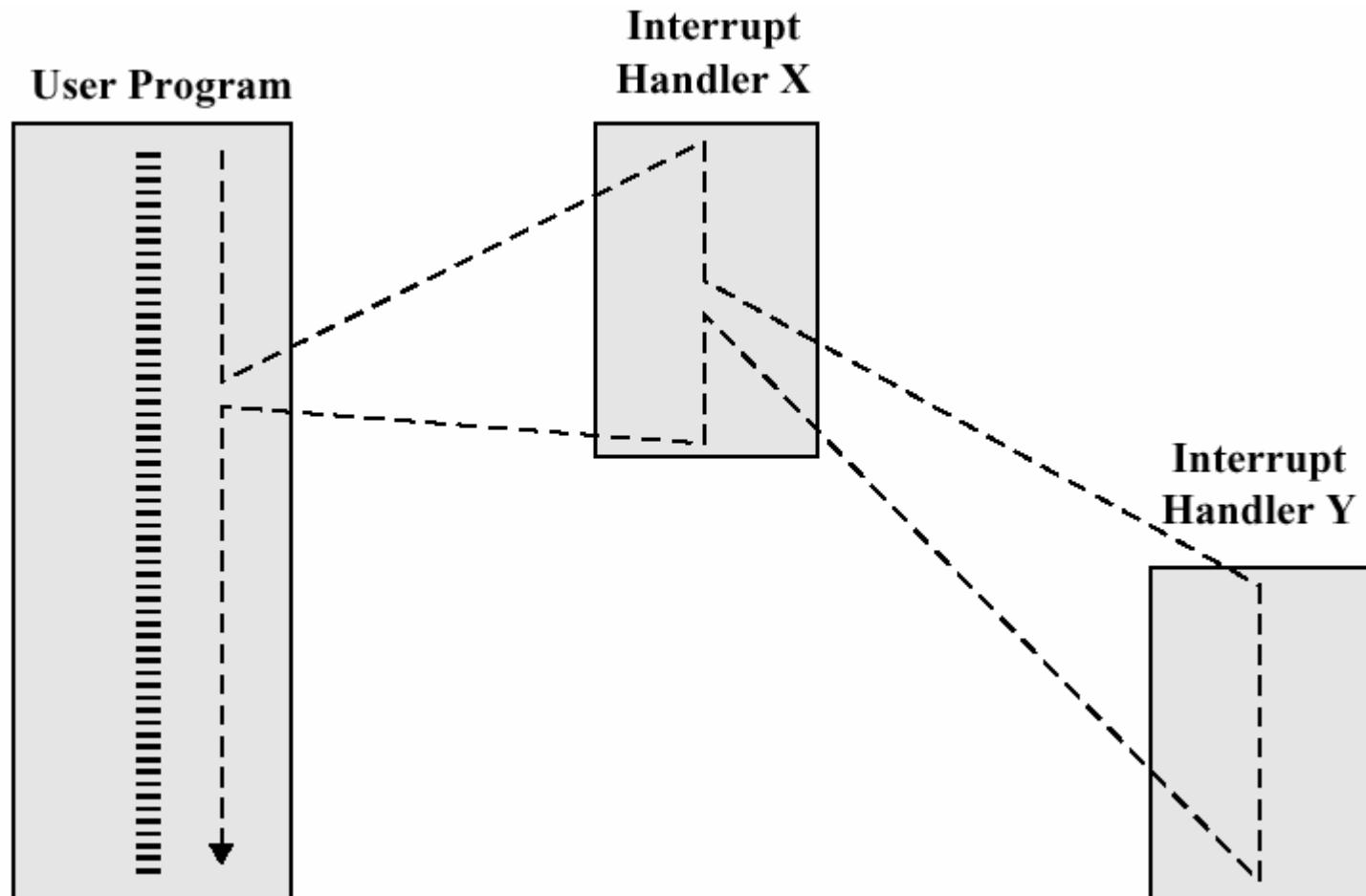
Przerwania wielokrotne (problemy 3 i 4)

- Zablokowanie (maskowanie przerwań)
 - CPU ignoruje kolejne sygnały przerwań i wykonuje aktualny handler
 - zgłoszenia przerwań są pamiętane; CPU zacznie je obsługiwać w kolejności zgłoszenia, po zakończeniu wykonywania aktualnego handlera
- Przerwania priorytetowe
 - obsługa przerwań o niższym priorytecie może być przerywana przez przerwania o wyższym priorytecie
 - po zakończeniu obsługi przerwania o wyższym priorytecie CPU wraca do obsługi przerwania o niższym priorytecie

Przerwania wielokrotne - sekwencyjne

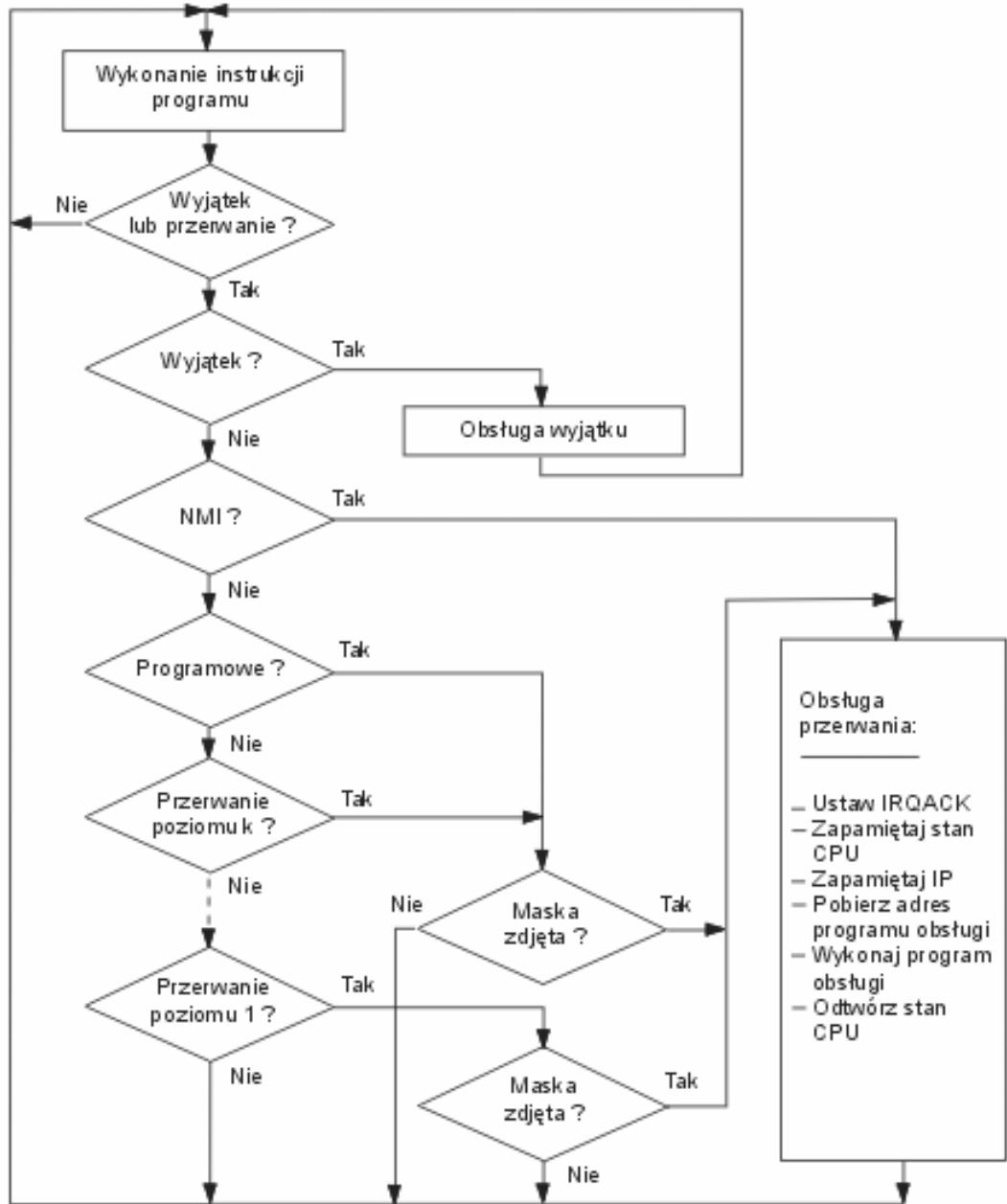


Przerwania wielokrotne - zagnieżdżone

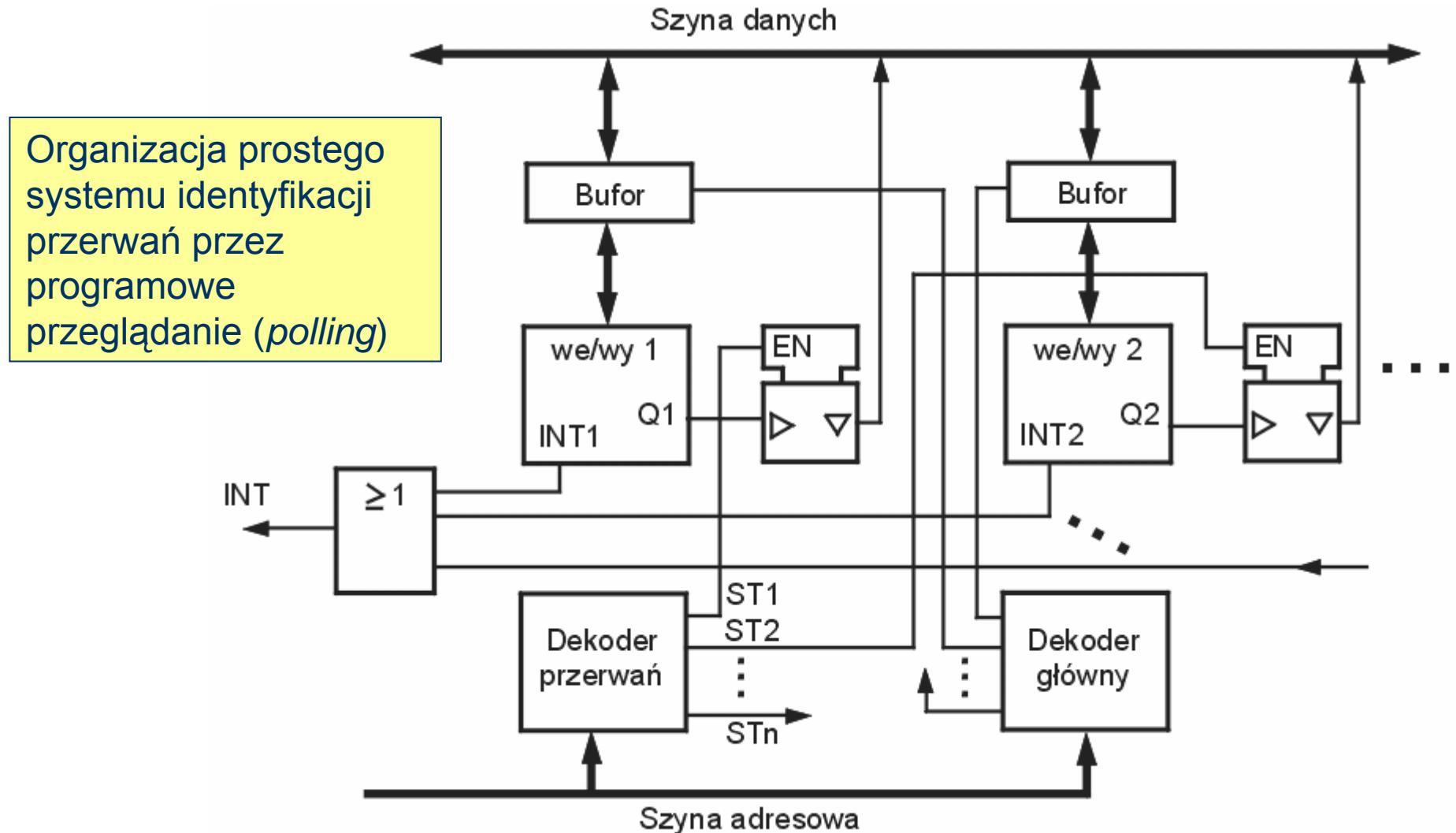


Cykł rozkazu

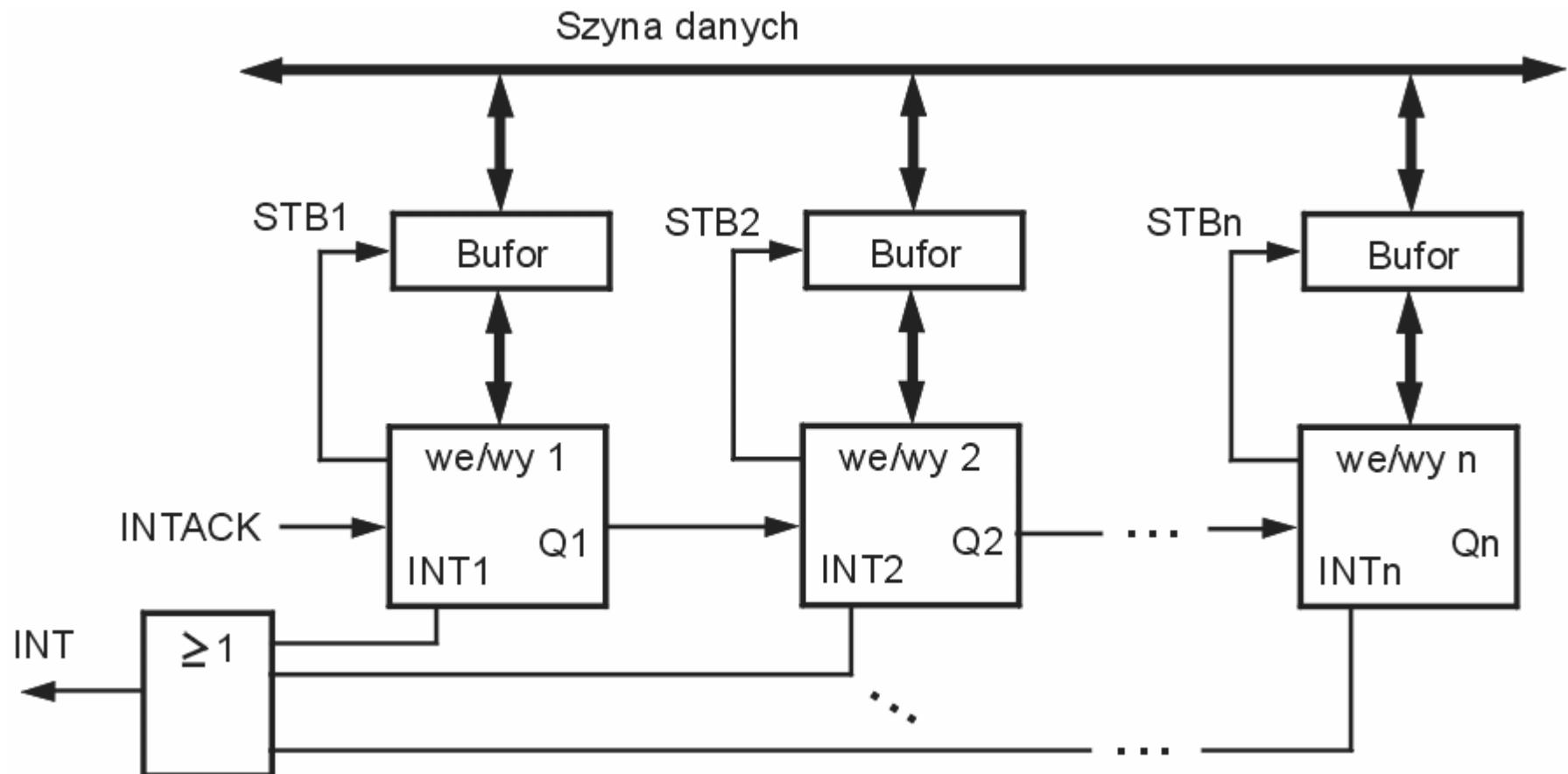
Diagram cyklu rozkazowego z uwzględnieniem obsługi wyjątków i przerwań



Identyfikacja źródła przerwań (1)

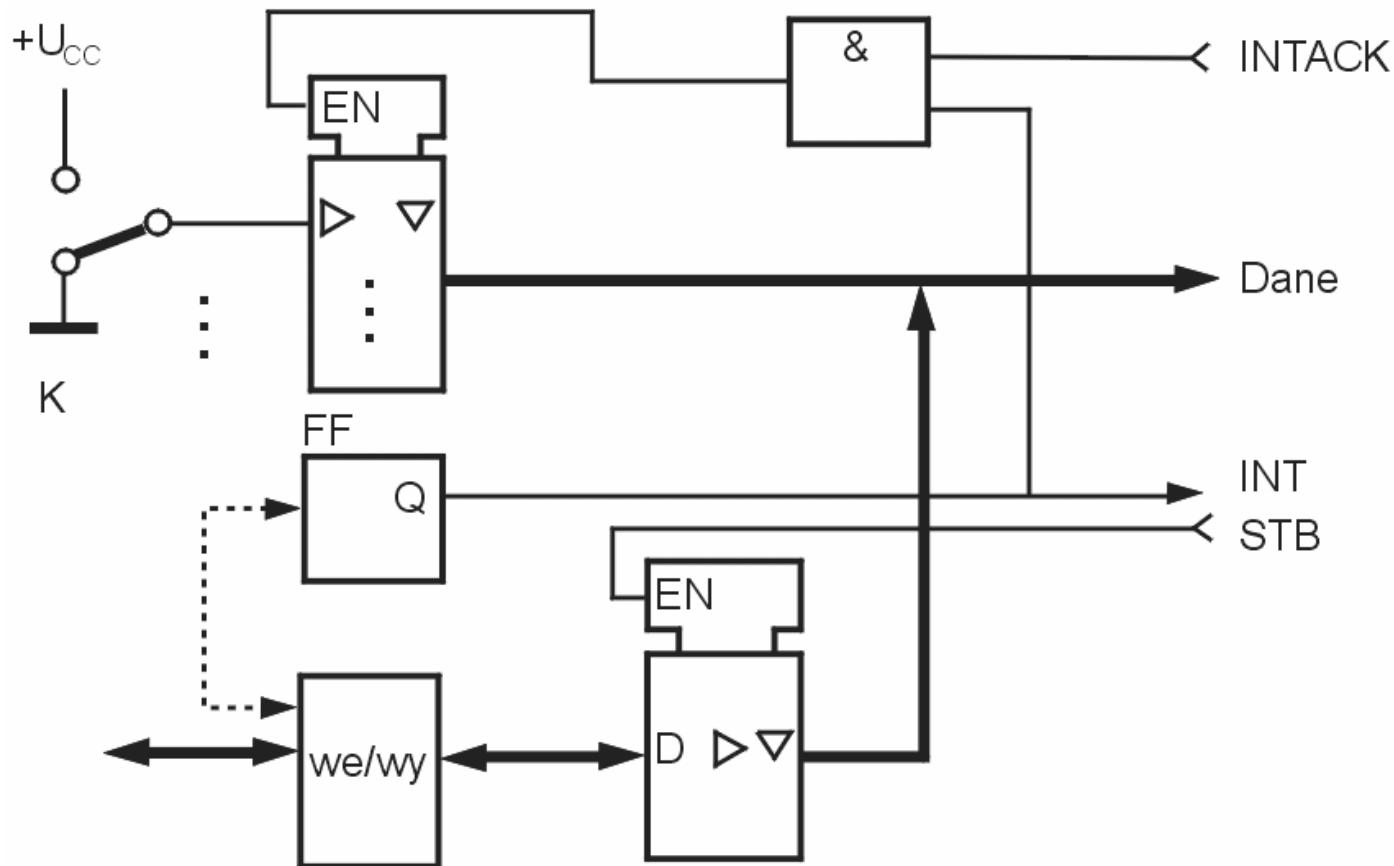


Identyfikacja źródła przerwań (2)



Organizacja systemu identyfikacji źródła przerwań z łańcuchowaniem urządzeń I/O (*daisy-chain*)

Identyfikacja źródła przerwań (3)



Prosty układ realizujący przerwania wektoryzowane

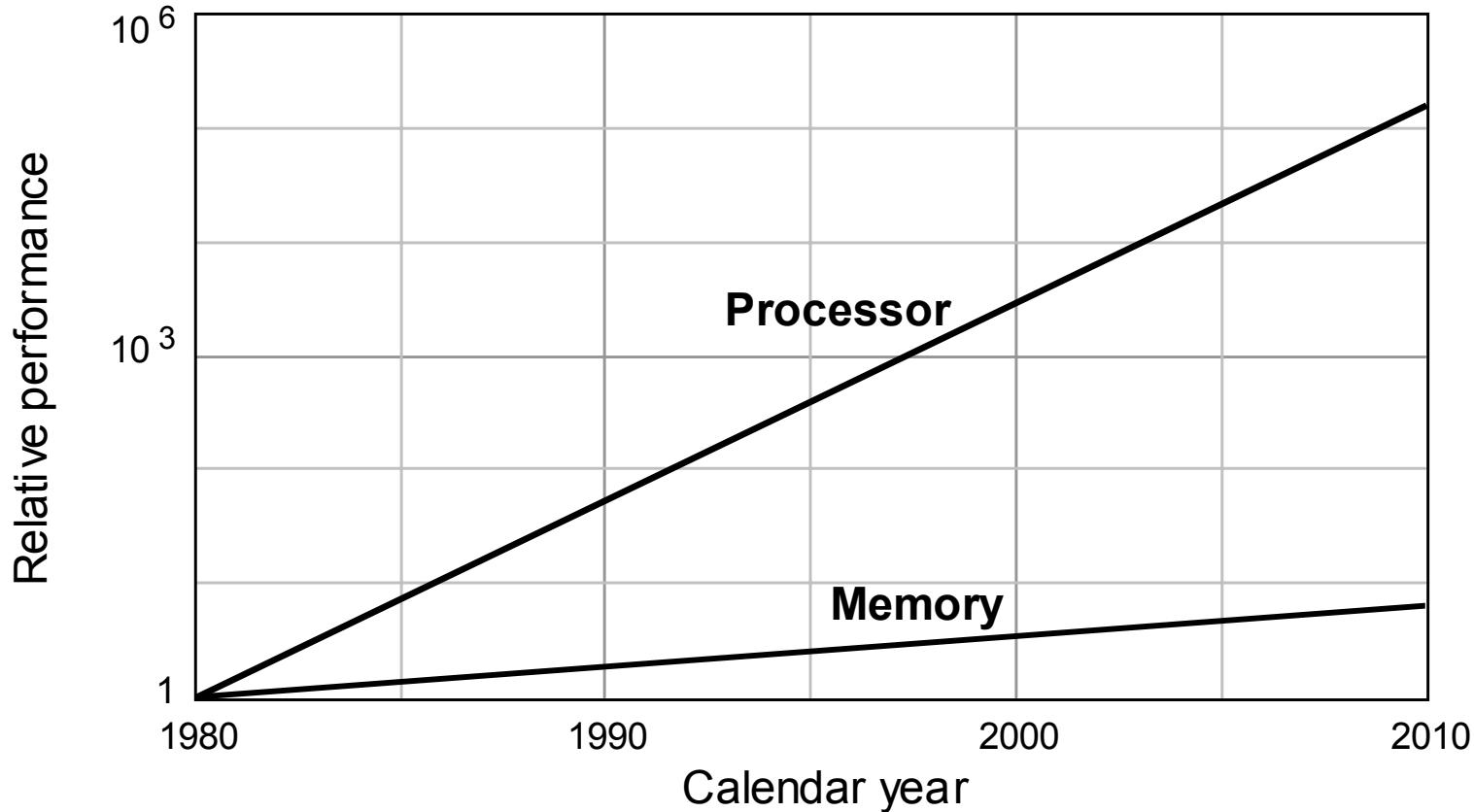
Podsumowanie

- Struktura i działanie CPU
 - rejestr wskaźników
 - plik rejestrów
 - segmentacja pamięci
- Stos
- Budowa i działanie jednostki sterującej
 - cykl wykonania rozkazu
 - warianty realizacji jednostki sterującej
 - mikroprogramowana jednostka sterująca
- Przerwania
 - koncepcja systemu przerwań, przerwania i wyjątki
 - handlery i ich adresy, przerwania wektoryzowane
 - przerwania wielokrotne
 - identyfikacja źródła przerwania

Organizacja i Architektura Komputerów

Pamięć cache

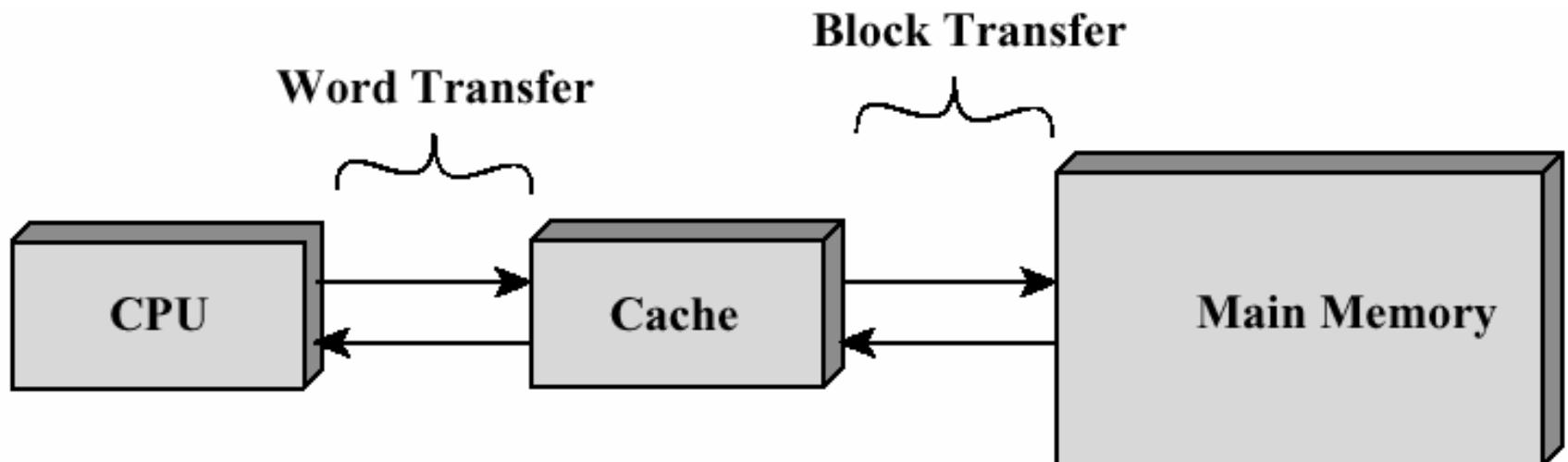
Wydajność: CPU i pamięć



Memory density and capacity have grown along with the CPU power and complexity, but memory speed has not kept pace.

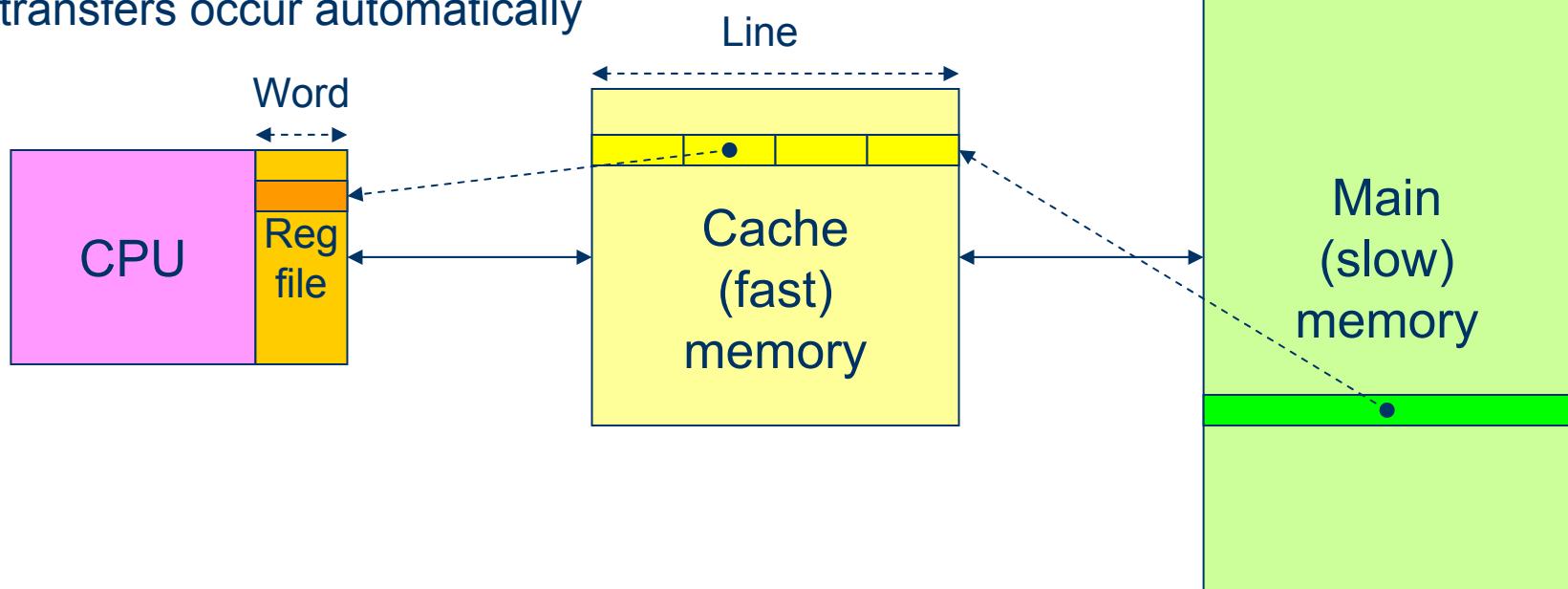
Pamięć cache – koncepcja

- niewielka, ale szybka pamięć RAM
- umieszczona między CPU a pamięcią główną (operacyjną)
- może być umieszczona w jednym układzie scalonym z CPU lub poza CPU

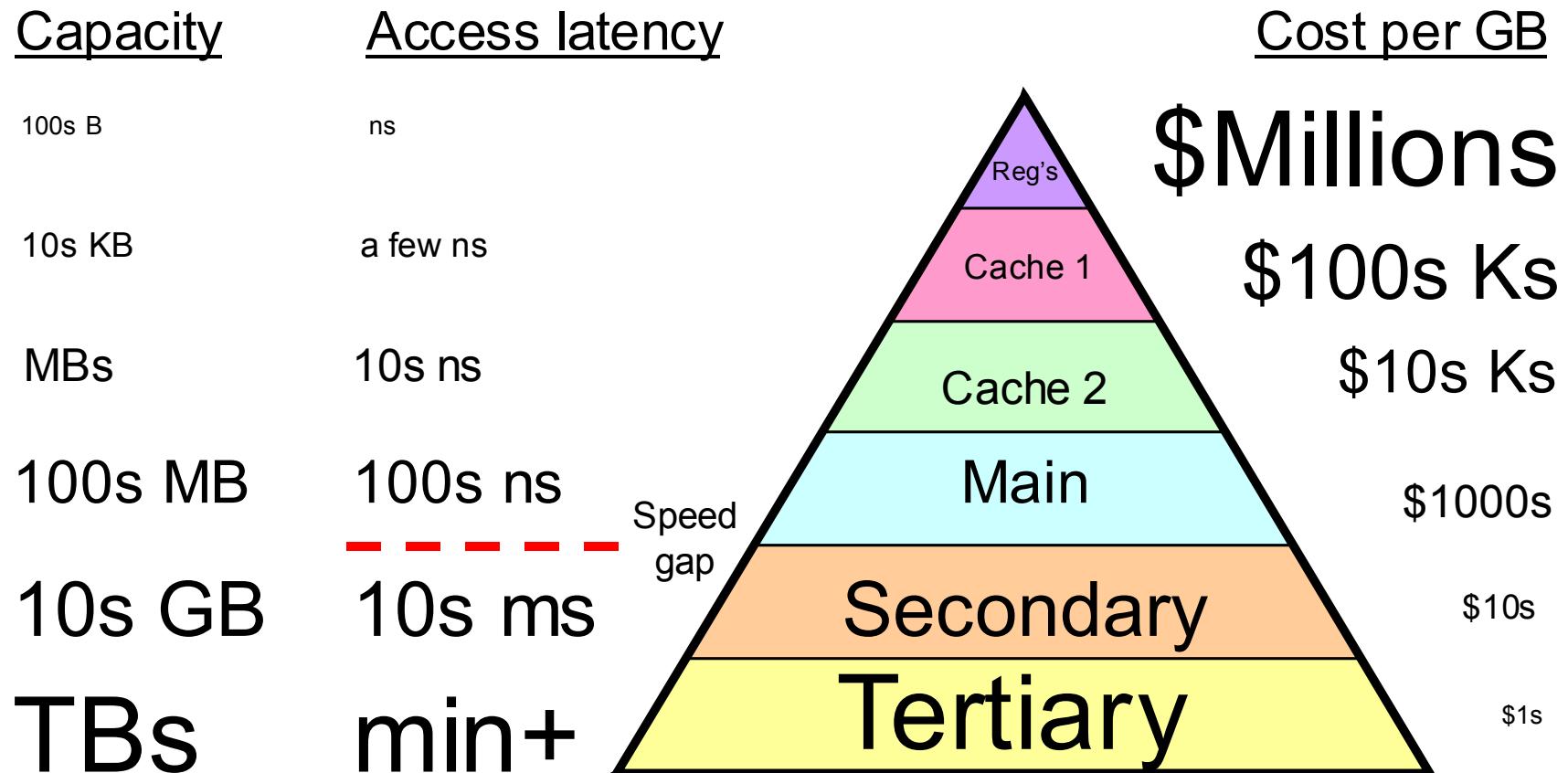


Pamięć cache – koncepcja

Cache is transparent to user;
transfers occur automatically



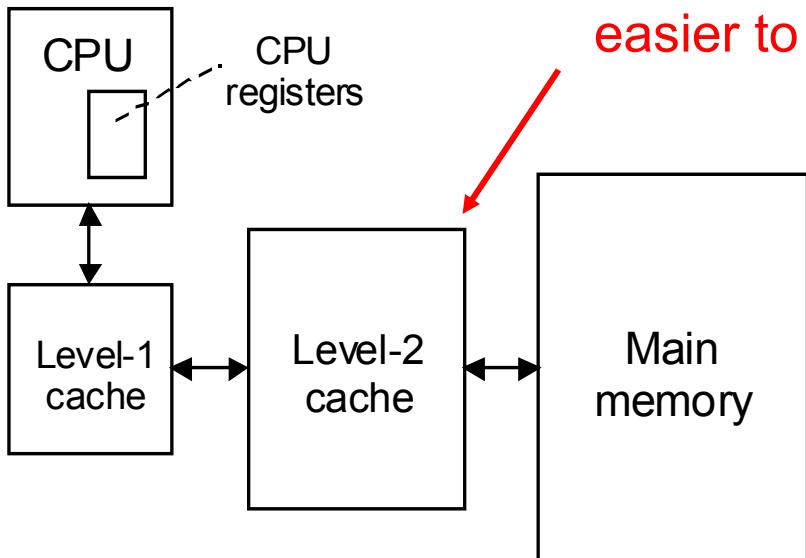
Hierarchiczny system pamięci



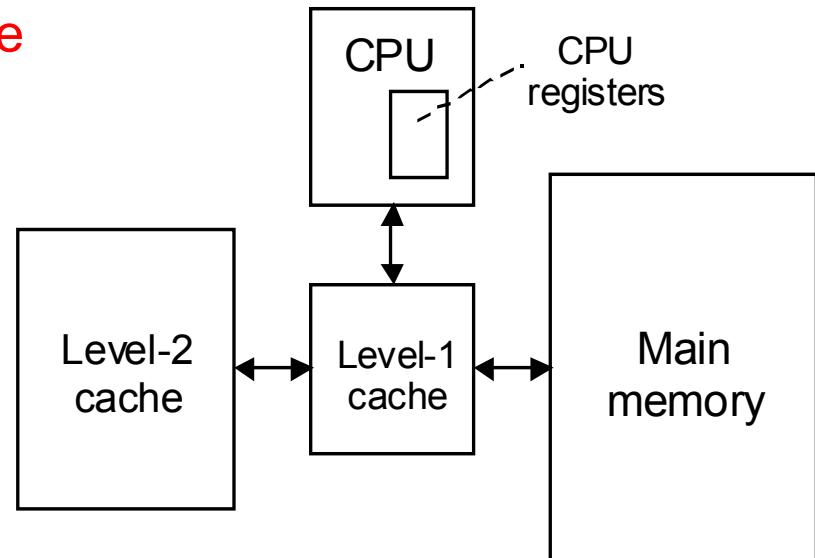
Names and key characteristics of levels in a memory hierarchy.

Połączenie CPU z cache

Cleaner and
easier to analyze



(a) Level 2 between level 1 and main



(b) Level 2 connected to “backside” bus

Cache memories act as intermediaries between the superfast processor and the much slower main memory.

Działanie pamięci cache

- CPU żąda odczytu pamięci
- sprawdza się, czy potrzebne dane znajdują się w cache
- jeśli tak, następuje szybki odczyt z cache
- jeśli nie, przesyła się potrzebny blok danych z pamięci głównej do cache
- następnie żądane dane są przesyłane do CPU
- pamięć cache jest wyposażona w znaczniki (*tags*) pozwalające ustalić, które bloki pamięci głównej są aktualnie dostępne w pamięci cache

Zasady lokalności

- Lokalność w sensie czasu: jeśli CPU odwołuje się do określonej komórki pamięci, jest prawdopodobne, że w najbliższej przyszłości nastąpi kolejne odwołanie do tej komórki
- Lokalność w sensie przestrzeni adresowej: jeśli CPU odwołuje się do określonej komórki pamięci, jest prawdopodobne, że w najbliższej przyszłości nastąpi odwołanie do sąsiednich komórek pamięci

Terminologia

- **cache hit** (trafienie): sytuacja, gdy żądana informacja znajduje się w pamięci cache wyższego poziomu
- **cache miss** (chybienie, brak trafienia): sytuacja, gdy żądanej informacji nie ma w pamięci cache wyższego poziomu
- **line, slot** (linijka, wiersz, blok): najmniejsza porcja (kwant) informacji przesyłanej między pamięcią cache, a pamięcią wyższego poziomu. Najczęściej stosuje się linijki o wielkości 32 bajtów

Terminologia cd.

- *hit rate* (współczynnik trafień): stosunek liczby trafień do łącznej liczby odwołań do pamięci
- *miss rate* (współczynnik chybień): $1 - \text{hit rate}$
- *hit time* (czas dostępu przy trafieniu): czas dostępu do żądanej informacji w sytuacji gdy wystąpiło trafienie. Czas ten składa się z dwóch elementów:
 - czasu potrzebnego do ustalenia, czy żądana informacja jest w pamięci cache
 - czasu potrzebnego do przesłania informacji z pamięci cache do procesora

Terminologia cd.

- *miss penalty* (kara za chybienie): czas potrzebny na wymianę wybranej linijki w pamięci cache i zastąpienie jej taką linijką z pamięci głównej, która zawiera żądaną informację
- czas *hit time* jest znacznie krótszy od czasu *miss penalty* (w przeciwnym przypadku stosowanie pamięci cache nie miałoby sensu!)

Problemy

- W jaki sposób ustalić, czy żądana informacja znajduje się w pamięci cache?
- Jeśli już ustalimy, że potrzebna informacja jest w pamięci cache, w jaki sposób znaleźć ją w tej pamięci (jaki jest jej adres?)
- Jeśli informacji nie ma w pamięci cache, to do jakiej linijki ją wpisać z pamięci głównej?
- Jeśli cache jest pełna, to według jakiego kryterium usuwać linijki aby zwolnić miejsce na linijki sprowadzane z pamięci głównej?
- W jaki sposób inicjować zawartość pamięci cache?

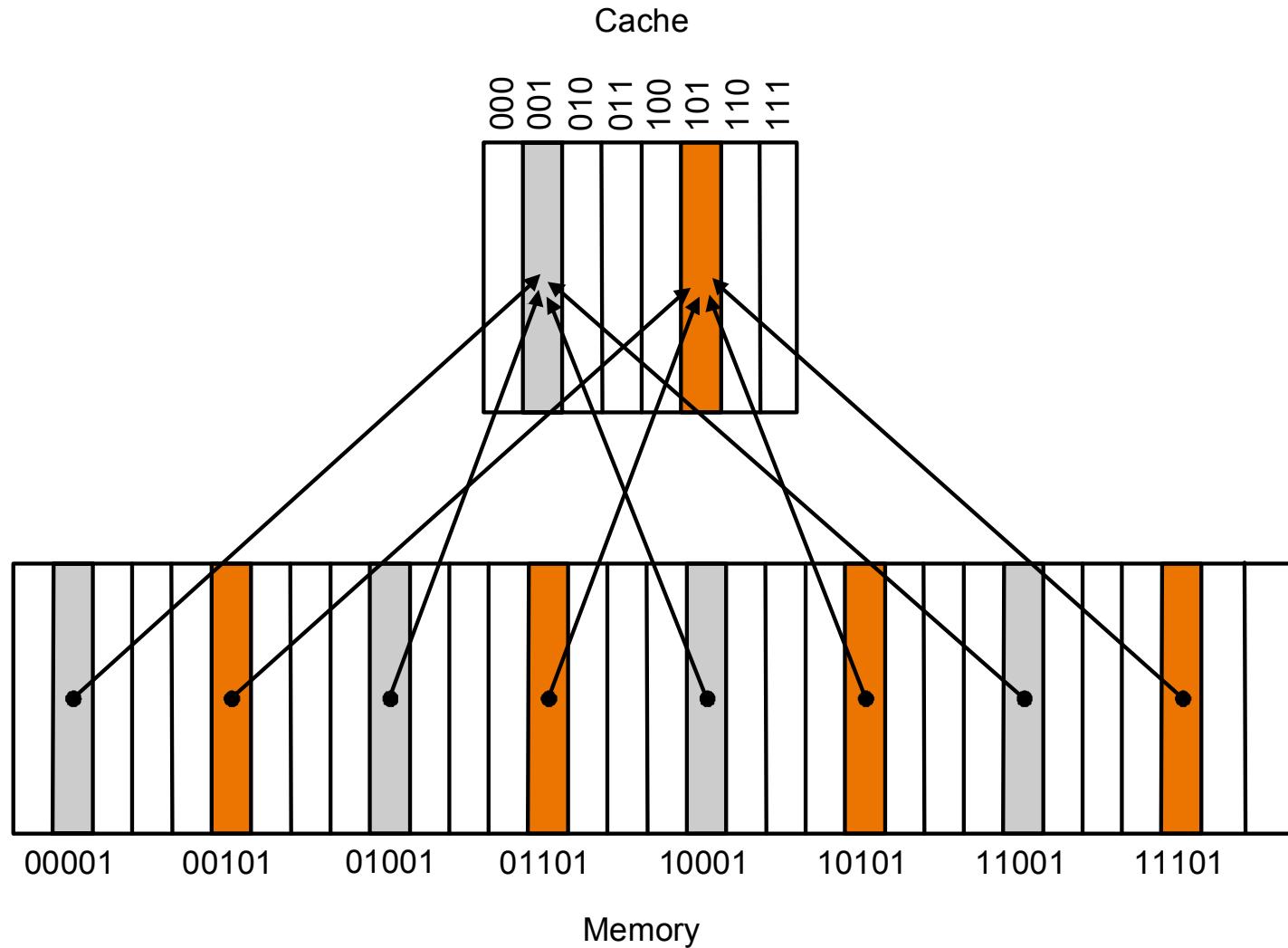
Odwzorowanie bezpośrednie

- *direct-mapped cache*
- każdy blok (linijka) w pamięci głównej jest odwzorowywana (przyporządkowana) tylko jednej linijce w pamięci cache
- ponieważ cache jest znacznie mniejsza od pamięci głównej, wiele różnych linijkę pamięci głównej jest odwzorowanych na tą samą linijkę w cache

Funkcja odwzorowująca

- Najprostsza metoda odwzorowania polega na wykorzystaniu najmniej znaczących bitów adresu
- Na przykład, wszystkie linijki w pamięci głównej, których adresy kończą się sekwencją 000 będą odwzorowane na tą samą linijkę w pamięci cache
- Powyższa metoda wymaga, by rozmiar pamięci cache (wyrażony w linijkach) był potęgą liczby 2

Odwzorowanie bezpośrednie

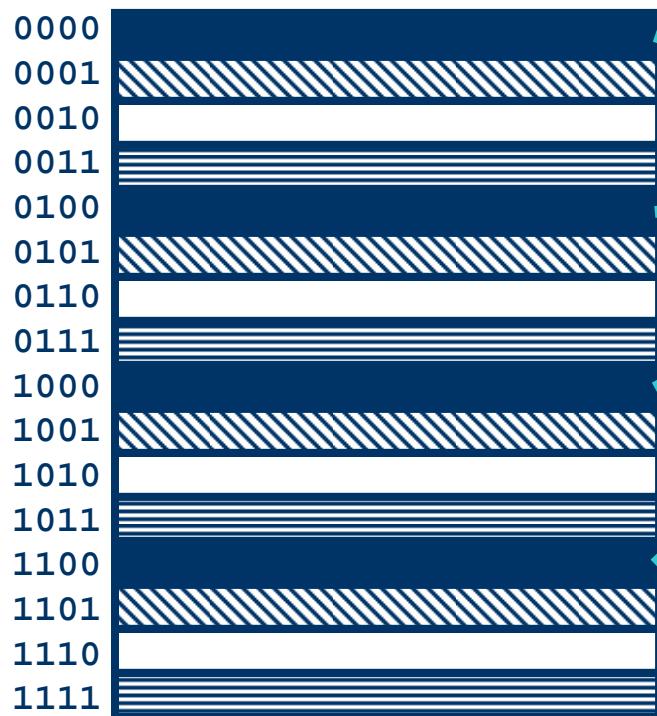


Co znajduje się w linijce 000?

- Wciąż nie wiemy, która linijka z pamięci głównej (z wielu możliwych) znajduje się aktualnie w konkretnej linijce pamięci cache
- Musimy zatem zapamiętać adres linijki przechowywanej aktualnie w linijkach pamięci cache
- Nie trzeba zapamiętywać całego adresu linijki, wystarczy zapamiętać tylko tę jego część, która nie została wykorzystana przy odwzorowaniu
- Tę część adresu nazywa się znacznikiem (*tag*)
- Znacznik związany z każdą linijką pamięci cache pozwala określić, która linijka z pamięci głównej jest aktualnie zapisana w danej linijce cache

Znaczniki – ilustracja

Pamięć główna



Dane

Znaczniki



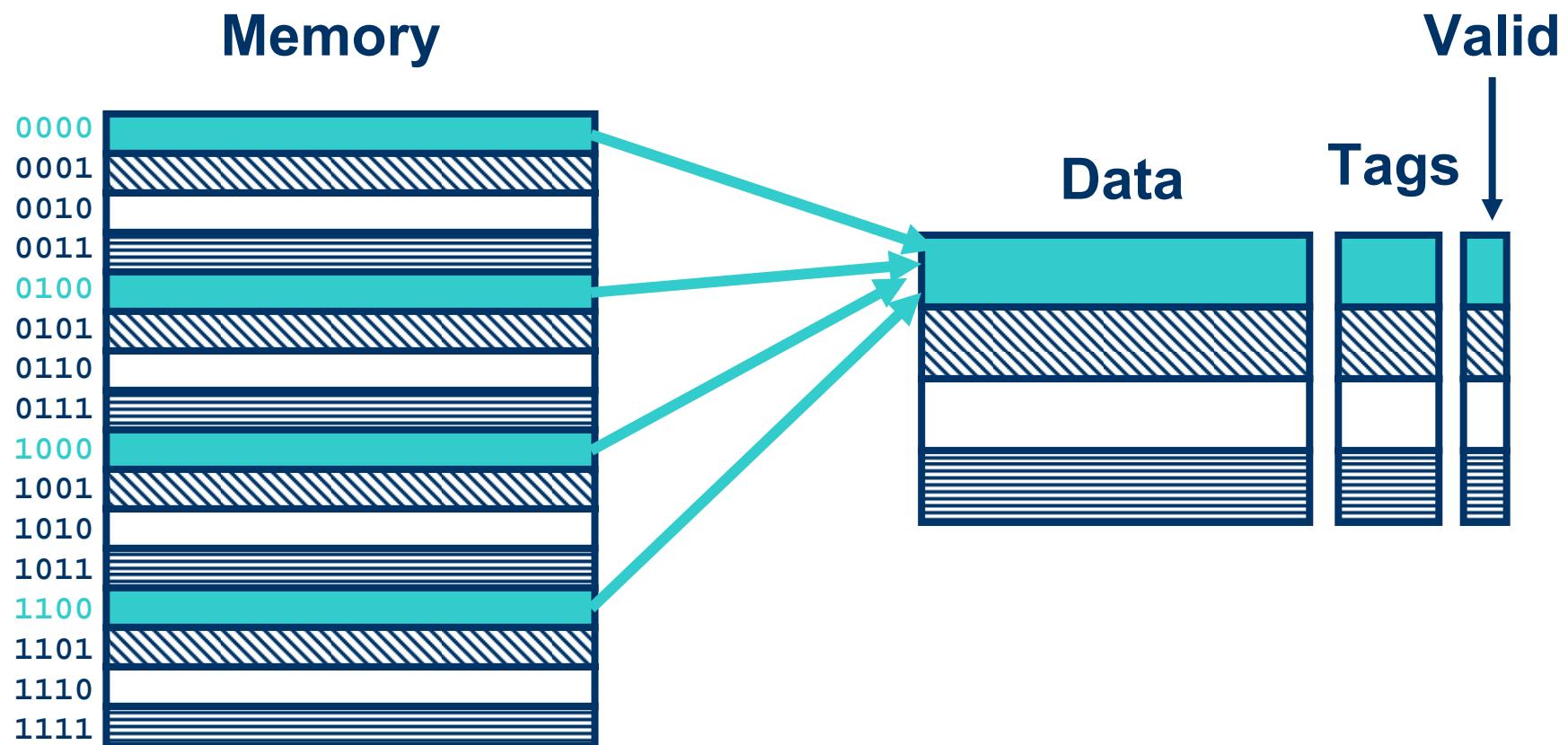
Iinicjalizacja cache

- Początkowo pamięć cache jest pusta
 - wszystkie bity w pamięci cache (włącznie z bitami znaczników) mają losowe wartości
- Po pewnym czasie pracy systemu pewna część znaczników ma określoną wartość, jednak pozostałe są nadal wielkościami losowymi
- Jak ustalić, które linijki zawierają rzeczywiste dane, a które nie zostały jeszcze zapisane?

Bit ważności (*valid bit*)

- Należy dodać przy każdej linijce pamięci cache dodatkowy bit, który wskazuje, czy linijka zawiera ważne dane, czy też jej zawartość jest przypadkowa
- Na początku pracy systemu bity ważności są zerowane, co oznacza, że pamięć cache nie zawiera ważnych danych
- Przy odwołaniach CPU do pamięci, w trakcie sprawdzania czy potrzebne dane są w cache należy ignorować linijki z bitem ważności równym 0
- Przy zapisie danych z pamięci głównej do pamięci cache należy ustawić bit ważności linijki

Rewizyta w cache



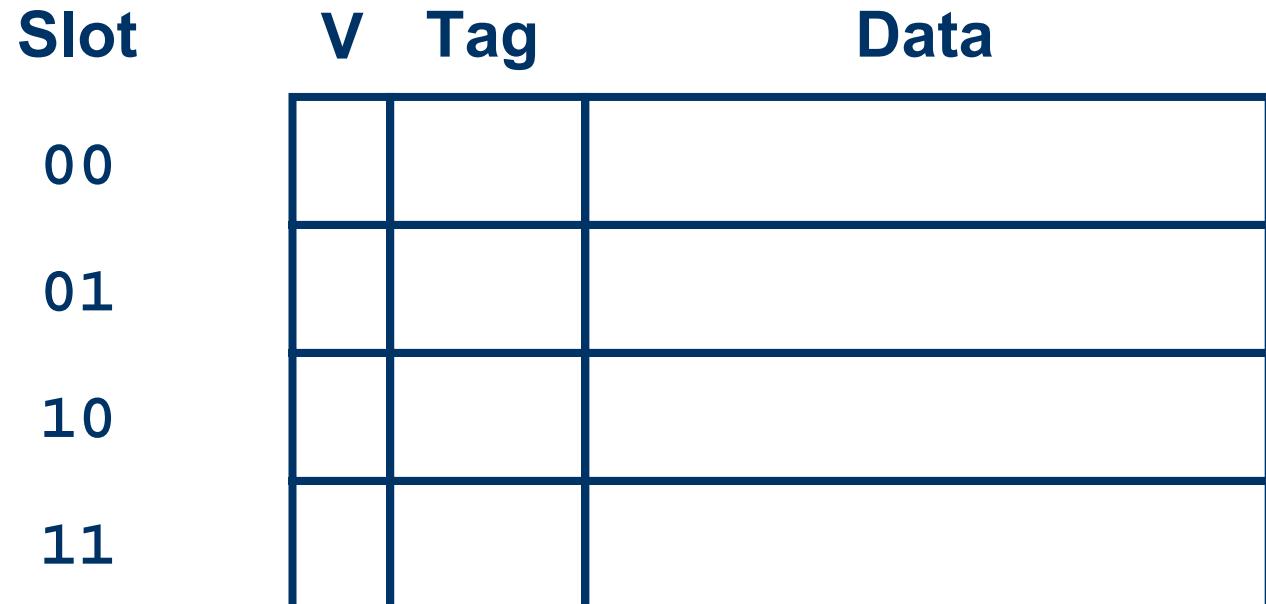
Prosta symulacja cache

- Użyjemy prostej struktury z pamięcią cache L1 zbudowaną tylko z czterech linijek, podobną do opisanej wcześniej
- Założymy, że na początku bity ważności zostały wyzerowane

Założenie: sekwencja odwołań CPU do pamięci

Adres	Adres binarny	Linijka	hit / miss
3	0011	11 (3)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>
3	0011	11 (3)	<i>hit</i>
2	0010	10 (2)	<i>miss</i>
4	0100	00 (0)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>
3	0011	11 (3)	<i>hit</i>
2	0010	10 (2)	<i>miss</i>
4	0100	00 (0)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>



Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>
3	0011	11 (3)	<i>hit</i>
2	0010	10 (2)	<i>miss</i>
4	0100	00 (0)	<i>miss</i>
8	1000	00 (0)	<i>miss</i>

Inicjalizacja
bitów
ważności

Slot	V	Tag	Data
00	0		
01	0		
10	0		
11	0		

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	miss
8	1000	00 (0)	miss
3	0011	11 (3)	hit
2	0010	10 (2)	miss
4	0100	00 (0)	miss
8	1000	00 (0)	miss

First Access Slot V Tag Data

00	0		
01	0		
10	0		
11	1	00	Mem[3]

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	miss
8	1000	00 (0)	miss
3	0011	11 (3)	hit
2	0010	10 (2)	miss
4	0100	00 (0)	miss
8	1000	00 (0)	miss

2nd Access Slot V Tag Data

00	1	10	Mem[8]
01	0		
10	0		
11	1	00	Mem[3]

→

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	miss
8	1000	00 (0)	miss
3	0011	11 (3)	hit
2	0010	10 (2)	miss
4	0100	00 (0)	miss
8	1000	00 (0)	miss

3rd Access Slot V Tag Data

00	1	10	Mem[8]
01	0		
10	0		
11	1	00	Mem[3]

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	miss
8	1000	00 (0)	miss
3	0011	11 (3)	hit
2	0010	10 (2)	miss
4	0100	00 (0)	miss
8	1000	00 (0)	miss

4th Access Slot V Tag Data

00	1	10	Mem[8]
01	0		
10	1	00	Mem[2]
11	1	00	Mem[3]

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	miss
8	1000	00 (0)	miss
3	0011	11 (3)	hit
2	0010	10 (2)	miss
4	0100	00 (0)	miss
8	1000	00 (0)	miss



5th Access Slot V Tag Data

00	1	01	Mem[4]
01	0		
10	1	00	Mem[2]
11	1	00	Mem[3]

Address	Binary Address	Slot	hit or miss
3	0011	11 (3)	miss
8	1000	00 (0)	miss
3	0011	11 (3)	hit
2	0010	10 (2)	miss
4	0100	00 (0)	miss
8	1000	00 (0)	miss



6th Access Slot V Tag Data

00	1	10	Mem[8]
01	0		
10	1	00	Mem[2]
11	1	00	Mem[3]

Problem

- Założenia projektowe:
 - 32-bitowe adresy (2^{32} bajtów w pamięci głównej, 2^{30} słów 4-bajtowych)
 - 64 KB cache (16 K słów). Każda linijka przechowuje 1 słowo
 - Odwzorowanie bezpośrednie (*Direct Mapped Cache*)
- Ile bitów będzie miał każdy znacznik?
- Ile różnych linijk z pamięci głównej będzie odwzorowanych na tę samą linijkę w pamięci cache?
- Ile bitów będzie zajmować łącznie kompletna linijka w pamięci cache? (data + tag + valid).

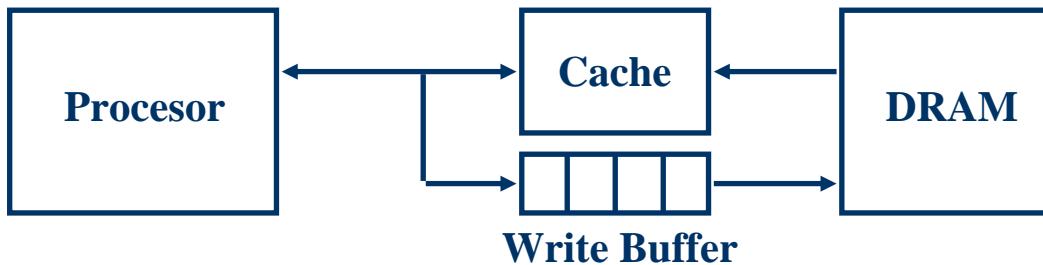
Rozwiążanie

- Pamięć zawiera 2^{30} słów
- Cache zawiera $16K = 2^{14}$ linijek (słów).
- Każda linijka może przechowywać jedną z $2^{30}/2^{14} = 2^{16}$ linijek pamięci głównej, stąd znacznik musi mieć 16 bitów
- 2^{16} linijek w pamięci głównej ma łączną pojemność równą $64K$ linijek – taki obszar jest przyporządkowany każdej linijce w pamięci cache
- Łączna pojemność pamięci cache wyrażona w bitach wynosi $2^{14} \times (32+16+1) = 49 \times 16K = 784$ Kb
(98 KB!)

Problem z zapisem

- Write through – zapis jednoczesny: dane są zapisywane jednocześnie do cache i pamięci głównej.
 - Zaleta: zapewnienie stałej aktualności danych w pamięci głównej (*memory consistency*)
 - Wada: duży przepływ danych do pamięci
- Write back – zapis opóźniony: aktualizuje się tylko pamięć cache. Fakt ten odnotowuje się w specjalnym dodatkowym bicie (*dirty, updated*). Przy usuwaniu linijki z cache następuje sprawdzenie bitu i ewentualna aktualizacja linijki w pamięci głównej.
 - Zaleta: mniejszy przepływ danych do pamięci
 - Wada: problemy ze spójnością danych w pamięci

Zapis jednoczesny (*write through*)

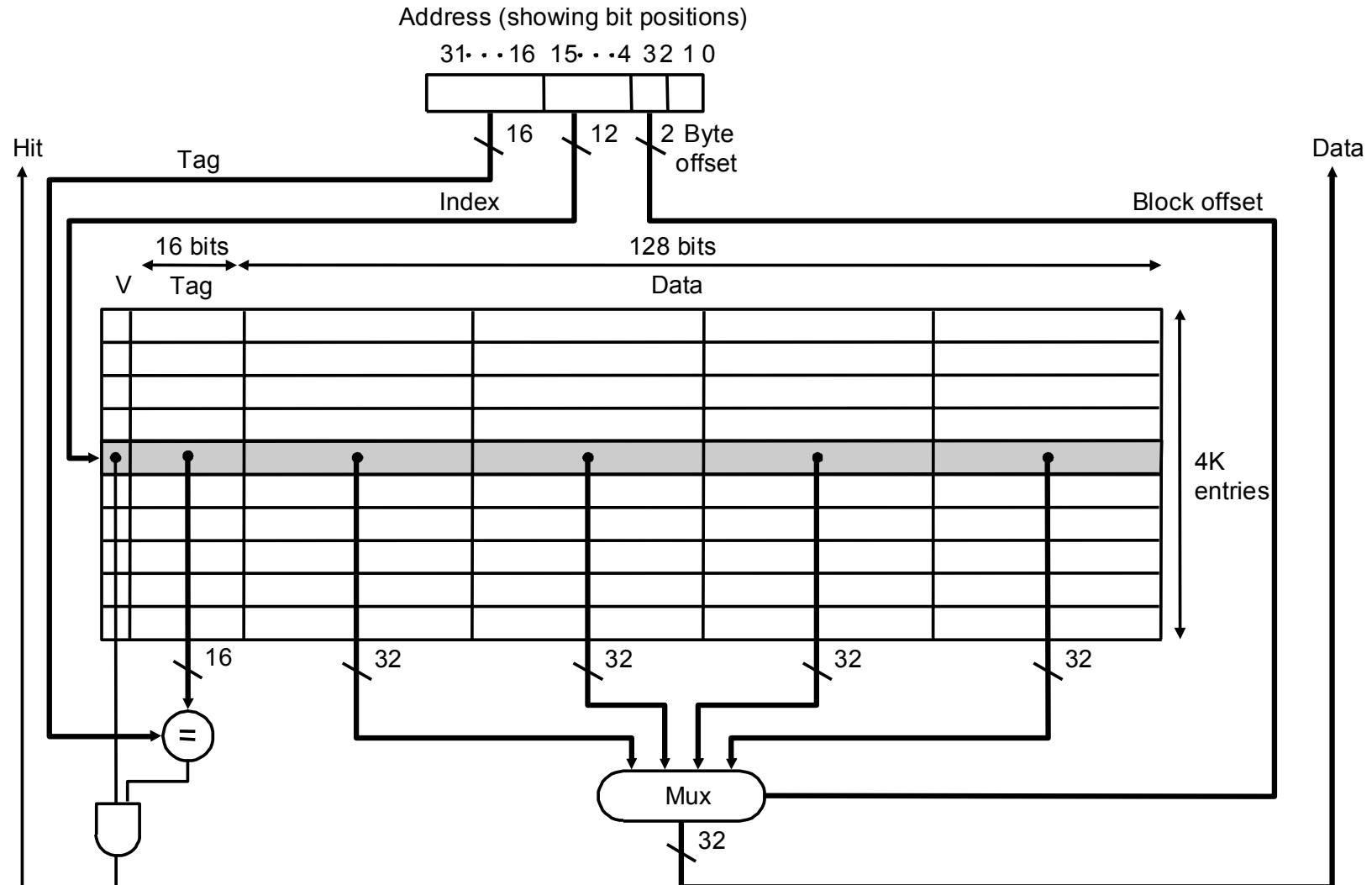


- W metodzie *write through* procesor zapisuje dane jednocześnie do cache i do pamięci głównej
 - w zapisie do pamięci głównej pośredniczy bufor
 - przepisywanie danych z bufora do pamięci głównej obsługuje specjalizowany kontroler
- Bufor jest niewielką pamięcią FIFO (*first-in-first-out*)
 - typowo zawiera 4 pozycje
 - metoda działa dobrze, jeśli częstotliwość operacji zapisu jest znacznie mniejsza od 1 / DRAM write cycle

Projekt cache - przykład

- Założenia projektowe:
 - 4 słowa w linijce
 - do przeprowadzenia odwzorowania pamięci używamy adresów linijk
 - adres linijk jest określany jako adres/4.
 - przy odwołaniu ze strony CPU potrzebne jest pojedyncze słowo, a nie cała linijka (można wykorzystać multiplekser sterowany dwoma najmniej znaczącymi bitami adresu)
 - Cache 64KB
 - 16 Bajtów w linijce
 - 4K linijk

Projekt cache - rozwiążanie



Wielkość linijki a wydajność

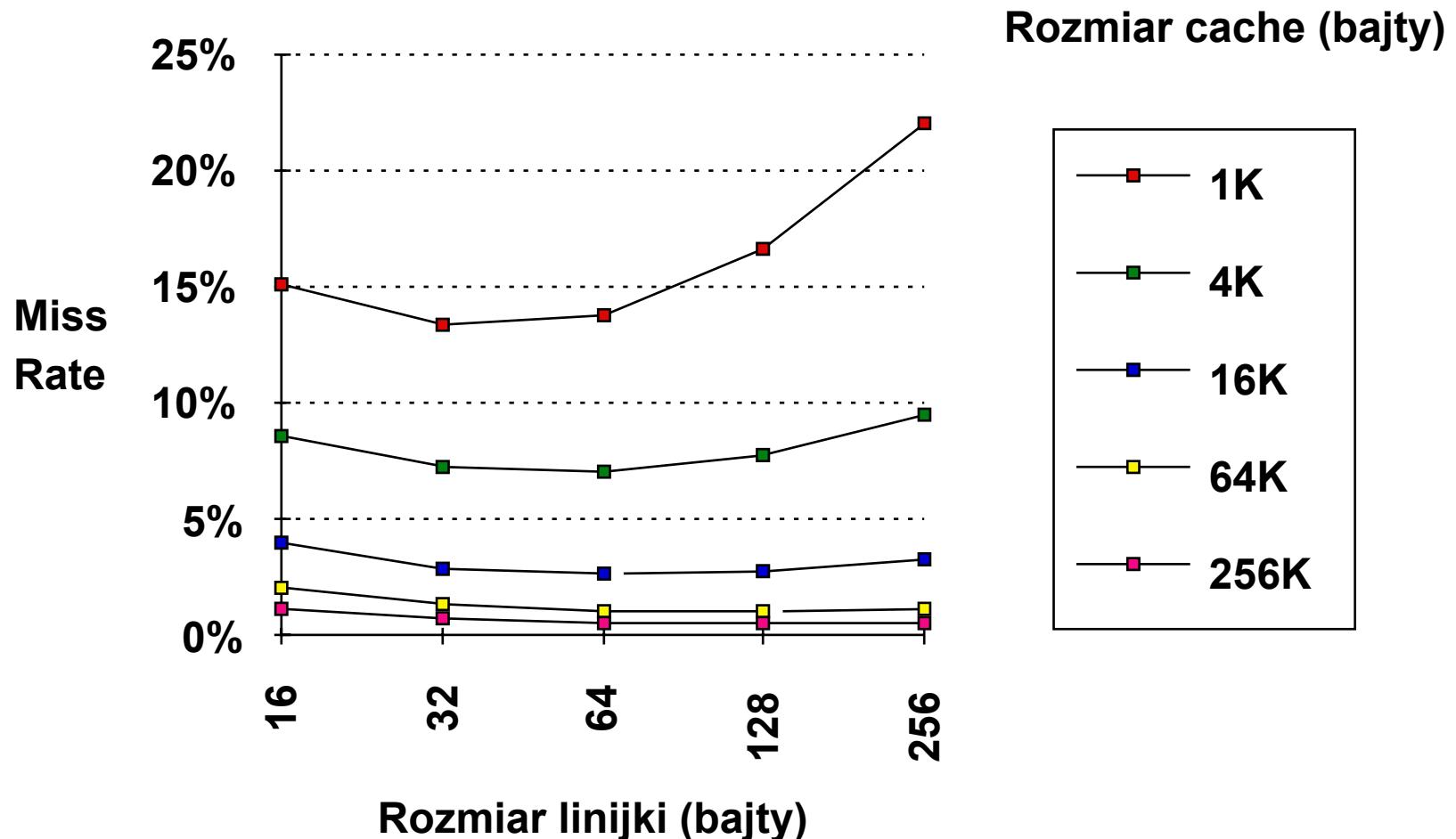
Przykład: DecStation 3100 cache z linijką o rozmiarze 1 lub 4 słów

Program	Rozmiar	Miss Rate		
	linijki	Instrukcje	Dane	Instr. + Dane
gcc	1	6.1%	2.1%	5.4%
gcc	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
spice	4	0.3%	0.6%	0.4%

Czy duża linijka jest lepsza?

- Zwiększenie rozmiaru linijki (przy ustalonym rozmiarze pamięci cache) oznacza, że liczba linijek przechowywanych w pamięci jest mniejsza
 - konkurencja o miejsce w pamięci cache jest większa
 - parametr *miss rate* wzrasta
- Proszę rozważyć skrajny przypadek, gdy cała pamięć cache tworzy jedną dużą linijkę!

Rozmiar linijki a *miss rate*



Rozmiar linijki a *miss penalty*

- Im większy jest rozmiar linijki, tym więcej czasu potrzeba na przesłanie linijki między pamięcią główną a pamięcią cache
- W przypadku chybienia czas sprowadzenia linijki wzrasta (parametr *miss penalty* ma większą wartość)
- Można budować pamięci zapewniające transfer wielu bajtów w jednym cyklu odczytu, ale tylko dla niewielkiej liczby bajtów (typowo 4)
- Przykład: hipotetyczne parametry dostępu do pamięci głównej:
 - 1 cykl CPU na wysłanie adresu
 - 15 cykli na inicjację każdego dostępu
 - 1 cykl na transmisję każdego słowa
- Parametr *miss penalty* dla linijki złożonej z 4 słów:
$$1 + 4 \times 15 + 4 \times 1 = 65 \text{ cykli.}$$

Wydajność cache

- Średni czas dostępu do pamięci
AMAT – *average memory access time*
AMAT = Hit time + Miss rate × Miss penalty
- Poprawa wydajności pamięci cache polega na skróceniu czasu AMAT przez:
 1. ograniczenie chybień (zmnieszenie *miss rate*, zwiększenie *hit rate*)
 2. zmniejszenie *miss penalty*
 3. zmniejszenie *hit time*

Wydajność cache

- Wydajność cache zależy głównie od dwóch czynników:
 - *miss rate*
 - Zależy od organizacji procesora (hardware) i od przetwarzanego programu (*miss rate* może się zmieniać).
 - *miss penalty*
 - Zależy tylko od organizacji komputera (organizacji pamięci i czasu dostępu do pamięci)
- Jeśli podwoimy częstotliwość zegara nie zmienimy żadnego z tych parametrów (czas dostępu do pamięci pozostanie taki sam)
- Dlatego parametr *speedup* wzrośnie mniej niż dwukrotnie

Wydajność cache - przykład

- Założymy, że w pamięci cache z odwzorowaniem bezpośrednim co 64 element pamięci jest odwzorowany w tej samej linijce. Rozważmy program:

```
for (i=0;i<10000;i++) {  
    a[i] = a[i] + a[i+64] + a[i+128];  
    a[i+64] = a[i+64] + a[i+128];  
}
```

- $a[i]$, $a[i+64]$ i $a[i+128]$ używają tej samej linijki
- Pamięć cache jest w powyższym przykładzie bezużyteczna

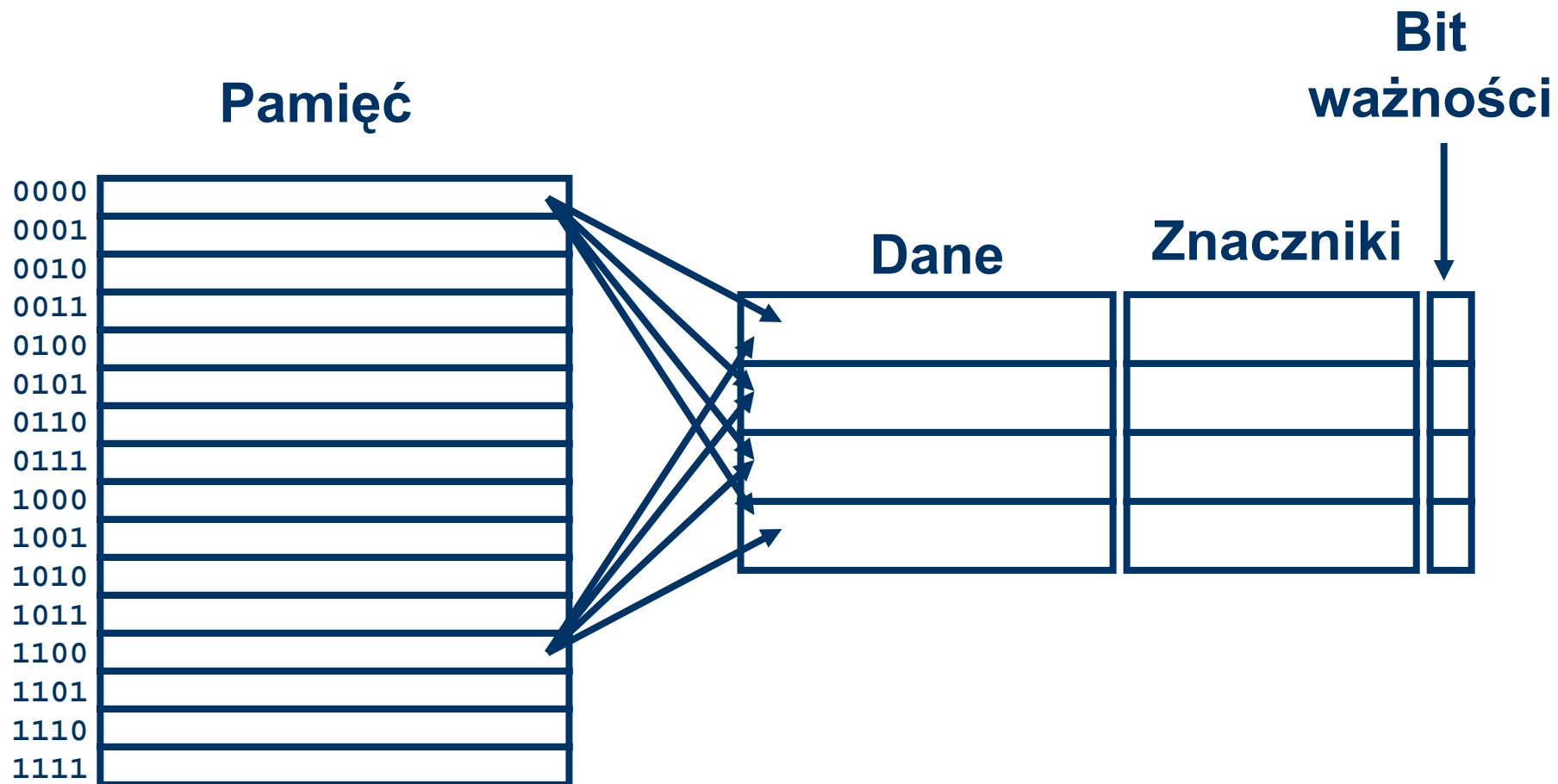
Jak zmniejszyć *miss rate*?

- Oczywiście zwiększenie pamięci cache spowoduje ograniczenie chybień, czyli zmniejszenie parametru *miss rate*
- Można też zmniejszyć *miss rate* ograniczając konkurencję przy obsadzie linijek pamięci cache
 - Trzeba pozwolić linijkom z pamięci głównej na rezydowanie w dowolnej linijce pamięci cache

Odwzorowanie skojarzeniowe

- *Fully associative mapping*
- Zamiast odwzorowania bezpośredniego zezwalamy na lokację linijek w dowolnym miejscu pamięci cache
- Teraz sprawdzenie czy dane są w pamięci cache jest trudniejsze i zajmie więcej czasu (parametr *hit time* wzrośnie)
- Potrzebne są dodatkowe rozwiązania sprzętowe (komparator dla każdej linijki pamięci cache)
- Każdy znacznik jest teraz kompletnym adresem linijki w pamięci głównej

Odwzorowanie skojarzeniowe



Rozwiązanie kompromisowe

- Odwzorowanie skojarzeniowe jest bardziej elastyczne, dlatego parametr *miss rate* ma mniejszą wartość
- Odwzorowanie bezpośrednie jest prostsze sprzętowo, czyli tańsze w implementacji
 - Ponadto lokalizacja danych w pamięci cache jest szybsza (parametr *hit time* ma mniejszą wartość)
- Szukamy kompromisu między *miss rate* a *hit time*.
- Rozwiązanie pośrednie: odwzorowanie sekcyjno-skojarzeniowe (*set associative*)

Set Associative Mapping

- Każda linijka z pamięci głównej może być ulokowana w pewnym dozwolonym podzbiorze linijk pamięci cache
- Pojęcie *n-way set associative mapping* (odwzorowanie n -drożne sekcyjno-skojarzeniowe) oznacza, że istnieje n miejsc (linijk) w pamięci cache, do których może trafić dana linijka z pamięci głównej placed.
- Pamięć cache jest zatem podzielona na pewną liczbę podzbiorów linijk, z których każdy zawiera n linijk

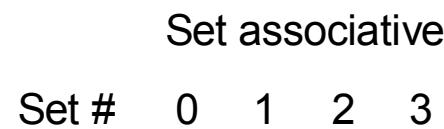
Porównanie odwzorowań

Przykład: cache składa się z 8 linijek, rozpatrujemy lokalizację linijki z pamięci głównej o adresie 12

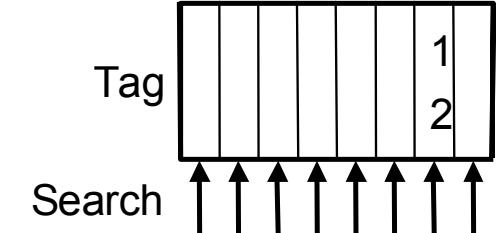
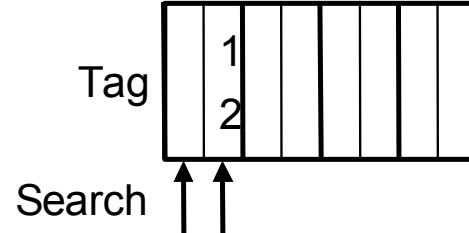
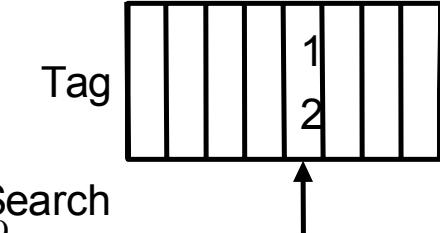
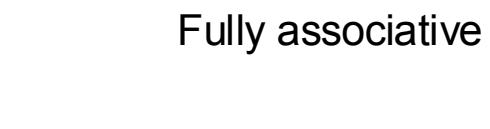
Linijka 12 może się znaleźć tylko w linijce 4



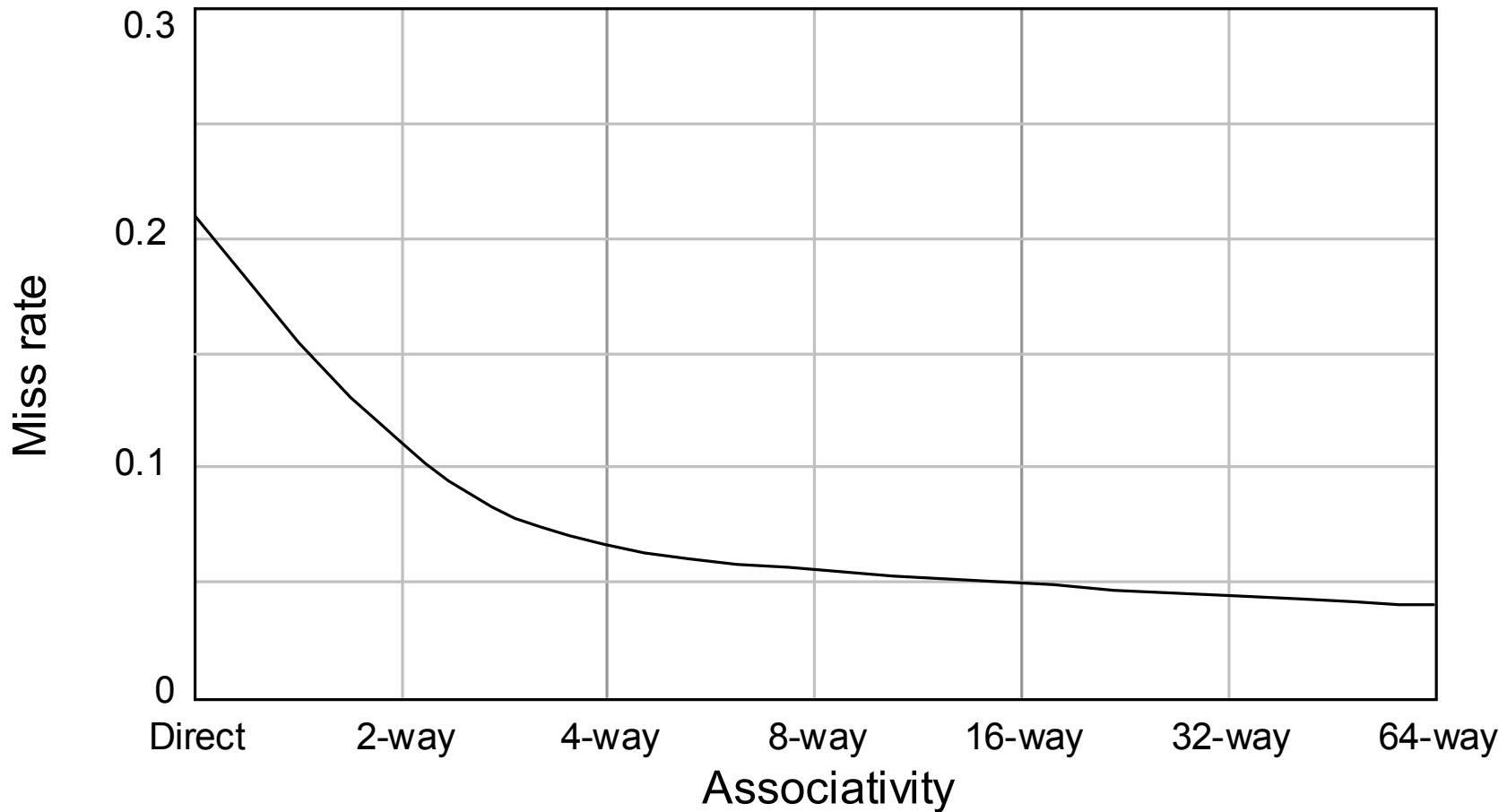
Linijka 12 może się znaleźć w jednej z dwóch linijk zbioru 0



Linijka 12 może się znaleźć w dowolnej linijce cache



Wydajność a n -drożność cache



Performance improvement of caches with increased associativity.

Odwzorowanie n -drożne

- Odwzorowanie bezpośrednie jest szczególnym przypadkiem n -drożnego odwzorowania sekcyjno-skojarzeniowego przy $n = 1$ (1 linijka w zbiorze)
- Odwzorowanie skojarzeniowe jest szczególnym przypadkiem n -drożnego odwzorowania sekcyjno-skojarzeniowego przy n równym liczbie linijk w pamięci cache
- Poprzedni przykład pokazuje, że odwzorowanie 4-drożne może nie dawać zauważalnej poprawy wydajności cache w porównaniu z odwzorowaniem 2-drożnym.

Algorytm wymiany linijek

- W metodzie odwzorowania bezpośredniego nie mamy wyboru przy zastępowaniu linijek – problem strategii wymiany linijek nie istnieje
- W metodzie odwzorowania sekcyjno-skojarzeniowego należy określić strategię wyboru linijek, które mają być usunięte przy wprowadzaniu do cache nowych linijek

Algorytm LRU

- **LRU:** *Least recently used.*
 - system zarządzania pamięcią cache rejestruje historię każdej linijki
 - jeśli trzeba wprowadzić do cache nową linijkę, a w dopuszczalnym zbiorze linijek nie ma wolnego miejsca, usuwana jest najstarsza linijka.
 - przy każdym dostępie do linijki (hit) wiek linijki jest ustawiany na 0
 - przy każdym dostępie do linijki wiek wszystkich pozostałych linijek w danym zbiorze linijek jest zwiększany o 1

Implementacja LRU

- Realizacja algorytmu LRU jest wykonywana sprzętowo
- Obsługa LRU w pamięciach 2-drożnych jest łatwa – wystarczy tylko 1 bit aby zapamiętać, która linijka (z dwóch) w zbiorze jest starsza
- Algorytm LRU w pamięciach 4-drożnych jest znacznie bardziej złożony, ale stosowany w praktyce
- W przypadku pamięci 8-drożnych sprzętowa realizacja LRU staje się kosztowna i skomplikowana. W praktyce używa się prostszego algorytmu aproksymującego działanie LRU

Wielopoziomowa pamięć cache

- Wielkość pamięci cache wbudowanej do procesora jest ograniczona ze względów technologicznych
- Ekonomicznym rozwiązaniem jest zastosowanie zewnętrznej pamięci cache L2
- Zwykle zewnętrzna pamięć cache jest szybką pamięcią SRAM (czas *miss penalty* jest znacznie mniejszy niż w przypadku pamięci głównej)
- Dodanie pamięci cache L2 wpływa na projekt pamięci cache L1 – dąży się do tego, by parametr *hit time* dla pamięci L1 był jak najmniejszy

Wspólna a rozdzielona cache

- We wspólnej pamięci cache dane i instrukcje są przechowywane razem (architektura von Neumanna)
- W rozdzielonej pamięci cache dane i instrukcje są przechowywane w osobnych pamięciach (architektura harwardzka)
- Dlaczego pamięć cache przechowująca instrukcje ma znacznie mniejszy współczynnik *miss rate*?

Size	Instruction Cache	Data Cache	Unified Cache
1 KB	3.06%	24.61%	13.34%
2 KB	2.26%	20.57%	9.78%
4 KB	1.78%	15.94%	7.24%
8 KB	1.10%	10.19%	4.57%
16 KB	0.64%	6.47%	2.87%
32 KB	0.39%	4.82%	2.48%
64 KB	0.15%	3.77%	1.35%
128 KB	0.02%	2.88%	0.95%

Wspólna a rozdzielona cache

- Które rozwiązanie zapewnia krótszy czas dostępu AMAT?
 - pamięć rozdzielona: 16 KB instrukcje + 16 KB dane
 - pamięć wspólna: 32 KB instrukcje + dane
- Założenia:
 - należy użyć danych statystycznych dotyczących miss rate z poprzedniego slajdu
 - zakładamy, że 75% dostępów do pamięci dotyczy odczytu instrukcji, a 25% dostępów dotyczy danych
 - $\text{miss penalty} = 50$ cykli
 - $\text{hit time} = 1$ cykl
 - operacje odczytu i zapisu zajmują dodatkowo 1 cykl, ponieważ jest tylko jeden port dla instrukcji i danych

Wspólna a rozdzielona cache

AMAT = %instr x (instr hit time + instr miss rate x instr miss penalty) +
%data x (data hit time + data miss rate x data miss penalty)

Dla rozdzielonej pamięci cache:

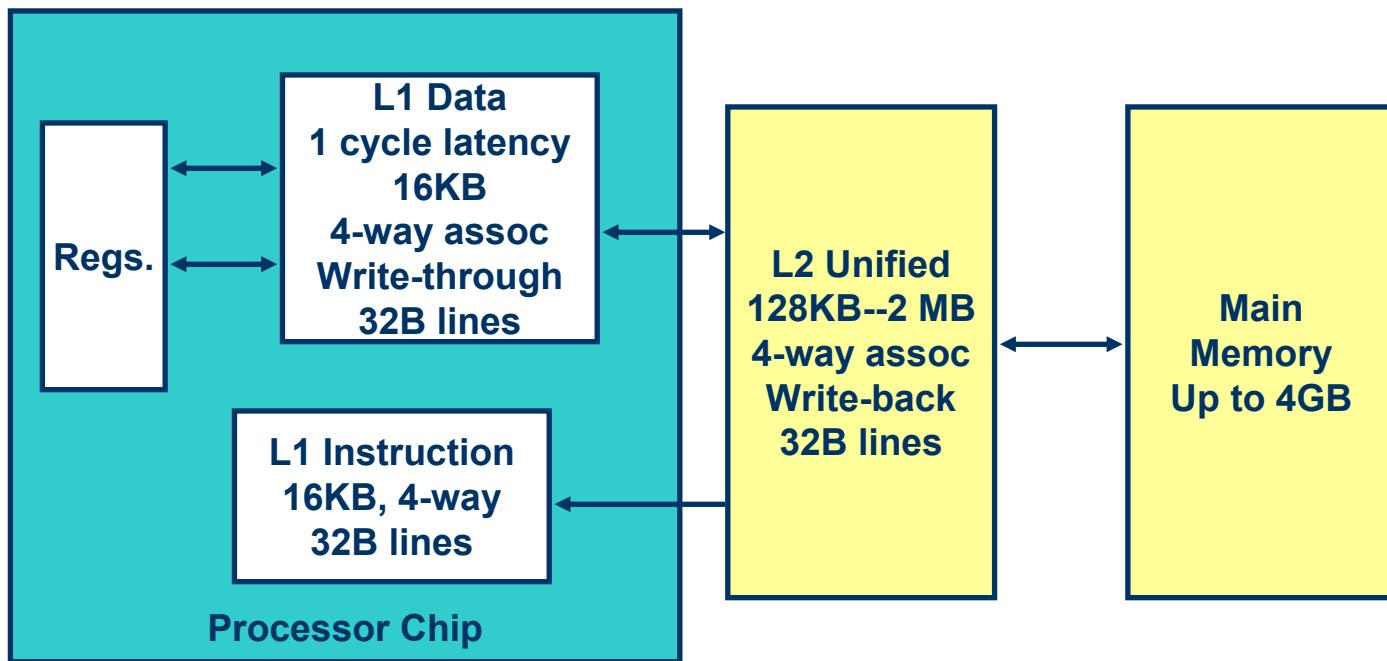
$$\text{AMAT} = 75\% \times (1 + 0.64\% \times 50) + 25\% \times (1 + 6.47\% \times 50) = \textcolor{red}{2.05}$$

Dla wspólnej pamięci cache:

$$\text{AMAT} = 75\% \times (1 + 2.48\% \times 50) + 25\% \times (1 + 2.48\% \times 50) = \textcolor{red}{2.24}$$

W rozważanym przykładzie lepszą wydajność (krótszy czas dostępu AMAT) zapewnia rozdzielona pamięć cache

Przykład – cache w Pentium



Podsumowanie

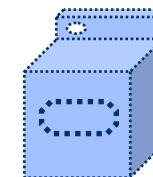
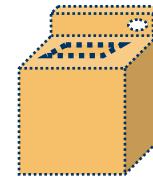
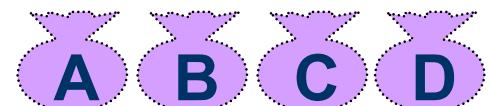
- Zasada lokalności
- Hierarchiczny system pamięci
- Koncepcja budowy i działania pamięci cache
- Metody odwzorowania pamięci głównej i pamięci cache
 - odwzorowanie bezpośrednie (mapowanie bezpośrednie)
 - odwzorowanie asocjacyjne (pełna asocjacja)
 - częściowa asocjacja
- Zapis danych i problem spójności zawartości pamięci
- Wydajność pamięci cache, metody optymalizacji, ograniczenia
- Wymiana linijek w pamięci cache - algorytm LRU
- Przykłady organizacji i architektury pamięci cache
- Wspólna a rozdzielona pamięć cache

Organizacja i Architektura Komputerów

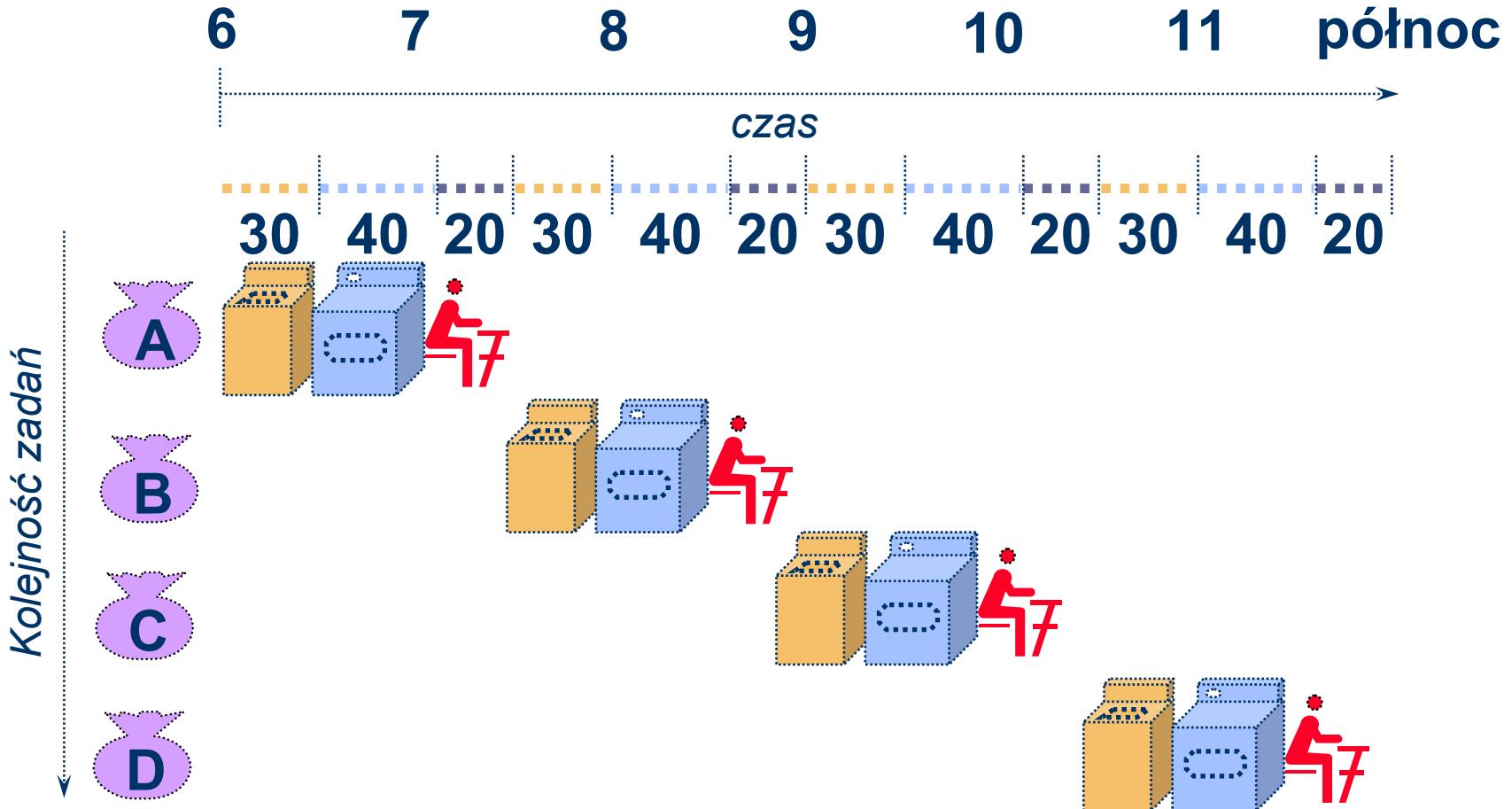
Przetwarzanie potokowe.
Architektury superskalarnie

Pipelining – wprowadzenie

- Pipelining: technika wykonywania ciągu instrukcji, w której kilka kolejnych instrukcji jest wykonywanych równolegle
- Przykład: pralnia
- Ania, Basia, Celina i Dorota mają po kompletnym ładunku odzieży, do prania, wysuszenia i wyprasowania
- Pranie trwa 30 minut
- Suszenie zajmuje 40 minut
- Prasowanie trwa 20 minut

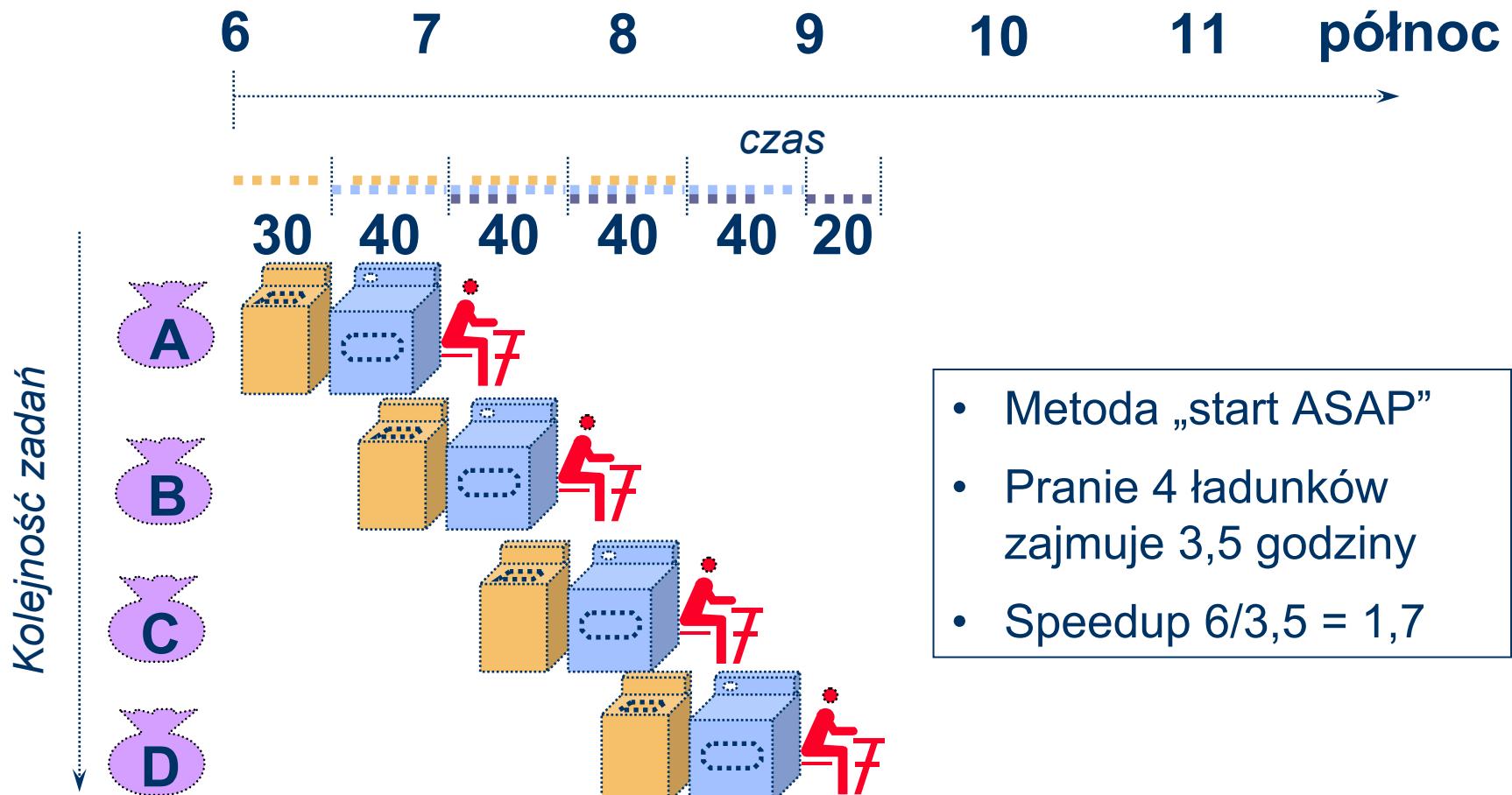


Wersja 1 – pralnia sekwencyjna



- Sekwencyjne pranie 4 ładunków zajmuje 6 godzin

Wersja 2 – pralnia potokowa



Pipelining – wnioski

- Pipelining nie zmienia czasu wykonania pojedynczej instrukcji (*latency*), wpływa natomiast na czas wykonania ciągu instrukcji (*throughput*)
- Częstość z jaką pracuje potok jest ograniczona przez czas pracy najwolniejszego stopnia w potoku (*slowest pipeline stage*)
- Wiele instrukcji jest wykonywanych równolegle (*instruction level parallelism*)
- Potencjalne zwiększenie wydajności jest tym większe, im większa jest liczba stopni w potoku
- Duże zróżnicowanie czasu trwania operacji w poszczególnych stopniach ogranicza wzrost wydajności
- Czas potrzebny na napełnienie potoku i opróżnienie go ogranicza wzrost wydajności

Wnioski cd.

- Architektura RISC zapewnia lepsze wykorzystanie możliwości przetwarzania potokowego niż architektura CISC. W architekturze RISC:
 - instrukcje mają taką samą długość
 - większość operacji dotyczy rejestrów
 - odwołania do pamięci, które mogą ewentualnie powodować kolizje są rzadsze i występują tylko w przypadku specjalnie do tego celu używanych instrukcji **load** i **store**
- W architekturze CISC instrukcje mają różną długość, występuje bogactwo sposobów adresowania danych w pamięci

Cykl prefetch

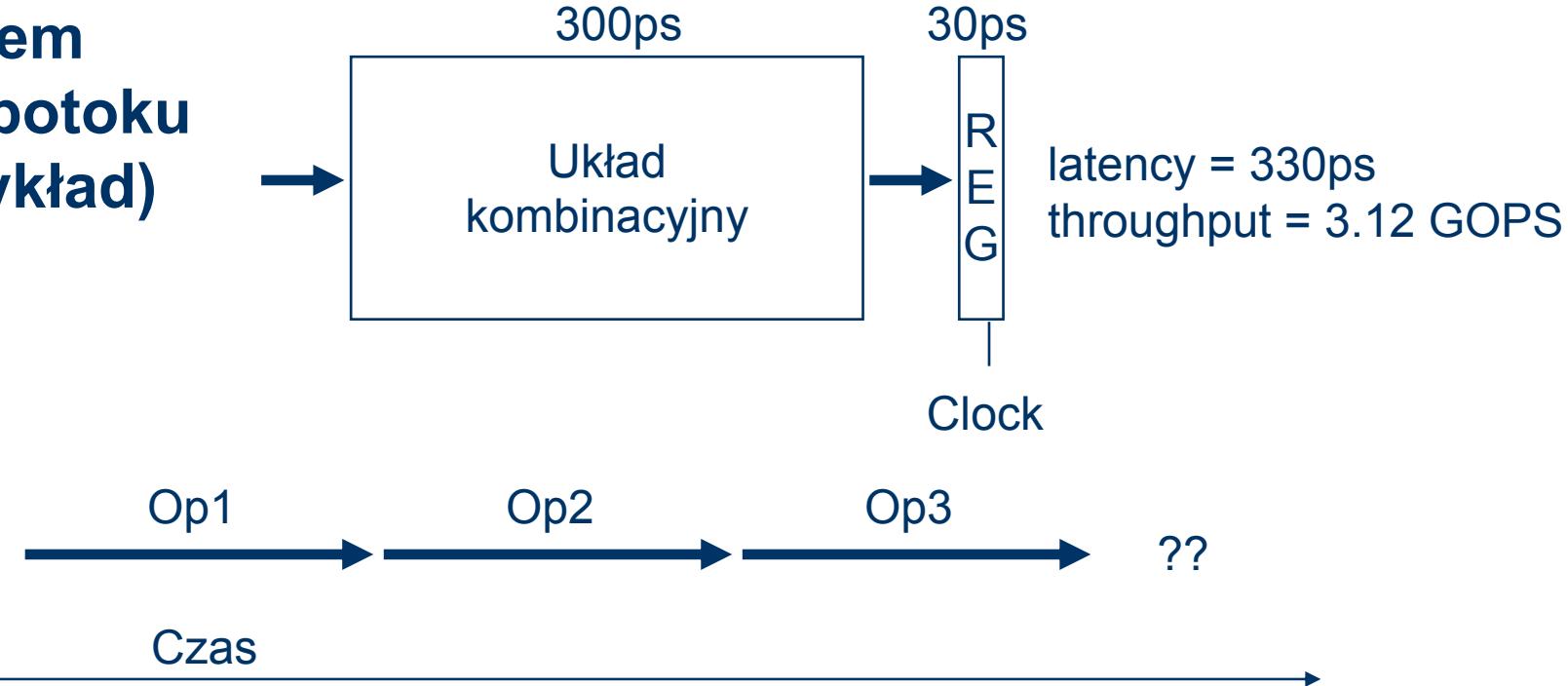
- Najprostsza wersja przetwarzania potokowego
- Pobranie kolejnej instrukcji (cykl *fetch*) wymaga dostępu do pamięci
- Wykonanie instrukcji zazwyczaj nie wymaga dostępu do pamięci
- Wniosek: można pobrać następną instrukcję z pamięci podczas wykonywania poprzedniej instrukcji
- Metoda nazywana „pobraniem wstępny” - *instruction prefetch*

Prefetch cd.

- Wprowadzenie cyklu *prefetch* poprawia wydajność (ale nie podwaja):
 - pobranie kodu operacji trwa zwykle krócej niż faza wykonania
 - może pobierać więcej niż jeden rozkaz w fazie *prefetch*?
 - każdy rozkaz skoku powoduje utratę korzyści z fazy *prefetch*
- Wnioski są niejednoznaczne:
 - z jednej strony korzystnie byłoby zwiększyć liczbę stopni w potoku
 - z drugiej jednak długi potok w przypadku skoku musi być opróżniony i napełniony od nowa

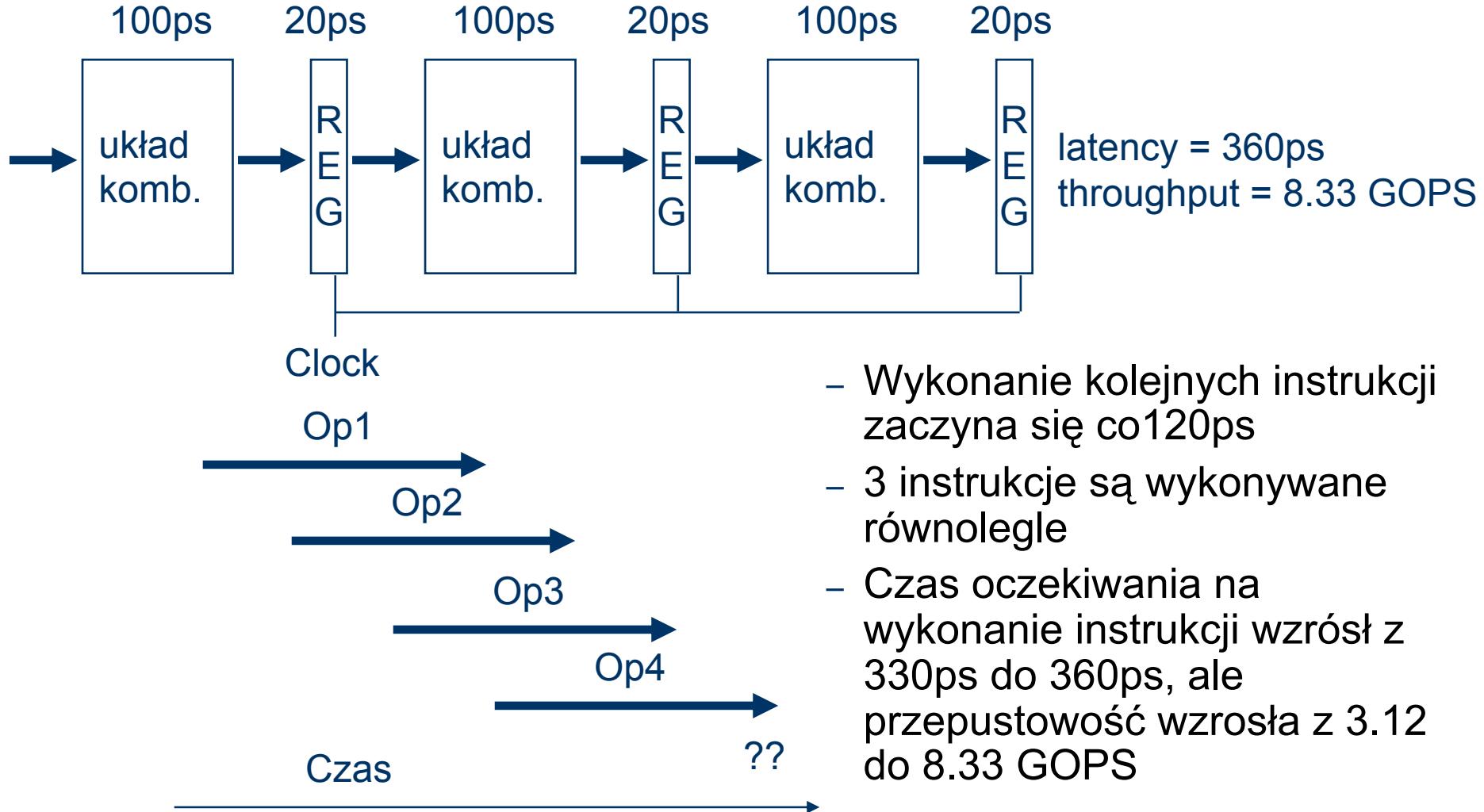
Punkt wyjścia – brak potoku

**System
bez potoku
(przykład)**



- Każda operacja musi być zakończona zanim zacznie się wykonywanie następnej operacji
- W podanym przykładzie czas wykonania operacji wynosi 330ps

Potok 3-stopniowy



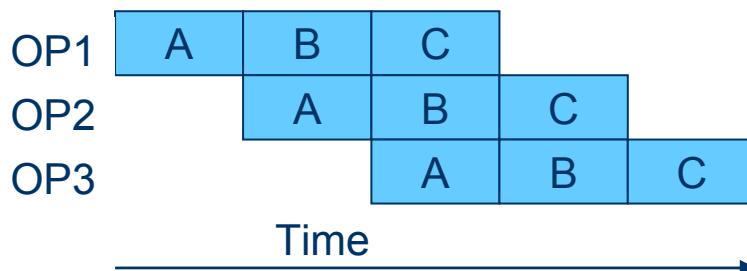
Diagramy czasowe

- Bez potoku



- Kolejna operacja nie może się zacząć przed zakończeniem poprzedniej

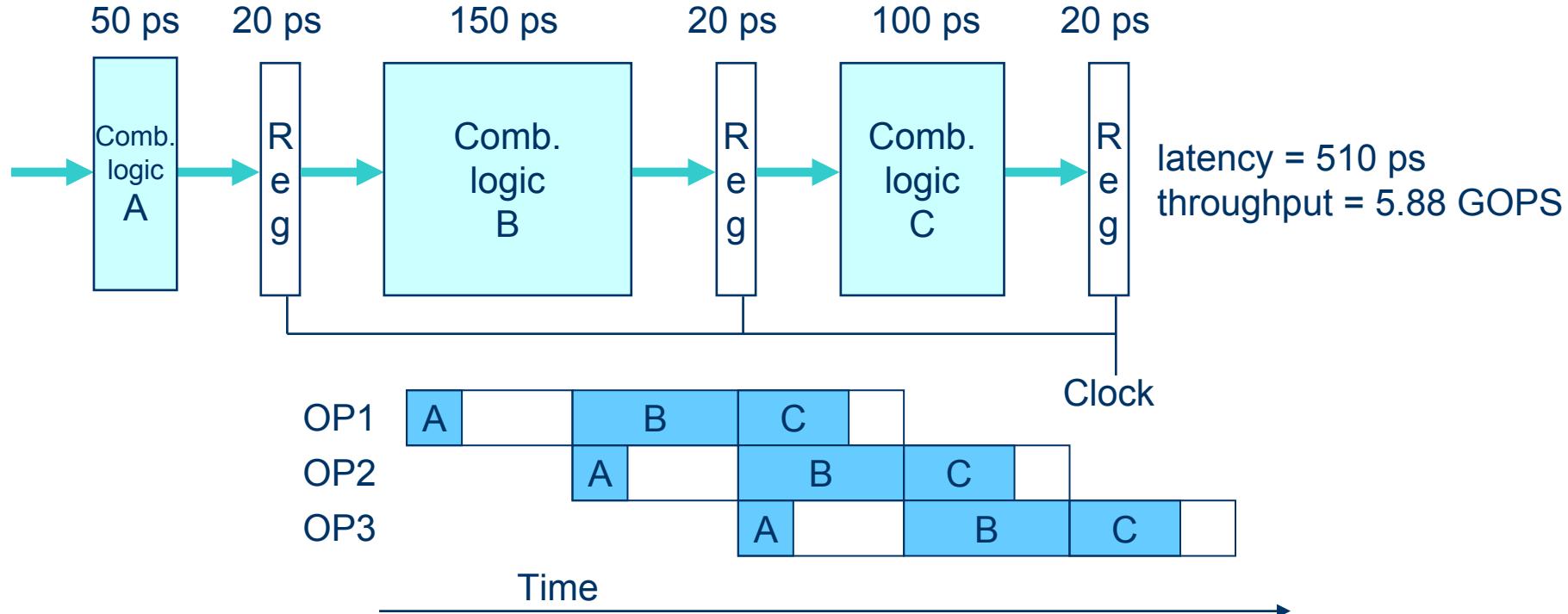
- 3-stopniowy potok



A, B, C – fazy wykonania instrukcji

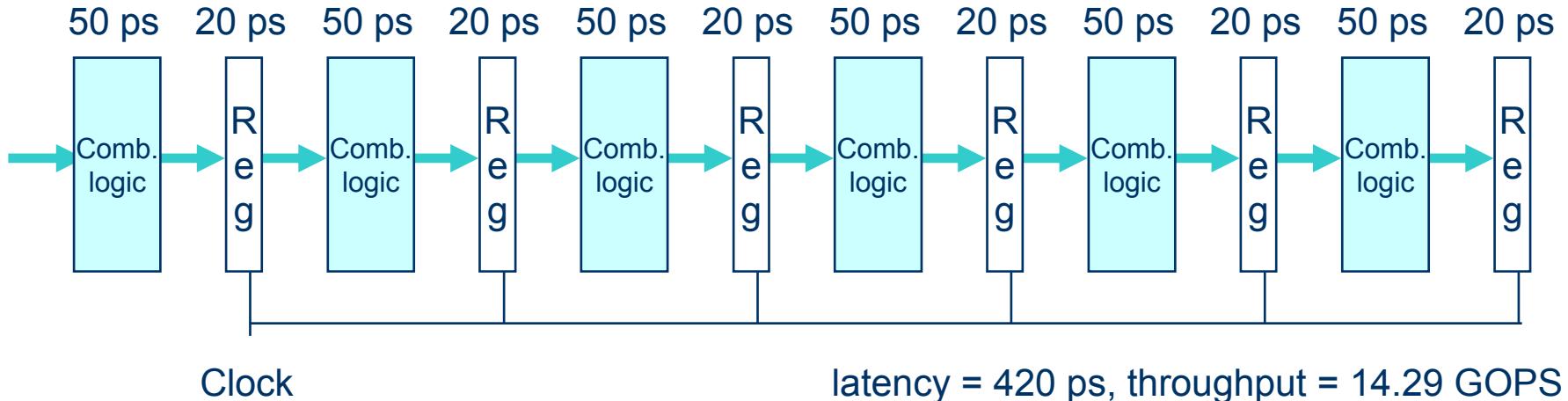
- Do 3 instrukcji wykonuje się równolegle

Ograniczenie – różnice opóźnień



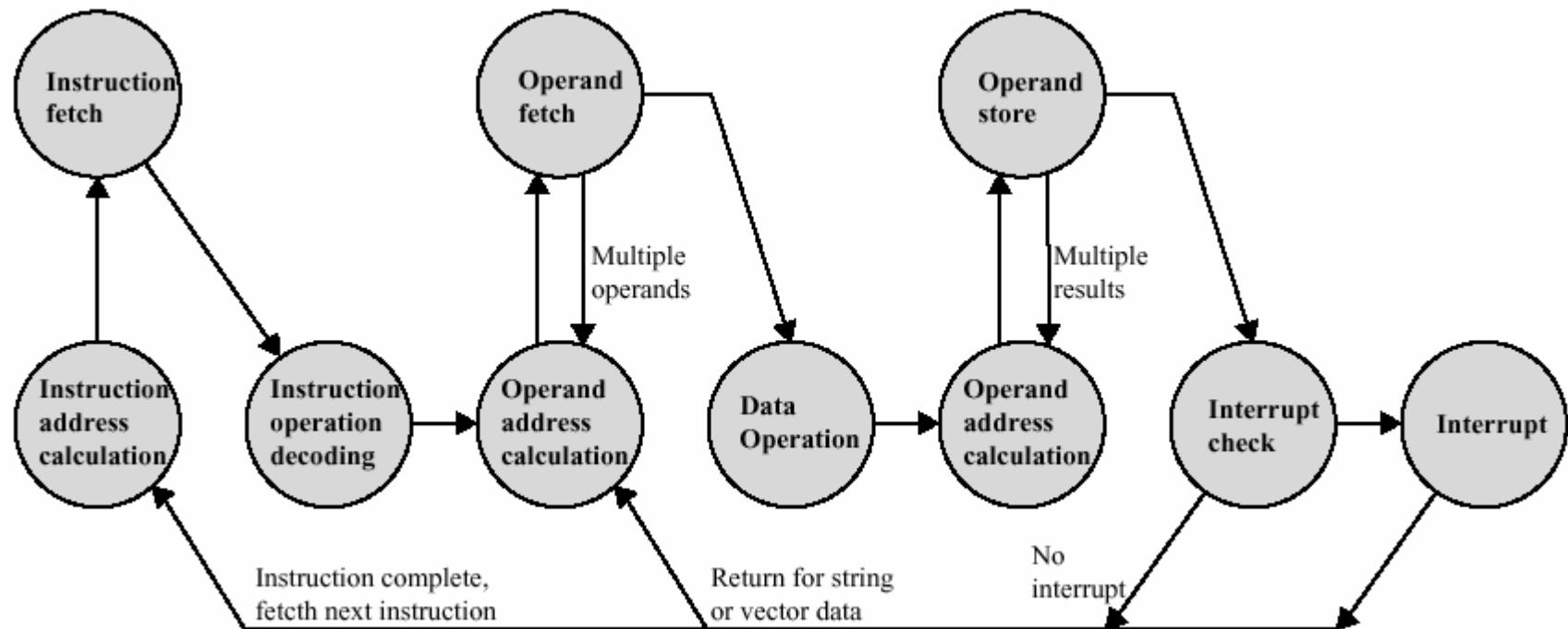
- Przepustowość ograniczona przez najwolniejszy stopień
- Pozostałe (szybsze) stopnie są przez znaczny czas bezczynne
- Należy dążyć do budowy potoku o zbliżonych czasach operacji w poszczególnych stopniach

Głęboki potok – problemy



- W miarę wydłużania (zwiększania głębokości) potoku opóźnienia rejestrów stają się coraz bardziej znaczące
- Procentowy udział czasu opóźnienia rejestrów w potoku:
 - 1-stopień: 6.25%
 - 3-stopnie: 16.67%
 - 6-stopni: 28.57%
- Problem ma duże znaczenie, ponieważ współczesne procesory o dużej wydajności mają coraz głębsze potoki

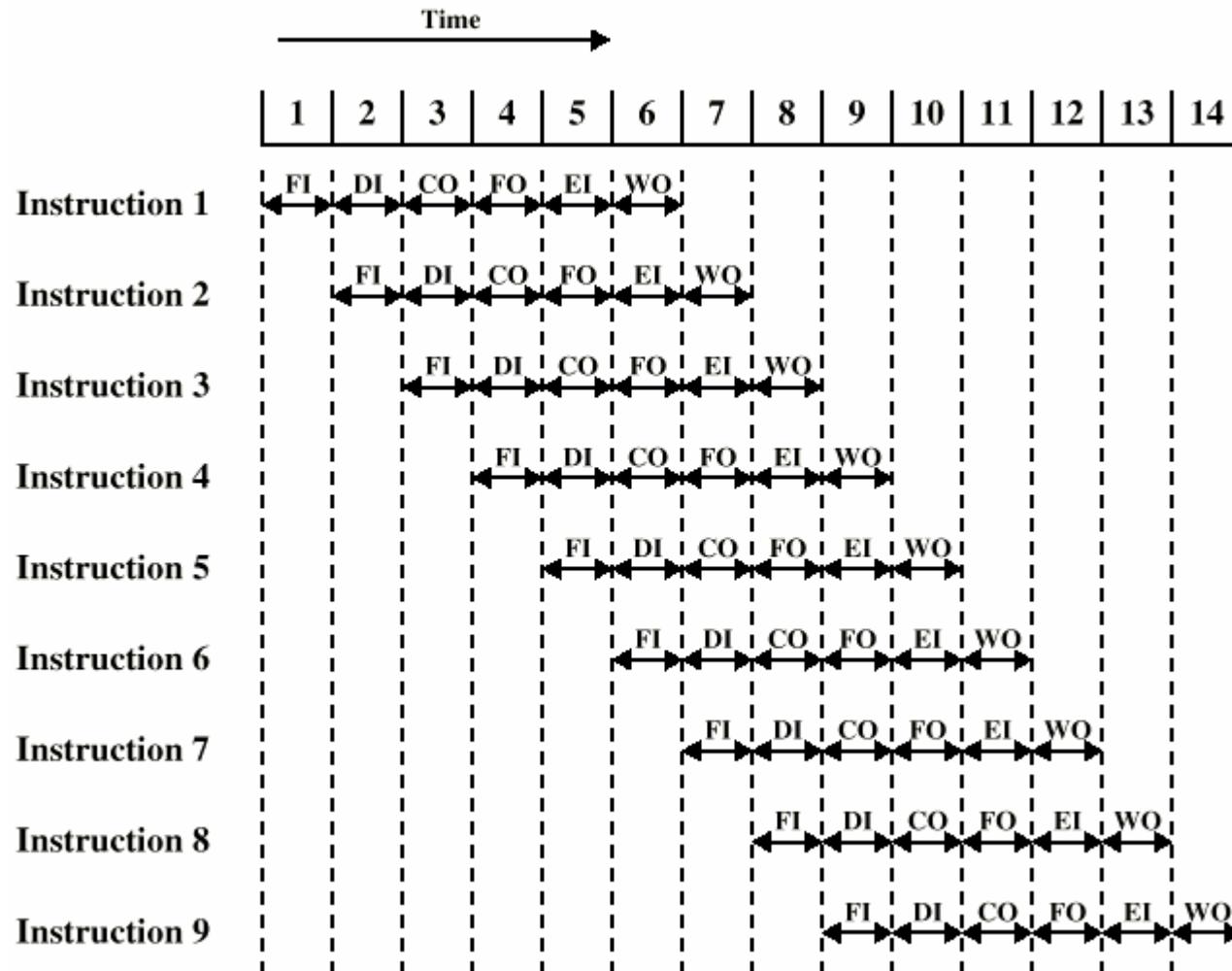
Cykl wykonania instrukcji



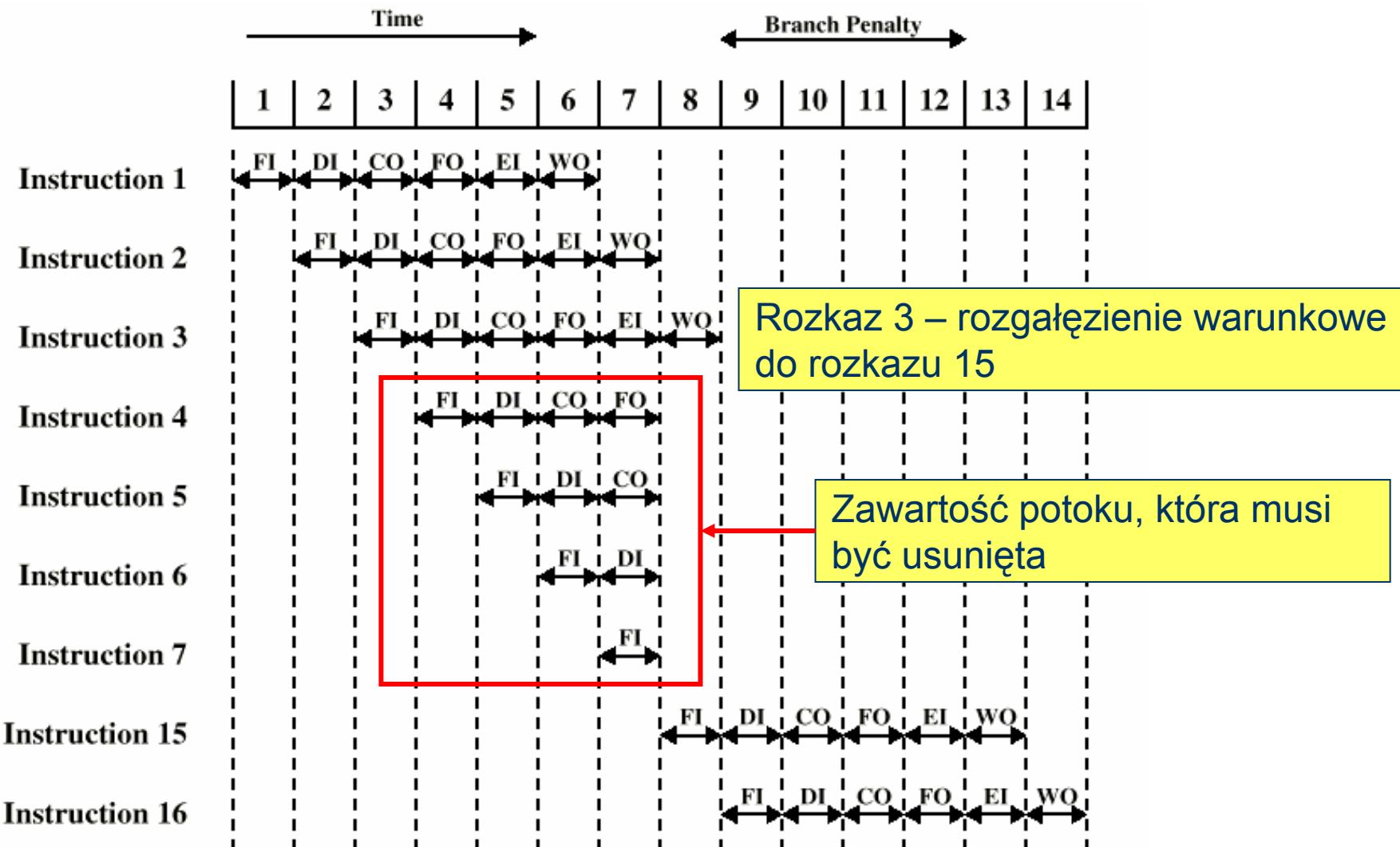
Typowe 6 stopni potoku

- Pobranie instrukcji (*fetch instruction - FI*)
- Dekodowanie instrukcji (*decode instruction - DI*)
- Obliczenie adresu argumentu (*calculate operand – CO*)
- Pobranie argumentu (*fetch operand – FO*)
- Wykonanie instrukcji (*execute instruction - EI*)
- Zapis wyniku (*write - WO*)

Potok 6-stopniowy



Instrukcja skoku



Skoki a potok

- Warianty przetwarzania potokowego w przypadku instrukcji skoków (*branch*):
 - **Multiple Streams** – dwa potoki; w przypadku rozgałęzienia każdy z potoków jest ładowany kodem odpowiadającym wykonaniu bądź niewykonaniu skoku; metoda zawodna w przypadku skoków wielokrotnych
 - **Prefetch Branch Target** – kod z obu dróg rozgałęzienia jest ładowany do potoku i przechowywany aż do chwili wykonania skoku; metoda zastosowana w IBM 360/91
 - **Loop buffer** – instrukcje są pobierane do małej szybkiej pamięci działającej podobnie jak cache, ale sterowanej przez układ sterowania potokiem; metoda efektywna w przypadku pętli obejmujących niewiele instrukcji; zastosowana w Cray-1

Skoki a potok cd.

- Warianty przetwarzania potokowego w przypadku instrukcji skoków (*branch*):
 - **Delayed Branching** – opóźnianie skoku; metoda szeregowania instrukcji poprzedzających i następujących po skoku wykonywana w trakcie kompilacji; kompilator stara się przenieść instrukcje poprzedzające skok i umieścić je za skokiem; dzięki temu w przypadku skoku potok nie musi być opróżniany i instrukcje mogą być wykonywane nadal; w takim przypadku wykonanie skoku jest wstrzymywane aż do zakończenia wykonania przestawionych instrukcji
 - **Branch Prediction** – przewidywanie skoków; najbardziej popularna i najbardziej efektywna metoda stosowana we współczesnych procesorach

Opóźnianie skoków - przykład

Adres	Normal	Delayed
100	LOAD A,X	LOAD A,X
101	ADD A,1	ADD A,1
102	JUMP 105	JUMP 106
103	ADD B,A	NOP
104	SUB B,C	ADD B,A
105	STORE Z,A	SUB B,C
106		STORE Z,A

one delay slot

- W tym przykładzie kompilator wstawia po rozkazie skoku JUMP jedną instrukcję pustą (*one delay slot*). Jest nią instrukcja NOP (*no operation* – nic nie rób)
- Wykorzystano język asemblera hipotetycznego procesora, dla uproszczenia założono, że każda instrukcja zajmuje 1 słowo

Opóźnianie skoków cd.

Można ulepszyć program z poprzedniego slajdu dokonując przestawienia instrukcji. Należy zauważyć, że ADD A,1 zawsze dodaje 1 do akumulatora A, zatem instrukcja ta może być przesunięta przez kompilator i użyta w miejsce NOP

Adres	Normal	Delayed	Optimized
100	LOAD A,X	LOAD A,X	LOAD A,X
101	ADD A,1	ADD A,1	JUMP 105
102	JUMP 105	JUMP 106	ADD A,1
103	ADD B,A	NOP	ADD B,A
104	SUB B,C	ADD B,A	SUB B,C
105	STORE Z,A	SUB B,C	STORE Z,A
106		STORE Z,A	

Efektywność opóźniania skoków

- W porównaniu z normalnym szeregowaniem rozkazów nic nie tracimy, a możemy zyskać
- Dobry kompilator zapewnia następującą efektywność przy opóźnianiu skoków:
 - około 60% slotów zostaje wypełnionych
 - około 80% instrukcji umieszczonych w slotach jest przydatnych w dalszym wykonywaniu programu
 - łącznie 48% ($60\% \times 80\%$) wszystkich slotów jest wykorzystanych z korzyścią dla wydajności systemu
- Zalety i wady metody opóźniania skoków
 - Zaleta: optymalizacja kompilatora ma wpływ na pracę pipeliningu
 - Wada: w nowoczesnych procesorach z głębokim potokiem (deep pipeline) metoda opóźniania skoków jest mało skuteczna

Przewidywanie skoków

- Metody statyczne
 - Zakładamy, że skok nie będzie nigdy wykonany
 - Metoda – *branch never will be taken*
 - Do potoku pobiera się zawsze instrukcję następującą po skoku
 - Metoda stosowana w procesorach Motorola 68020 i VAX 11/780
 - Zakładamy, że skok będzie zawsze wykonany
 - Metoda – *branch always will be taken*
 - Do potoku pobiera się instrukcję wskazywaną przez adres skoku (*branch target*)
 - O tym czy skok będzie, czy też nie będzie wykonany wnioskujemy na podstawie rodzaju skoku (kodu operacji)
 - Przesłanką metody jest spostrzeżenie, że pewne rodzaje skoków są wykonywane z dużym, a inne z małym prawdopodobieństwem
 - Badania pokazują, że można uzyskać nawet 75% sukcesów

Przewidywanie skoków

- Metody dynamiczne

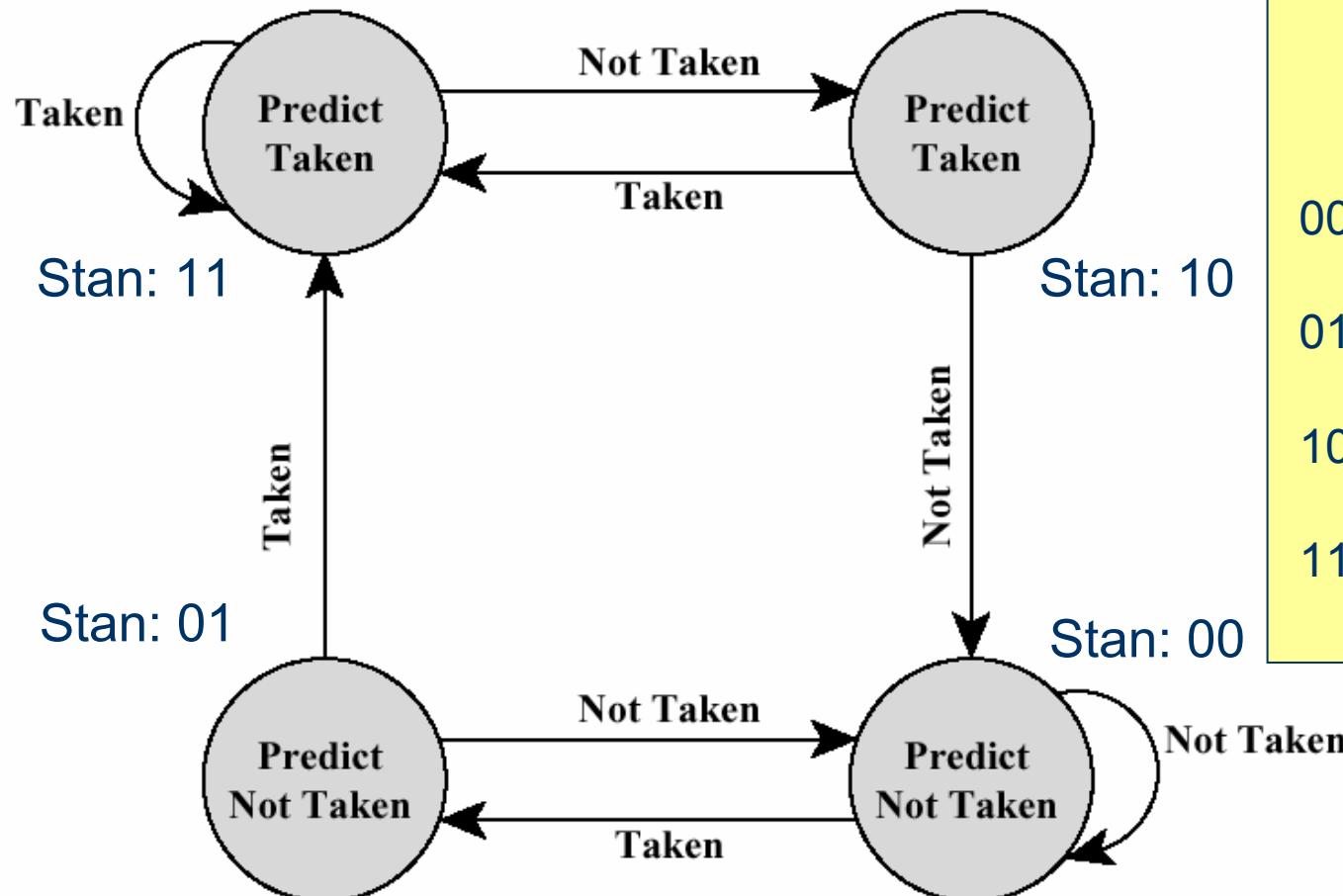
- Prognoza skoku jest ustalana dynamicznie, w trakcie wykonania programu
- Przewidywanie jest wykonywane na podstawie historii każdego skoku zapisanej w tablicy historii skoków BTB (*branch target buffer*)
- BTB składa się zwykle z 128 – 1024 rekordów o postaci:

Adres instrukcji

Adres skoku (target)

Stan

Predykcja skoków – diagram BTB



Przykładowe kodowanie stanów:

- 00 – mocna hipoteza o braku skoku
- 01 – słaba hipoteza o braku skoku
- 10 – słaba hipoteza o skoku
- 11 – mocna hipoteza o skoku

Rozmiar BTB

Im większy jest rozmiar BTB (czyli im więcej rekordów mieści tablica), tym bardziej trafne jest przewidywanie skoków

Rozmiar BTB	Średnia liczba trafnych prognoz [%]
16	40
32	50
64	65
128	72
256	78
512	80
1024	85
2048	87

Głębokość potoku

- Wyniki badań pokazują, że istnieje teoretyczna optymalna wartość głębokości potoku równa około 8 stopni
 - z jednej strony im głębszy potok, tym lepiej działa równoległe przetwarzanie instrukcji
 - z drugiej strony głęboki potok w przypadku źle przewidzianego skoku powoduje dużą stratę czasu (*branch penalty*)
- W nowych procesorach stosuje się ulepszone warianty przetwarzania potokowego (*hyperpipeline*) dostosowane do pracy z bardzo szybkim zegarem
 - w takich procesorach optymalna głębokość potoku jest większa i dochodzi nawet do 20 (przykład – Pentium 4)

Głębokość potoku cd.

Liczba stopni przetwarzania danych typu *integer* w potokach popularnych procesorów

CPU	liczba stopni w potoku
P	5
MMX	6
P-Pro	12
P-II	12
P-III	10
P-4	20
M1-2	7
K5	7
K6	6
K7	11

P – Pentium
M – Cyrix
K – AMD

Zjawisko hazardu

- Równoległe przetwarzanie instrukcji w potoku prowadzi często do niekorzystnych zjawisk nazywanych hazardem. Hazard polega na braku możliwości wykonania instrukcji w przewidzianym dla niej cyklu. Wyróżnia się trzy rodzaje hazardu:
 1. **Hazard zasobów** (*structural hazard*) – kiedy dwie instrukcje odwołują się do tych samych zasobów
 2. **Hazard danych** (*data hazard*) – kiedy wykonanie instrukcji zależy od wyniku wcześniejszej, nie zakończonej jeszcze instrukcji znajdującej się w potoku
 3. **Hazard sterowania** (*control hazard*) – przy przetwarzaniu instrukcji skoków i innych instrukcji zmieniających stan licznika rozkazów (np. wywołania podprogramów)
- Hazard sterowania został omówiony już wcześniej – teraz zajmiemy się hazardem zasobów i danych

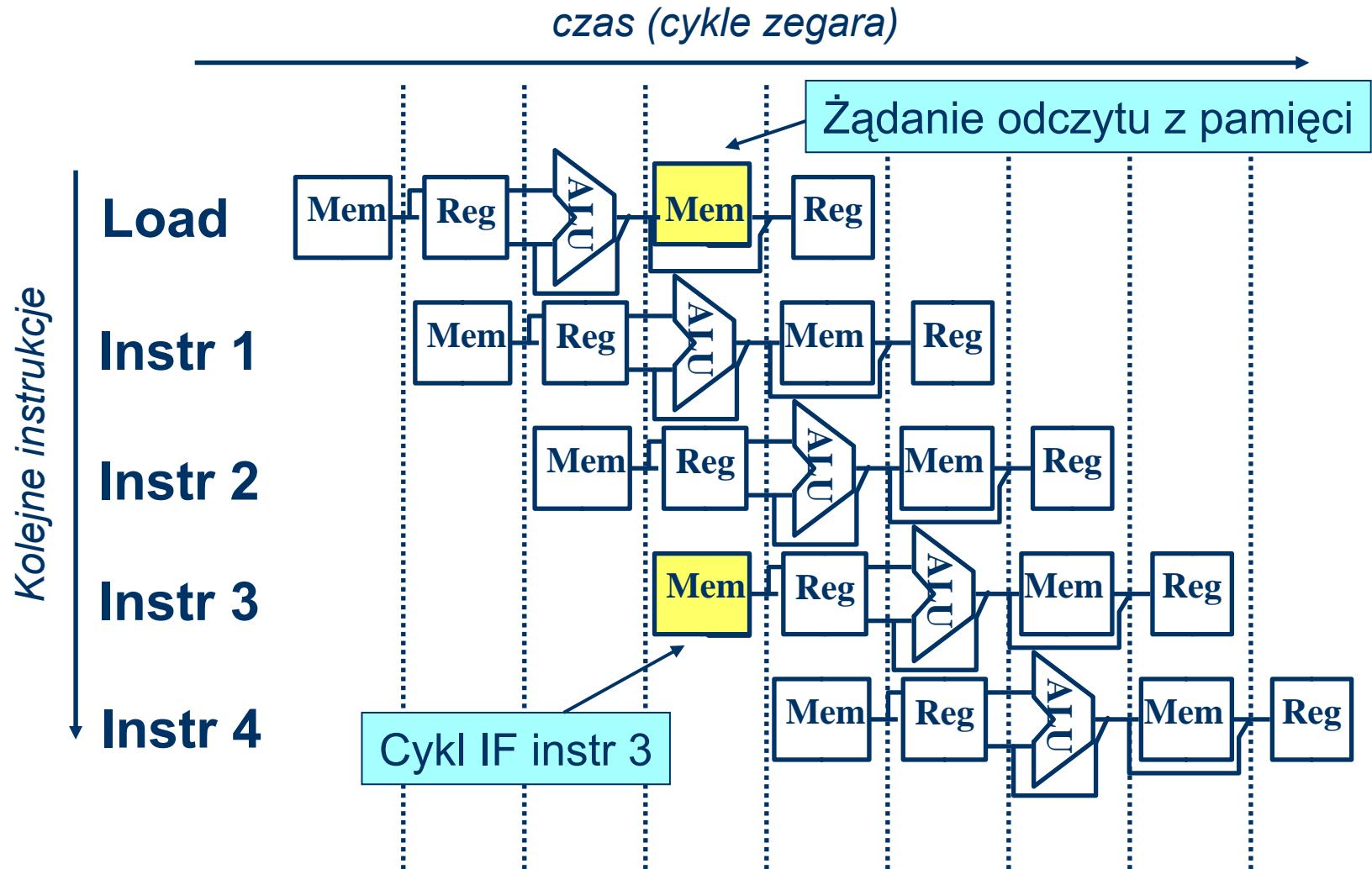
Zjawisko hazardu cd.

- Podstawowym sposobem rozwiązywania problemów wynikających z hazardu jest chwilowe zatrzymanie potoku na jeden lub więcej cykli zegara. W tym celu do potoku wprowadza się tzw. przegrody (*stalls*), które są faktycznie operacjami pustymi („*bubbles*”)
- Przegrody wpływają oczywiście na zmniejszenie wydajności CPU
- Ze względu na występowanie zjawisk hazardu rzeczywista wydajność CPU jest zawsze mniejsza od wydajności teoretycznej, obliczonej przy założeniu że hazardy nie występują

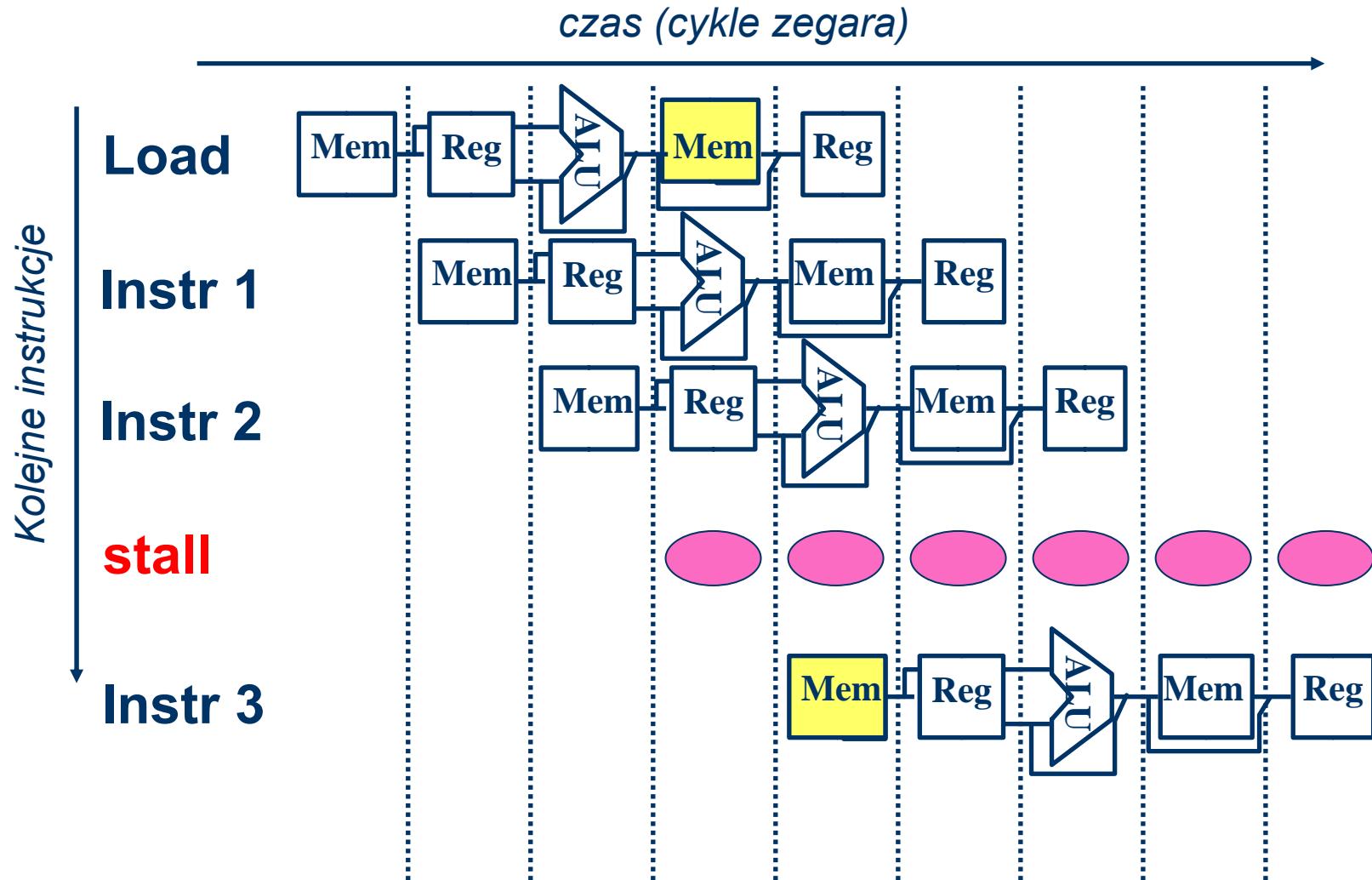
Hazard zasobów

- Nazywany też hazardem strukturalnym lub konfliktom zasobów (*resource conflict*)
- Powodem hazardu tego typu jest jednoczesne żądanie dostępu do tego samego zasobu (zwykle pamięci) przez dwa różne stopnie potoku
- Typowy hazard zasobów występuje, gdy jedna z instrukcji wykonuje ładowanie danych z pamięci podczas gdy inna ma być w tym samym cyklu ładowana (*instruction fetch*)
- Rozwiążanie polega na wprowadzeniu jednej (lub kilku) przegród). W efekcie wzrasta CPI i spada wydajność procesora

Hazard zasobów - przykład



Hazard zasobów – przykład cd.



Hazard danych

- Hazard danych występuje, ponieważ potok zmienia kolejność operacji odczytu/zapisu argumentów w stosunku do kolejności, w jakiej te operacje występują w sekwencyjnym zapisie programu
- Przykład:
`add ax,bx`
`mov cx,ax`

Instrukcja `mov cx,ax` ma pobrać zawartość akumulatora `ax`, podczas gdy potok jeszcze nie określił tej zawartości w poprzedniej instrukcji dodawania

Hazard typu RAW

- Wyróżnia się trzy typy (rodzaje) hazardu danych: RAW, WAR i WAW:
- Hazard RAW (*read after write*) występuje gdy pojawi się żądanie odczytu danych przed zakończeniem ich zapisu
- Przykład (taki sam jak na poprzednim slajdzie)

```
add  ax,bx  
mov  cx,ax
```

Hazard typu WAR

- Hazard WAR (*write after read*) występuje gdy pojawi się żądanie zapisu danych przed zakończeniem ich odczytu
- Przykład:

```
mov  bx, ax  
add  ax, cx
```

- CPU chce zapisać do akumulatora ax nową wartość równą sumie $ax+cx$, podczas gdy instrukcja przesłania `mov` jeszcze nie odczytała starej zawartości `ax` i nie przesyłała jej do rejestrów `bx`

Hazard typu WAW

- Hazard WAW (*write after write*) występuje gdy pojawi się żądanie zapisu danych przed zakończeniem wcześniejszej operacji zapisu
- Przykład:

```
mov ax, [mem]  
add ax,bx
```

- CPU chce zapisać do akumulatora ax nową wartość równą sumie ax+bx, podczas gdy poprzednia instrukcja nie zdążyła jeszcze pobrać do ax zawartości komórki pamięci [mem]

Problem

- Czy może powstać hazard danych RAR (*read after read*) ?
- Przykład:

```
add  bx,ax  
mov cx,ax
```

- Odpowiedź: taka sytuacja nie powoduje hazardu danych, ponieważ zawartość rejestru ax nie zmienia się. Może natomiast wystąpić konflikt równoczesnego dostępu (hazard zasobów). Problem można rozwiązać stosując rejestr typu *dual-port*

Zapobieganie hazardom danych

- Najprostszą metodą jest, jak poprzednio, zatrzymywanie potoku (*stalls*)
 - metoda ta obniża wydajność CPU i jest stosowana w ostateczności
- Szeregowanie statyczne (*static scheduling*)
 - wykonywane programowo podczas kompilacji
 - polega na zmianie kolejności instrukcji, tak aby zlikwidować hazardy
 - likwidacja hazardów nie zawsze jest możliwa
 - dobre kompilatory potrafią usunąć znaczną część potencjalnych hazardów danych

Szeregowanie statyczne

- Przykład:

Kod oryginalny

add ax,15

mov cx,ax

mov [mem],bx

Kod po uszeregowaniu

add ax,15

mov [mem],bx

mov cx,ax

- Zmiana kolejności instrukcji nie wpływa na działanie programu
- Hazard RAW został zlikwidowany, ponieważ instrukcja dodawania zdążyć być wykonana przed pobraniem argumentu przez instrukcję przesłania ax do cx

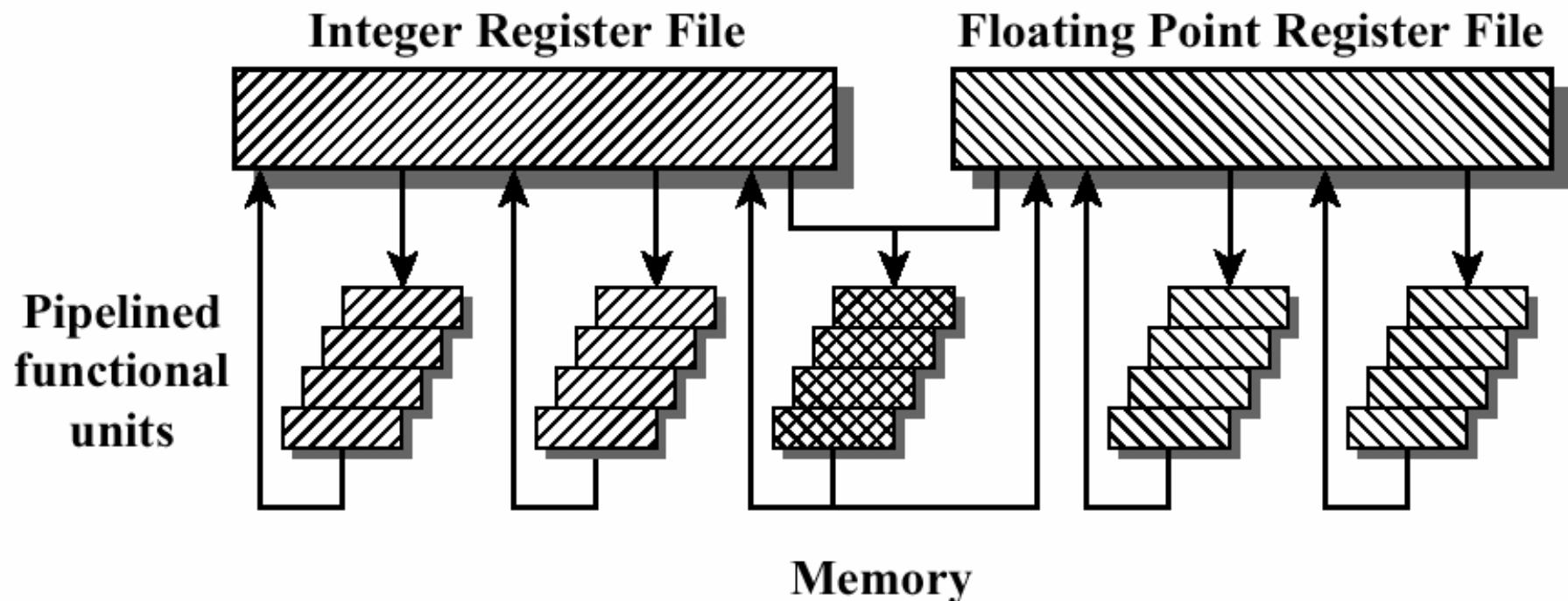
Szeregowanie dynamiczne

- *Dynamic scheduling* – wspólna nazwa technik polegających na usuwaniu problemów z hazardem danych w trakcie wykonywania programu, a nie w fazie kompilacji
 - realizacja sprzętowa w CPU
 - najważniejsze techniki:
 - wyprzedzanie argumentów (*operand forwarding*)
 - wyprzedzanie wyników operacji (*result forwarding*)
 - w procesorach superskalarnych stosuje się ponadto
 - notowanie (*scorebording*)
 - przemianowywanie rejestrów (*register renaming*)

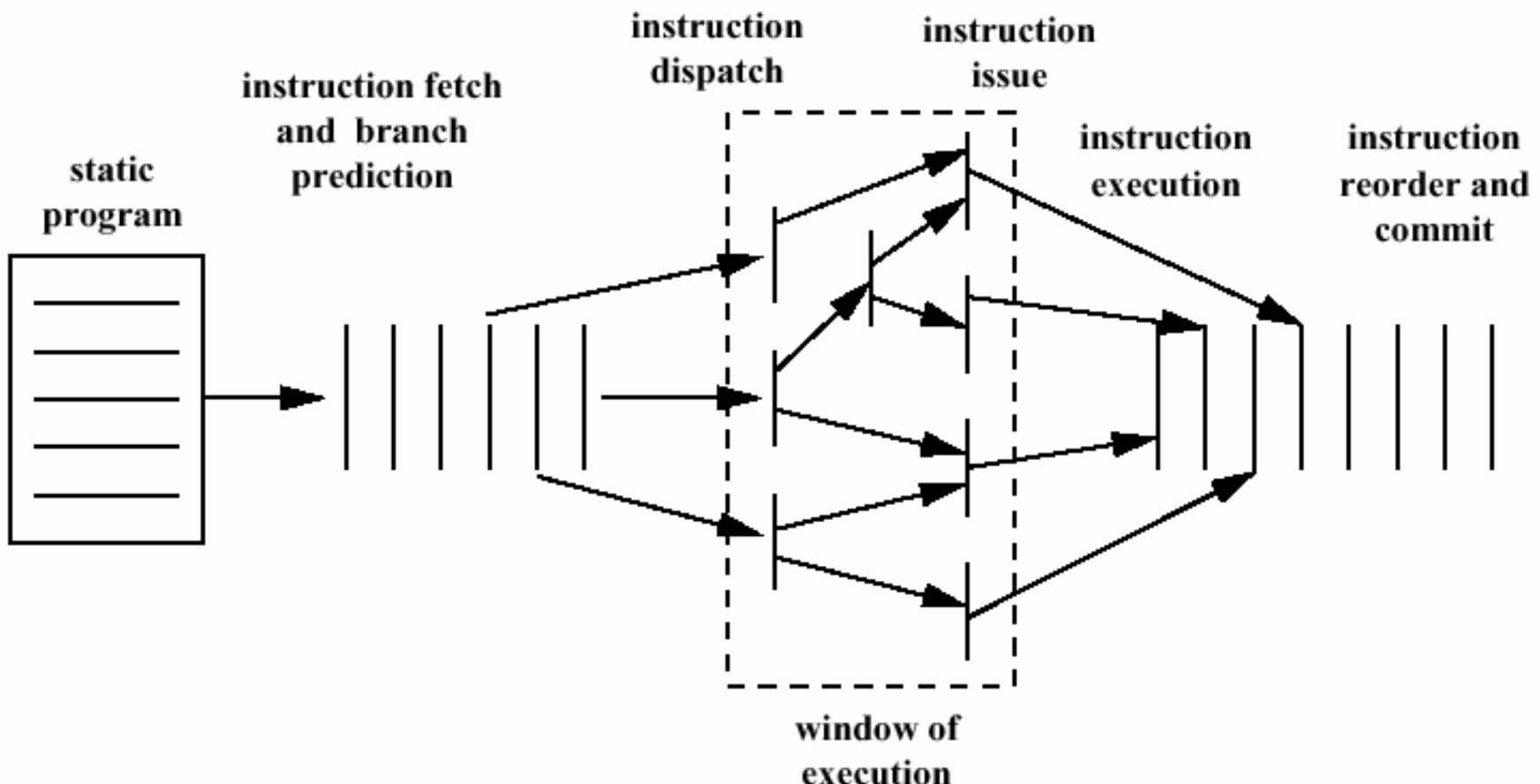
Procesory superskalarne

- Procesory, które wykonują równolegle więcej niż jeden rozkaz (paralelizm na poziomie rozkazu)
- Podstawowe bloki funkcjonalne CPU są zwielenokrotnione:
 - dwa potoki (lub więcej)
 - kilka jednostek ALU (zwykle osobne ALU dla operacji integer i FP)
- Termin „superskalarny” użyty po raz pierwszy w 1987 roku oznacza, że CPU przetwarza w danej chwili kilka argumentów skalarnych (liczb), a nie tylko jedną wielkość skalarną

Procesor superskalarny - koncepcja



Superskalar – koncepcja cd.



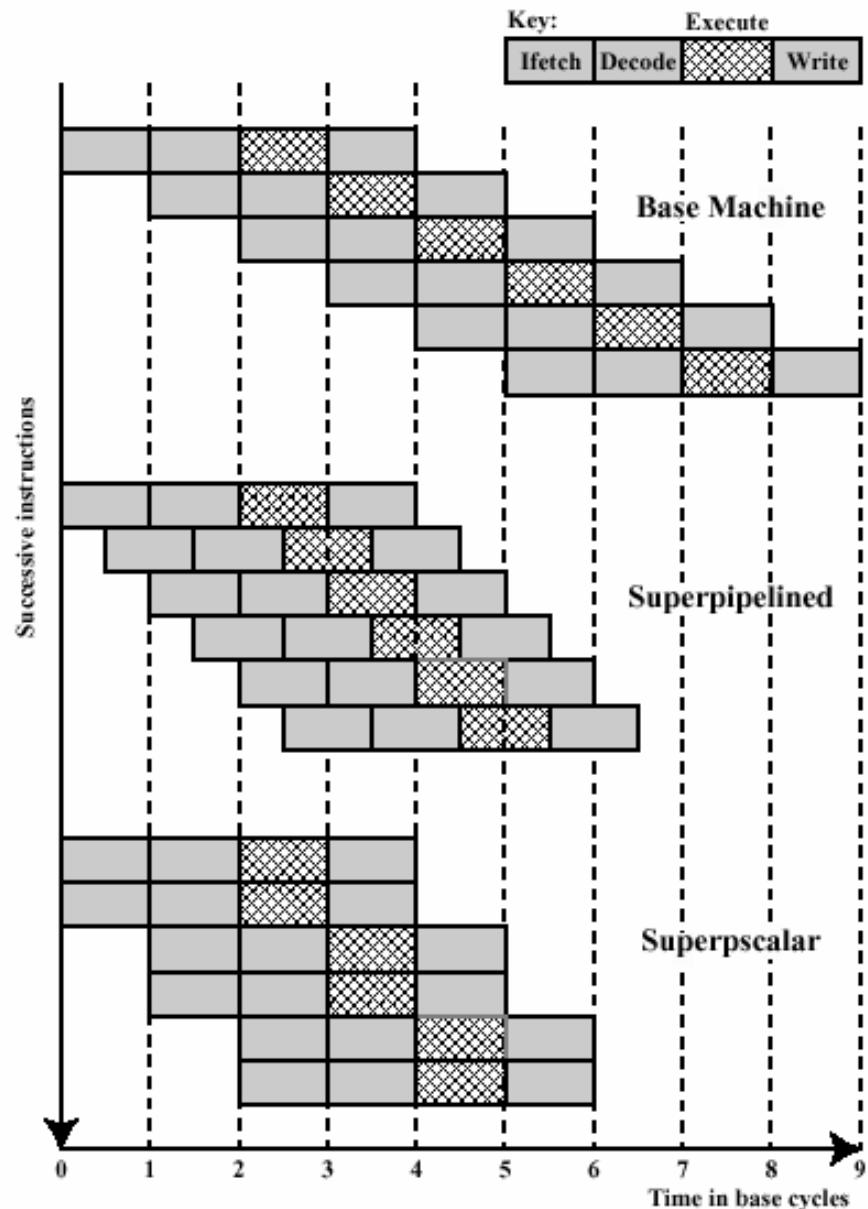
Superskalar cd.

Porównanie koncepcji CPU:

- 1) z potokiem instrukcji
- 2) z superpotokiem
- 3) superskalarnej

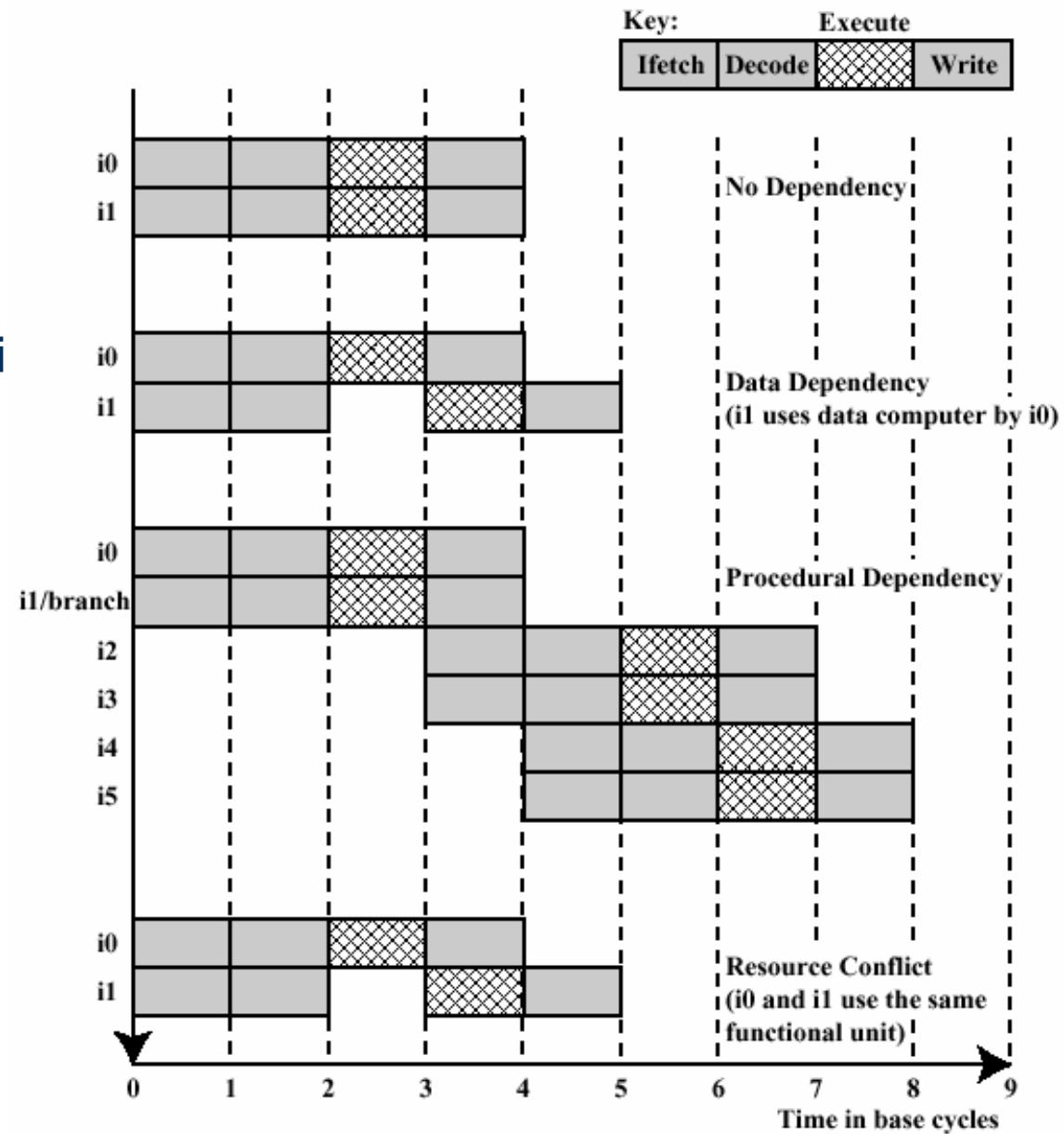
Superpotok jest taktowany podwójną częstotliwością zegara

Rozwiązanie superskalarne zapewnia największą wydajność



Superskalar - problemy

Zagadnienie
współzależności instrukcji
w procesorach
superskalarnych jest
jeszcze trudniejsze niż
problem hazardów w
pojedynczym potoku

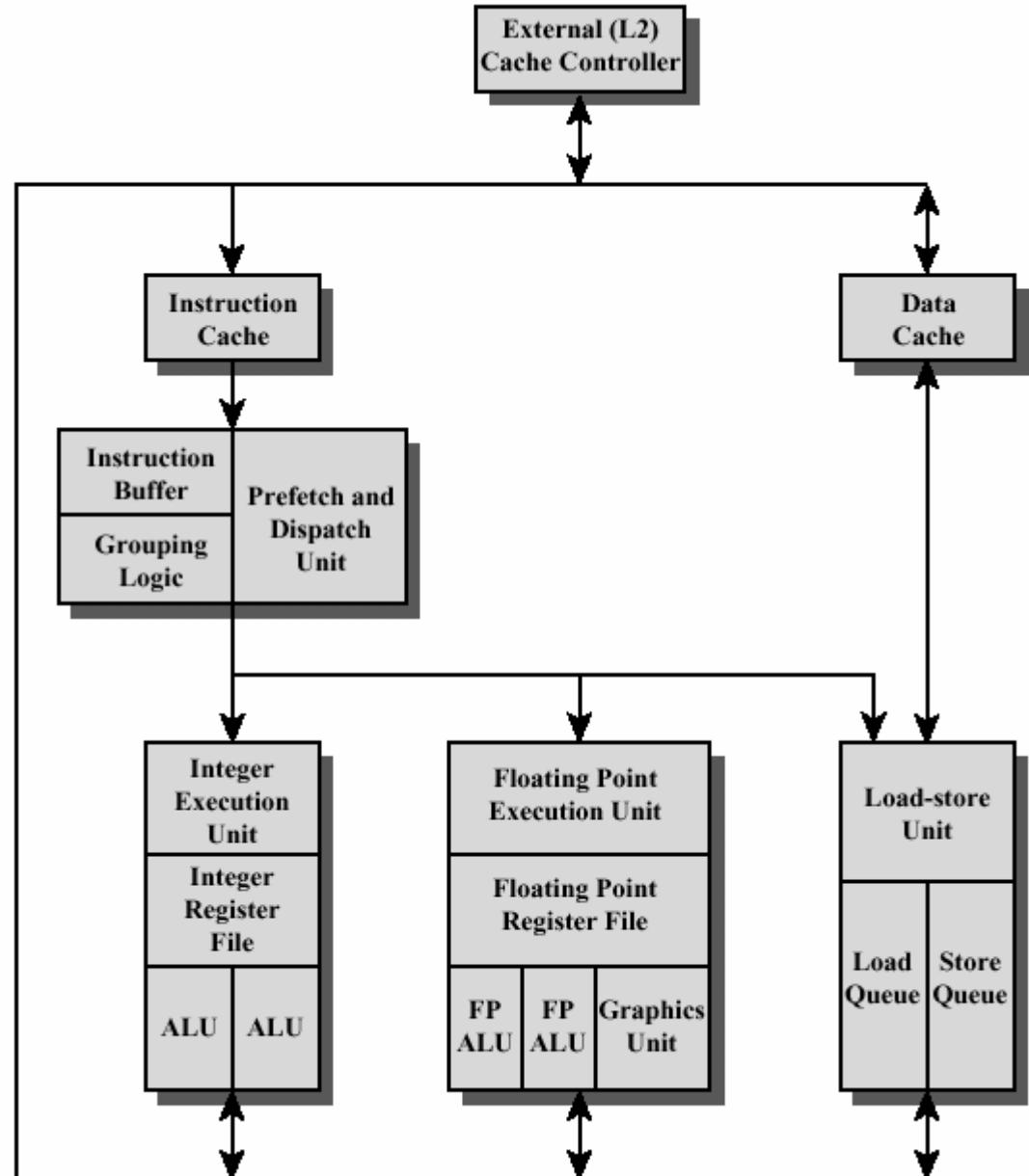


Superskalar – problemy cd.

- W celu odpowiedniego wykorzystania zalet architektury superskalarnej stosuje się rozmaite techniki:
- Przemianowywanie rejestrów – metoda usuwania uzależnień między instrukcjami przy użyciu zbioru pomocniczych rejestrów
- Okna rejestrów – parametry są przekazywane w bloku rejestrów nazywanym oknem; zmiana bloku (i tym samym parametrów) wymaga zmiany samego wskaźnika okna
- Statyczna optymalizacja kodu (w fazie kompilacji)
- Zaawansowane dynamiczne metody szeregowania instrukcji (parowanie, technika zmiany kolejności wykonywania rozkazów – *out-of-order completion*)
- Ze względu na swoje cechy architektura RISC znacznie lepiej sprzyja technice superskalarnej niż architektura CISC

Superskalar – przykład

Uproszczony schemat
blokowy procesora
superskalarnego
UltraSparc III (Sun)



Podsumowanie

- Koncepcja przetwarzania potokowego
 - głębokość potoku
 - potok a skoki – przewidywanie skoków
 - zjawisko hazardu
 - sterowania
 - danych
 - zasobów
 - szeregowanie statyczne i dynamiczne
- Architektura superskalarna
 - metody zwiększania wydajności
 - przemianowywanie rejestrów
 - okna rejestrów
 - statyczne i dynamiczne szeregowanie rozkazów

Organizacja i Architektura Komputerów

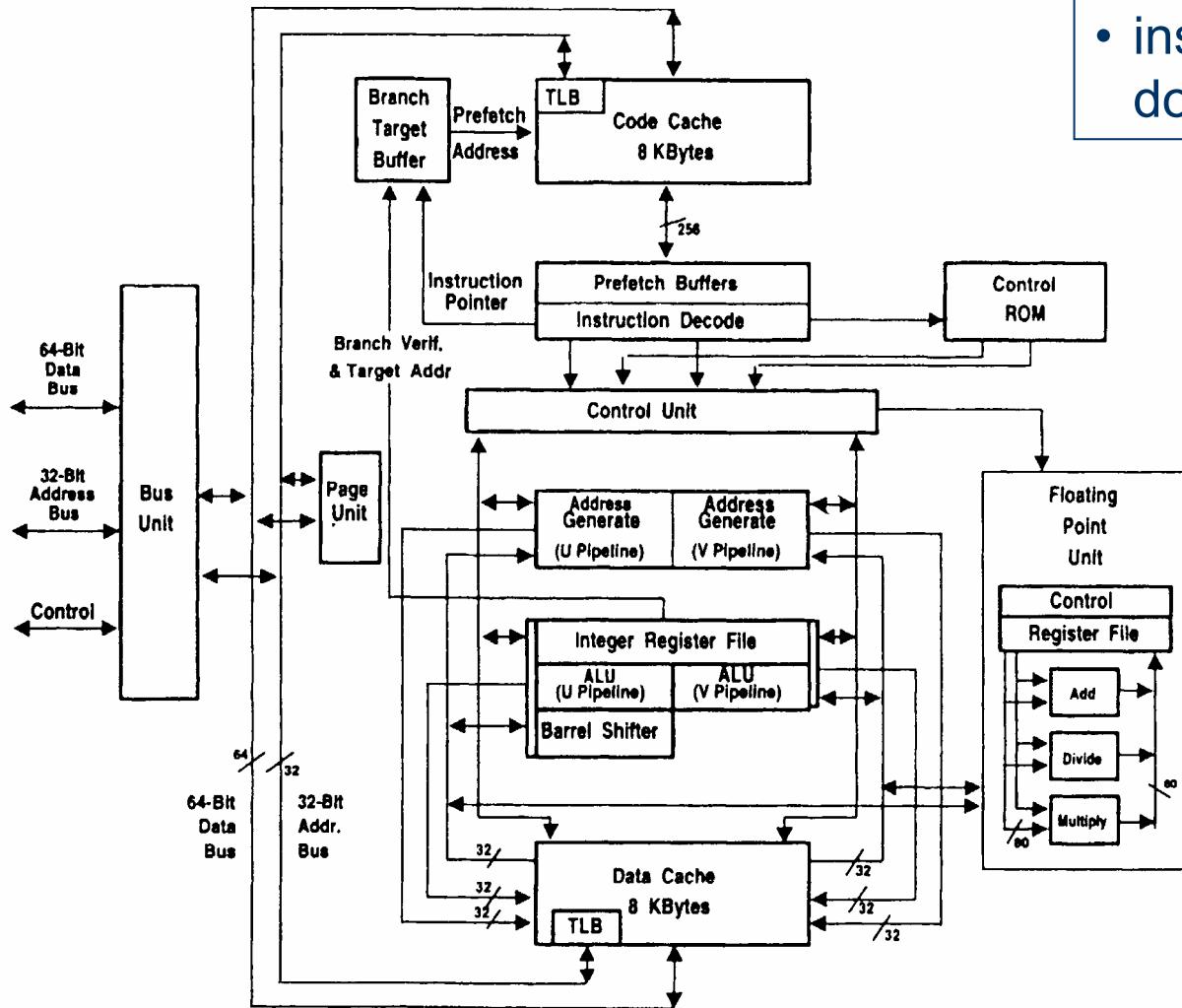
Architektury IA-32 i IA-64

Architektura IA-32

- Intel Architecture 32
 - pierwszy procesor IA-32 – 8086 (1978)
 - podstawowa architektura utrzymana do 1993 r (pierwszy procesor Pentium – superskalar, 2 potoki)
- Architektura P6
 - pierwszy procesor – Pentium Pro (1995)
 - 3-potokowy superskalar
 - 5 jednostek wykonawczych
 - 8+8KB L1 cache, 256 KB L2 cache
 - Pentium III (1999)
 - 16+16 KB L1 cache, 256 lub 512 L2 cache
 - Streaming SIMD Extension (SSE) – równoległe operacje na spakowanych 32-bitowych liczbach FP w rejestrach SSE o rozmiarze 128 bitów
 - Pentium III Xeon – ulepszona pamięć cache

Pentium CPU (1993)

- dwa potoki U i V
- instrukcje są dobierane parami



Mikroarchitektura P6

- Zapoczątkowana w Pentium Pro (1995)
 - Używana w procesorach Intel wcześniejszych od P4
 - Mieszana architektura CISC-RISC
 - lista instrukcji typu CISC
 - translacja kodu programu na ciąg mikrooperacji RISC
 - 3 potokowy superskalar, out-of-order execution
 - 14-stopniowy potok instrukcji
 - Potok składa się z 3 części:
 - wydawanie instrukcji: in-order
 - wykonanie instrukcji: out-of-order (RISC core)
 - kończenie instrukcji (retire): in-order

Mikroarchitektura P6 cd.

■ Fetch/decode unit

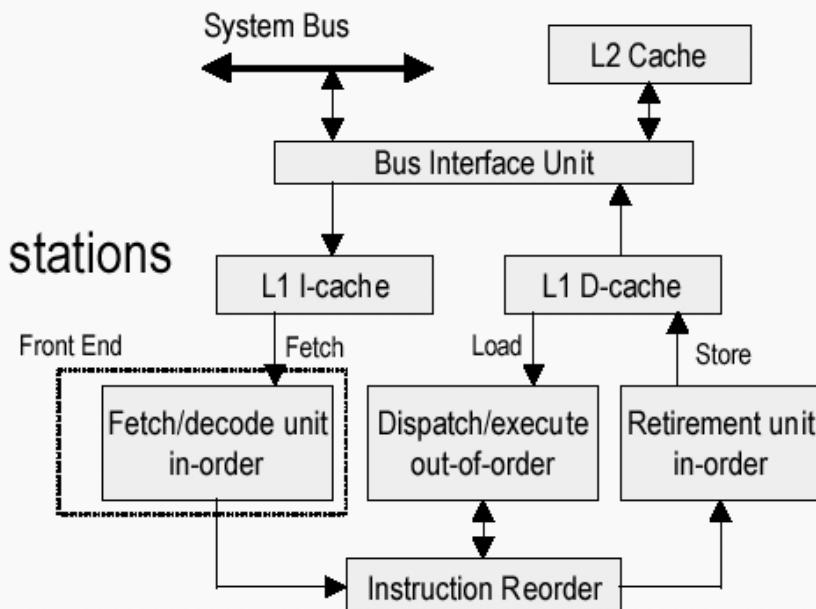
- ◆ fetches instruction from instruction cache / memory
- ◆ decodes instructions to µops
- ◆ branch prediction

■ Dispatch/execute unit

- ◆ schedules and executes µops
- ◆ reorder buffer and reservation stations
- ◆ 5 functional units

■ Retirement unit

- ◆ retires executed instructions in program order
- ◆ writes results to registers and memory



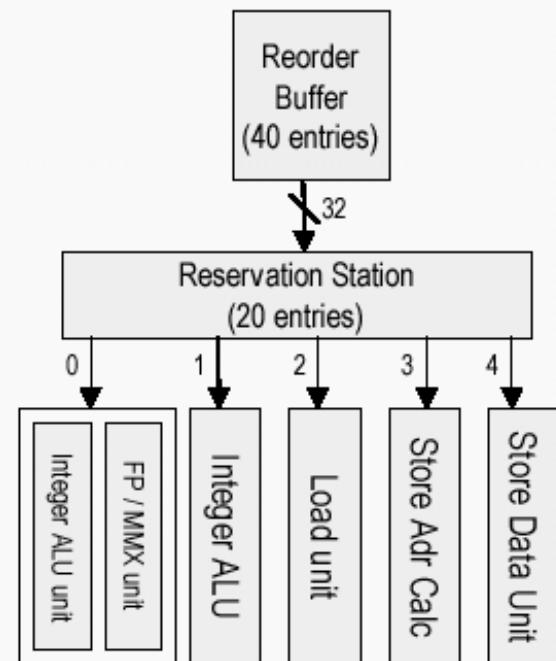
Mikroarchitektura P6 cd.

- Six pipelined execution units, connected to a 20 unit reservation station with 5 ports

- ◆ **port 0:** integer ALU, LEA, integer multiply, FP add, mul, div, MMX ALU, SSE FP mul, div, sqrt, move
- ◆ **port 1:** integer ALU, MMX ALU and shift, SSE add, sqrt, shuffle, move
- ◆ **port 2:** load unit, SSE load
- ◆ **port 3:** store address unit, SSE store
- ◆ **port 4:** store data unit, SSE store

- FP divide is not pipelined

- ◆ latency: single/double/extended
18 / 32 / 38 cycles



P6 – Pentium III Xeon

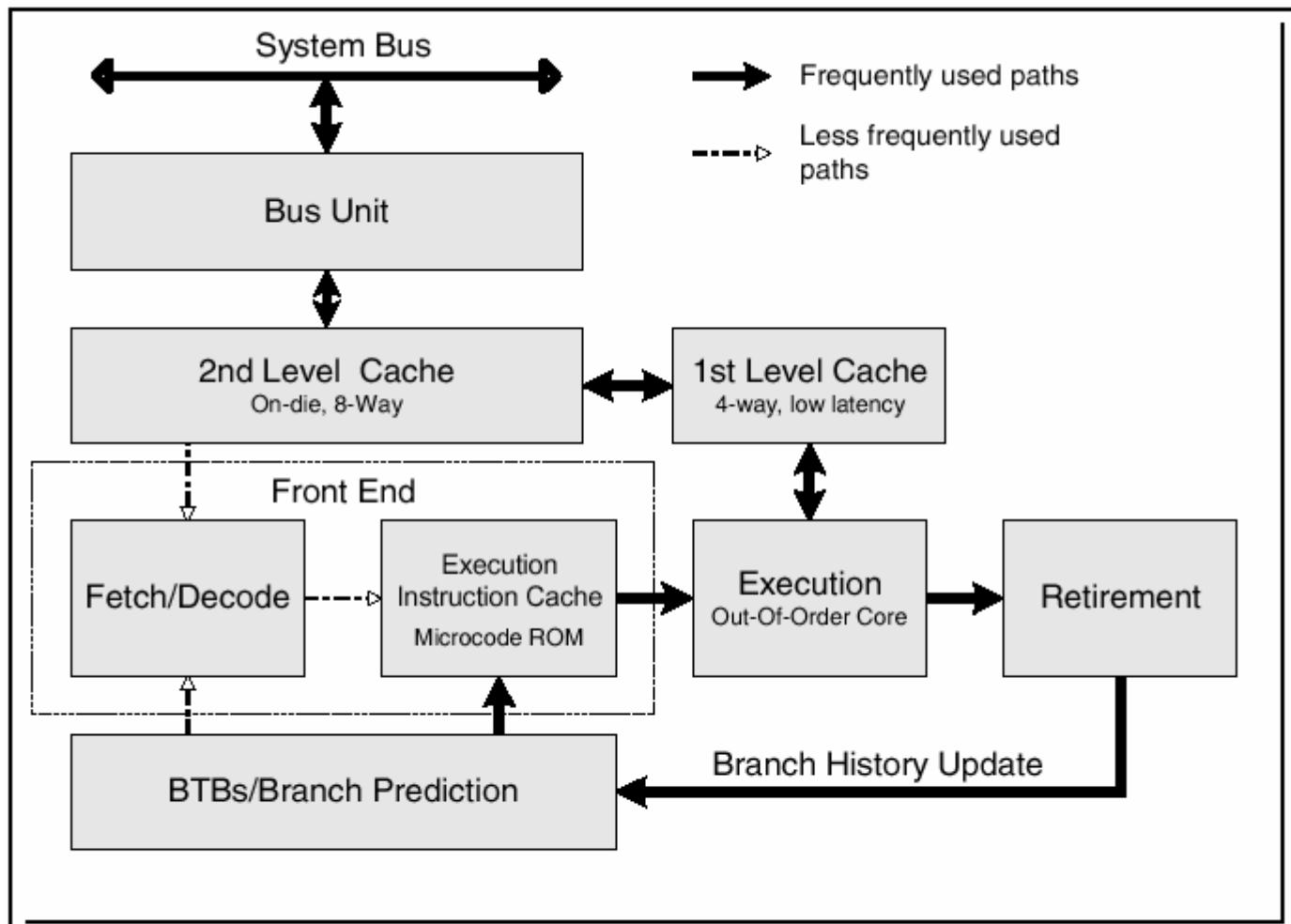


Figure 2-1. The P6 Processor Micro-Architecture with Advanced Transfer Cache Enhancement

Pentium 4

- Nazwa projektu: Willamette
- Architektura NetBurst
- Głęboki potok – 20 stopni
 - zaprojektowany do pracy z szybkim zegarem >1,5 GHz
 - różne części CPU pracują z różną częstotliwością zegara
- Trzy sekcje:
 - wydawanie instrukcji (dispatch) – in-order
 - wykonanie instrukcji – out-of-order
 - kończenie instrukcji – in-order
- Technologia SSE2
 - ulepszone technologie MMX i SSE

Intel NetBurst

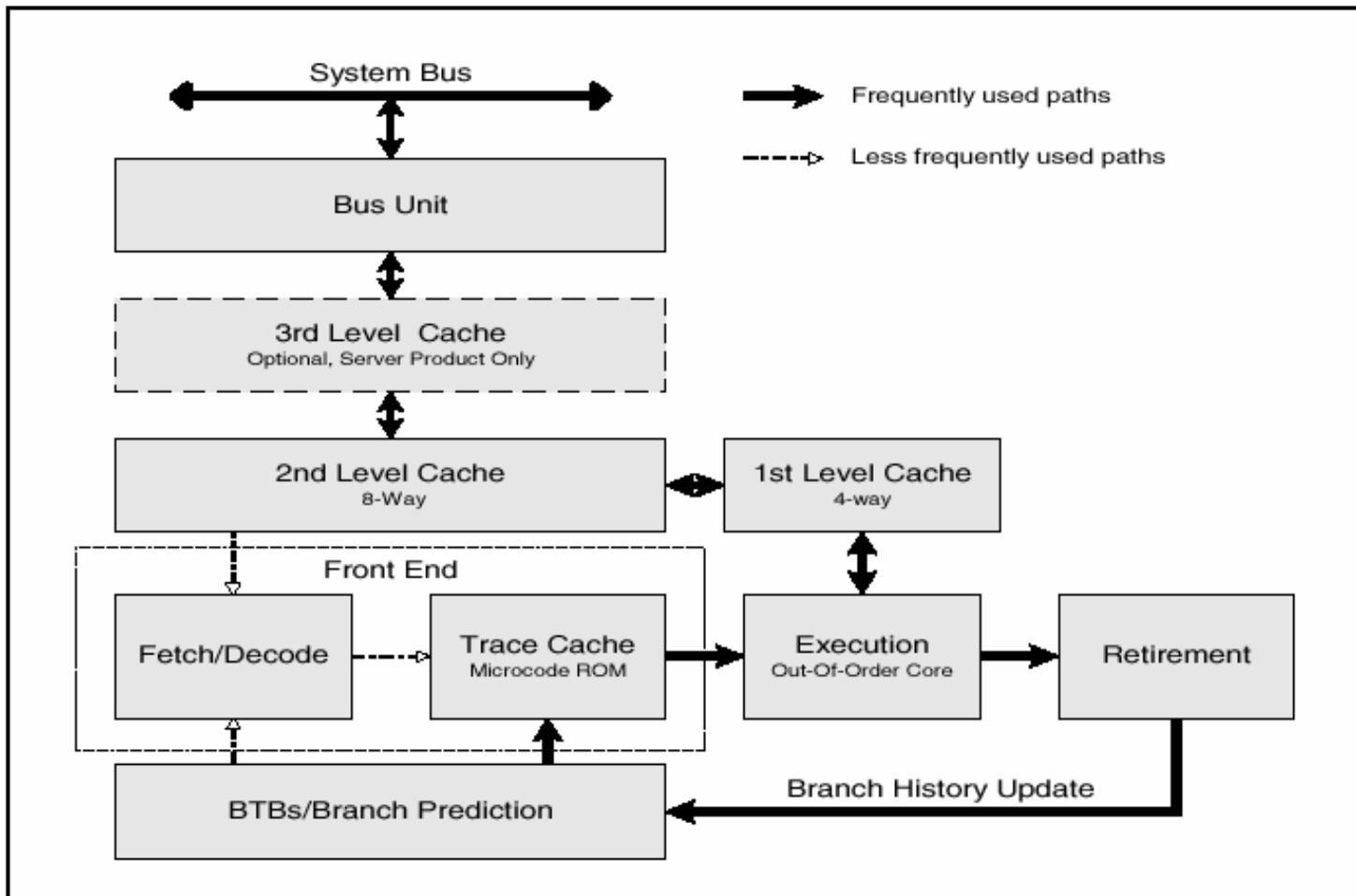
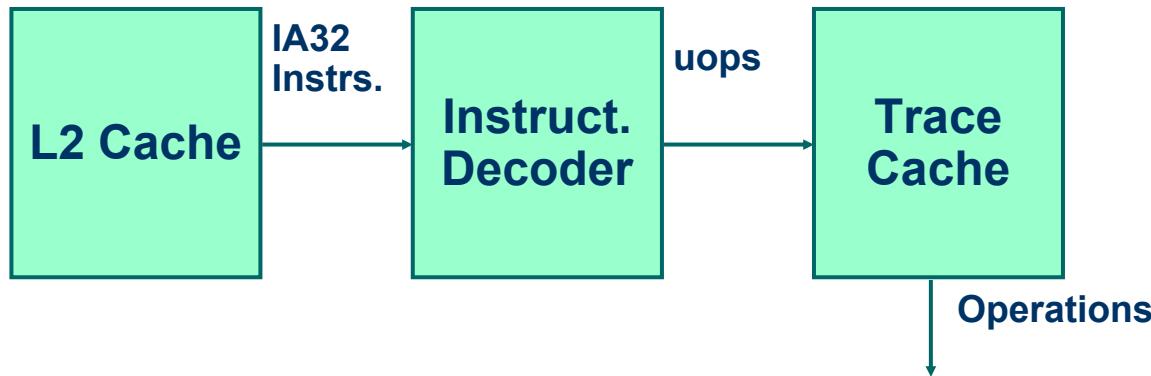


Figure 2-2. The Intel NetBurst Micro-Architecture

Pentium 4 – trace cache

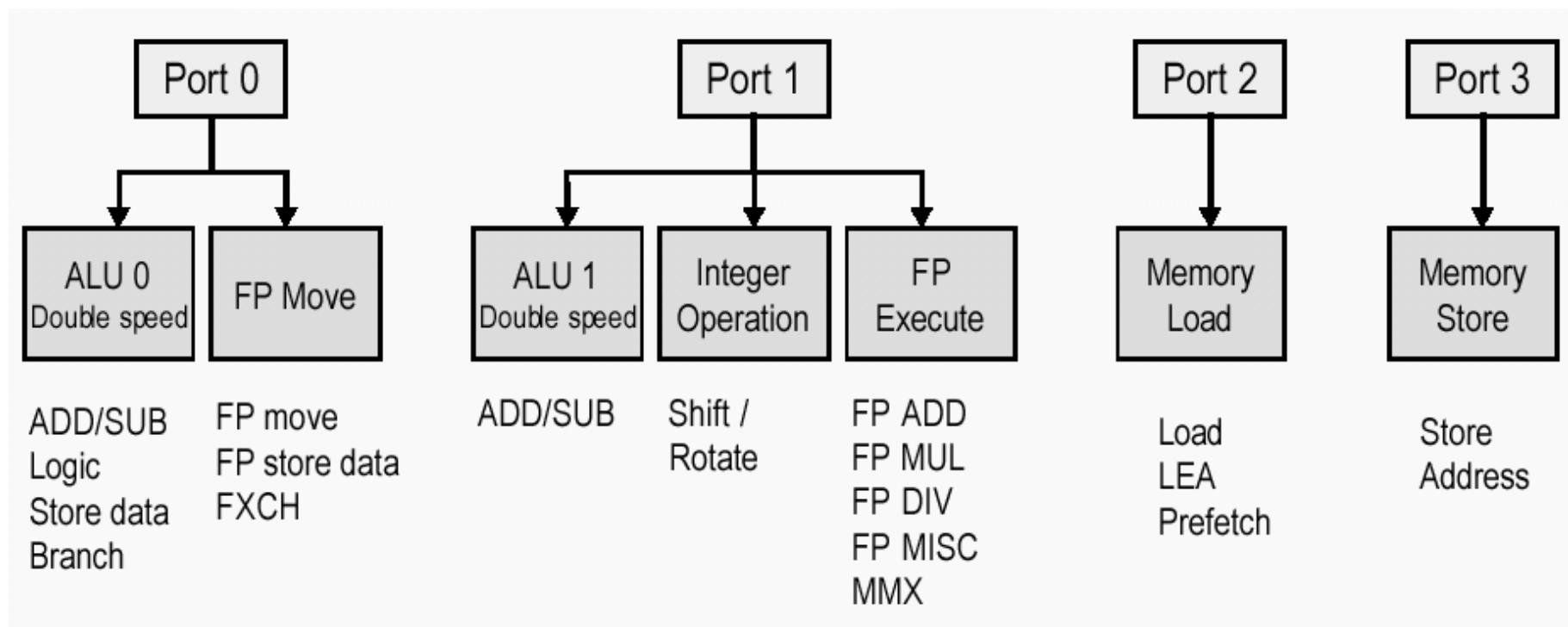


- Trace Cache
 - zastępuje tradycyjną pamięć cache
 - instrukcje są przechowywane w zdekodowanej postaci (jako mikrooperacje)
 - zmniejsza wymagania dotyczące szybkości pracy dekodera

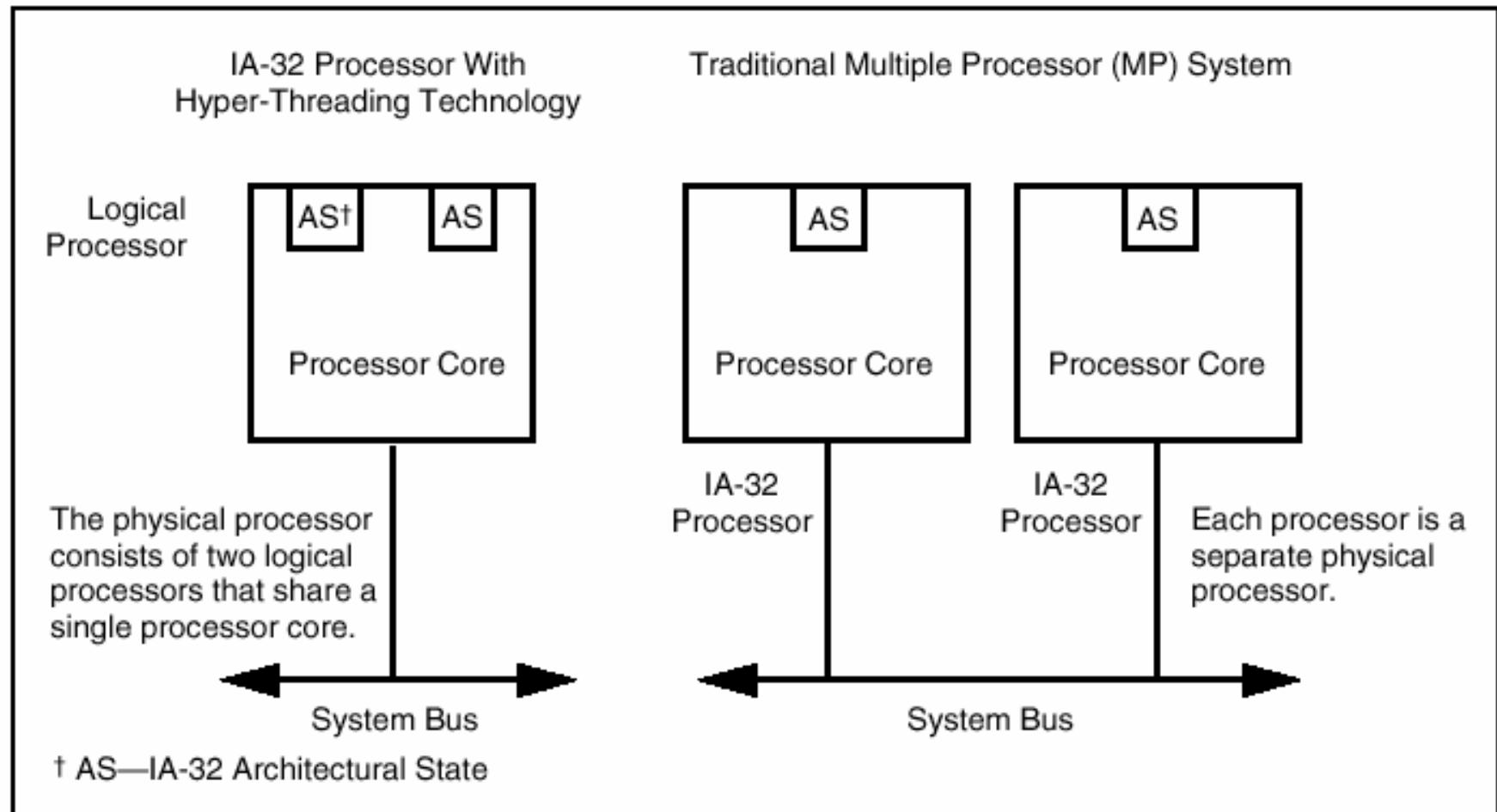
Ulepszenia w NetBurst wobec P6

- Trace cache – 12K mikrooperacji
- 8 KB L1 cache, czas dostępu 2 cykle
- 256 KB L2 cache, czas dostępu 7 cykli
- ALU taktowane podwójną częstotliwością zegara
- 20-stopniowy potok, częstotliwość > 1,5 GHz
- Poczwórna przepływność szyny (400 MHz) – quad-pumped
- Technologia SSE2

Pentium 4 – układy wykonawcze



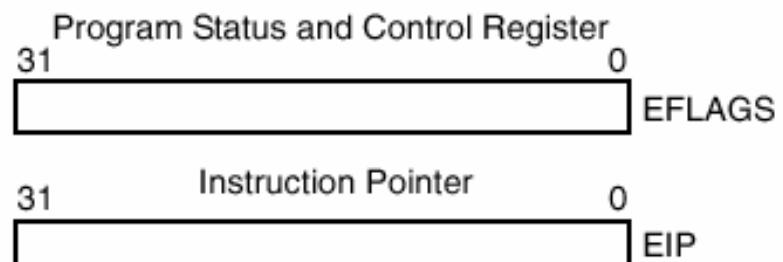
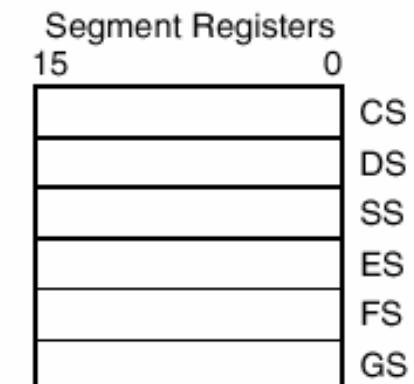
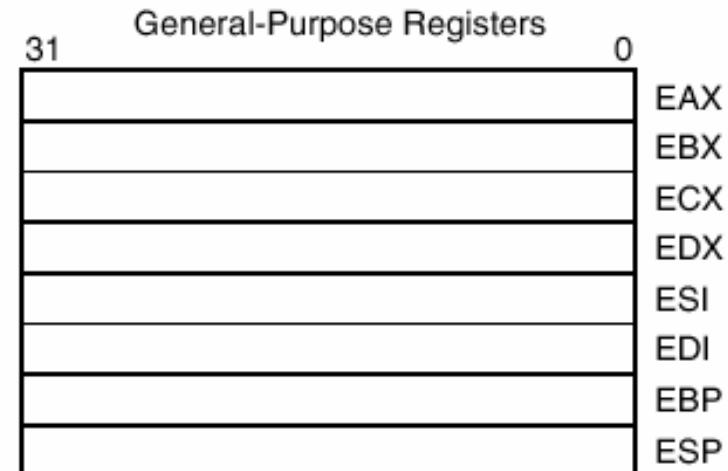
Hyper-Threading



Rejestry P4

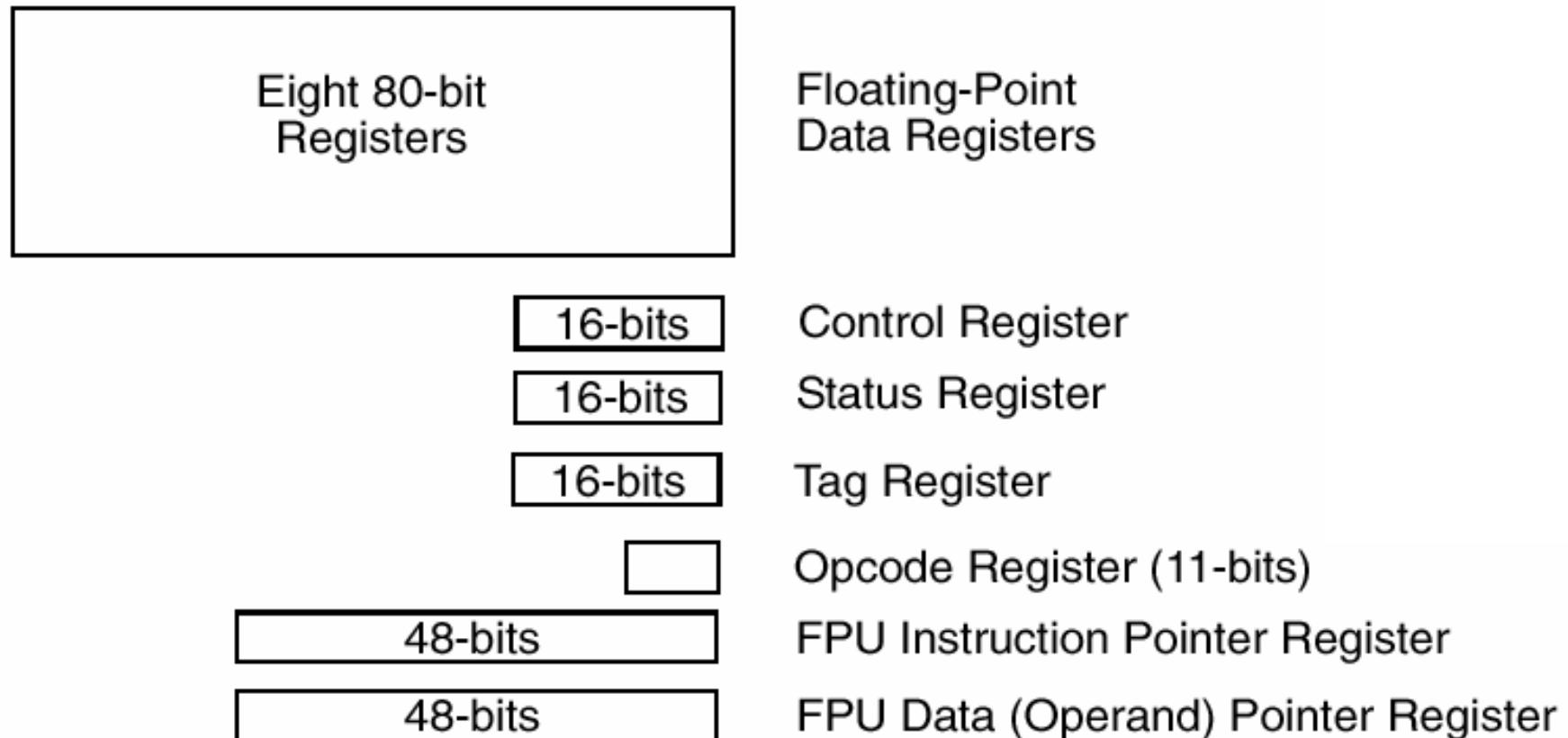
		General-Purpose Registers			
31	16	15	8	7	0
		AH	AL		
		BH	BL		
		CH	CL		
		DH	DL		
		BP			
		SI			
		DI			
		SP			

16-bit	32-bit
AX	EAX
BX	EBX
CX	ECX
DX	EDX
	EBP
	ESI
	EDI
	ESP



Rejestry P4 cd.

FPU Registers



Rejestry P4 cd.

MMX Registers

Eight 64-bit
Registers

MMX Registers

SSE and SSE2 Registers

Eight 128-bit
Registers

XMM Registers

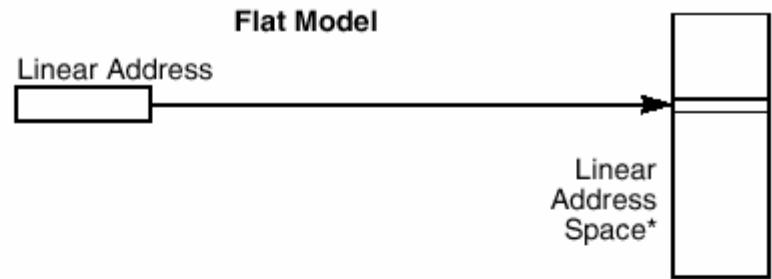
32-bits

MXCSR Register

Modele pamięci

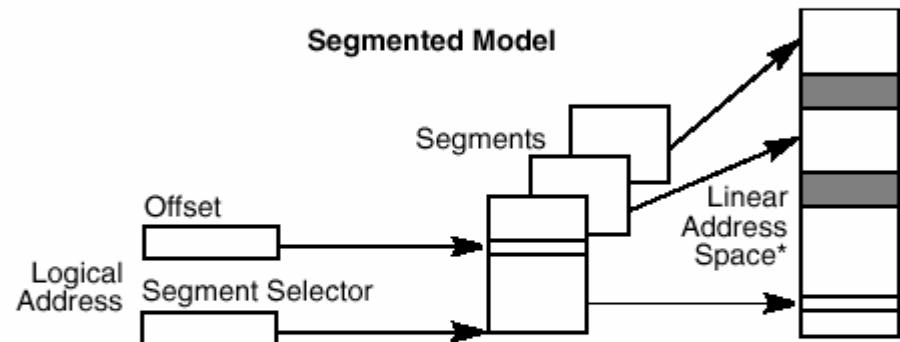
- **Flat Model**

płaski model pamięci – bez podziału na segmenty



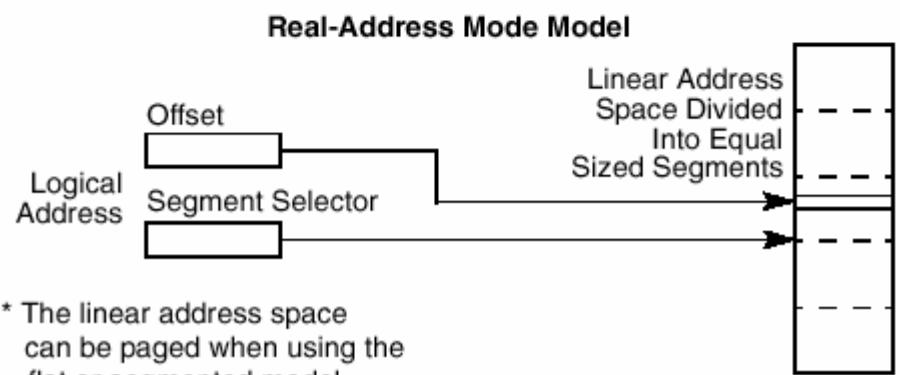
- **Segmented Model**

pamięć podzielona na segmenty



- **Real-Address-Model**

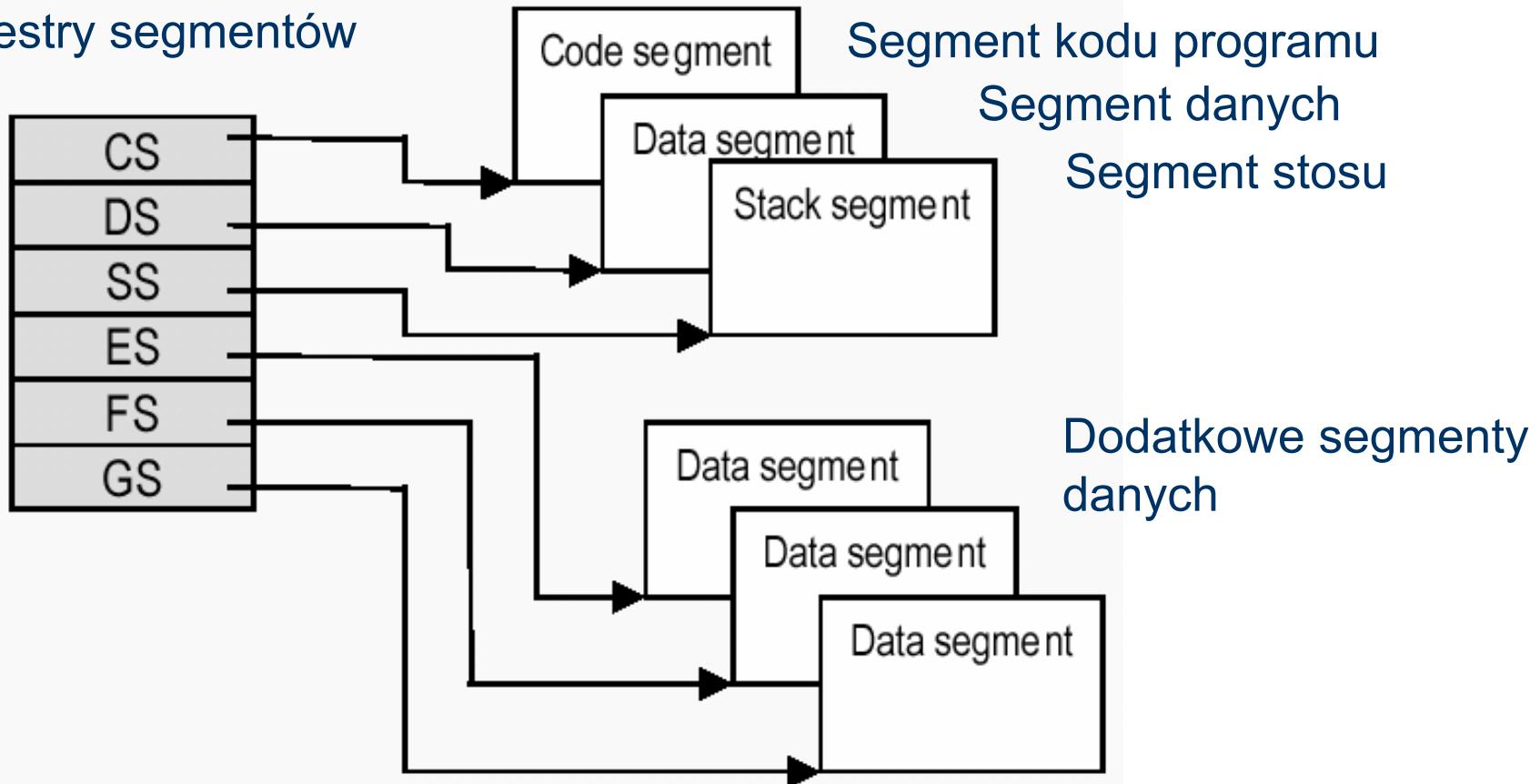
segmentacja typu x86



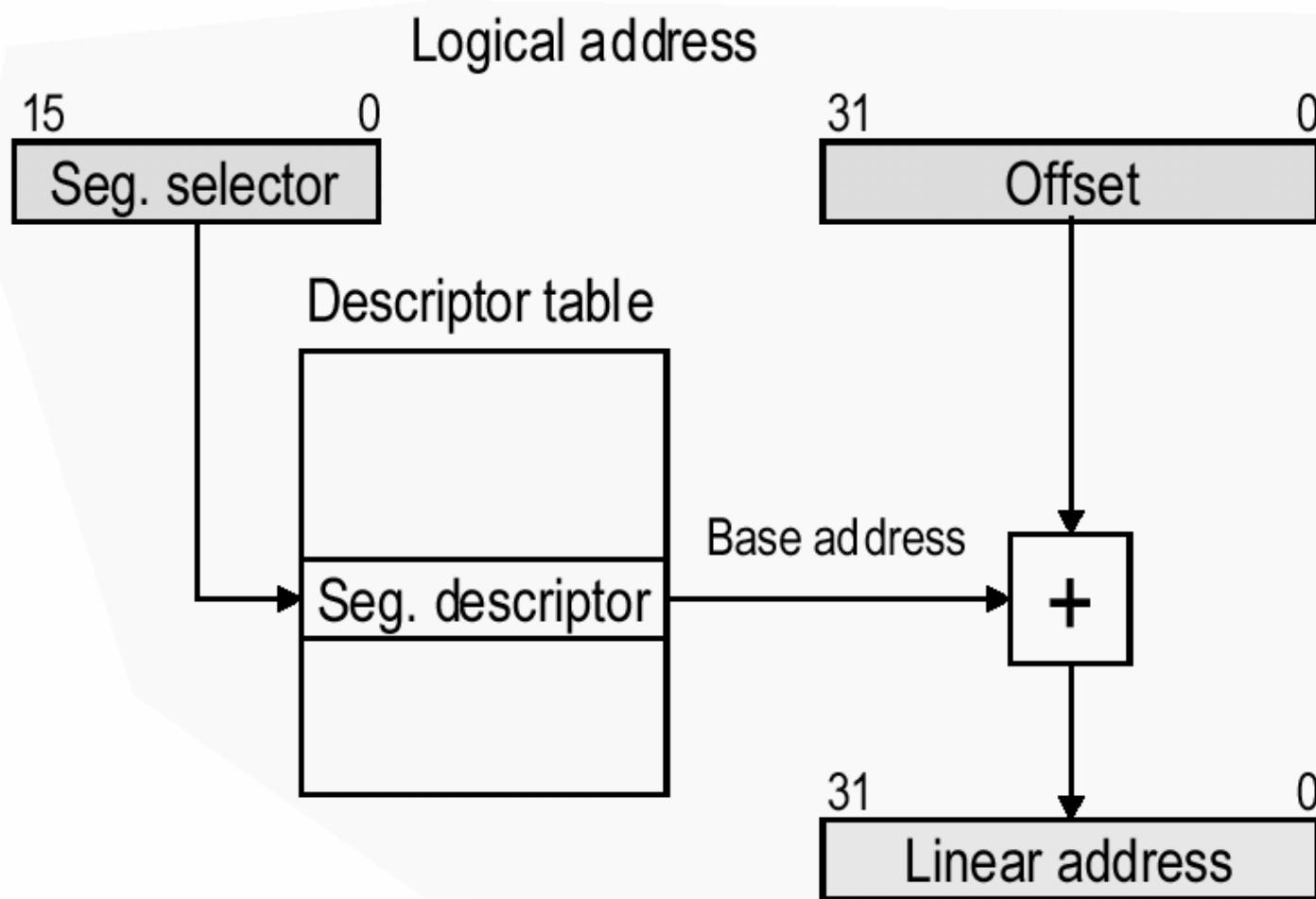
* The linear address space can be paged when using the flat or segmented model.

Segmentacja pamięci

Rejestry segmentów



Segmentacja pamięci cd.



Segmentacja

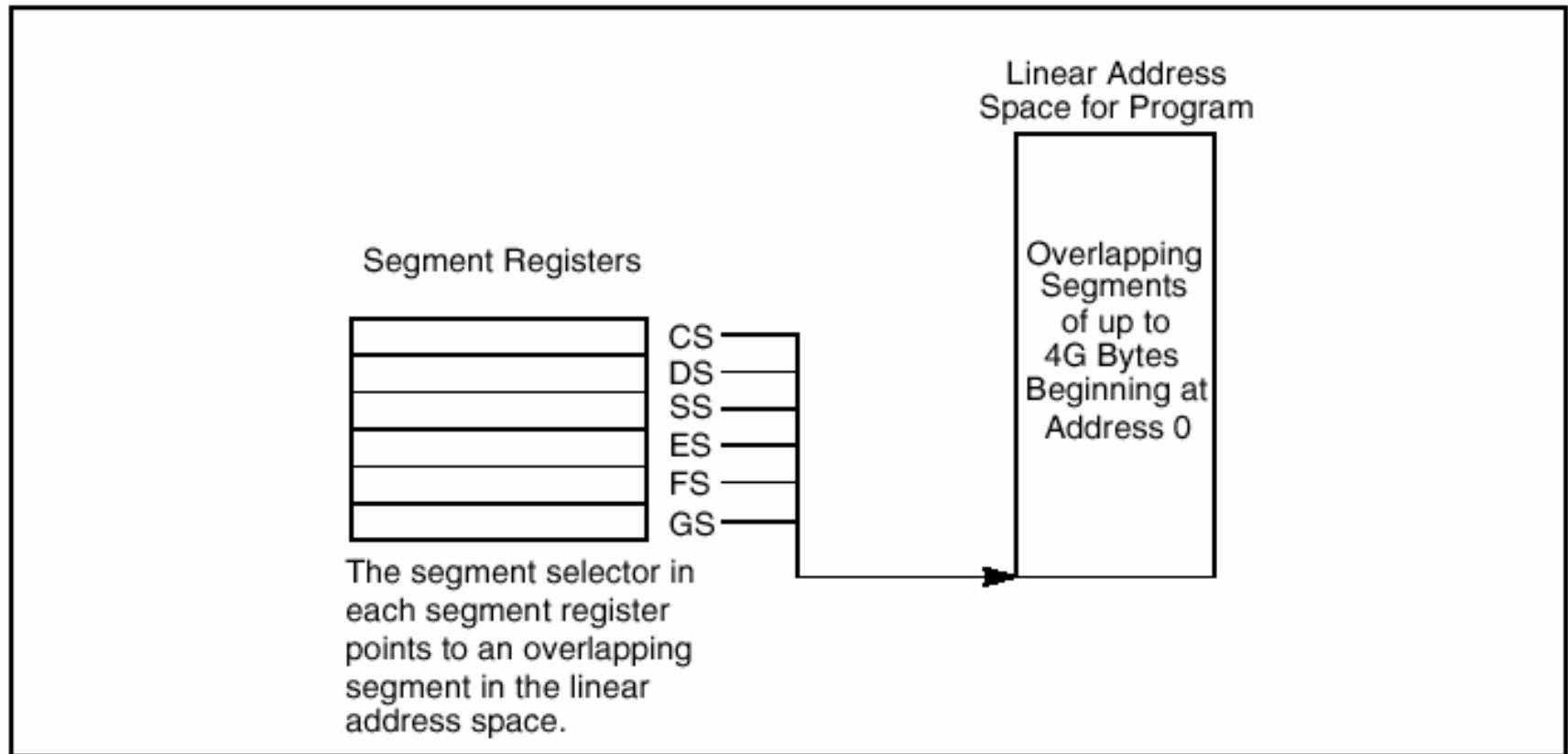


Figure 3-5. Use of Segment Registers for Flat Memory Model

Segmentacja cd.

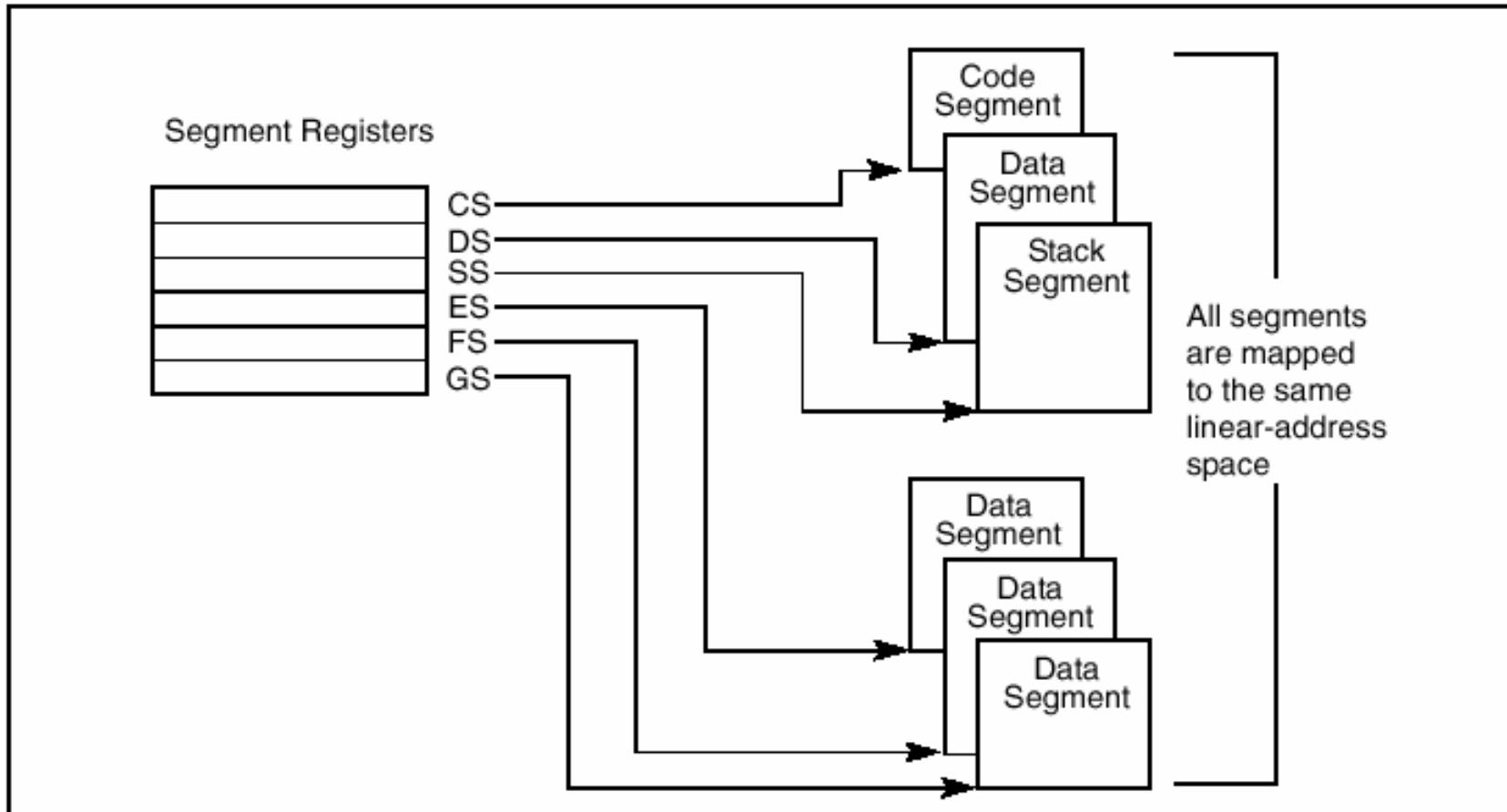
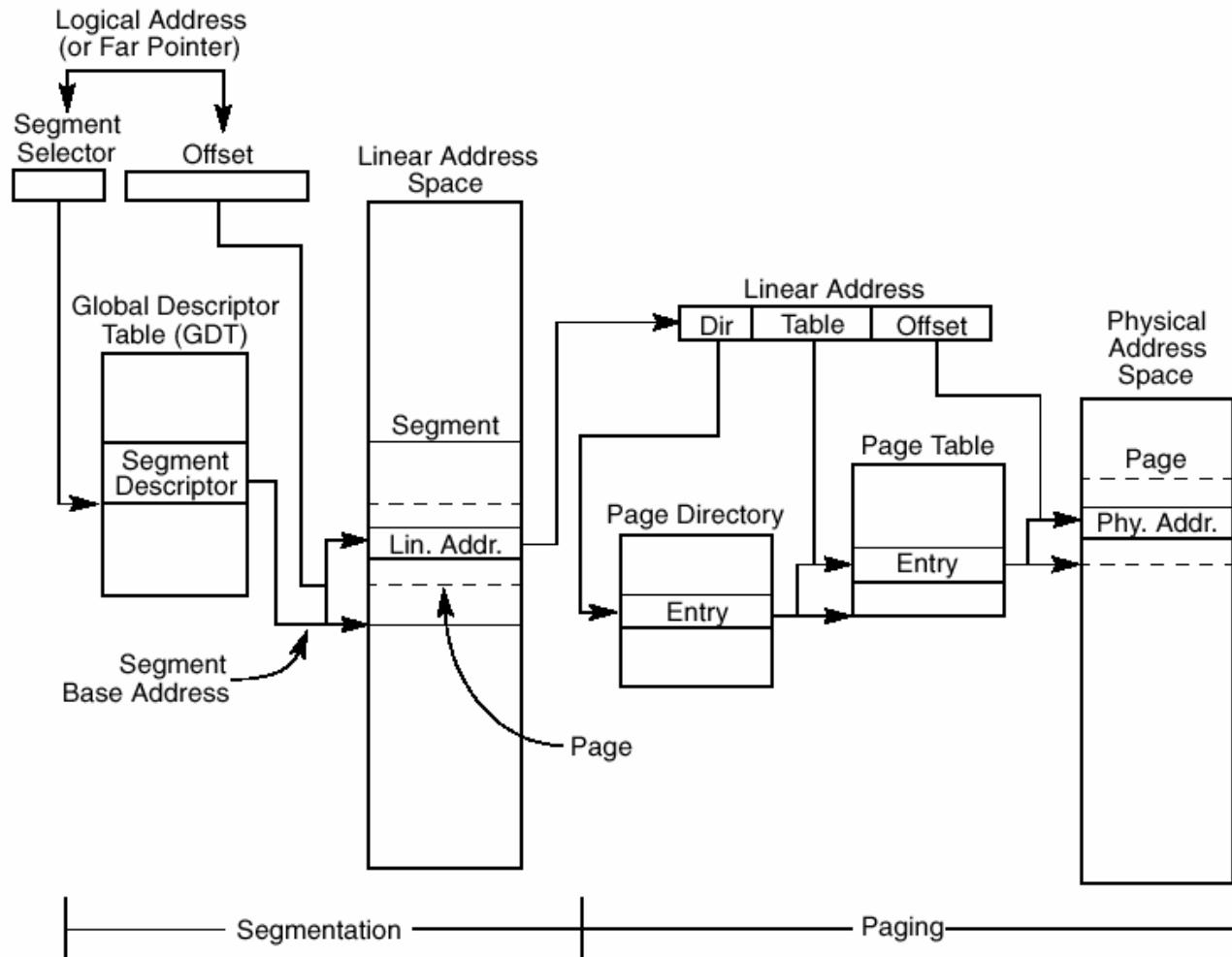


Figure 3-6. Use of Segment Registers in Segmented Memory Model

Segmentacja cd.



Segmentacja cd.

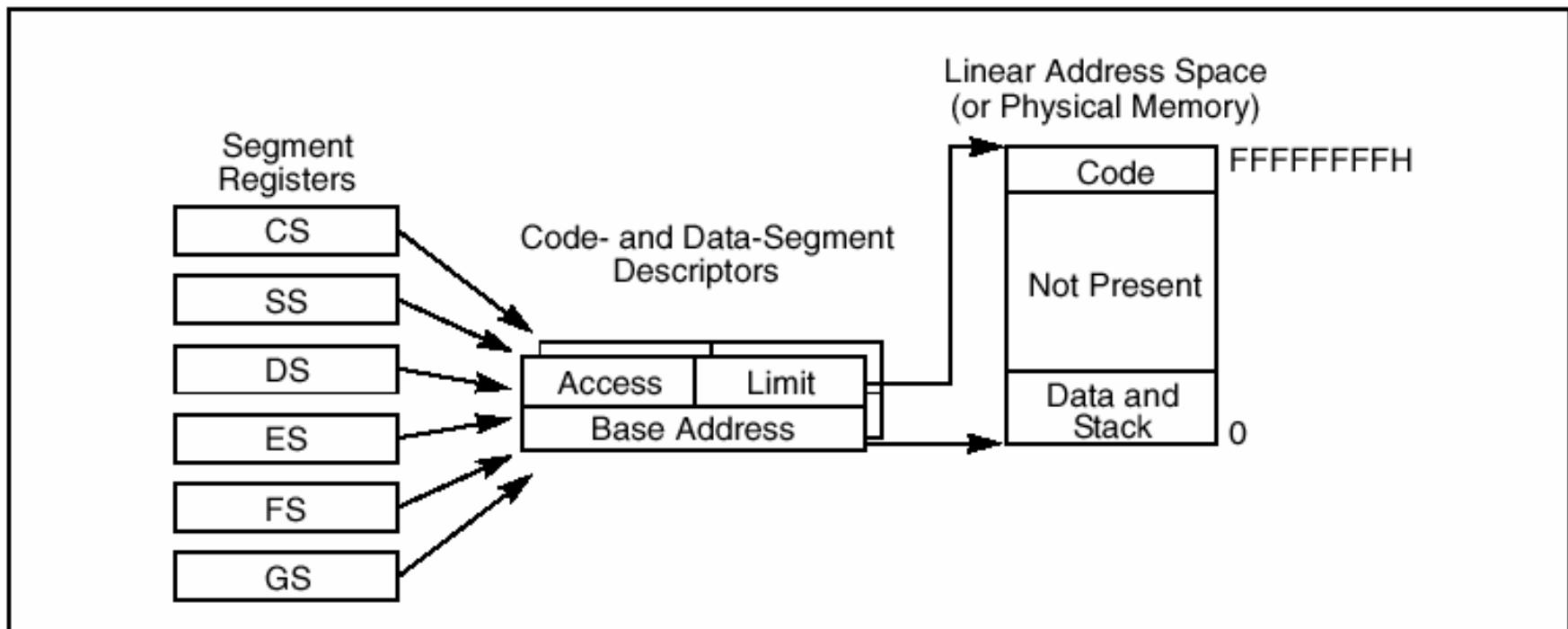


Figure 3-2. Flat Model

Segmentacja cd.

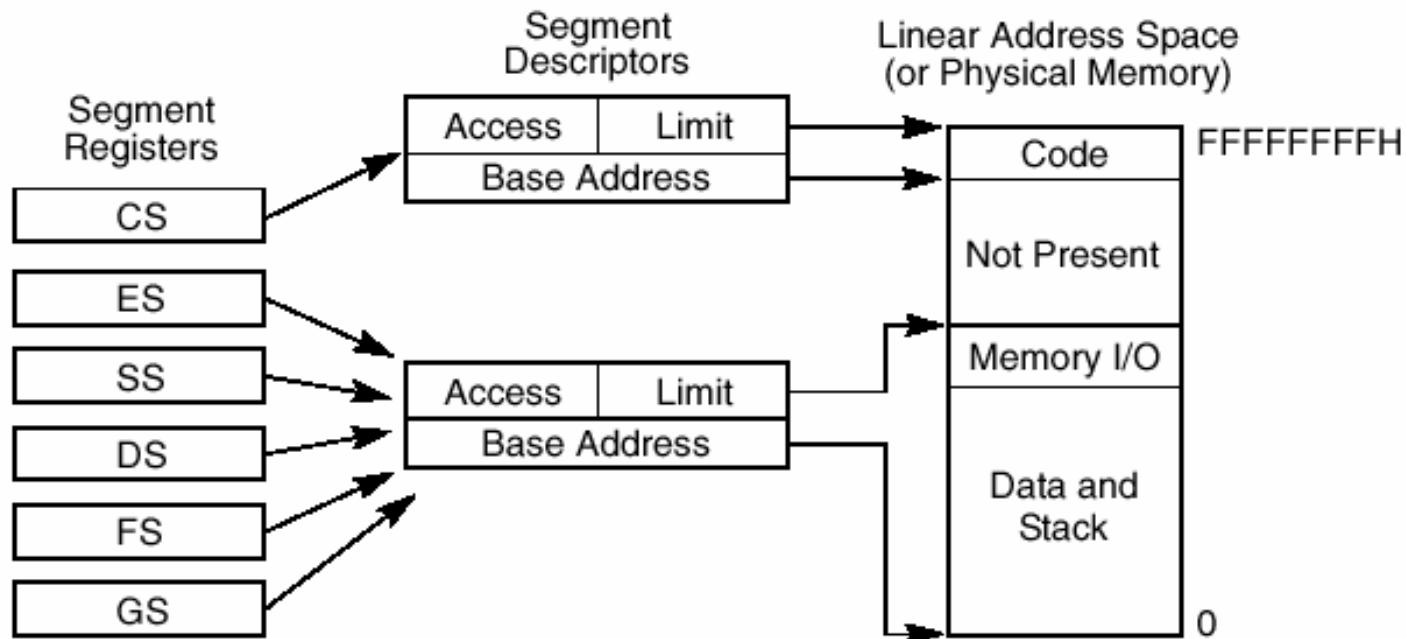


Figure 3-3. Protected Flat Model

Segmentacja cd.

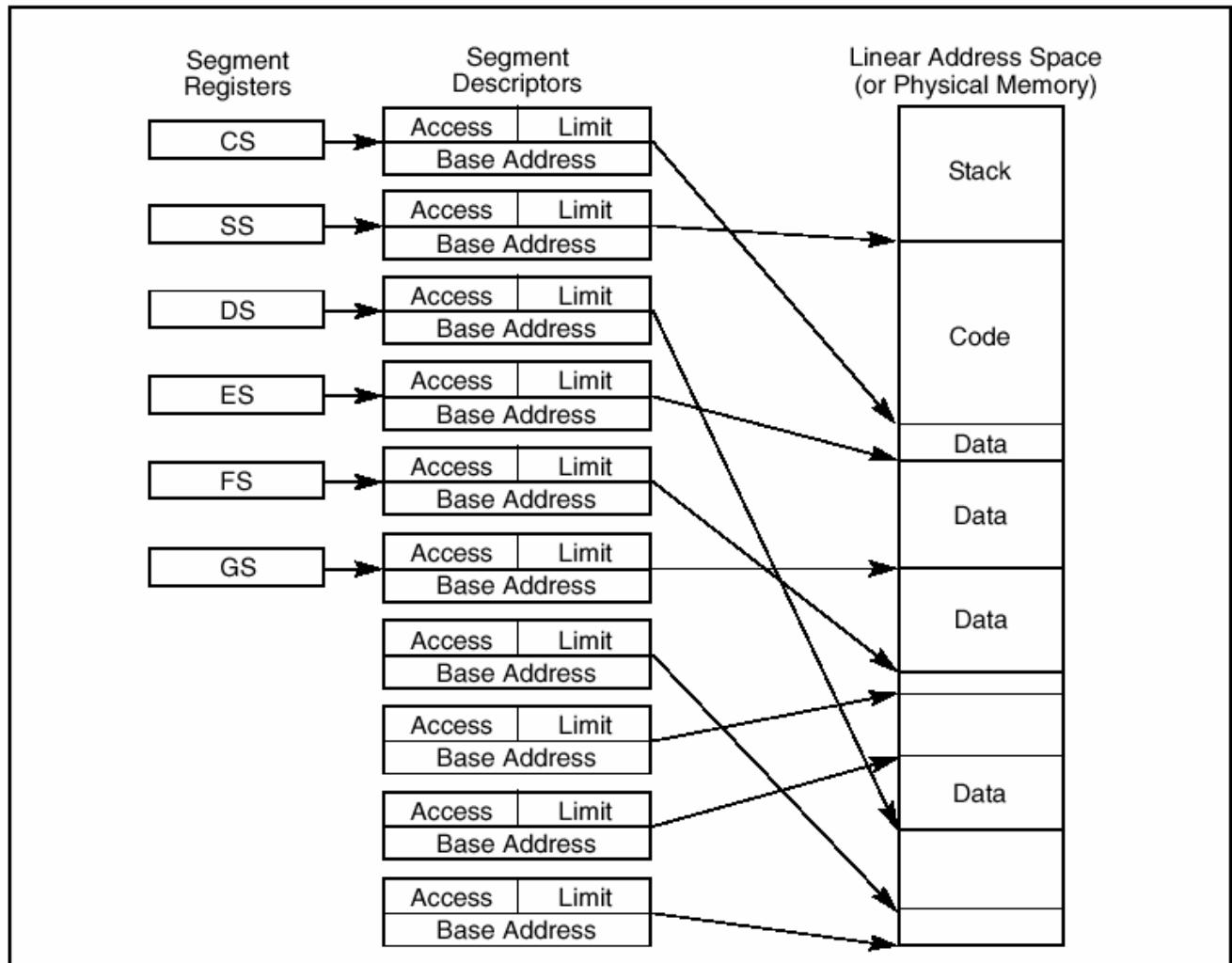
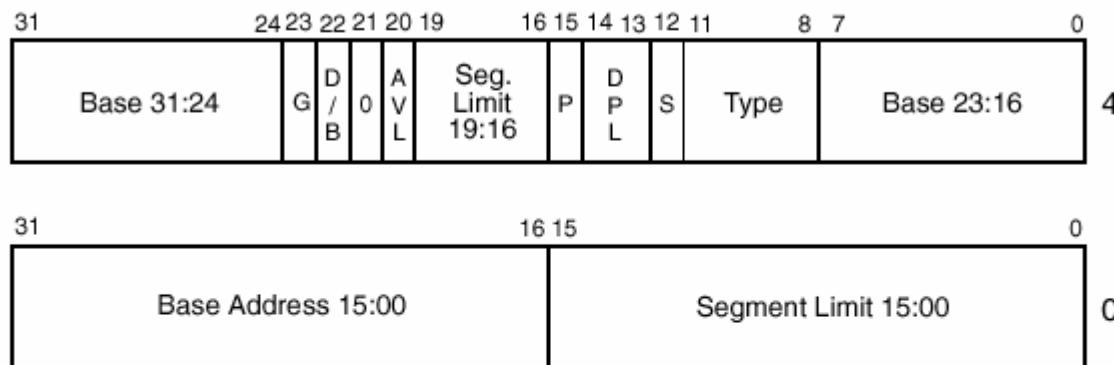


Figure 3-4. Multi-Segment Model

Segmentacja cd.



AVL — Available for use by system software

BASE — Segment base address

D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)

DPL — Descriptor privilege level

G — Granularity

LIMIT — Segment Limit

P — Segment present

S — Descriptor type (0 = system; 1 = code or data)

TYPE — Segment type

Figure 3-8. Segment Descriptor

Protekcja

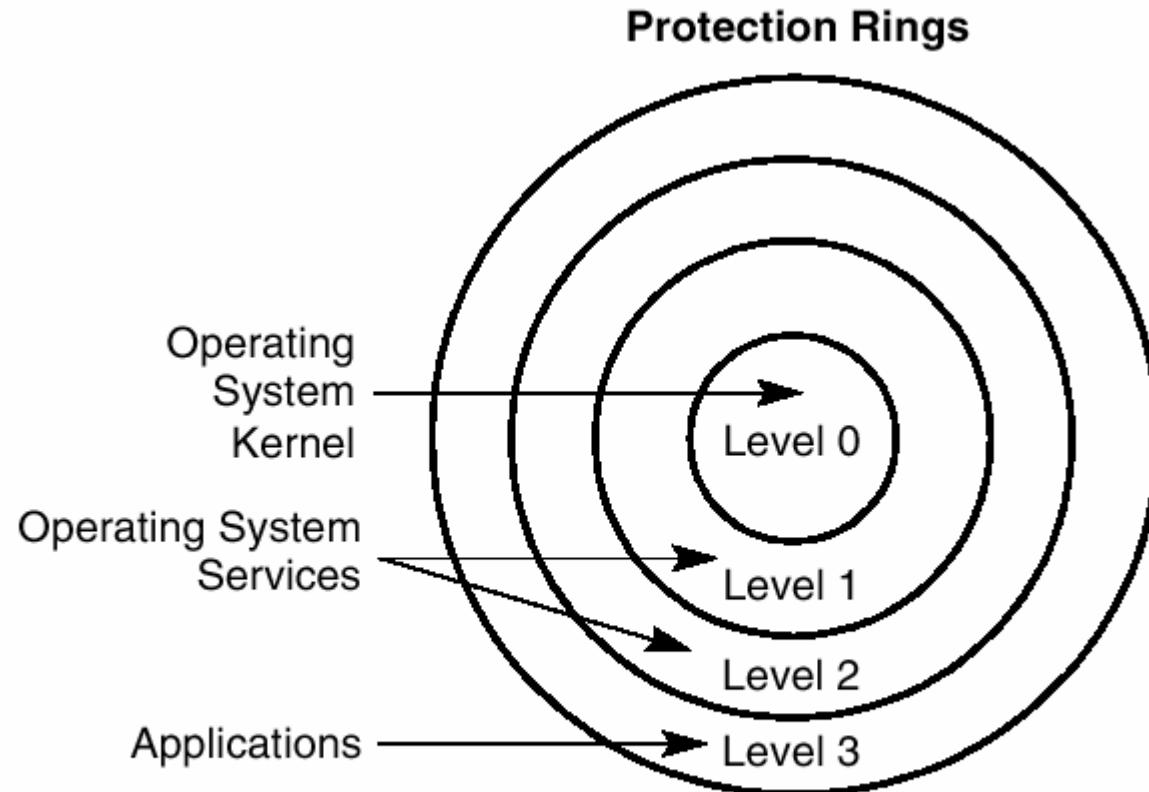
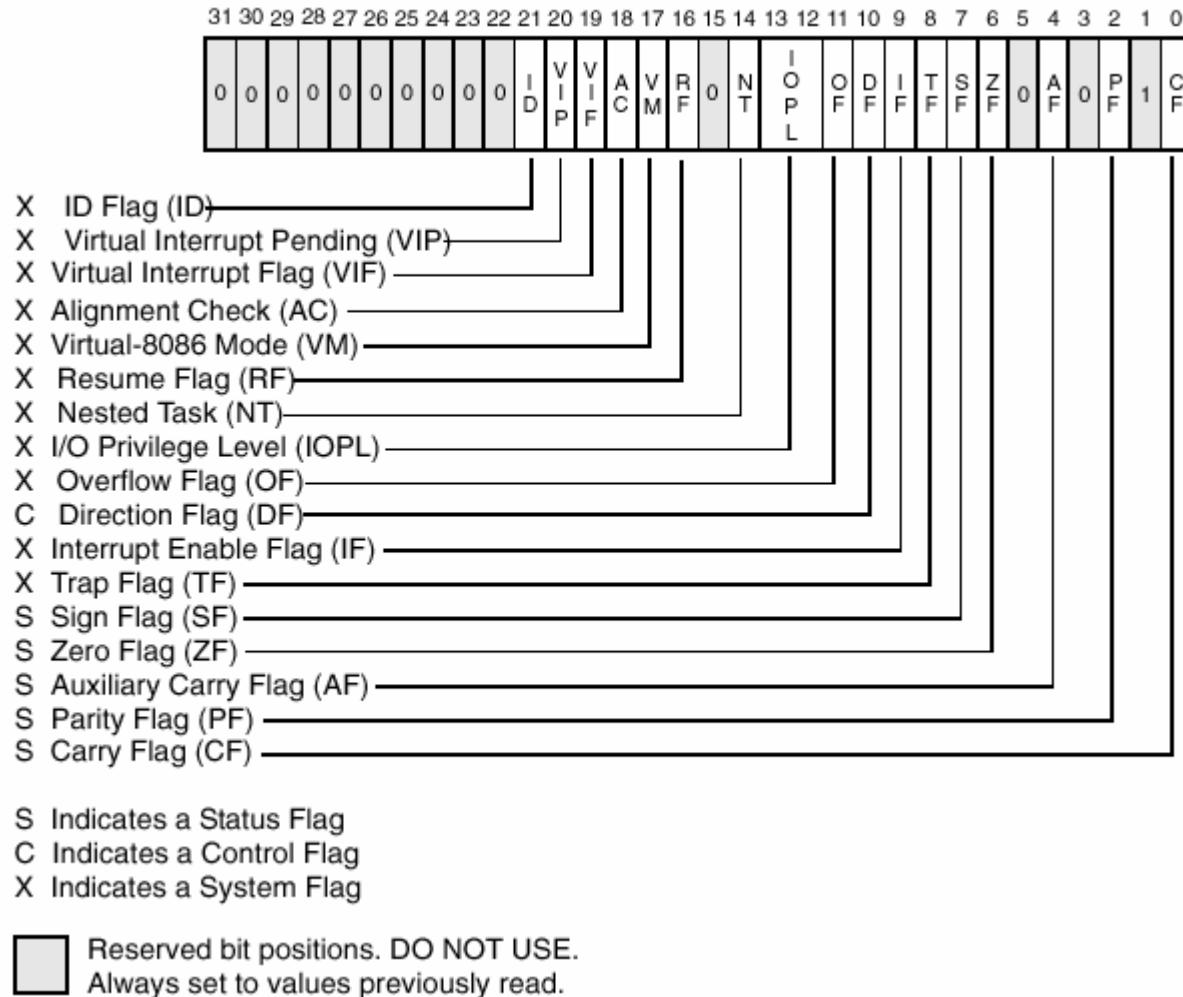
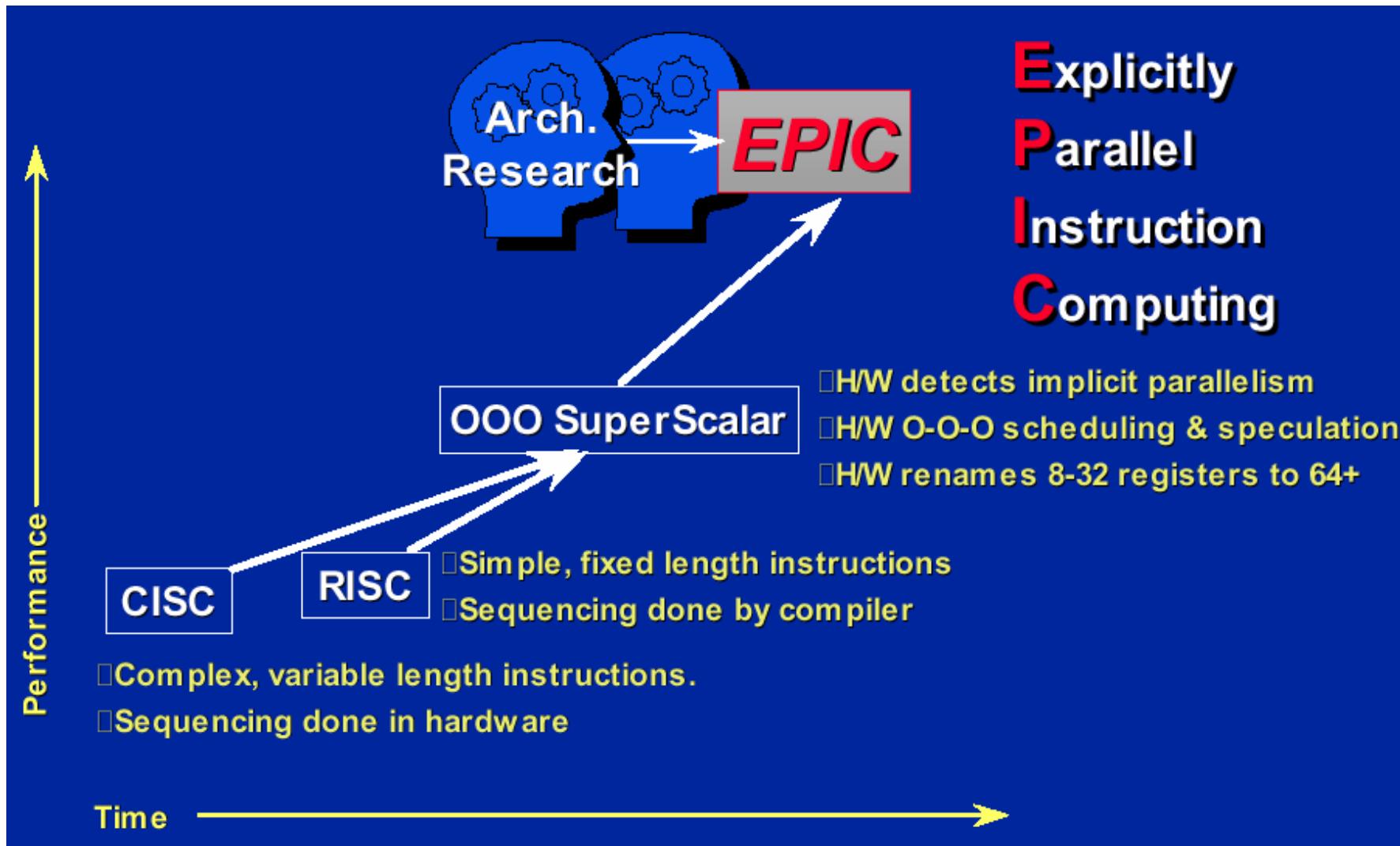


Figure 4-2. Protection Rings

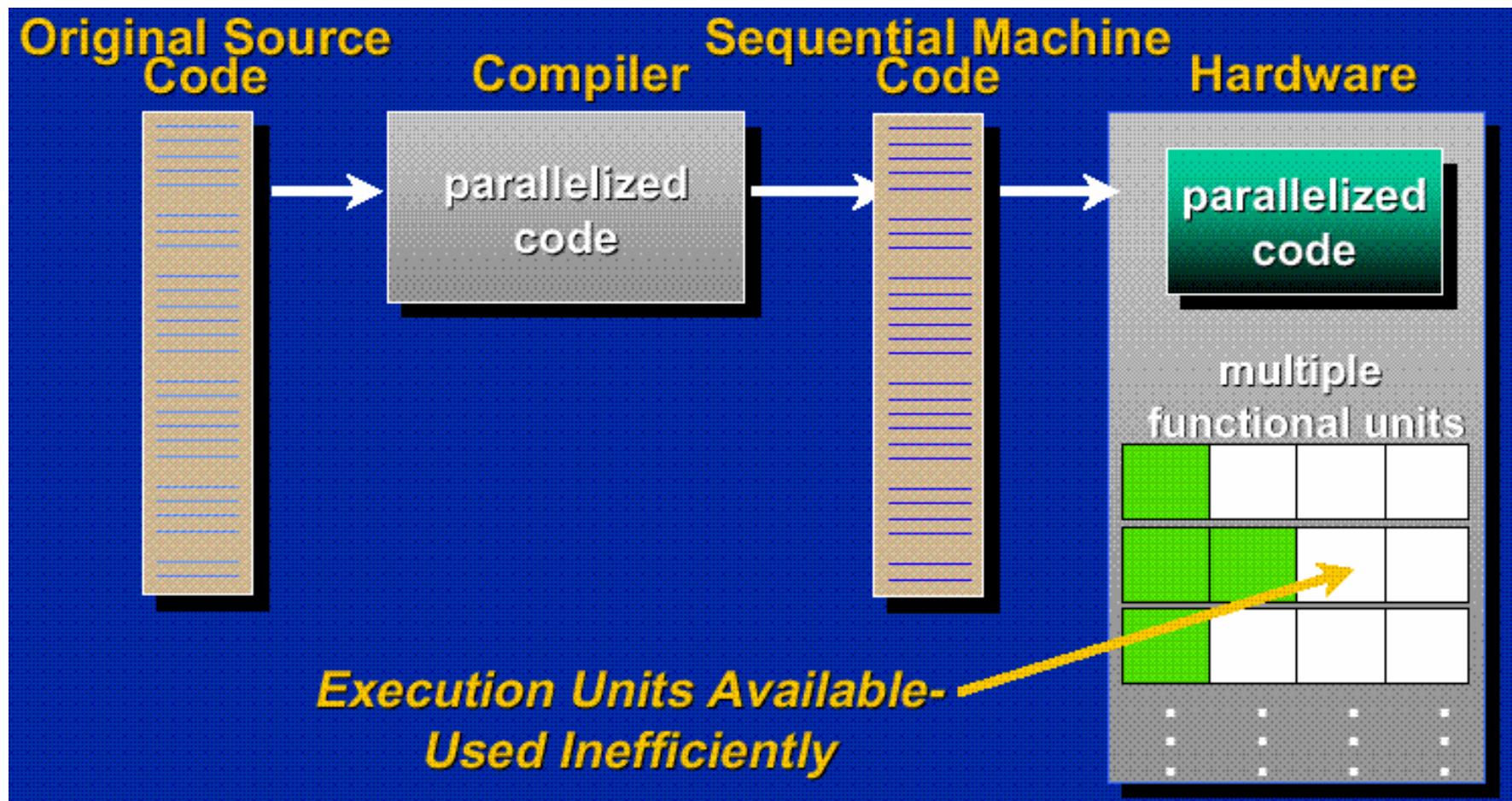
Wskaźniki P4



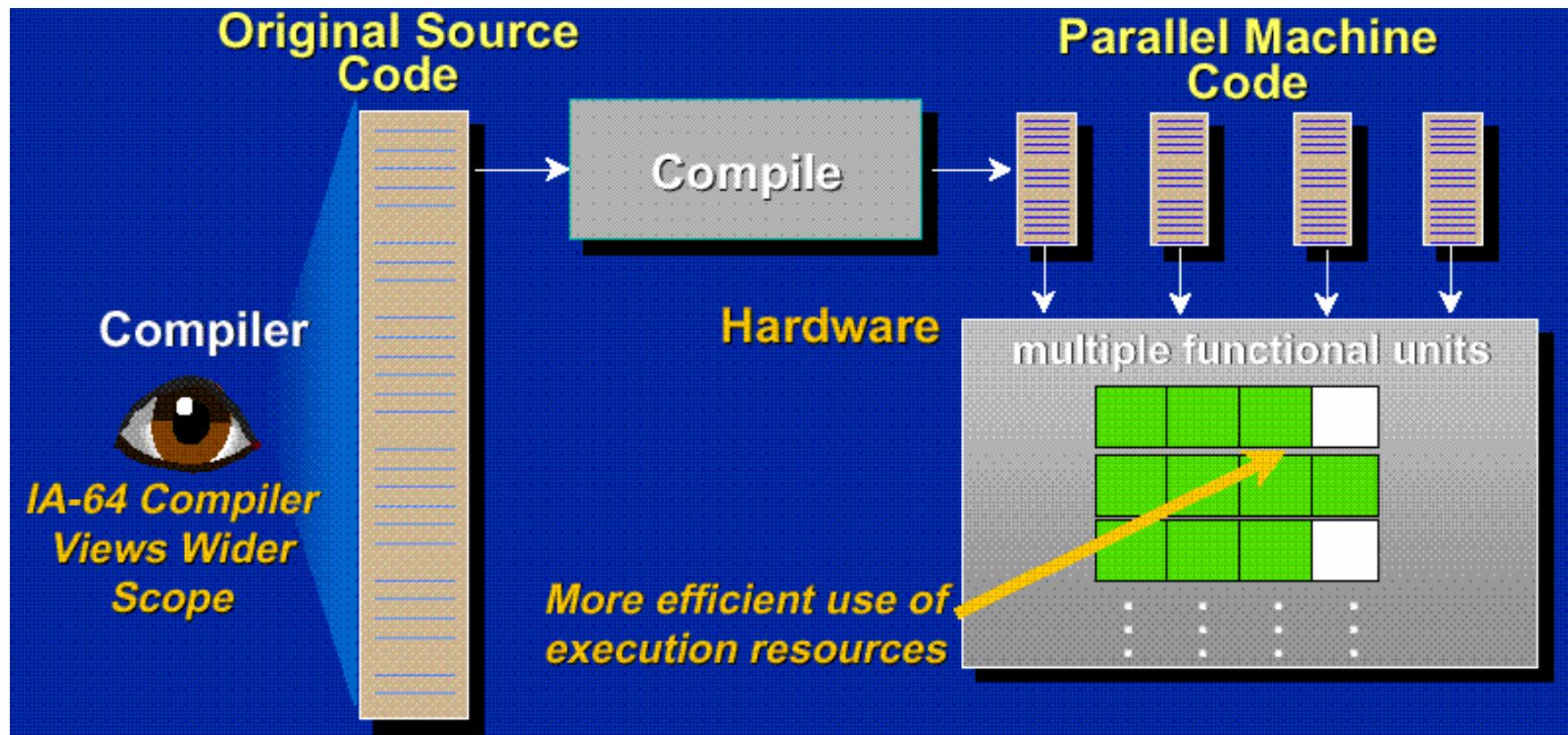
Koncepcja IA-64



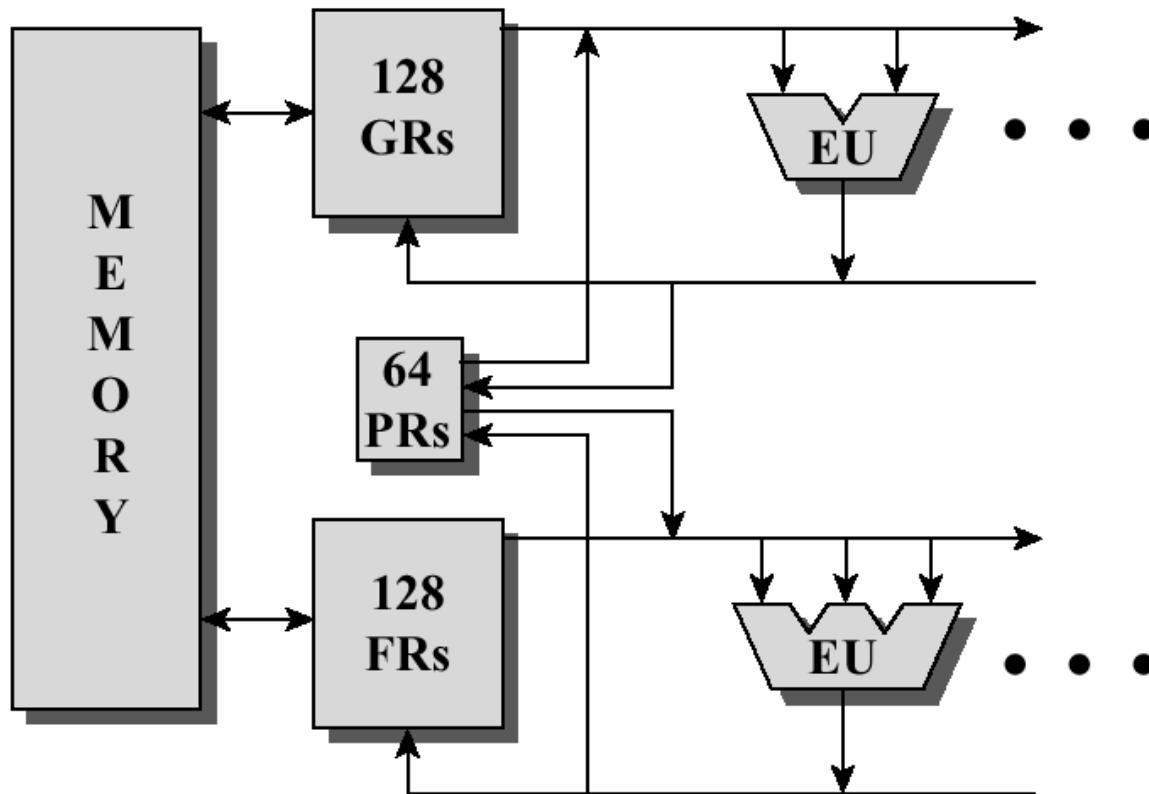
Koncepcja IA-64



Koncepcja IA-64 cd.



Architektura IA-64



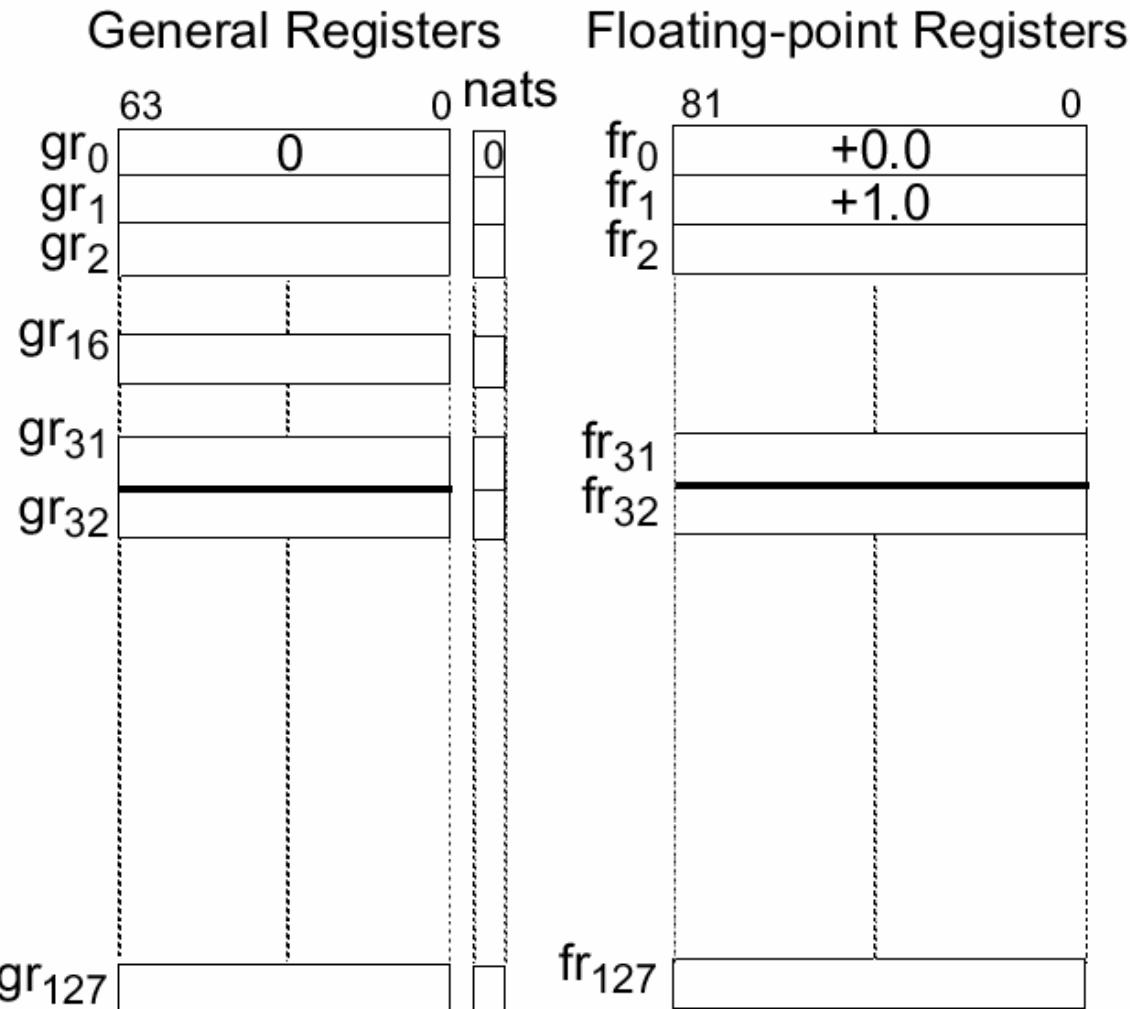
GR = General-purpose or integer register

FR = Floating-point or graphics register

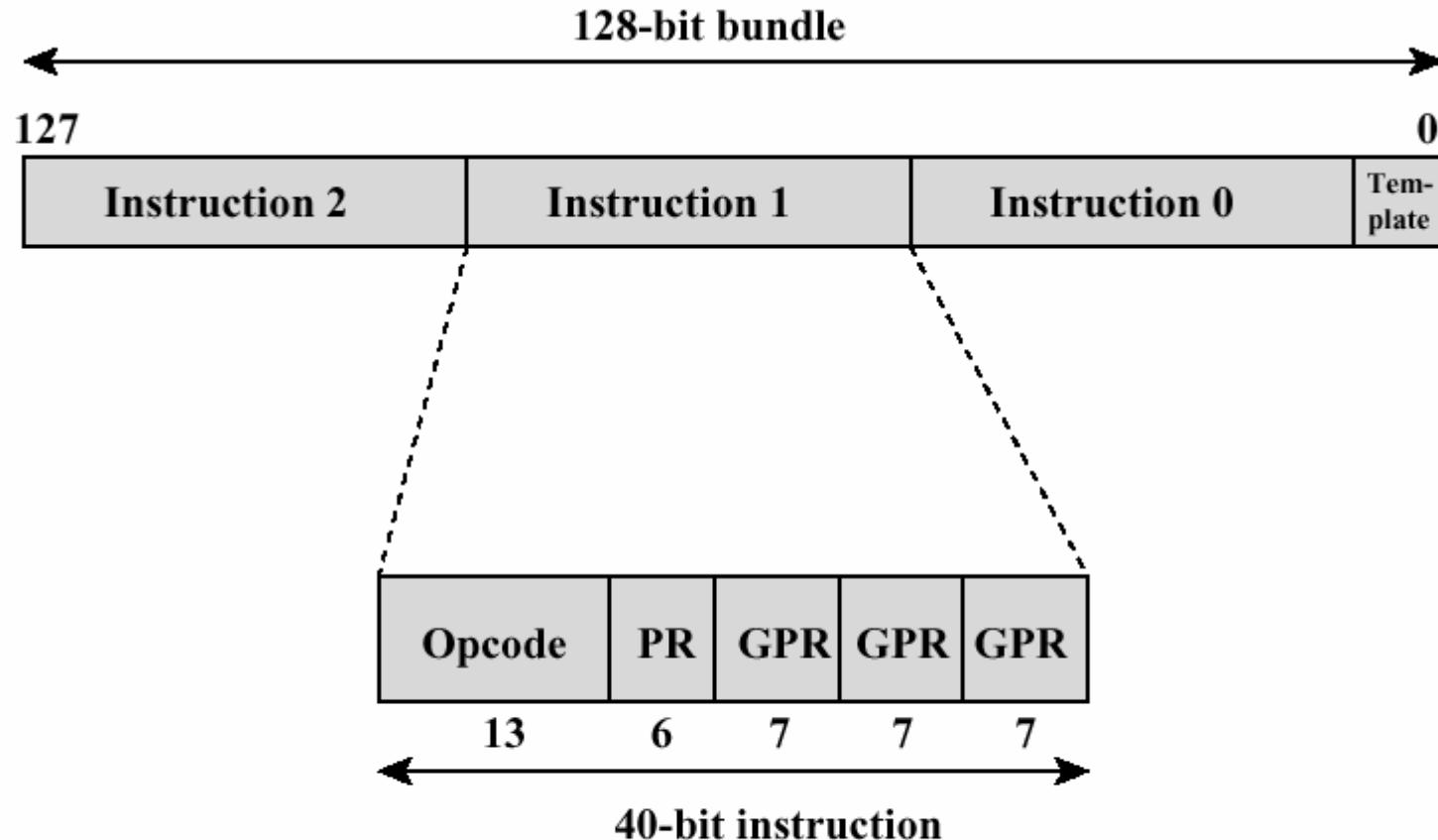
PR = One-bit predicate register

EU = Execution unit

Rejestry IA-64



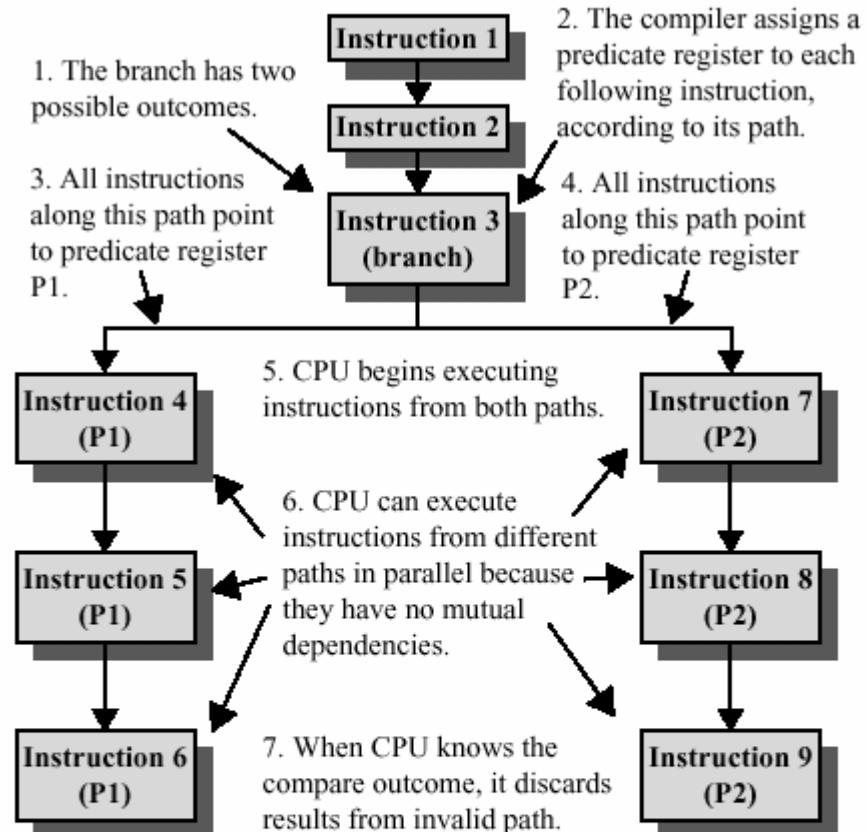
Format instrukcji IA-64



PR = Predicate register

GPR = General-purpose register (integer or floating-point)

Predykaty IA-64

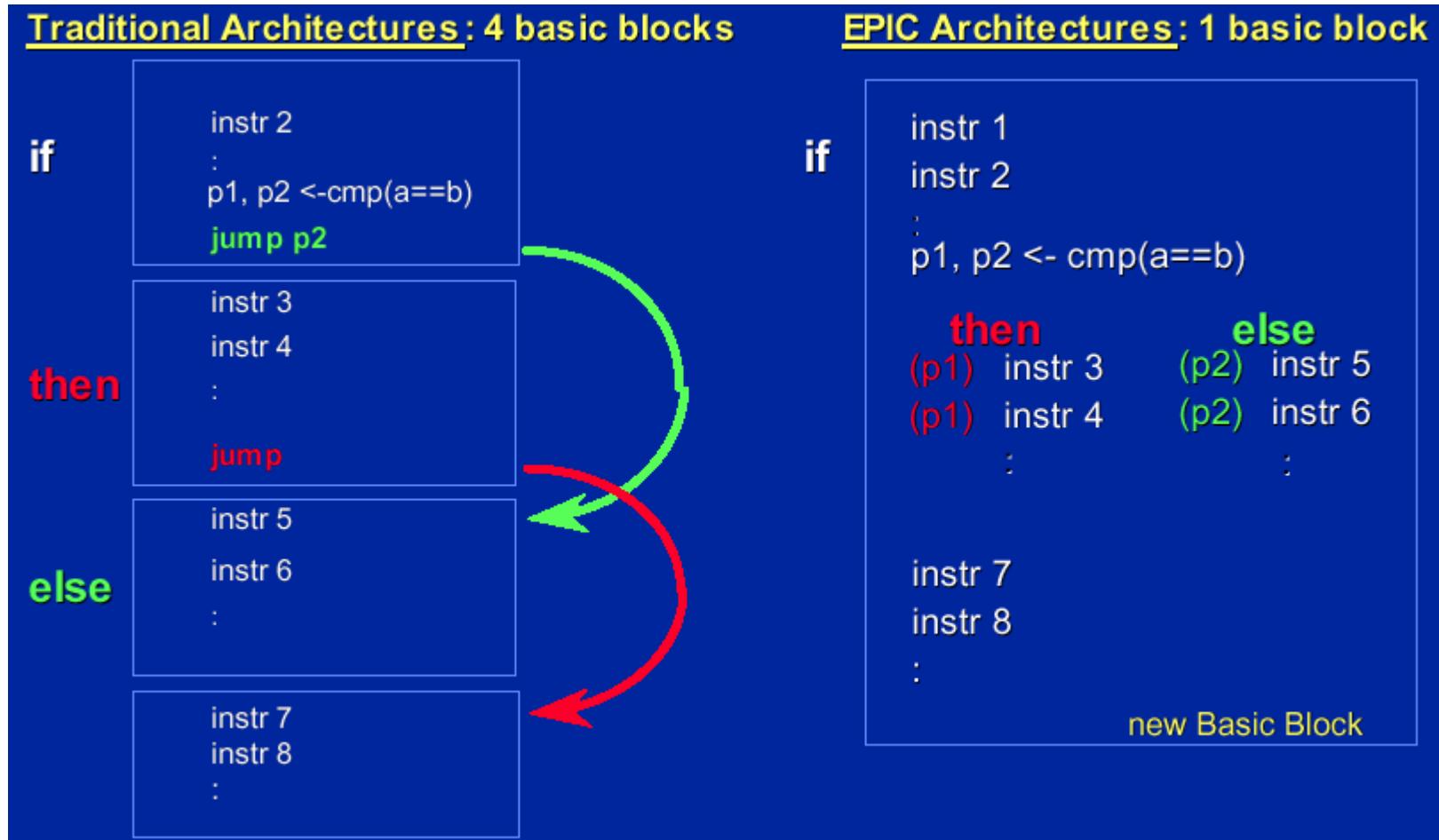


The compiler might rearrange instructions in this order, pairing instructions 4 and 7, 5 and 8, and 6 and 9 for parallel execution.

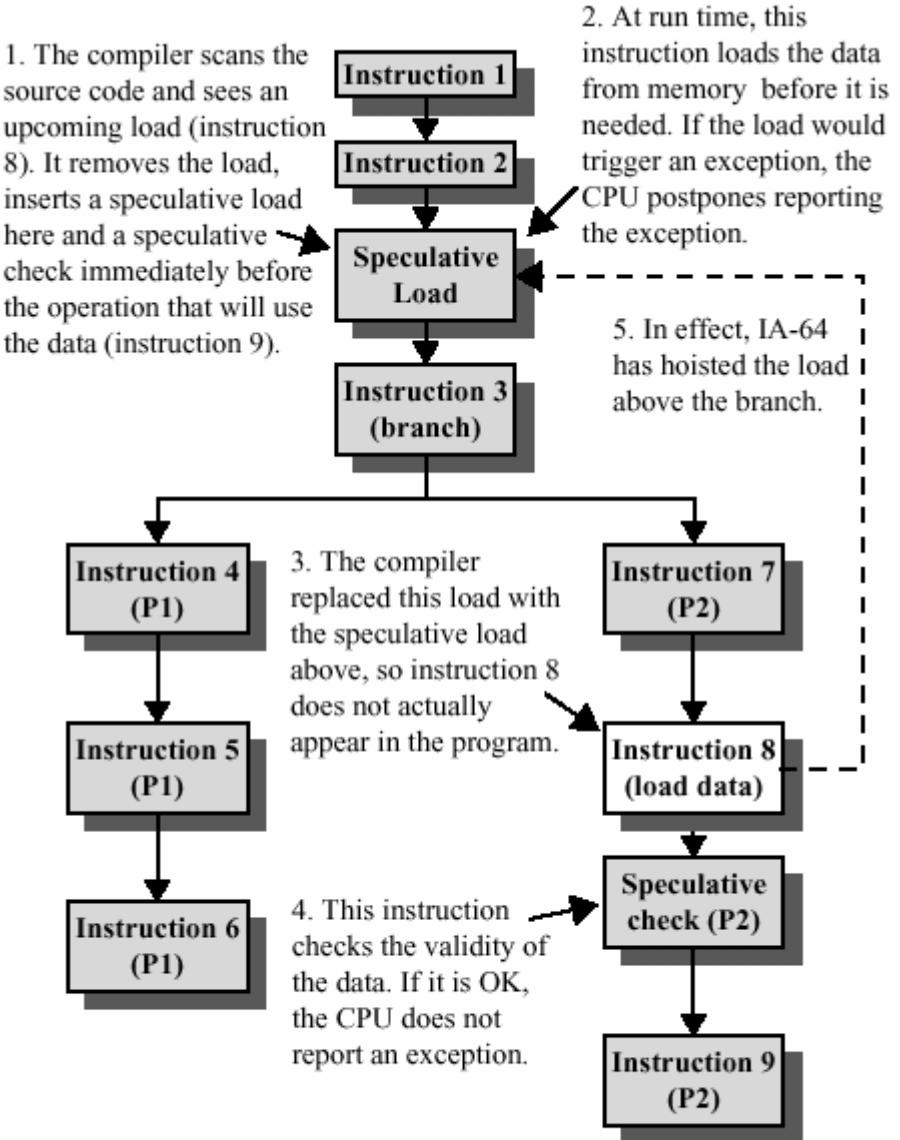
Instruction 1	Instruction 2	Instruction 3
Instruction 4	Instruction 7	Instruction 5
Instruction 8	Instruction 6	Instruction 9

(a) Predication

Predykaty IA-64



Spekulacja IA-64



(b) Speculative loading

Spekulacja IA-64

- Compiler can issue a load prior to a preceding, possibly-conflicting store

Traditional Architectures



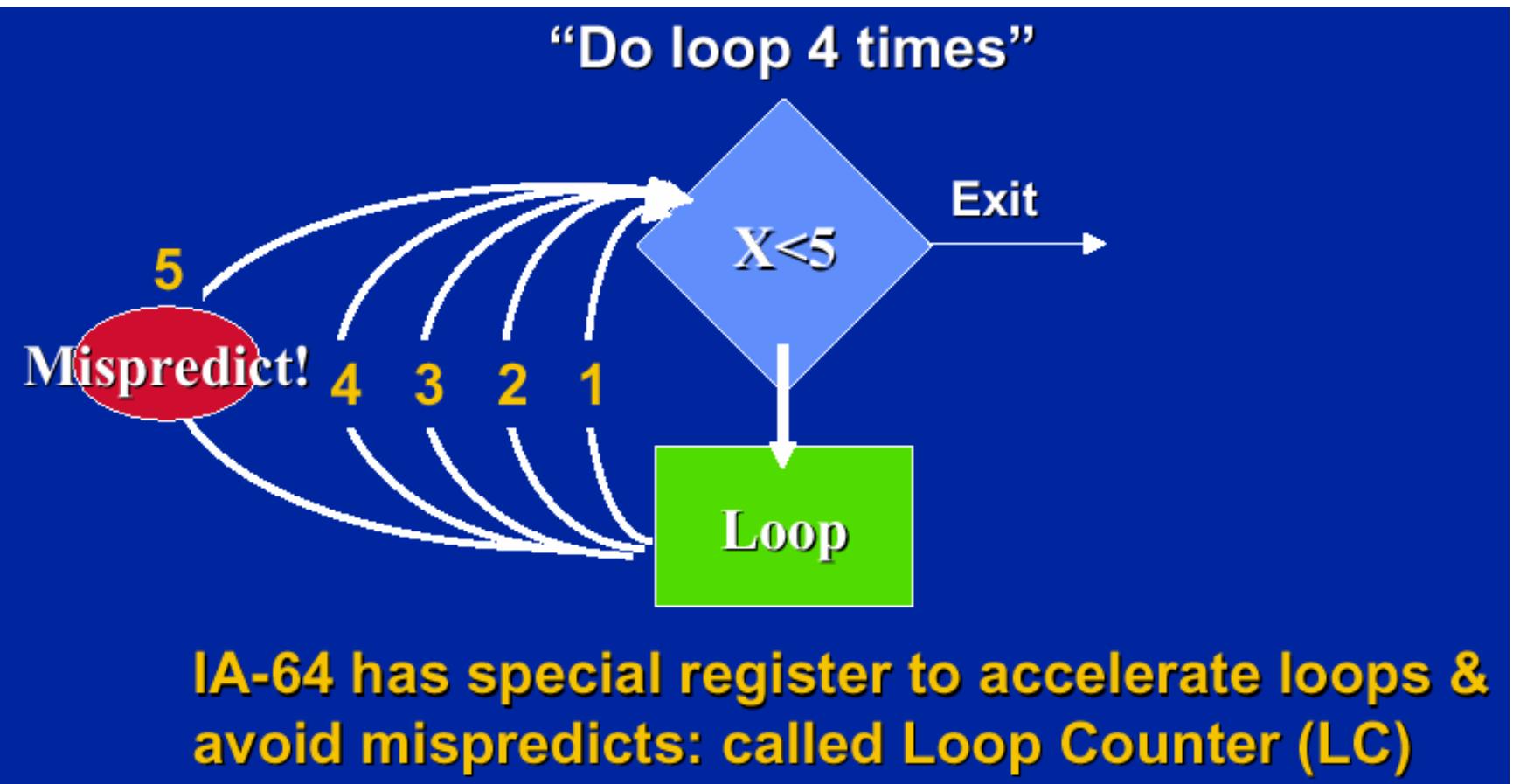
Memory latency
stalls CPU

IA-64

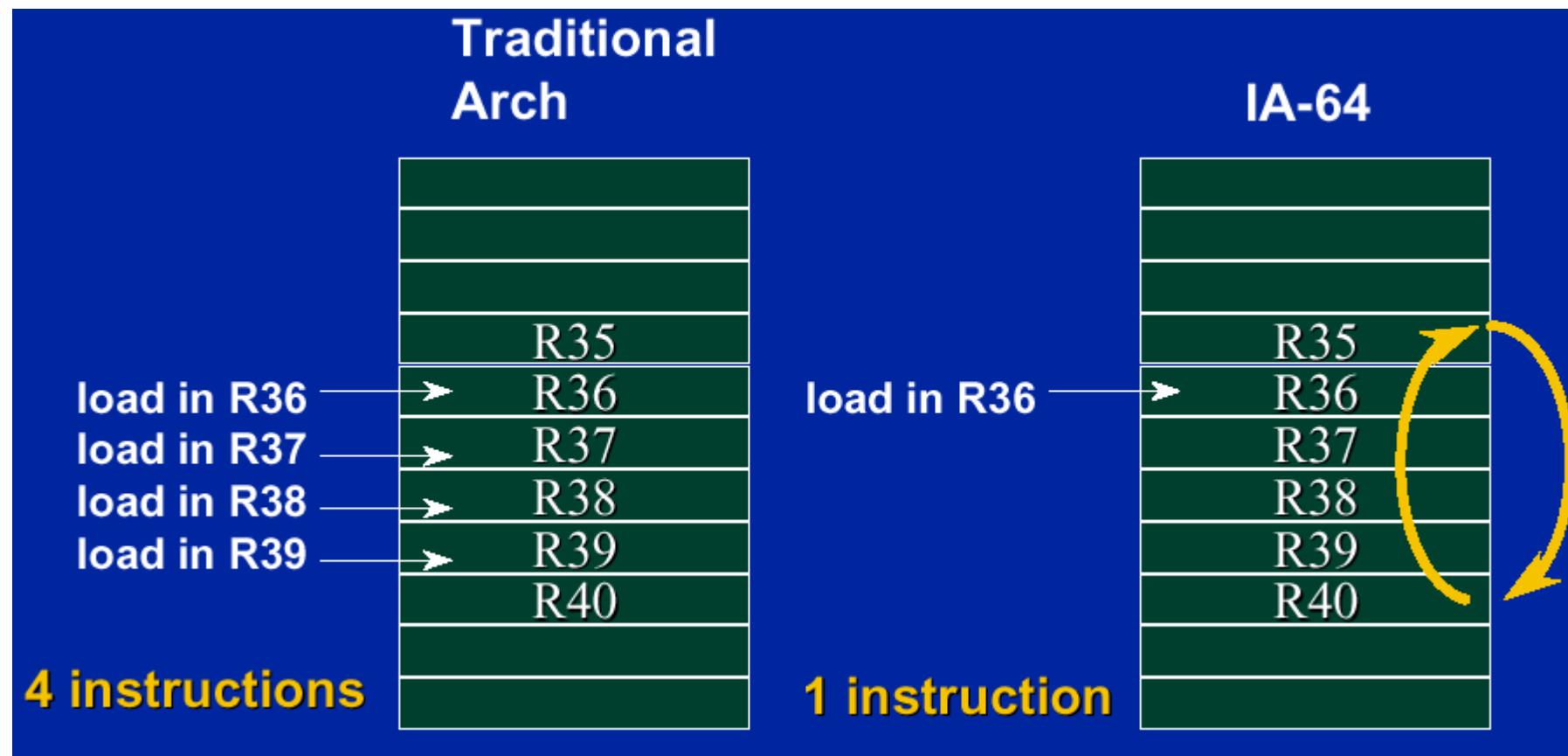


Elevates load above
“store barrier”

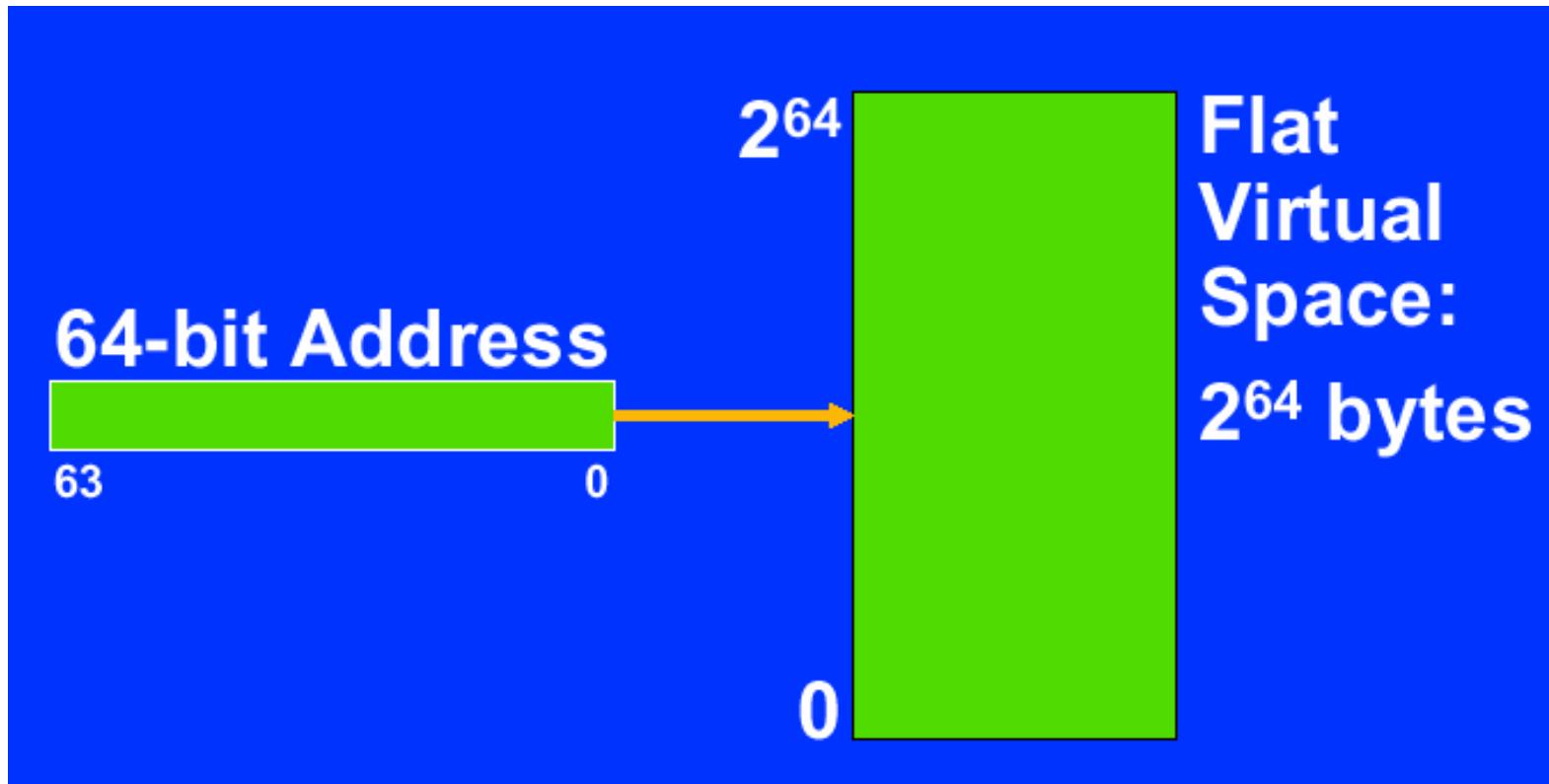
Licznik pętli IA-64



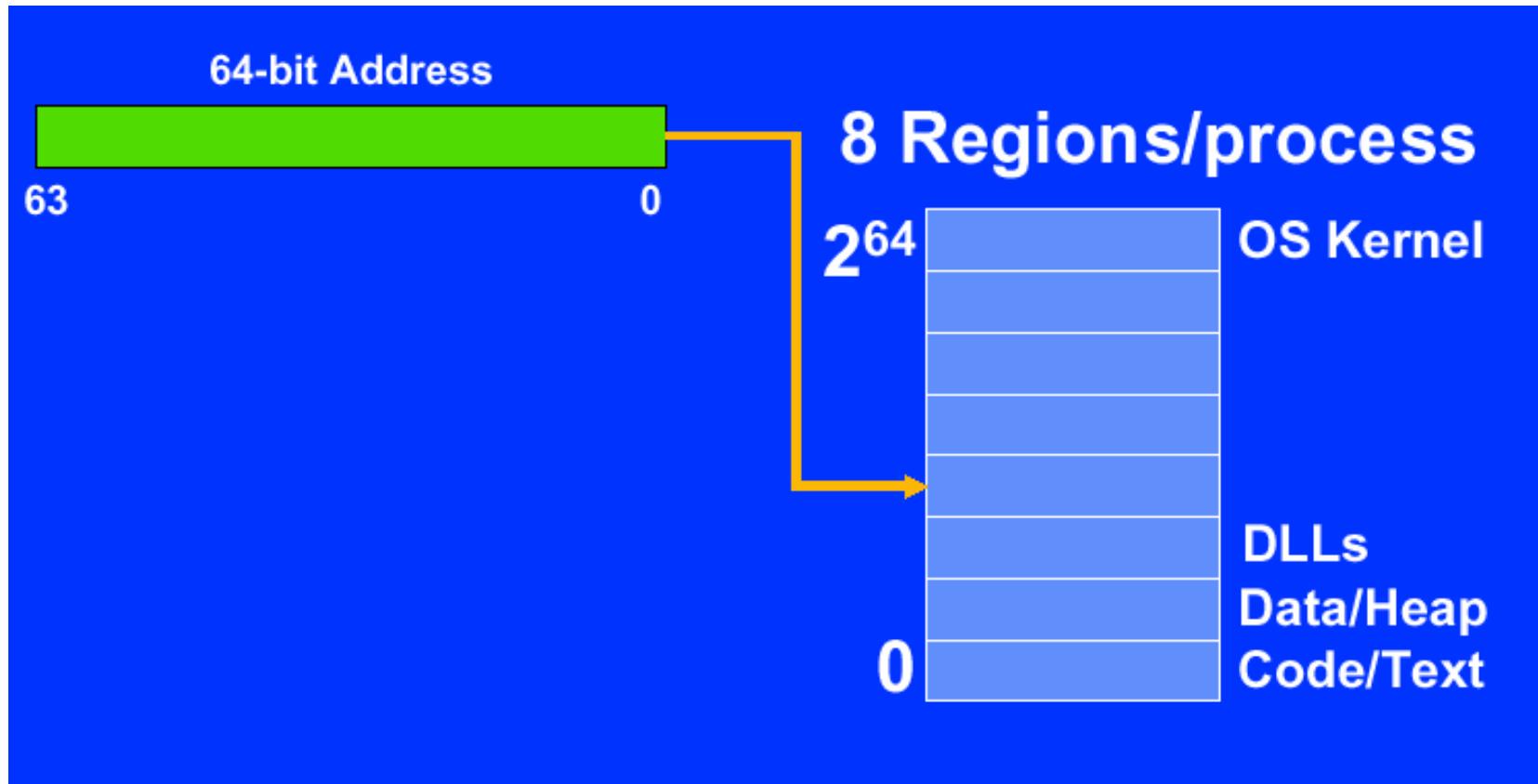
Obroty rejestrów IA-64



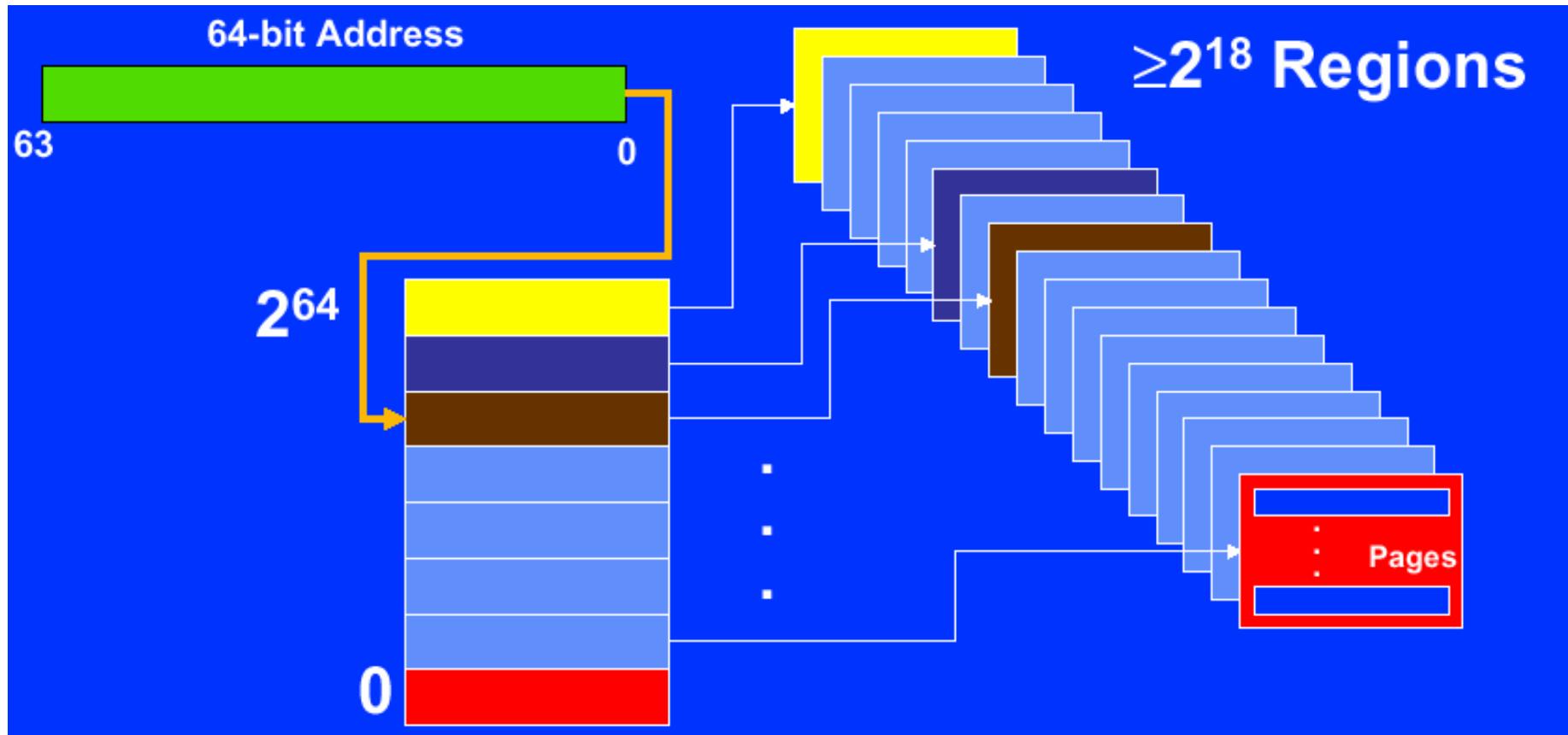
Pamięć IA-64 – proces



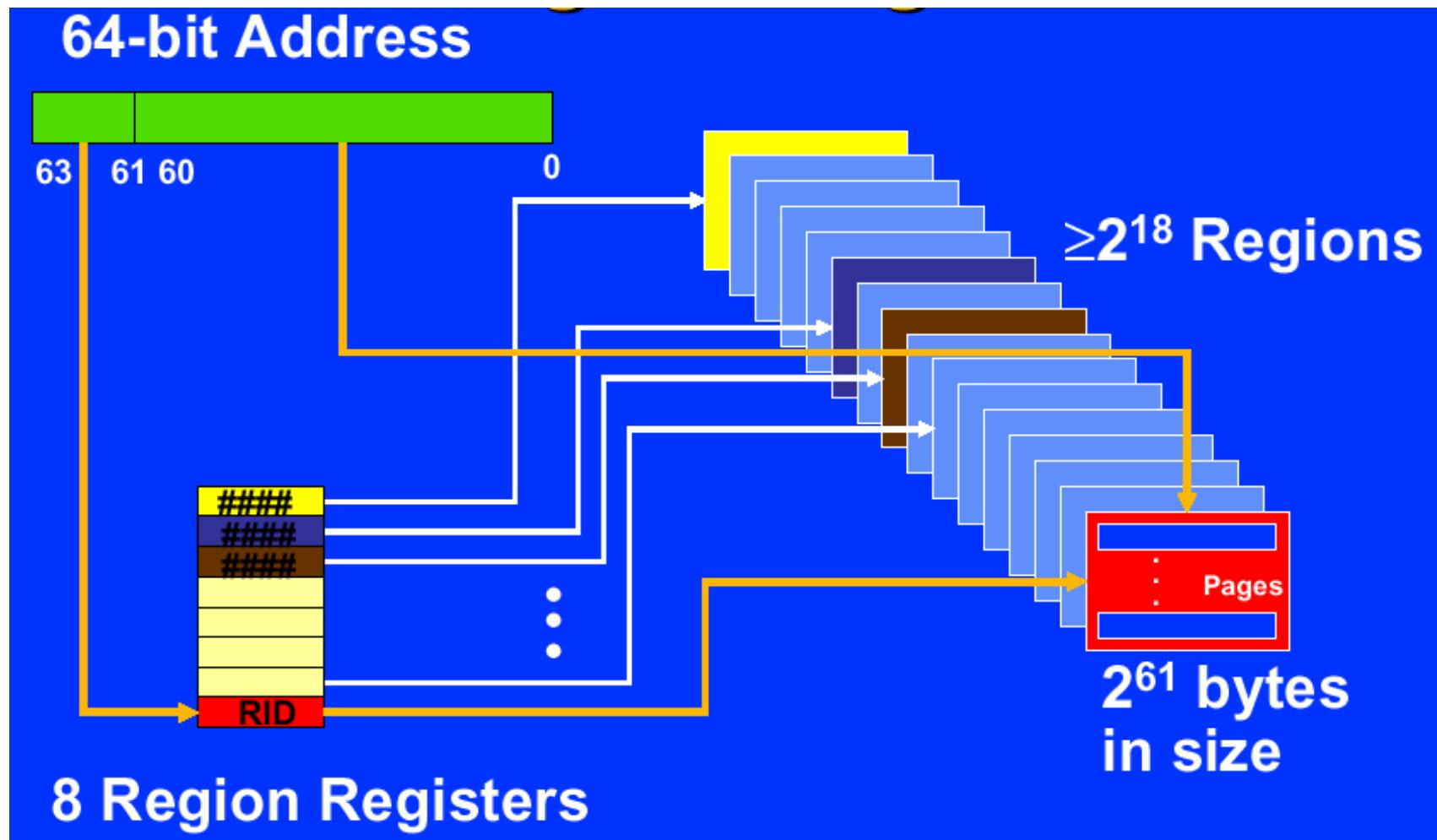
Pamięć IA-64 – proces



Pamięć IA-64 – system



Pamięć IA-64 – regiony



Architektura IA-64

- Itanium
 - opracowany w 2000 roku
 - około 10 mln tranzystorów

- Itanium 2
 - opracowany w 2002 roku
 - cache L3 on chip
 - 6 jednostek ALU
 - linijka L1 – 64 bajty

Podsumowanie

- Mikroarchitektura P6
- Architektura Pentium 4
 - NetBurst
 - trace cache
 - Hyper-Threading
- Modele pamięci
- Segmentacja pamięci
- Protekcja
- Architektura IA-64 (Itanium)
 - EPIC (wariant VLIW)
 - predykaty
 - spekulacja
 - organizacja pamięci (regiony)

Organizacja i Architektura Komputerów

Architektura listy instrukcji – ISA

Listy instrukcji

- Zbiór poleceń (instrukcji) zrozumiałych dla CPU
 - lista instrukcji powinna być funkcjonalnie pełna, tzn. umożliwiać zapis dowolnego algorytmu
- Instrukcje są zapisywane w postaci binarnej
 - każda instrukcja ma swój unikatowy kod binarny (kod operacji)
- W programach źródłowych zapisanych w języku asemblera zamiast kodów binarnych stosuje się zapis symboliczny operacji
 - np. `add`, `mov`

Elementy instrukcji

1. Kod operacji (op code), określa czynność wykonywaną przez instrukcję
2. Wskazanie na argument źródłowy (*source operand*)
3. Wskazanie na argument docelowy (*destination operand*)
4. Wskazanie na kolejną instrukcję programu, która ma być wykonana jako następna

Przykład: **mov ax, cx ;prześlij cx do ax**

Mnemonik (rodzaj operacji)

Argument docelowy

Argument źródłowy

Element 1) jest obligatoryjny, elementy 2) – 4) są opcjonalne

Reprezentacja instrukcji

- Postać symboliczna (język asemblera)

mov ax, cx

- Zapis funkcjonalny

ax ← cx

- Postać binarna w pamięci programu

10001011

11000001

w postaci binarnej kod operacji jest zapisywany jako pierwszy, dalej następują argumenty

Reprezentacja instrukcji cd.

Typowy format instrukcji z odniesieniami do dwóch argumentów

Kod operacji	Argument 1	Argument 2
Binarny kod operacji określający jednoznacznie rodzaj rozkazu; zdeterminowany przez mnemonik instrukcji (np. MOV, ADD itp.)	Odniesienie do argumentu nr. 1 <ul style="list-style-type: none">• jawnie (wprost) podana wartość argumentu lub• adres lub• inne wskazanie na argument	Odniesienie do argumentu nr. 2 <ul style="list-style-type: none">• jawnie (wprost) podana wartość argumentu lub• adres lub• inne wskazanie na argument

Liczba argumentów

- Istotnym czynnikiem przy projektowaniu ISA jest liczba argumentów (lub odniesień do argumentów) zawartych w instrukcji
 - wpływ na długość słowa instrukcji oraz złożoność CPU
 - duża liczba argumentów wymaga długiego słowa instrukcji
- Rozważmy, ile argumentów wymaga instrukcja dodawania ADD
 - jeśli chcemy dodać do siebie zawartość dwóch komórek pamięci potrzebujemy **dwóch** adresów
 - gdzie umieścić wynik dodawania? Potrzebny jest **trzeci** adres
 - która instrukcja w programie ma być wykonana jako następna?
 - zwykle domyślnie przyjmuje się, że będzie to kolejna instrukcja w programie, ale czasem chcemy wykonać skok w inne miejsce programu
 - do wskazania miejsca skoku potrzebny jest **czwarty** adres

Liczba argumentów cd.

- W praktyce procesory czteroadresowe nie są stosowane
 - zbyt skomplikowana budowa CPU
 - długie słowo instrukcji
 - nie zawsze instrukcja wymaga aż czterech adresów
- Większość instrukcji wymaga jednego, dwóch lub najwyżej trzech adresów
 - adres kolejnej instrukcji w programie jest określany przez inkrementację licznika rozkazów (z wyjątkiem instrukcji skoków)
- Rozważmy ciąg instrukcji hipotetycznego procesora wykonującego obliczenie wartości wyrażenia:

$$Y = (A - B) / (C + (D * E))$$

Przykład – procesor 3–adresowy

- Jeśli mamy procesor 3 – adresowy, możemy określić adresy dwóch argumentów i adres wyniku.
- Poniżej podano sekwencję instrukcji obliczającą wartość wyrażenia:

$$Y = (A - B) / (C + (D * E))$$

SUB R1, A, B	; Rejestr R1 $\leftarrow A - B$
MUL R2, D, E	; Rejestr R2 $\leftarrow D * E$
ADD R2, R2, C	; Rejestr R2 $\leftarrow R2 + C$
DIV R1, R1, R2	; Rejestr R1 $\leftarrow R1 / R2$

- Instrukcje 3-adresowe są wygodne w użyciu, bowiem można wskazać adres wyniku. Należy zauważyć, że w podanym przykładzie oryginalne zawartości komórek A, B, C, D i E nie zostały zmienione

Procesor 2–adresowy

- Jak zmniejszyć liczbę adresów?
 - dążymy do skrócenia słowa instrukcji i uproszczenia budowy CPU
- Typowa metoda polega na **domyślnym** założeniu odnośnie adresu wyniku operacji
 - wynik jest umieszczany w miejscu, skąd pobrano jeden z argumentów źródłowych
 - taka metoda oczywiście niszczy (zamazuje) pierwotną wartość argumentu
- Najczęściej przyjmuje się, że wynik jest umieszczany w miejscu, skąd pobrano pierwszy argument (np. Pentium)
 - pierwszy argument może się znajdować w rejestrze, komórce pamięci itp.

Przykład – procesor 2–adresowy

- Przy założeniu, że argumenty znajdują się w rejestrach, podana sekwencja instrukcji oblicza wartość wyrażenia:

$$Y = (A - B) / (C + (D * E))$$

SUB A, B	; Rejestr A $\leftarrow A - B$
MUL D, E	; Rejestr D $\leftarrow D * E$
ADD D, C	; Rejestr D $\leftarrow D + C$
DIV A, D	; Rejestr A $\leftarrow A / D$

- Otrzymujemy ten sam wynik co wcześniej, ale teraz zawartość rejestrów A i D uległa zmianie

Przykład – procesor 2–adresowy cd.

- Jeśli chcemy zachować oryginalną zawartość rejestrów, musimy skopiować ją do innych rejestrów:

MOV R1, A	; Kopiuj A do rejestru R1
MOV R2, D	; Kopiuj D do rejestru R2
SUB R1, B	; Rejestr R1 \leftarrow R1-B
MUL R2, E	; Rejestr R2 \leftarrow R2*E
ADD R2, C	; Rejestr R2 \leftarrow R2+C
DIV R1, R2	; Rejestr R1 \leftarrow R1 / R2

- Teraz wszystkie rejesty A-E zostają zachowane, ale kosztem kilku dodatkowych instrukcji

Procesor 1–adresowy

- W instrukcji jawnie wskazany jest tylko jeden argument
- Drugi argument znajduje się w **domyślnym** miejscu
 - zwykle w akumulatorze
 - akumulator w różnych językach asemblera ma różne oznaczenie, najczęściej A, AX, ACC
- Wynik jest umieszczany również w domyślnym miejscu, typowo w akumulatorze
 - wartość drugiego argumentu zostaje zniszczona (zamazana przez wynik operacji)

Przykład – procesor 1–adresowy

- Obliczamy wartość wyrażenia: $Y = (A-B) / (C + (D * E))$

LDA D	; Load ACC with D
MUL E	; Acc \leftarrow Acc * E
ADD C	; Acc \leftarrow Acc + C
STO R1	; Store Acc to R1
LDA A	; Acc \leftarrow A
SUB B	; Acc \leftarrow Acc - B
DIV R1	; Acc \leftarrow Acc / R1

- W wielu wczesnych komputerach stosowano procesory 1-adresowe, ponieważ mają one prostszą konstrukcję
- Jak widać z powyższego przykładu, architektura 1-adresowa wymaga jednak dłuższych programów

Procesor 0–adresowy

- W skrajnym przypadku można stosować instrukcje nie zawierające adresów
- Obliczenia są wykonywane na stosie
 - argumenty umieszcza się w odpowiedniej kolejności na stosie
 - operacje są wykonywane na szczytowych komórkach stosu
 - wynik umieszcza się na szczycie stosu
- Koncepcję przetwarzania 0-adresowego podał po raz pierwszy polski logik i filozof Jan Łukasiewicz (1878-1956)
 - notacja beznawiasowa
 - RPN – *Reverse Polish Notation*, odwrotna notacja polska
 - stosowana w niektórych komputerach i kalkulatorach (np. HP)
 - korzystając z metody RPN opracowano ciekawe języki programowania, np. FORTH

Przykład – procesor 0–adresowy

- Obliczamy wyrażenie: $Y = (A-B) / (C + (D * E))$

PUSH B	; B
PUSH A	; B, A
SUB	; (A-B)
PUSH E	; (A-B), E
PUSH D	; (A-B), E, D
MUL	; (A-B), (E*D)
PUSH C	; (A-B), (E*D), C
ADD	; (A-B), (E*D+C)
DIV	; (A-B) / (E*D+C)

Liczba adresów – wnioski

- Więcej adresów
 - bardziej skomplikowane, ale też wygodne w użyciu instrukcje
 - dłuższe słowo instrukcji
 - większa liczba rejestrów
 - operacje na rejestrach są szybsze niż operacje na zawartości pamięci
 - krótsze programy (w sensie liczby instrukcji)
- Mniej adresów
 - mniej skomplikowane instrukcje
 - dłuższe programy (w sensie liczby instrukcji)
 - krótszy cykl pobrania instrukcji
- Większość współczesnych procesorów stosuje architekturę 2-adresową lub 3-adresową
- Szczególne koncepcje ISA (RISC, VLIW, EPIC)

Rodzaje argumentów

- Adresy (addresses)
- Liczby (numbers)
 - Integer/floating point
- Znaki (characters)
 - ASCII
- Dane logiczne (logical data)
 - Bits or flags

Format danych – Pentium

- 8 bit byte
- 16 bit word
- 32 bit double word
- 64 bit quad word
- adresowanie pamięci odbywa się bajtami (każdy kolejny bajt w pamięci ma swój adres)
- dane 32-bitowe (double word) muszą być ulokowane począwszy od adresu podzielnego przez 4

Interpretacja (kodowanie) danych

- General – dowolna liczba binarna
- Integer – liczba binarna ze znakiem (U2)
- Ordinal – liczba całkowita bez znaku (NKB)
- Unpacked BCD – jedna cyfra BCD na bajt
- Packed BCD – 2 cyfry BCD na bajt
- Near Pointer – 32-bitowy offset w segmencie
- Bit field – pole bitowe, ciąg bitów
- Byte String – ciąg bajtów
- Floating Point – liczba zmiennoprzecinkowa

Typy instrukcji

- Data Transfer – przesyłanie danych (np. MOV)
- Arithmetic – operacje arytmetyczne (np. ADD)
- Logical – operacje logiczne (np. OR)
- Conversion – konwersja danych, np. liczb binarnych na BCD
- I/O – operacje wejścia/wyjścia
- System Control
 - specjalne operacje systemowe, np. sterowanie stanem procesora, zarządzanie protekcją itp.
- Transfer of Control (np. JMP)
 - skoki, wywołanie podprogramów itp..

Tryby adresowania

- Sposoby wskazywania na argumenty instrukcji
- Podstawowe tryby adresowania używane we współczesnych procesorach:
 - Register – rejestrowy, wewnętrzny
 - Immediate – natychmiastowy
 - Direct – bezpośredni
 - Register indirect – rejestrowy pośredni
 - Base + Index – bazowo-indeksowy
 - Register relative – względny
 - Stack addressing – adresowanie stosu

Adresowanie rejestrowe

- Argumenty operacji znajdują się w rejestrach procesora
- Przykład:

```
    mov  al,bl      ;prześlij bl do al
    inc  bx          ;zwiększ bx o 1
    dec  al          ;zmniejsz al o 1
    sub  dx,cx      ;odejmij cx od dx
```

- Ponieważ procesory zawierają niewiele rejestrów GP, wystarczy tylko kilka bitów do wskazania rejestrów w instrukcji; ich kody są zwykle umieszczone w kodzie instrukcji
- Zalety: krótki kod binarny instrukcji, szybkie wykonanie

Adresowanie rejestrowe cd.

- W procesorach RISC stosuje się zwykle zasadę pełnej symetrii, która oznacza, że instrukcje mogą operować na dowolnych rejestrach
- W procesorach CISC niektóre kombinacje instrukcji i rejestrów nie są dozwolone:

Przykład (x86):

```
mov cs,ds      ;nie można przesyłać  
                 ;rejestrów segmentów  
mov bl,bx      ;rozmiary rejestrów  
                 ;muszą być takie same  
                 ;bl - 8 bitów, bx - 16 bitów
```

Adresowanie natychmiastowe

- Argument operacji znajduje się w postaci jawnej w instrukcji, zaraz po kodzie operacji
- Przykłady:

```
mov bl,44      ;prześlij 44 (dziesiętnie) do bl  
mov ah,44h    ;prześlij 44 (hex) do ah  
mov di,2ab4h  ;prześlij 2ab4 (hex) do di  
add ax,867    ;dodaj 867 (dec) do ax
```

- Argument natychmiastowy jest pobierany przez procesor razem z kodem operacji, dlatego nie jest potrzebny osobny cykl pobrania argumentu (*data fetch*)
- Tryb natychmiastowy można stosować tylko w odniesieniu do stałych, których wartości są znane w trakcie pisania programu

Adresowanie bezpośrednie

- Instrukcje stosujące adresowanie bezpośredni odwołują się do argumentów umieszczonych w pamięci
- Adres argumentu w architekturze x86 składa się z dwóch elementów: selektora segmentu i ofsetu (offset) i jest zapisywany w postaci:

segment:offset

- Przykład

1045h:a4c7h

oznacza komórkę o adresie **a4c7h** w segmencie **1045h**

- Ofset bywa nazywany również przesunięciem (ang. displacement, w skrócie disp)

Adresowanie bezpośrednie cd.

- Zwykle w instrukcjach podaje się tylko offset, natomiast wskazanie na segment znajduje się w domyślnym rejestrze
 - Instrukcje operujące na danych w pamięci używają domyślnie rejestru DS.
 - Instrukcje operujące na stosie używają domyślnie rejestru SS
 - Instrukcje skoków używają domyślnie segmentu CS
- Jeśli jest taka potrzeba, można zastosować inny register segmentu niż domyślny, wtedy adres trzeba poprzedzić tzw. prefiksem podającym register segmentu

Przykład: `mov ax, [di]` ;domyślny adres `ds:di`
`mov ax, es:[di]` ;adres `es:di`

Adresowanie bezpośrednie cd.

- Obliczanie adresu efektywnego (EA) na podstawie wskazania segmentu i offsetu odbywa się na dwa różne sposoby, w zależności od zastosowanego modelu pamięci.
- Wyróżnia się dwa podstawowe rodzaje segmentacji:
 - Standardową (*segmented mode*) stosowaną powszechnie przez aplikacje 32-bitowe IA-32
 - Tryb adresów rzeczywistych (*real-address mode*) stosowaną w starych aplikacjach 16-bitowych

Adresowanie bezpośrednie cd.

- W trybie segmentacji standardowej obliczanie adresu efektywnego na przebiega następująco:
 - 16-bitowy rejestr segmentu zawiera selektor segmentu, który wskazuje pewien deskryptor segmentu w tablicy deskryptorów segmentów
 - Z deskryptora segmentu pobiera się 32-bitowy adres bazowy segmentu
 - Adres efektywny oblicza się jako sumę 32-bitowego adresu bazowego i 32-bitowego offsetu podanego w instrukcji
- Przykład
 - w trakcie wykładu

Adresowanie bezpośrednie cd.

- W trybie adresów rzeczywistych obliczanie adresu efektywnego na przebiega następująco:
 - 16-bitowy rejestr segmentu po pomnożeniu przez 16 daje bezpośrednio adres bazowy segmentu (nie stosuje się deskryptorów segmentów)
 - Adres efektywny oblicza się jako sumę adresu bazowego i 16-bitowego offsetu
- Adres ds:offset jest zatem przekształcany na adres efektywny według zależności:
$$EA = 16*ds + offset$$
- Przykład
 - w trakcie wykładu

Adresowanie bezpośrednie cd.

- Argument jest umieszczony w komórce pamięci; adres tej komórki jest podany w instrukcji
- Przykłady:

```
mov al,DANA ;prześlij bajt z komórki  
;pamięci o adresie DS:DANA  
;do rejestru al.  
mov ax,news ;prześlij ax do adresu DS:NEWS
```

Uwaga: x86 stosuje konwencję *little-endian*, dlatego jeśli założymy, że offset **DANA=1002h**, pierwsza z powyższych przykładowych instrukcji będzie w pamięci zapisana jako ciąg bajtów:

kod operacji
02
10

Adresowanie bezpośrednie cd.

- Zamiast offsetu w postaci nazwy symbolicznej można podać offset w postaci liczbowej (dziesiętnie lub szesnastkowo):
Przykład: `mov al, [1234h]`
- Nawias użyty w powyższym przykładzie wskazuje kompilatorowi, że zastosowano adresowanie bezpośredni
- Jeżeli nawias został pominięty, kompilator zinterpretowałby argument jako natychmiastowy
 - w powyższym przykładzie liczba 1234h zostałaby potraktowana jako gotowy argument natychmiastowy, a nie offset adresu argumentu znajdującego się w pamięci

Adresowanie pośrednie rejestrowe

- Adresowanie pośrednie rejestrowe działa podobnie jak adresowanie bezpośrednie, z tą różnicą, że offset jest podany w rejestrze, a nie bezpośrednio w instrukcji
 - żeby uzyskać offset trzeba zatem wykonać dodatkowy, pośredni krok polegający na sięgnięciu do zawartości rejestru
- W procesorach x86 do adresowania rejestrowego pośredniego można wykorzystywać rejesty: **bx**, **bp**, **si**, **di**
- Należy pamiętać, że rejesty **bx**, **si**, **di** domyślnie współpracują z rejestrem segmentu **ds**, natomiast rejestr **bp** współpracuje domyślnie z rejestrem segmentu **ss**

Adresowanie pośrednie rejestrowe

- Przykłady:

mov cx, [bx]

- 16-bitowe słowo z komórki adresowanej przez offset w rejestrze **bx** w segmencie danych (**ds**) jest kopiowane do rejestru

mov [bp], dl

- bajt z **dl** jest kopiowany do segmentu stosu pod adres podany w rejestrze **bp**

mov [di], [bx]

- przesłania typu pamięć-pamięć nie są dozwolone (z wyjątkiem szczególnych instrukcji operujących na łańcuchach znaków – stringach)

Adresowanie bazowo-indeksowe

- Podobne do adresowania rejestrowego pośredniego
- Różnica polega na tym, że offset jest obliczany jako suma zawartości dwóch rejestrów
- Przykład: **mov dx, [bx+di]**
- Adresowanie bazowo-indeksowe jest przydatne przy dostępie do tablic
 - Rejestr bazowy wskazuje adres początku tablicy
 - Rejestr indeksowy wskazuje określony element tablicy
- Jako rejestr bazowy można wykorzystać rejesty **bp** lub **bx** (uwaga: **bx** współpracuje z **ds**, natomiast **bp** współpracuje z **ss**)
- Jako rejesty indeksowe można wykorzystać rejesty **di** lub **si**

Adresowanie rejestrowe względne

- Podobne do adresowania bazowo-indeksowego
- Offset jest obliczany jako suma zawartości rejestru bazowego lub indeksowego (**bp**, **bx**, **di** lub **si**) oraz liczby określającej bazę
- Przykład: **mov ax, [di+100h]**
powyższa instrukcja przesyła do **ax** słowo z komórki pamięci w segmencie **ds** o adresie równym sumie zawartości rejestru **di** i liczby **100h**
- Podobnie jak w innych trybach adresowych, rejesty **bx**, **di** oraz **si** współpracują domyślnie z rejestrem segmentu danych **ds**, natomiast rejestr **bp** współpracuje z rejestrem segmentu stosu **ss**

Adresowanie rejestrowe względne cd.

- Adresowanie rejestrowe względne jest alternatywnym (w stosunku do adresowania bazowo-indeksowego) sposobem dostępu do elementów tablic
- Przykłady:

`mov array[si],bl`

bajt z rejestru **bl** jest zapamiętyany w segmencie danych, w tablicy o adresie początkowym (podanym przez nazwę symboliczną) **array**, w elemencie o numerze podanym w **si**

`mov di, set[bx]`

rejestr **di** jest załadowany słowem 16-bitowym z segmentu danych, z tablicy **set**, z elementu o numerze podanym w **bx**

Adr. względne bazowo-indeksowe

- Adresowanie względne bazowo-indeksowe jest kombinacją dwóch trybów:
 - bazowo-indeksowego
 - rejestrówego względnego
- Przykłady:

mov dh, [bx+di+20h]

rejestr **dh** jest ładowany z segmentu danych, z komórki o adresie równym sumie **bx+di+20h**

mov ax, file [bx+di]

rejestr **ax** jest ładowany z segmentu danych, z komórki o adresie równym sumie adresu początkowego tablicy **file**, rejestrów **bx** oraz rejestrów **di**

mov list [bp+di], cl

rejestr **cl** jest zapamiętywany w segmencie stosu, w komórce o adresie równym sumie adresu początkowego tablicy **list**, rejestrów **bp** oraz rejestrów **di**

Adresowanie stosu

- Jest używane przy dostępie do stosu
- Przykłady

popf	;pobierz słowo wskaźników (flagi) ze stosu
pushf	;wyślij słowo wskaźników na stos
push ax	;wyślij ax na stos
pop bx	;odczytaj bx ze stosu
push ds	;wyślij rejestr segmentu ds na stos
pop cs	;odczytaj rejestr segmentu ze stosu
push [bx]	;zapisz komórkę pamięci o adresie podanym ;w bx na stos

Adresowanie danych – uwaga

- We wszystkich przykładach z kilkunastu ostatnich plansz ilustrujących tryby adresowania argumentów użyto dla uproszczenia wyłącznie rejestrów 8-bitowych (np. **a1**, **ah**, **b1**) lub 16-bitowych (np. **ax**, **bx**, **cx**)
- W ogólnym przypadku, w architekturze IA-32 można również używać rejestrów 32-bitowych. Wówczas w podanych przykładach mogłyby wystąpić rejesty **eax**, **ebx** itd.

Adresowanie w instrukcjach skoku

- Adres w instrukcjach skoków może mieć różną postać:
 - krótką (short)
 - bliską (near)
 - daleką (far)

- Adresy mogą być podane w różny sposób:
 - Wprost w części adresowej instrukcji – tryb bezpośredni (direct)
 - W rejestrze
 - W komórce pamięci adresowanej przez rejestr

Skoki krótkie (short)

- Adres jest jednobajtową liczbą w kodzie U2
- Skok polega na dodaniu tej liczby do rejestru IP
- Krótki skok ma zakres od –128 do +127 bajtów w stosunku do aktualnej wartości IP i odbywa się w obrębie tego samego segmentu wskazanego przez **cs**
- Przykład:

```
label2      jmp  short label1  
            ....  
label1      jmp  label2
```

Uwaga: w pierwszej instrukcji skoku trzeba użyć słowa **short** aby wskazać kompilatorowi, że skok jest krótki, ponieważ podczas kompilacji w tym miejscu nie wiadomo jeszcze jaka jest jego długość. W drugiej instrukcji skoku słowo **short** jest zbędne.

Kompilator sam zoptymalizuje kod i zastosuje skok krótki, ponieważ znana jest już odległość do etykiety **label2**

Skoki bliskie (near)

- Skok może być wykonany do dowolnej instrukcji w obrębie segmentu wskazanego przez aktualną wartość w rejestrze **cs**
- W trybie adresów rzeczywistych adres skoku jest 16-bitowy, natomiast segment ma rozmiar 64 kB
- W trybie segmentowym IA-32 adres jest 32-bitowy
- Ogólna postać skoku:
jmp near ptr etykieta_docelowa
- Przykład:

jmp near ptr dalej

. . . .

. . . .

dalej . . .

Skoki dalekie (far)

- Skok może być wykonany do dowolnej instrukcji w obrębie dowolnego innego segmentu
- W rozkazie skoku należy podać nie tylko adres skoku tak jak w przypadku skoków bliskich, ale ponadto również nową wartość rejestru segmentu **cs**
- Ogólna postać skoku:
jmp far ptr etykieta_docelowa
etykieta docelowa znajduje się w innym segmencie kodu programu niż instrukcja skoku
- Elementy instrukcji skoku dalekiego są w zakodowanej instrukcji przechowywane w następującej kolejności: kod operacji skoku, offset (adres skoku), nowa zawartość **cs**. Offset i cs są przechowywane w postaci *little-endian*

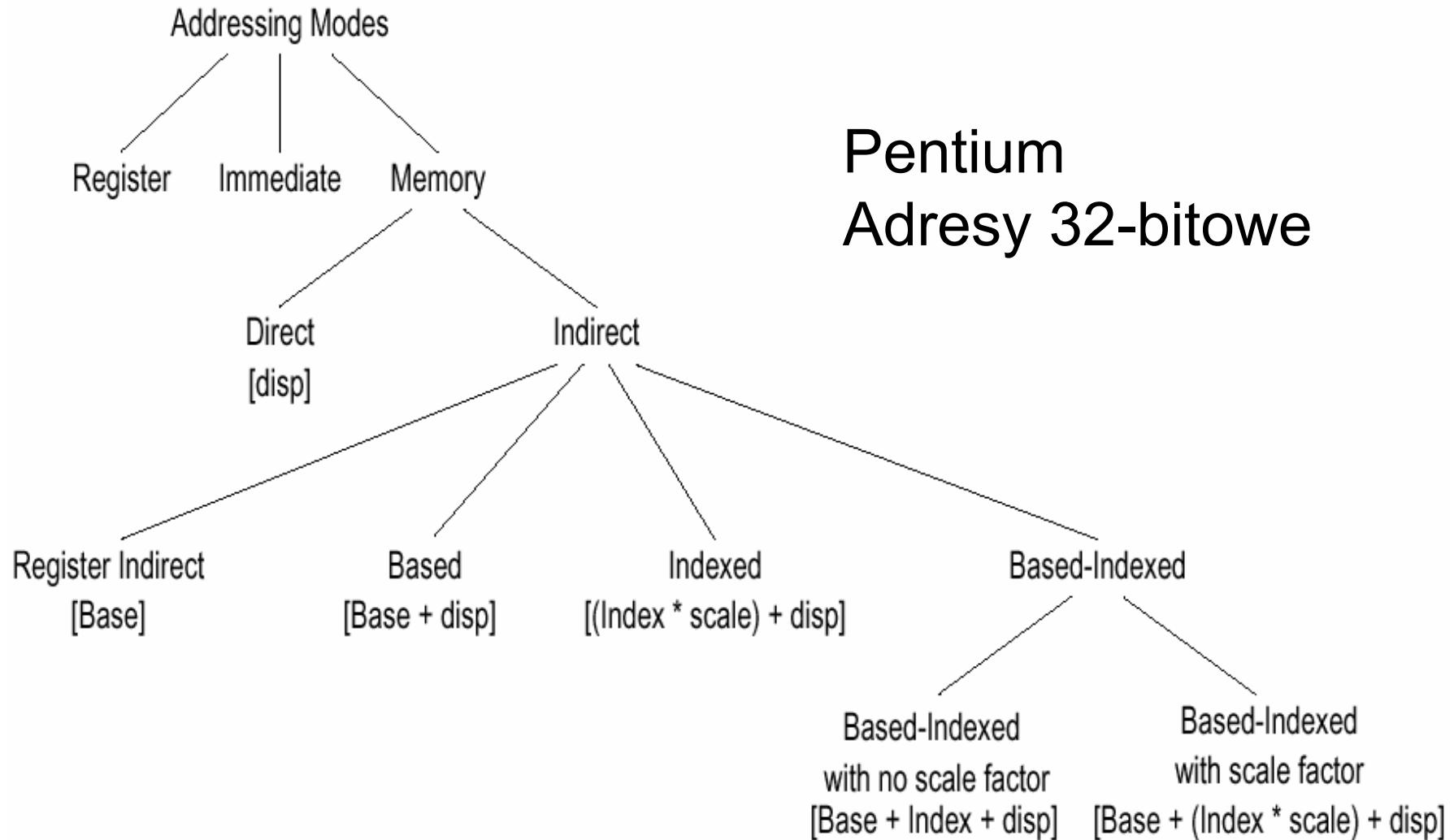
Skoki – przykłady

jmp ax ; skocz w obrębie tego samego segmentu
; do adresu podanego w **ax**

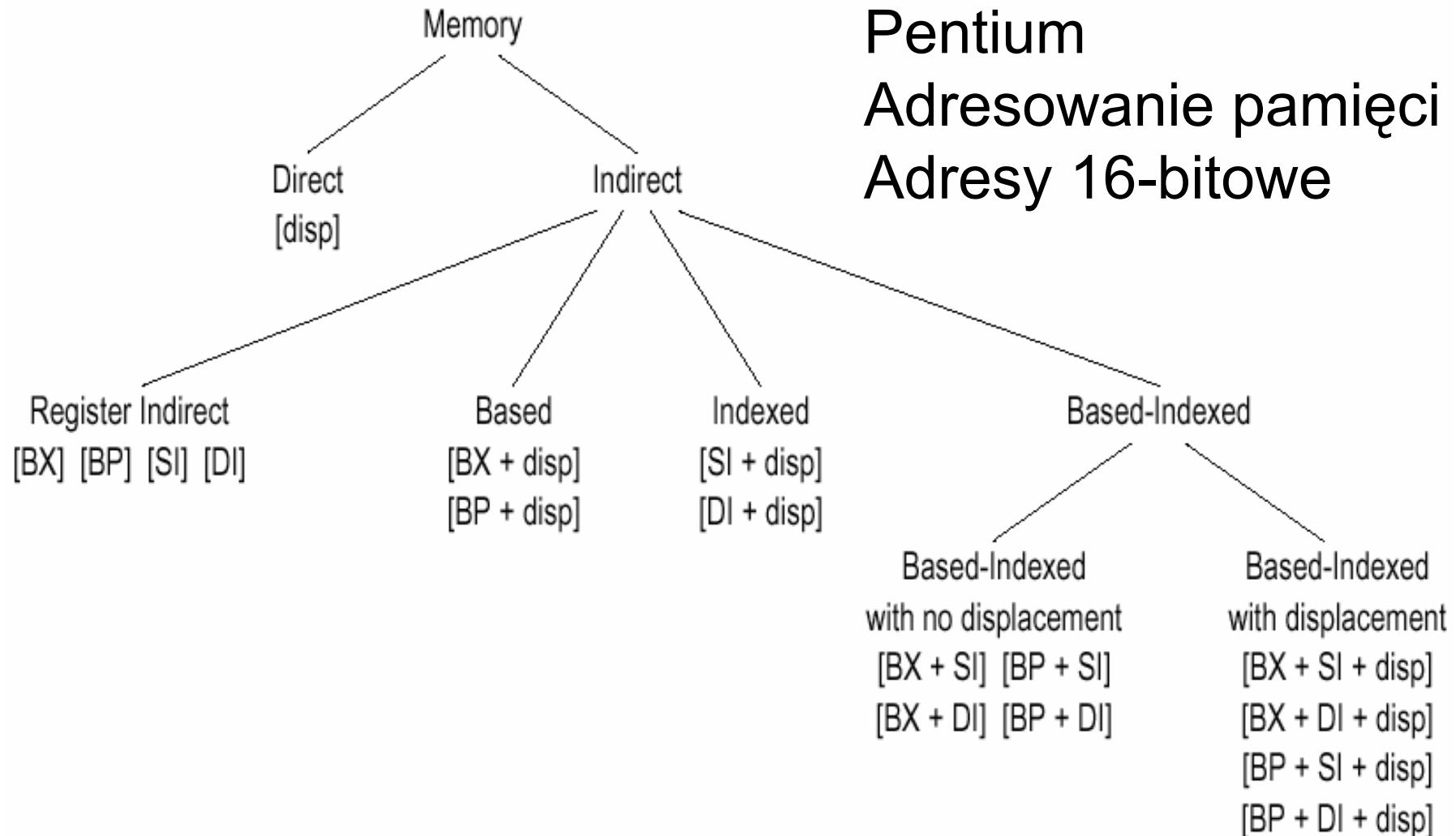
jmp near ptr [bx]
; skocz w obrębie tego samego segmentu
; do adresu wskazanego w komórce pamięci
; w segmencie danych wskazanej przez **bx**

jmp table[bx]
; skocz w obrębie tego samego segmentu
; do adresu wskazanego w wektorze **table**,
; przechowywanym w segmencie danych,
; w elemencie o numerze podanym w **bx**

Tryby adresowania – podsumowanie



Tryby adresowania – podsumowanie



Tryby adresowania – podsumowanie

Pentium – adresy 32-bitowe

Segment + Base + (Index * Scale) + displacement

CS	EAX	EAX	1	no displacement
SS	EBX	EBX	2	8-bit displacement
DS	ECX	ECX	4	32-bit displacement
ES	EDX	EDX	8	
FS	ESI	ESI		
GS	EDI	EDI		
	EBP	EBP		
	ESP			

Tryby adresowania – podsumowanie

Pentium – różnice adresowania 32- i 16-bitowego

	16-bit addressing	32-bit addressing
Base register	BX, BP	EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
Index register	SI, DI	EAX, EBX, ECX, EDX, ESI, EDI, EBP
Scale factor	None	1, 2, 4, 8
Displacement	0, 8, 16 bits	0, 8, 32 bits

Tryby adresowania – podsumowanie

- Skąd procesor Pentium wie czy adres jest 32- czy 16-bitowy?
- Wykorzystuje się bit D w deskryptorze segmentu CS

D = 0

» domyślny rozmiar argumentów i adresów – 16 bitów

D = 1

» domyślny rozmiar argumentów i adresów – 32 bity

- Programista może zmienić wartości domyślne
- W Pentium instrukcję można poprzedzić tzw. Prefiksem (jednobajtowym przedrostkiem, który zmienia domyślne długości argumentów i adresów)
 - 66H prefiks zmiany długości argumentu
 - 67H prefiks zmiany długości adresu
- Stosując prefiksy można mieszać ze sobą tryby 32- i 16-bitowe

Tryby adresowania – podsumowanie

Przykład: niech domyślnym trybem będzie tryb 16-bitowy

Przykład 1: zmiana rozmiaru argumentu

```
mov ax,123 ==> b8 007b
```

```
mov eax,123 ==> 66 | b8 0000007b
```

Przykład 2: zmiana długości adresu

```
mov ax,[ebx*esi+2] ==> 67 | 8b0473
```

Przykład 3: zmiana rozmiaru argumentu i długości adresu

```
mov eax,[ebx*esi+2] ==> 66 | 67 | 8b0473
```

Podsumowanie

- Sposób zapisu instrukcji
 - mnemonik, argumenty
 - zapis w języku asemblera, zapis funkcjonalny, zapis binarny
- Liczba adresów instrukcji
 - procesory 3-, 2-, 1- i 0-adresowe
- Tryby adresowania argumentów (danych)
- Tryby adresowania programu w instrukcjach skoków

Organizacja i Architektura Komputerów

Język asemblera (Pentium)

Asembler i języki wysokiego poziomu

Niektóre instrukcje języka C można zastąpić pojedynczymi instrukcjami w języku asemblera

Assembly Language	C
inc result	result++;
mov size, 45	size = 45;
and mask1, 128	mask1 &= 128;
add marks, 10	marks += 10;

Asembler i języki wysokiego poziomu

Większość instrukcji języka C jest jednak równoważna
większej liczbie instrukcji w języku assemblera

C	Assembly Language
<code>size = value;</code>	<code>mov AX,value</code> <code>mov size,AX</code>
<code>sum += x + y + z;</code>	<code>mov AX,sum</code> <code>add AX,x</code> <code>add AX,y</code> <code>add AX,z</code> <code>mov sum,AX</code>

Program w języku asemblera

- Trzy rodzaje poleceń
 - Instrukcje
 - Mówią procesorowi co ma robić
 - Są wykonywane przez CPU, rodzaj instrukcji jest określony przez kod operacji (Opcode)
 - Dyrektywy (pseudo-ops)
 - Wskazówki dla kompilatora (asemblera) dotyczące procesu tłumaczenia
 - Nie są wykonywane przez CPU, nie generują instrukcji w kodzie wynikowym programu
 - Macro
 - Sposób skróconego zapisu grupy instrukcji
 - Technika podobna do podprogramów, ale realizowana na poziomie kompilacji, a nie w trakcie wykonania podprogramu

Linia programu asemblera

- Format linii programu w języku asemblera:

[label] mnemonic [operands] [;comment]

- Pola w nawiasach [] są opcjonalne
- Etykieta służy do dwóch celów:
 - Oznacza instrukcje, do których są wykonywane skoki
 - Oznacza identyfikatory zmiennych i stałych, do których można w programie odwoływać się przez nazwę etykiety
- Mnemonik określa rodzaj operacji, np. **add**, **or**
- Operandy (argumenty) oznaczają dane wykorzystywane przez instrukcję
 - W Pentium instrukcje mogą mieć 0, 1, 2 lub 3 operandy

Linia programu asemblera cd.

- Komentarze, zaczynają się od znaku średnika
 - Mogą być umieszczone w tej samej linii co instrukcja lub tworzyć osobną linię
- Przykład

```
repeat: inc result ; zwiększMieenną result o 1
CR           equ 0dh      ; znak powrotu karetki (CR)
```

- W powyższym przykładzie etykieta **repeat** umożliwia wykonanie skoku do instrukcji **inc**
- Etykieta CR oznacza stałą o nazwie CR i o wartości **0dh** nadanej przez dyrektywę **equ**

Alokacja danych

- Deklaracja zmiennych w języku wysokiego poziomu, np. w języku C:

```
char      response  
int       value  
float     total  
double   average_value
```

- Określa
 - Wymagany rozmiar pamięci (1 bajt, 2 bajty, ...)
 - Nazwę (etykietę) identyfikującą zmienną (**response**, **value**, ...)
 - Typ zmiennej (**int**, **char**, ...)
 - Ta sama liczba binarna może być interpretowana w różny sposób, np. 1000 1101 1011 1001 oznacza
 - Liczbę -29255 (signed integer)
 - Liczbę 36281 (unsigned)

Alokacja danych cd.

- W języku asemblera do alokacji danych używa się dyrektywy **define**
 - Rezerwuje miejsce w pamięci
 - Etykietuje miejsce w pamięci
 - Inicjalizuje daną
- Dyrektywa **define** nie określa typu danej
 - Interpretacja typu danej jest zależna od programu, który operuje na danej
- Dyrektywy **define** są lokalizowane w części **.DATA** programu
- Ogólny format dyrektywy **define**:
[var-name] d? init-value [,init-value], ...

Alokacja danych cd.

- Mamy 5 rodzajów dyrektywy **define**:

db definiuj bajt (alokacja 1 bajtu)

dw definiuj słowo (alokacja 2 bajtów)

dd definiuj podwójne słowo (alokacja 4 bajtów)

dq definiuj poczwórne słowo (alokacja 8 bajtów)

dt definiuj 10 bajtów (alokacja 10 bajtów)

- Przykłady:

sorted **db** 'y'

response **db** ? ;brak inicjalizacji

value **dw** 25159

float1 **dq** 1.234

Alokacja danych cd.

- Wielokrotne definicje mogą być zapisane w postaci skróconej
- Przykład

message	db	'B'
	db	'y'
	db	'e'
	db	0dh
	db	0ah

można zapisać jako:

message	db	'B' , 'y' , 'e' , 0dh , 0ah
----------------	-----------	-----------------------------

lub w jeszcze bardziej skróconej postaci:

message	db	'Bye' , 0dh , 0ah
----------------	-----------	-------------------

Alokacja danych cd.

- Wielokrotna inicjalizacja

- Dyrektywa DUP umożliwia nadanie tej samej wartości wielu obiektom (bajtom, słowom. itd.)
- Tablica o nazwie **marks** złożona z 8 słów może być zadeklarowana i wstępnie wyzerowana w następujący sposób:
marks dw 8 DUP (0)

- Przykłady:

table dw 10 DUP (?)	;10 słów, bez ;inicjalizacji
message db 3 DUP ('Bye!')	;12 bajtów ;zainicjowanych jako ;'Bye!Bye!Bye!'
name1 db 30 DUP ('?')	;30 bajtów, każdy ;zainicjowany jako ;znak '?'

Alokacja danych cd.

- Dyrektywa DUP może być zagnieżdżana
- Przykład

```
stars db 4 DUP(3 DUP ('*'),2 DUP ('?'),5 DUP ('!'))
```

rezerwuje 40 bajtów i nadaje im wartość:

```
***??!!!!***??!!!!!*!!**??!!!!!*!!**??!!!!
```

- Przykład

```
matrix dw 10 DUP (5 DUP (0))
```

- definiuje macierz 10 x 5 i wstępnie zeruje jej elementy
- tę samą deklarację można zapisać jako:

```
matrix dw 50 DUP (0)
```

Alokacja danych cd.

- Podczas komplikacji asembler buduje tabelę symboli, w której umieszcza napotkane w programie etykiety, w tym etykiety oznaczające dane
- Przykład:

```
.DATA  
value    DW   0  
sum      DD   0  
marks    DW   10 DUP (?)  
message  DB   'The grade is:',0  
char1    DB   ?
```

name	offset
value	0
sum	2
marks	6
message	26
char1	40

Alokacja danych cd.

- Rozmiary danych w języku asemblera i odpowiadające im typy danych w języku C

Directive	C data type
DB	char
DW	int, unsigned
DD	float, long
DQ	double
DT	internal intermediate float value

Alokacja danych cd.

- Dyrektywa **label**

- Jest innym sposobem nadania nazwy danym umieszczonym w pamięci
- Ogólny format:

name label type

- Dostępnych jest pięć typów danych

byte	1 bajt
word	2 bajty
dword	4 bajty
qword	8 bajtów
tword	10 bajtów

Alokacja danych cd.

- Dyrektywa `label`
- Przykład

```
.DATA  
    count      label   word  
    Lo_count  db      0  
    Hi_count  db      0  
.CODE  
    ...  
    mov  Lo_count,al  
    mov  Hi_count,cl
```

- `count` oznacza wartość 16-bitową (dwa bajty)
- `Lo_count` oznacza mniej znaczący bajt
- `Hi_count` oznacza bardziej znaczący bajt

Adresowanie argumentów

- Adresowanie rejestrowe (register)
 - Argumenty znajdują się w wewnętrznych rejestrach CPU
 - Najbardziej efektywny tryb adresowania
- Przykłady

mov eax,ebx

mov bx,cx

- Instrukcja **mov**

mov destination, source

- Kopiuje dane z **source** do **destination**

Adresowanie argumentów cd.

- Adresowanie natychmiastowe (immediate)
 - Argument jest zapisany w instrukcji jako jej część
 - Tryb jest efektywny, ponieważ nie jest potrzebny osobny cykl ładowania argumentu
 - Używany w przypadku danych o stałej wartości
- Przykład
 - mov al, 75**
 - Liczba 75 (dziesiętna) jest ładowana do rejestru **al** (8-bitowego)
 - Powyższa instrukcja używa trybu rejestrowego do wskazania argumentu docelowego oraz trybu natychmiastowego do wskazania argumentu źródłowego

Adresowanie argumentów cd.

- Adresowanie bezpośrednie (direct)
 - Dane znajdują się w segmencie danych
 - Adres ma dwa elementy: **segment:offset**
 - Ofset w adresowaniu bezpośrednim jest podawany w instrukcji, natomiast wskazanie na segment odbywa się z wykorzystaniem domyślnego rejestru segmentu (**DS**), chyba że w instrukcji zastosowano prefiks zmiany segmentu na inny
- Jeśli w programie używamy etykiet przy dyrektywach **db**, **dw**, **label**,
 - Asembler oblicza offset na podstawie wartości liczbowej etykiety
 - W tym celu wykorzystywana jest tablica symboli

Adresowanie argumentów cd.

- Adresowanie bezpośrednie
- Przykład

mov al, response

- Asembler zastępuje symbol **response** przez jego liczbową wartość (offset) z tablicy symboli

mov table1, 56

- Założymy, że obiekt **table1** jest zadeklarowany jako
table1 dw 20 DUP (0)
- Instrukcja **mov** odnosi się w tym przypadku do pierwszego elementu wektora **table1** (elementu o numerze 0)
- W języku C równoważny zapis jest następujący:
table1[0] = 56

Adresowanie argumentów cd.

- Adresowanie bezpośrednie – problemy
 - Tryb bezpośredni jest wygodny tylko przy adresowaniu prostych zmiennych
 - Przy adresowaniu tablic powstają poważne kłopoty
 - Jako przykład warto rozważyć dodawanie elementów tablic
 - Adresowanie bezpośrednie nie nadaje się do użycia w pętlach programowych z iteracyjnie zmienianym wskaźnikiem elementów tablic
 - Rozwiązaniem jest zastosowanie adresowania pośredniego (indirect)

Adresowanie argumentów cd.

- Adresowanie pośrednie (indirect)
 - Ofset jest określony pośrednio, z wykorzystaniem rejestru
 - Tryb ten czasem jest nazywany pośrednim rejestrówym (register indirect)
 - Przy adresowaniu 16-bitowym offset znajduje się w jednym z rejestrów: **bx**, **si** lub **di**
 - Przy adresowaniu 32-bitowym offset może znajdować się w dowolnym z 32-bitowych rejestrów GP procesora
- Przykład
 - mov ax, [bx]**
 - Nawiąsy [] oznaczają, że register **bx** zawiera offset, a nie argument

Adresowanie argumentów cd.

- Używając adresowania pośredniego można przetwarzać tablice przy pomocy pętli programowej
- Przykład: sumowanie elementów tablicy
 - Załaduj adres początkowy tablicy (czyli offset) do rejestru **bx**
 - Powtarzaj następujące czynności w pętli
 - Pobierz element z tablicy używając offsetu z **bx**
 - Wykorzystaj adresowanie pośrednie
 - Dodaj element do bieżącej sumy częściowej
 - Zmień offset w **bx** aby wskazywał kolejny element w tablicy

Adresowanie argumentów cd.

- Ładowanie offsetu do rejestru
 - Założmy, że chcemy załadować do bx offset tablicy o nazwie **table1**
 - Nie możemy zapisać:
`mov bx, table1`
 - Dwa sposoby załadowania offsetu do **bx**
 - Z wykorzystaniem dyrektywy **offset**
 - Operacja zostanie wykonana podczas kompilacji
 - Z wykorzystaniem instrukcji **lea** (load effective address)
 - Operacja zostanie wykonana przez CPU w trakcie realizacji programu

Adresowanie argumentów cd.

- Wykorzystanie dyrektywy `offset`
 - Poprzedni przykład można rozwiązać pisząc
`mov bx,offset table1`
- Wykorzystanie instrukcji `lea`
 - Ogólny format instrukcji `lea`
`lea register,source`
 - Zastosowanie `lea` w poprzednim przykładzie
`lea bx,table1`

Adresowanie argumentów cd.

- Co lepiej zastosować – **offset** czy **lea** ?
 - Należy stosować dyrektywę **offset** jeśli tylko jest to możliwe
 - Dyrektywa **offset** wykonuje się podczas kompilacji
 - Instrukcja **lea** wykonuje się w trakcie realizacji programu
 - Czasem użycie instrukcji **lea** jest konieczne
 - Jeśli dana jest dostępna tylko w trakcie wykonywania programu
 - np. gdy dostęp do danej jest przez indeks podany jako parametr podprogramu
 - Możemy zapisać

```
lea bx, table[si]
```

aby załadować **bx** adresem elementu tablicy **table1**, którego indeks jest podany w rejestrze **si**
 - W tym przypadku nie można użyć dyrektywy **offset**

Instrukcje przesyłu danych

- Najważniejsze instrukcje tego typu w asemblerze Pentium to:
 - **mov** (move) – kopiuj daną
 - **xchg** (exchange) – zamień ze sobą dwa argumenty
 - **xlat** (translate) – transluje bajt używając tablicy translacji
- Instrukcja **mov**
 - Format
mov destination, source
 - Kopiuje wartość z **source** do **destination**
 - Argument źródłowy **source** pozostaje niezmieniony
 - Obydwa argumenty muszą mieć ten sam rozmiar (np. bajt, słowo itp.)
 - **Source** i **destination** nie mogą równocześnie oznaczać adresu pamięci
 - Jest to cecha większości instrukcji Pentium
 - Pentium ma specjalne instrukcje, które ułatwiają kopowanie bloków danych z pamięci do pamięci

Instrukcje przesyłań danych cd.

- W instrukcji **mov** można stosować 5 różnych kombinacji trybów adresowania:
- Tryby te można stosować również we wszystkich innych instrukcjach Pentium z dwoma argumentami

Instruction type	Example
mov register, register	mov DX, CX
mov register, immediate	mov BL, 100
mov register, memory	mov BX, count
mov memory, register	mov count, SI
mov memory, immediate	mov count, 23

Instrukcje przesyłań danych cd.

- Niejednoznaczność operacji `mov`
- Założymy następującą sekwencję deklaracji:

```
. DATA  
table1 dw 20 DUP (0)  
status db 7 DUP (1)
```

- W podanej niżej sekwencji instrukcji przesyłań dwie ostatnie są niejednoznaczne:

```
mov bx,offset table1  
mov si,offset status  
mov [bx],100  
mov [si],100
```

- Nie jest jasne, czy asembler ma użyć bajtu czy słowa do reprezentacji liczby 100

Instrukcje przesyłań danych cd.

- Niejednoznaczność instrukcji **mov** z poprzedniego przykładu można usunąć używając dyrektywy **ptr**
- Ostatnie dwie instrukcje **mov** z poprzedniej planszy można zapisać jako:
 - mov word ptr [bx],100**
 - mov byte ptr [si],100**
 - Słowa **word** i **byte** są nazywane specyfikacją typu
- Można używać ponadto następujących specyfikacji typu:
 - **dword** – dla wartości typu doubleword
 - **qword** – dla wartości typu quadword
 - **tword** – dla wartości złożonych z 10 bajtów

Instrukcje przesyłań danych cd.

- Instrukcja **xchg**
- Ogólny format

xchg operand1,operand2

Zamienia wartości argumentów **operand1** i **operand2**

- Przykłady

xchg eax,edx

xchg response,cl

xchg total,dx

- Bez instrukcji **xchg** operacja zamiany argumentów wymagałaby dwóch instrukcji **mov** i pomocniczego rejestru

Instrukcje przesyłań danych cd.

- Instrukcja **xchg** jest przydatna przy konwersji 16-bitowych danych między formatami little-endian i big-endian
- Przykład

xchg al,ah

Zamienia dane w **ax** na przeciwny format endian

- Pentium ma podobną instrukcję **bswap** do konwersji danych 32-bitowych:

bswap 32-bit_register

Instrukcja **bswap** działa tylko na rejestrach 32-bitowych

Instrukcje przesyłań danych cd.

- Instrukcja **xlat** transluje bajty
- Format
xlatb
- Przed użyciem instrukcji **xlat** należy:
 - W rejestrze **bx** umieścić adres początkowy tablicy translacji
 - **a1** Musi zawierać indeks do tablicy (indeksowanie zaczyna się od wartości równej 0)
 - Instrukcja **xlat** odczytuje z tablicy translacji bajt wskazany przez indeks i zapisuje go w **a1**
 - Wartość w **a1** zostaje utracona
- Tablica translacji może mieć najwyżej 256 elementów (rejestr **a1** jest 8-bitowy)

Instrukcje przesyłań danych cd.

- Przykład działania instrukcji **xlat** – szyfrowanie
 - Cyfry na wejściu: 0 1 2 3 4 5 6 7 8 9
 - Cyfry na wyjściu: 4 6 9 5 0 3 1 8 7 2

```
.DATA  
xlat_table    db    '4695031872'  
...  
.CODE  
mov     bx,offset xlat_table  
getCh  al  
sub    al,'0' ;konwersja cyfry (ASCII) na indek  
xlatb  
putCh  al
```

Instrukcje asemblera - przegląd

- Oprócz omówionych instrukcji przesyłań danych procesor Pentium realizuje wiele innych operacji.
- Najważniejsze grupy operacji w Pentium:
 - Instrukcje arytmetyczne
 - Instrukcje skoków
 - Instrukcje pętli
 - Instrukcje logiczne
 - Instrukcje przesunięć logicznych (shift)
 - Instrukcje przesunięć cyklicznych (rotate)
- W dalszej części wykładu są omówione najważniejsze instrukcje z tych grup, które pozwalają napisać wiele prostych, ale bardzo użytecznych programów w języku asemblera

Instrukcje arytmetyczne

- Instrukcje **inc** oraz **dec**
- Format:

```
inc destination  
dec destination
```

- Działanie
 $\text{destination} = \text{destination} +/- 1$
- Argument **destination** może być 8-, 16- lub 32-bitowy, w pamięci lub w rejestrze
 - Nie można stosować trybu natychmiastowego
- Przykłady

```
inc bx          ; bx = bx + 1  
dec value      ; value = value - 1
```

Instrukcje arytmetyczne cd.

- Instrukcja dodawania **add**

- Format

```
add destination,source
```

- Działanie

```
destination = destination + source
```

- Przykłady

```
add ebx,eax
```

```
add value,35
```

- Użycie **inc eax** jest lepsze niż **add eax,1**

- Instrukcja **inc** jest krótsza w kodzie binarnym

- Obydwie instrukcje wykonują się w tym samym czasie

Instrukcje arytmetyczne cd.

- Instrukcja odejmowania **sub**

- Format

```
sub destination,source
```

- Działanie

```
destination = destination - source
```

- Przykłady

```
sub ebx,eax
```

```
sub value,35
```

- Użycie **dec eax** jest lepsze niż **sub eax,1**

- Instrukcja **dec** jest krótsza w kodzie binarnym

- Obydwie instrukcje wykonują się w tym samym czasie

Instrukcje arytmetyczne cd.

- Instrukcja porównania **cmp**

- Format

cmp destination, source

- Działanie

destination - source

– zarówno **destination** jak i **source** nie ulegają zmianie, CPU wykonuje jedynie wirtualne odejmowanie by porównać argumenty (zbadać relację między nimi: $>$, $=$)

- Przykłady

cmp ebx, eax

cmp count, 100

- Instrukcja **cmp** jest używana razem z instrukcjami skoków warunkowych do realizacji rozgałęzień w programie uzależnionych od relacji między argumentami **cmp**

Instrukcje skoków

- Skok bezwarunkowy **jmp**

- Format

jmp label

- Działanie

- CPU przechodzi do realizacji instrukcji oznaczonej etykietą **label**

- Przykład – nieskończona pętla

```
mov eax,1
inc_again:
    inc eax
    jmp inc_again
    mov ebx,eax      ; ta instrukcja nigdy się
                      ; nie wykona
```

Instrukcje skoków cd.

- Skoki warunkowe
- Format

`j<cond> label`

- Działanie
 - CPU przechodzi do realizacji instrukcji oznaczonej etykietą `label` jeśli warunek `<cond>` jest spełniony
- Przykład – testowanie wystąpienia znaku powrotu karetki CR

```
getCh    al  
cmp      al,0dh ; 0dh = ASCII CR  
je       CR_received  
inc      cl  
...  
CR_received:  
...
```

Instrukcje skoków cd.

- Wybrane instrukcje skoków warunkowych
 - Argumenty w poprzedzającej instrukcji cmp są traktowane jako liczby ze znakiem

je	skocz gdy równe (equal)
jg	skocz gdy większe (greater)
jl	skocz gdy mniejsze (less)
jge	skocz gdy większe lub równe (greater or equal)
jle	skocz gdy mniejsze lub równe (less or equal)
jne	skocz gdy różne (not equal)

Instrukcje skoków cd.

- Skoki warunkowe mogą również testować stan poszczególnych bitów w rejestrze wskaźników (**eflags**)

jz skocz gdy zero (jump if ZF = 1)

jnz skocz gdy nie zero (jump if ZF = 0)

jc skocz gdy przeniesienie (jump if CF = 1)

jnc skocz gdy brak przeniesienia (jump if CF = 0)

- Instrukcja **jz** jest synonimem **je**
- Instrukcja **jnz** jest synonimem **jne**

Instrukcje skoków cd.

- Instrukcja pętli **loop**

- Format

loop target

- Działanie

- Dekrementuje **cx** i skacze do etykiety **target** jeśli **cx** nie jest zerem
 - Przed wejściem w pętlę do rejestru należy wpisać liczbę powtórzeń pętli

- Przykład

```
    mov  cx,50          ; licznik pętli = 50
repeat:                         ; etykieta początku pętli
    ...
    ...
loop   repeat           ; koniec pętli
    ...
```

Instrukcje skoków cd.

- Poprzedni przykład jest równoważny zapisowi

```
    mov  cx,50           ; licznik pętli = 50
repeat:                           ; etykieta początku pętli
    ...
    ...
    dec  cx             ; dekrementuj licznik
    jnz  repeat         ; skoczy do etykiety repeat, jeśli cx ≠ 0
    ...
```

- Nieco zaskakujący jest fakt, że para instrukcji **dec** oraz **jnz** wykonuje się szybciej niż instrukcja **loop** z przykładu na poprzedniej planszy

Instrukcje logiczne

- Format

`and destination,source`

`or destination,source`

`not destination`

- Działanie

- Wykonuje odpowiednią operację logiczną na bitach
 - Wynik jest umieszczany w `destination`

- Instrukcja `test` jest odpowiednikiem instrukcji `and`, ale nie niszczy `destination` (podobnie jak `cmp`) tylko testuje argumenty

`test destination,source`

Instrukcje logiczne cd.

- Przykład – testowanie, czy w rejestrze **a1** jest liczba parzysta, czy nieparzysta

```
test al,01h ; testuj najmniej znaczący bit al
jz liczba_parzysta
liczba_nieparzysta:
    ...
    ; przetwarzaj przypadek liczby
    ; nieparzystej
    jmp skip
liczba_parzysta:
    ...
    ; przetwarzaj przypadek liczby
    ; parzystej
skip:
    ...
```

Instrukcje przesunięć logicznych

- Format

- Przesunięcia w lewo

- `shl destination, count`

- `shl destination, cl`

- Przesunięcia w prawo

- `shr destination, count`

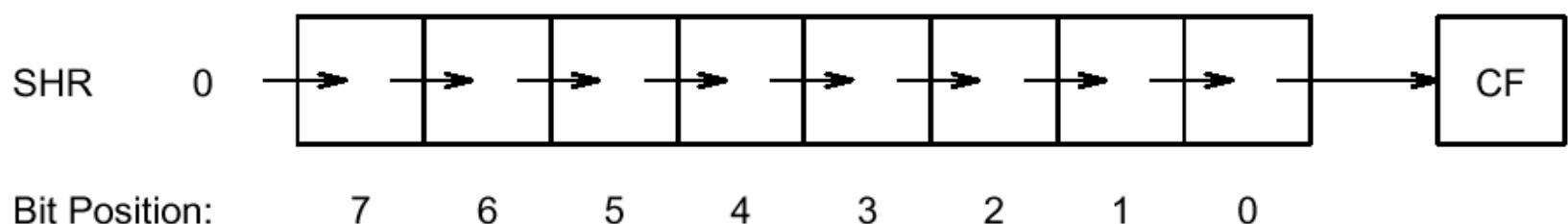
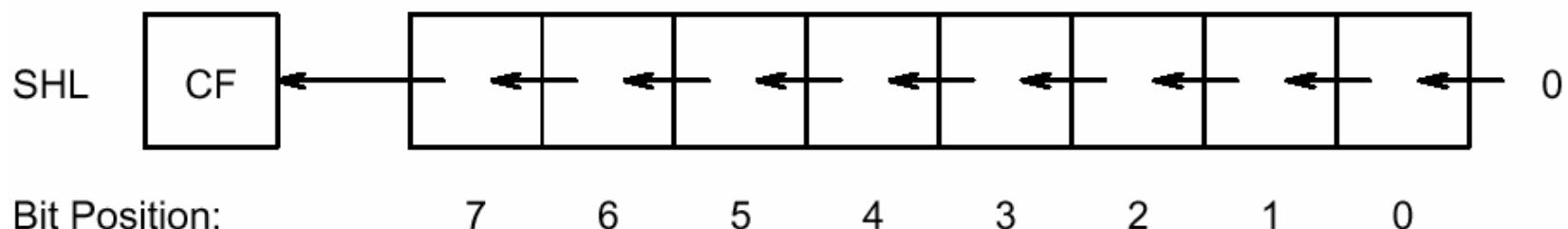
- `shr destination, cl`

- Działanie

- Wykonuje przesunięcie argumentu `destination` w lewo lub w prawo o liczbę bitów podaną w `count` lub w rejestrze `cl`
 - Zawartość rejestrów `cl` nie ulega zmianie

Instrukcje przesunięć logicznych

- Bit opuszczający przesuwany obiekt trafia do bitu przeniesienia w rejestrze wskaźników
- Na wolną pozycję wprowadzany jest bit równy zero



Instrukcje przesunięć logicznych

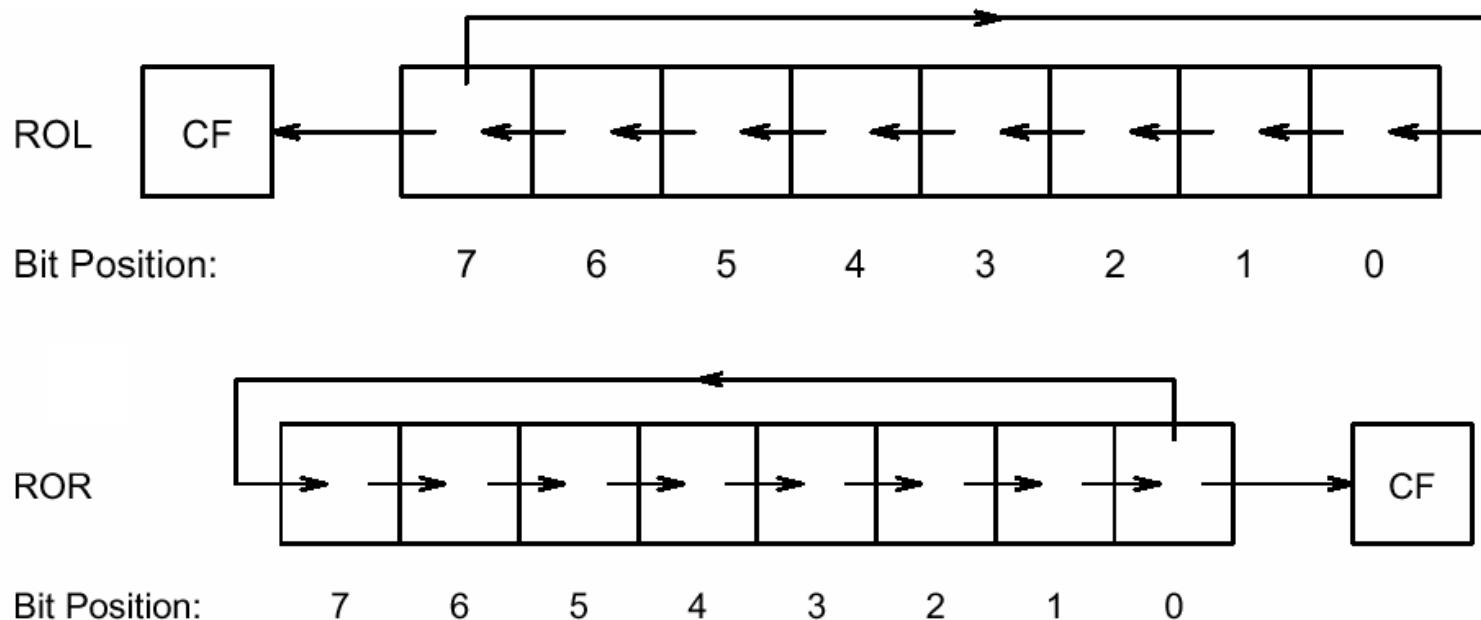
- Liczba **count** jest argumentem natychmiastowym
 - Przykład
`shl ax,5`
- Specyfikacja **count** o wartości większej niż 31 jest niedozwolona
 - Jeśli podana liczba **count** jest większa niż 31 instrukcja używa tylko 5 najmniej znaczących bitów **count**
- Wersja instrukcji z użyciem rejestru **c1** jest przydatna, jeśli liczba bitów, o które należy przesunąć argument jest znana dopiero w czasie wykonania programu

Instrukcje przesunięć cyklicznych

- Dwa typy instrukcji rotate
 - Przesunięcie cykliczne przez bit przeniesienia
rcl (rotate through carry left)
rcr (rotate through carry right)
 - Przesunięcie cykliczne bez bitu przeniesienia
rol (rotate left)
ror (rotate right)
- Format instrukcji **rotate** jest taki sam jak instrukcji **shift**
 - Można stosować natychmiastowy argument **count** lub podawać liczbę pozycji, o które ma nastąpić rotacja używając rejestrów **c1**

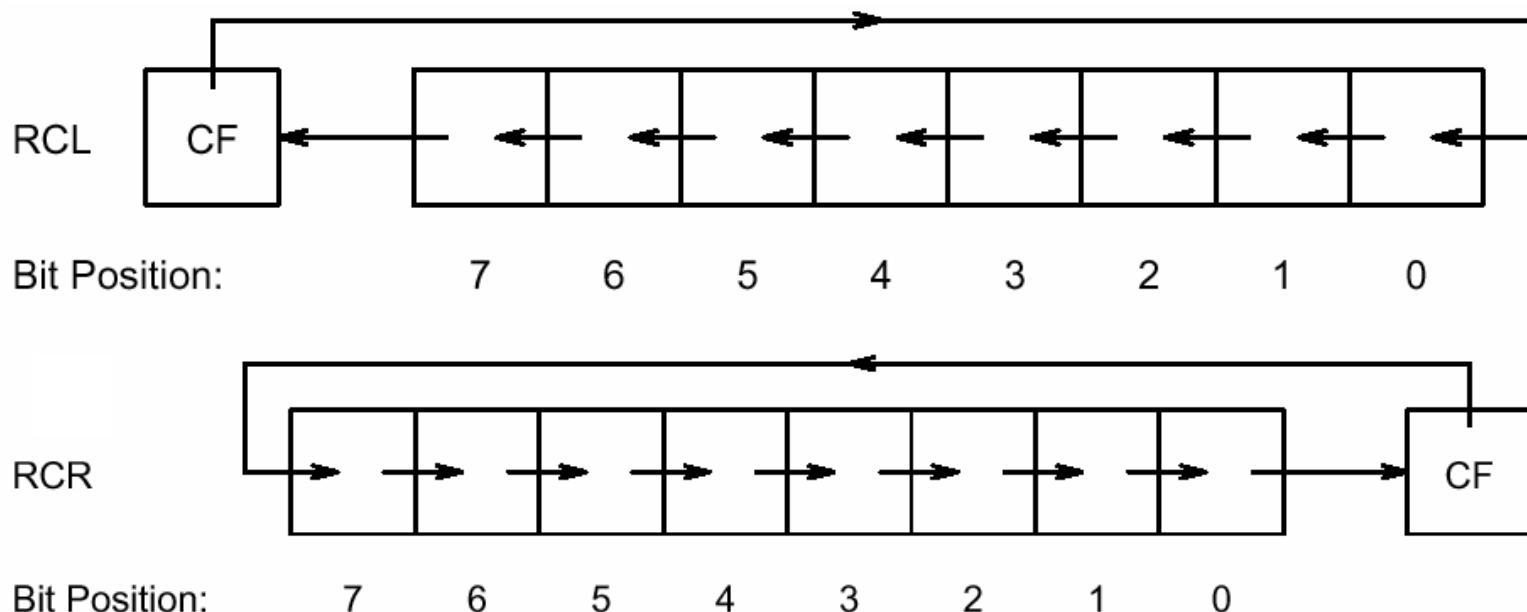
Instrukcje przesunięć cyklicznych

- Instrukcje przesunięć cyklicznych z wyłączeniem bitu przeniesienia działają według poniższego diagramu:



Instrukcje przesunięć cyklicznych

- Instrukcje przesunięć cyklicznych z udziałem bitu przeniesienia działają według poniższego diagramu:



Definiowanie stałych

- Asembler przewiduje dwa sposoby definiowania stałych:
 - Dyrektywa **equ**
 - Definicja nie może być zmieniona
 - Można definiować łańcuchy (strings)
 - Dyrektywa **=**
 - Definicja może być zmieniona
 - Nie można definiować łańcuchów
- Definiowanie stałych zamiast podawanie ich w instrukcjach w postaci natychmiastowej ma dwie ważne zalety:
 - Poprawia czytelność programu
 - Ułatwia dokonywanie zmian w programie (wystarczy zmienić definicję i przekompilować program, zamiast zmieniać wszystkie wystąpienia stałych liczbowych)

Definiowanie stałych

- Dyrektywa **equ**
- Składnia

name equ expression

- Przypisuje wartość wyrażenia **expression** stałej o nazwie **name**
- Wyrażenie jest obliczane podczas komplikacji
 - Podobne działanie jak #define w języku C

- Przykłady

NUM_OF_ROWS equ 50

NUM_OF_COLS equ 10

ARRAY_SIZE equ NUM_OFROWS * NUM_OF_COLS

- Można także definiować łańcuchy znaków, np.

JUMP equ jmp

Definiowanie stałych

- Dyrektywa =
- Składnia

name = expression

- Działanie podobne działanie jak dla dyrektywy **equ**
- Dwie zasadnicze różnice
 - Redefiniowanie jest dozwolone

count = 0

...

count = 99

jest legalnym zapisem

- Nie można definiować stałych łańcuchów i przedefiniowywać słowa kluczowe asemblera lub mnemoniki instrukcji
- Przykład
 - Zapis **JUMP = jmp** jest nielegalny

Przykłady programów

- W załączonym pliku programy_asm.pdf znajdują się teksty źródłowe kilku prostych programów napisanych w języku asemblera procesora Pentium:
 - BINCHAR.ASM – konwersja tekstu ASCII na reprezentację binarną
 - HEXI1CHAR.ASM – konwersja tekstu ASCII na kod szesnastkowy
 - HEX2CHAR.ASM – konwersja tekstu ASCII na kod szesnastkowy z wykorzystaniem instrukcji **xlat**
 - TOUPPER.ASM – konwersja małych liter na duże litery przez manipulację znakami
 - ADDIGITS.ASM – sumowanie poszczególnych cyfr w liczbie

Podsumowanie

- Linie programu asemblerowego
- Alokacja danych
- Tryby adresowania argumentów
 - rejestrowy, natychmiastowy, bezpośredni, pośredni
- Instrukcje przesyłu danych
- Przegląd najważniejszych instrukcji asemblera Pentium
 - instrukcje arytmetyczne, skoki bezwarunkowe i warunkowe, instrukcje logiczne, przesunięcia logiczne i cykliczne
- Definiowanie stałych
- Przykłady programów