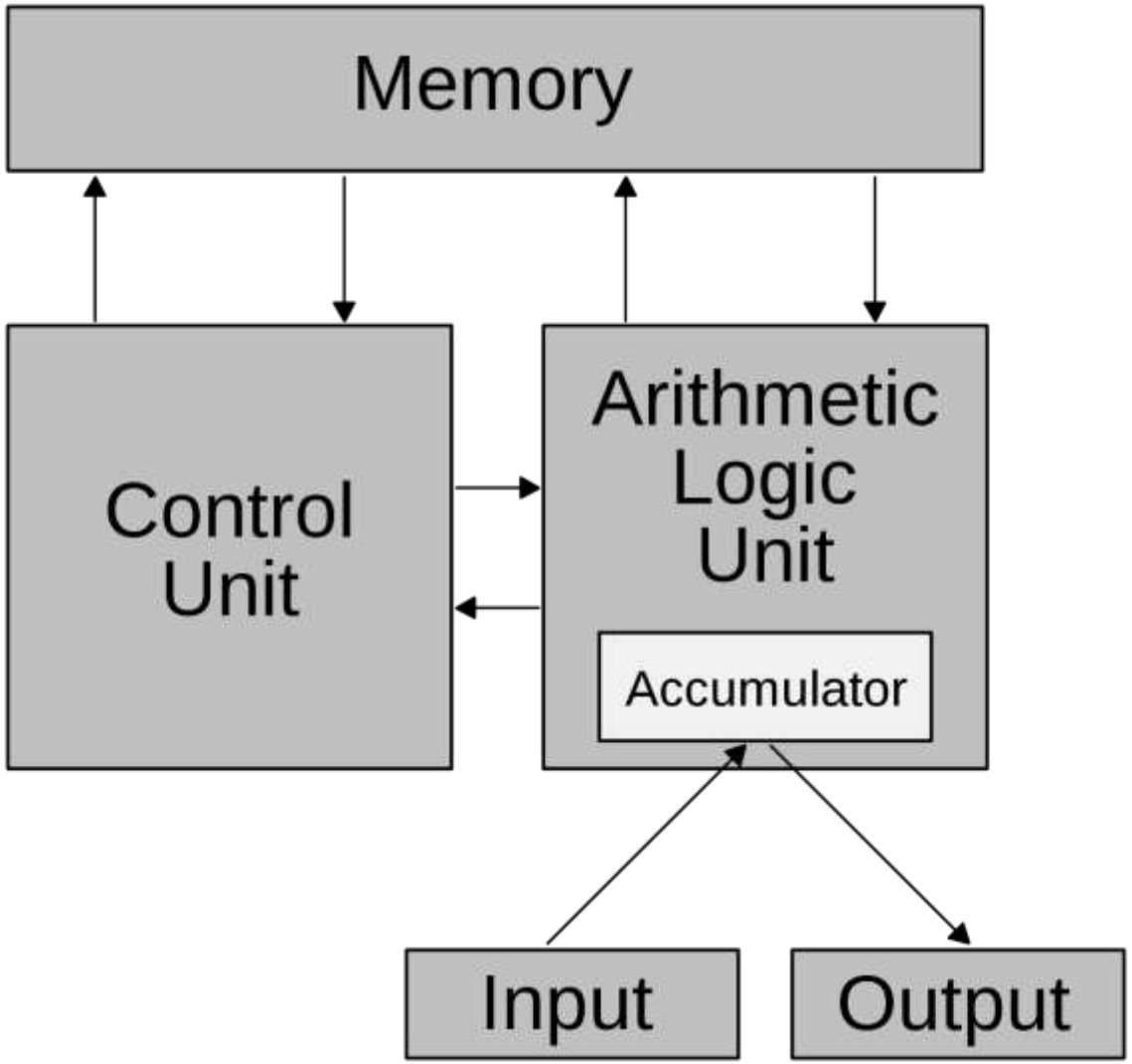
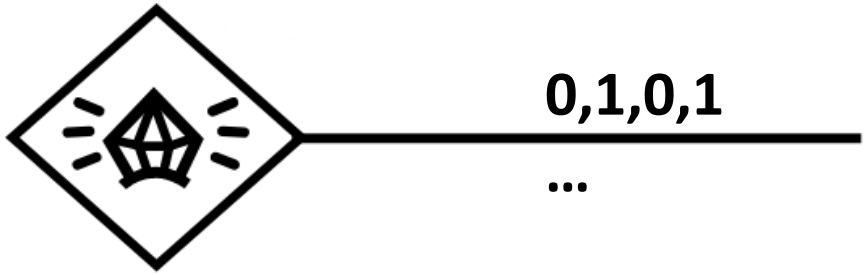

Uniwersytet
w Siedlcach
Architektura
Systemów
Komputerowych
dr Marcin
Stępniak

Komputer von Neumanna



Zegar (cd.)

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



Pamięć

- Wielkie tablice komórek (słowa kilku bajtowe)
- RAM : Random Access Memory
- w RAM jest przechowywana informacja (dane) podczas działania komputera
  - dane jako wartości zmiennych
  - te dane to także program (sekwencja instrukcji z instrukcjami kontrolnymi, tj. skokami) , który jest wykonywany

Program to dane w pamięci

- Jedna z najbardziej istotnych cech architektury von Neumanna:
  - **Instrukcje**
  - **dane**, które są przetwarzane przez te instrukcje

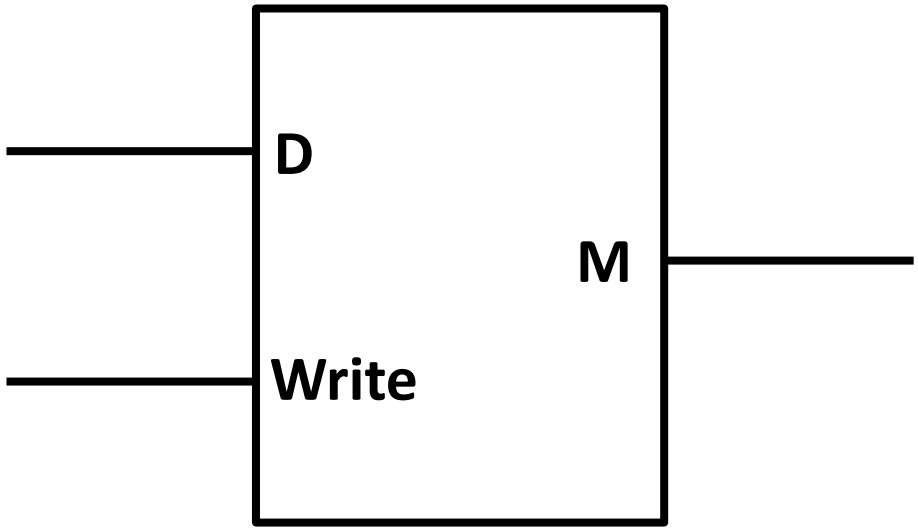
  są w tej samej pamięci RAM.
- Proste a genialne w swojej prostocie: tak jest we współczesnych komputerach

Program to dane w pamięci

- Podstawa dla współczesnych metod obliczeniowych
- Programy mogą być zmieniane podczas ich wykonywania:
  - Dodawane/ usuwane instrukcje w programie
- Ale wielkim odkryciom/wynalazkom towarzyszy zawsze wielkie ryzyko:
  - Wirusy komputerowe

Co to jest RAM i jak to działa

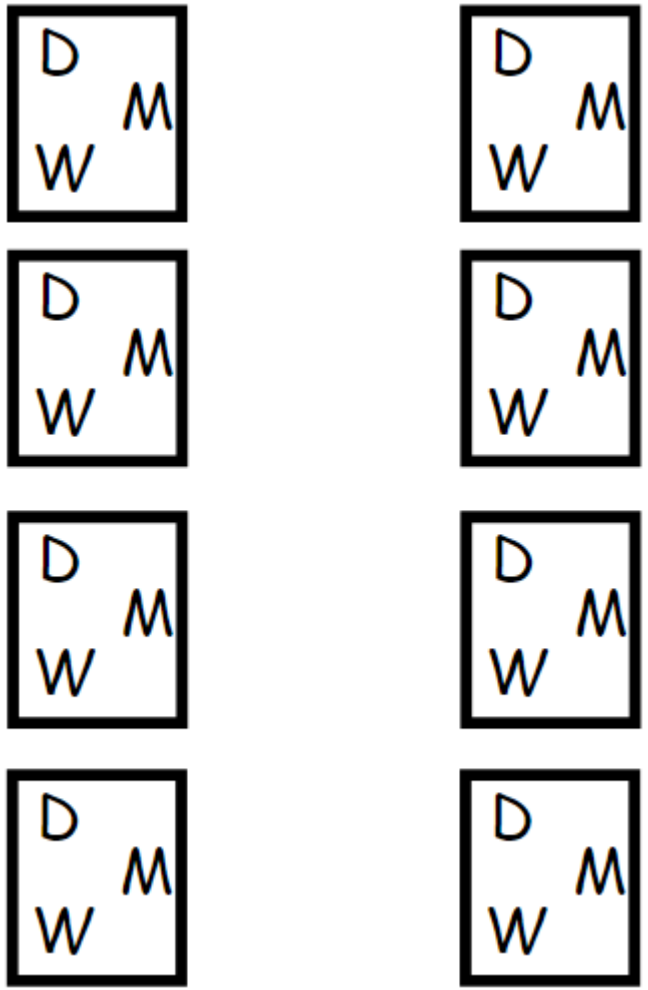
- pojedynczy bit w pamięci RAM…



RAM (cd.)

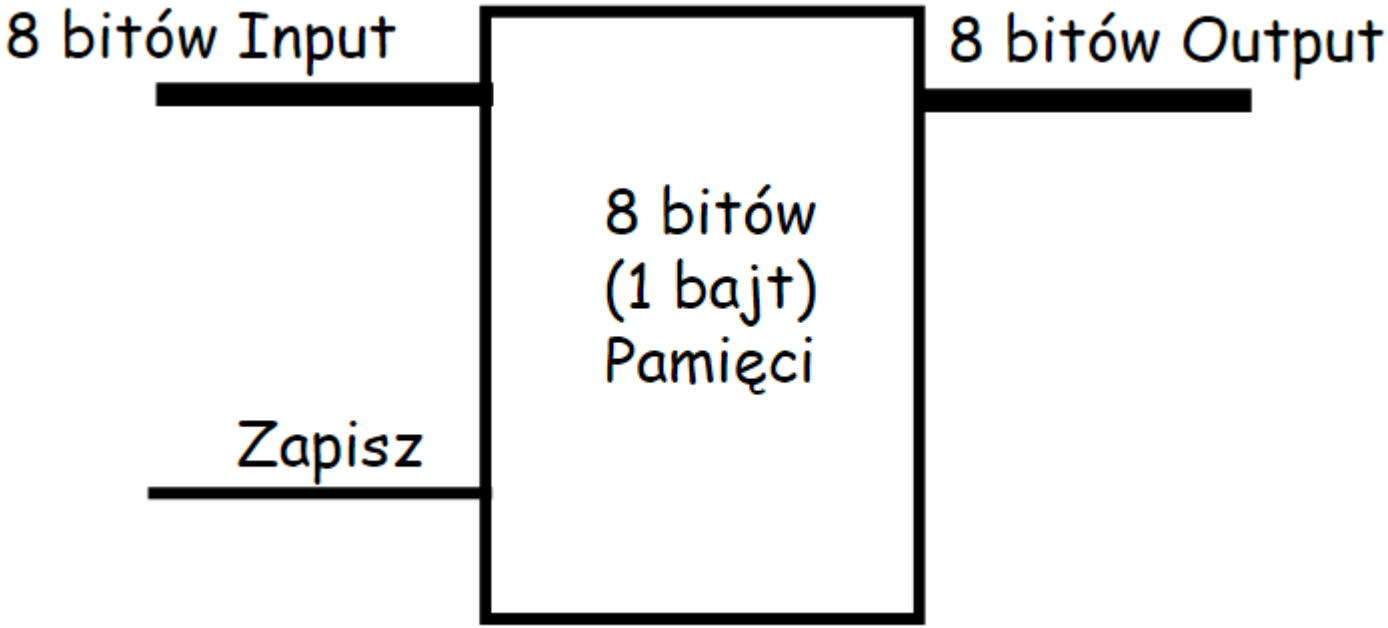
- Połączenie 8 bitów razem w bajt

- 8 bitów (b) = bajt (B)
- RAM składa się z bajtów



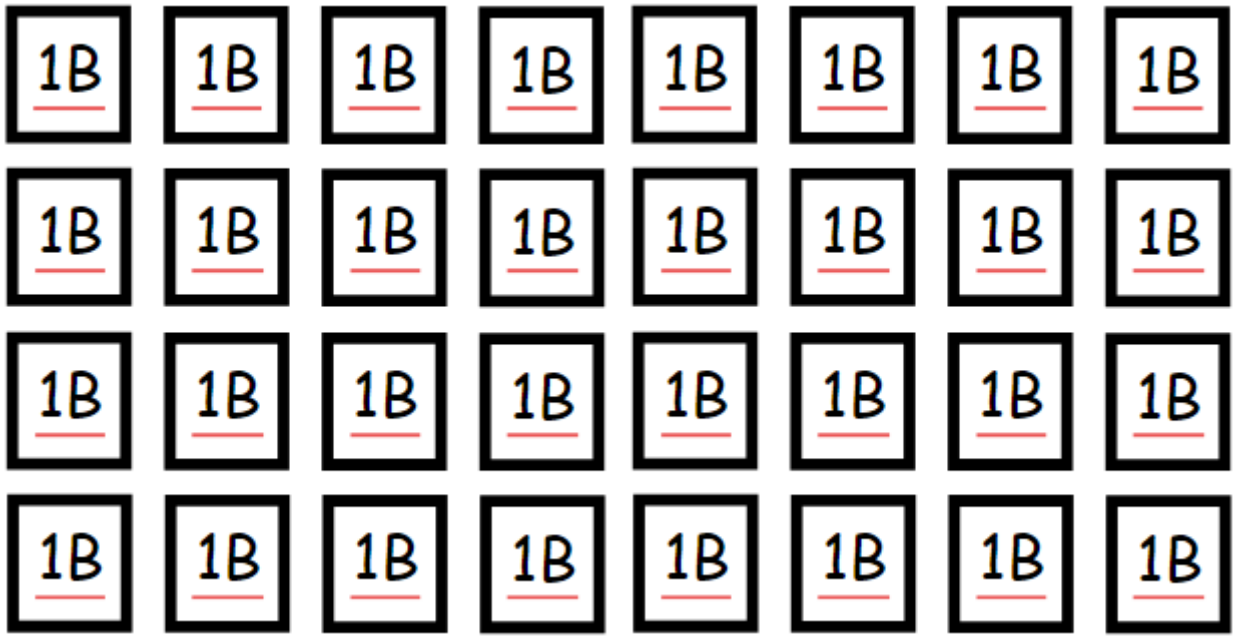
RAM (cd.)

- Bajt pamięci (8-bitowy rejestr)



RAM (cd.)

- potrzebujemy ogromnych pamięci (obecnie gigabajtowych) składających się z 1 bajtowych komórek

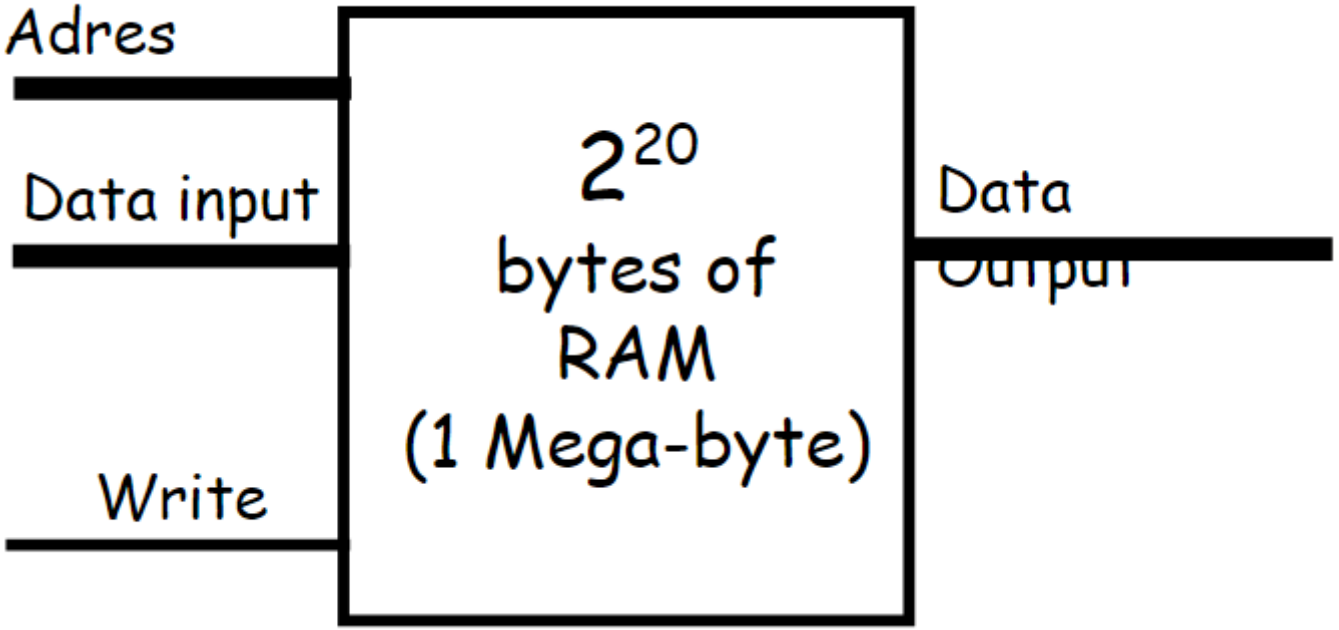


Problem: Jak wyznaczać komórkę pamięci do czytania /zapisywania ?

RAM (cd.)

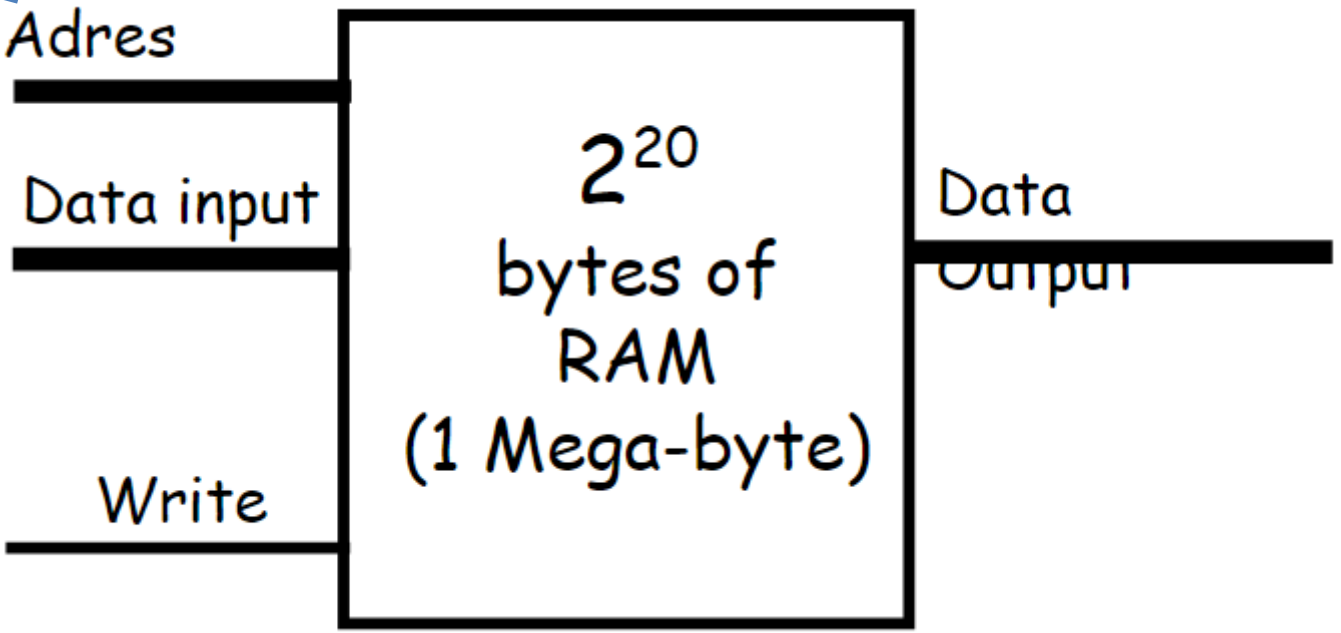
- Rozwiązanie: przypisz każdej komórce unikalny adres (liczbę binarna o ustalonej długości)
- W ten sposób ponumerujemy wszystkie komórki
- Wtedy możemy zapytać RAM:
  - jaka jest zawartość komórki pamięci o adresie 011010101010?
- Lub kazać RAM:
  - wpisz “11101101” do komórki pamięci o adresie 011010101010

RAM (cd.)

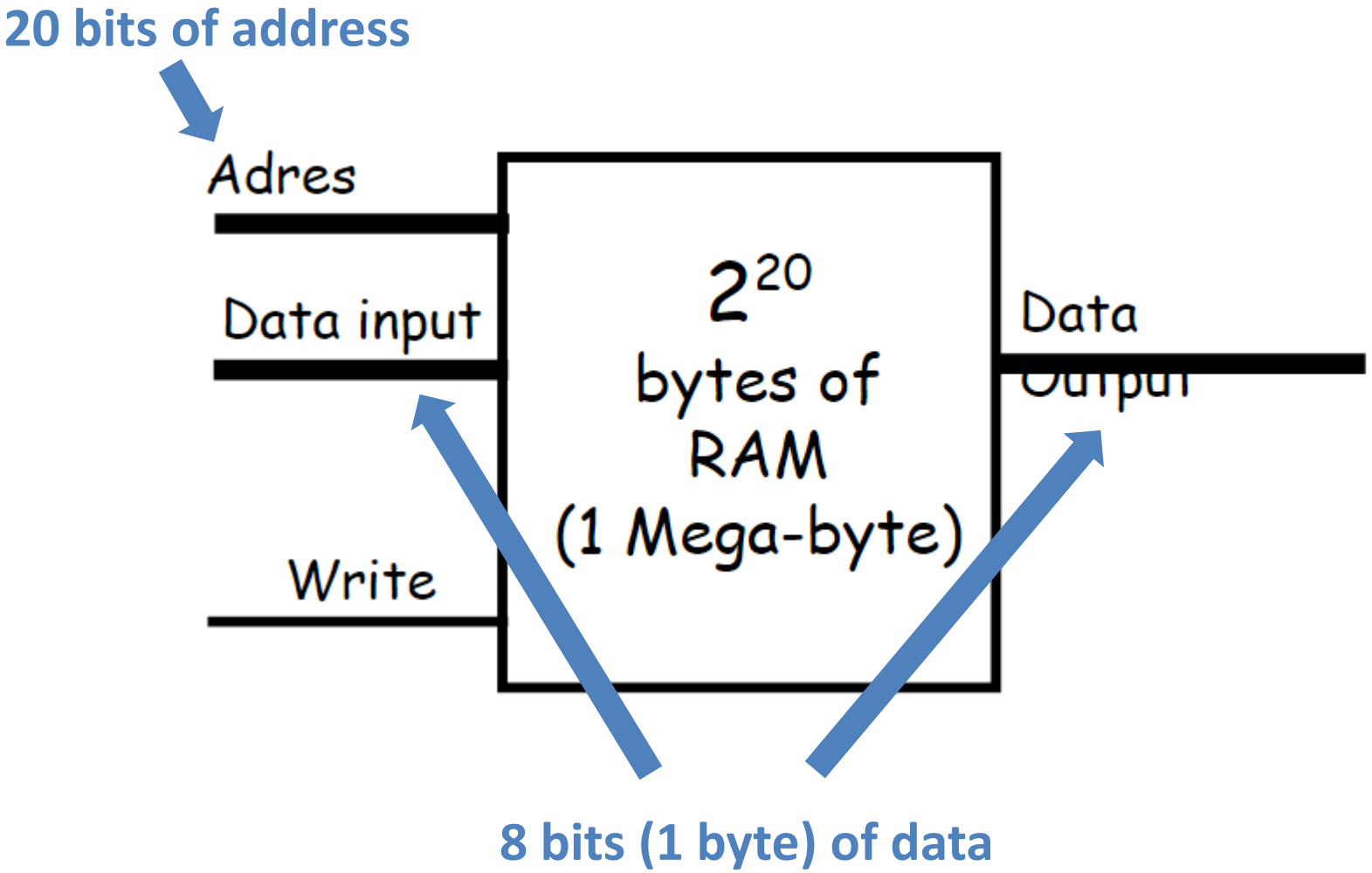


RAM (cd.)

20 bitowy adres

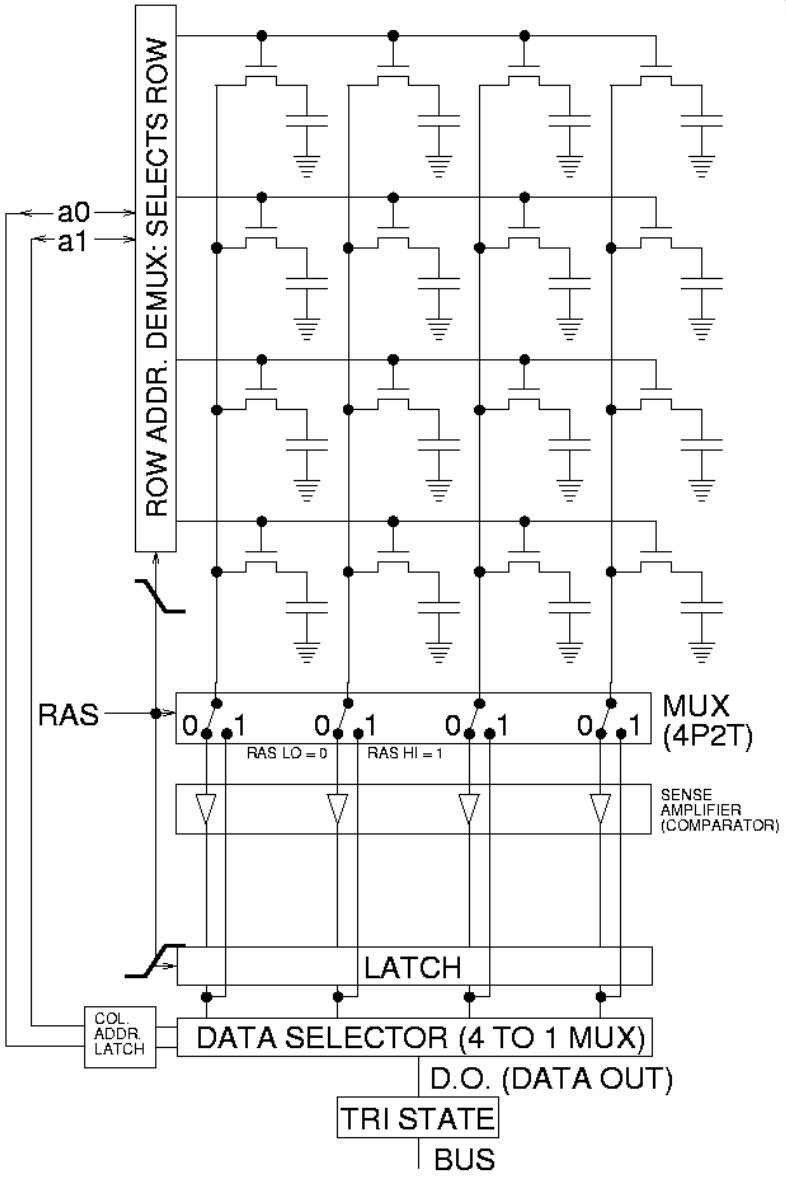


RAM (cd.)


20 bits of address
Adres
Data input
Write
2^{20}
bytes of
RAM
(1 Mega-byte)
Data
8 bits (1 byte) of data

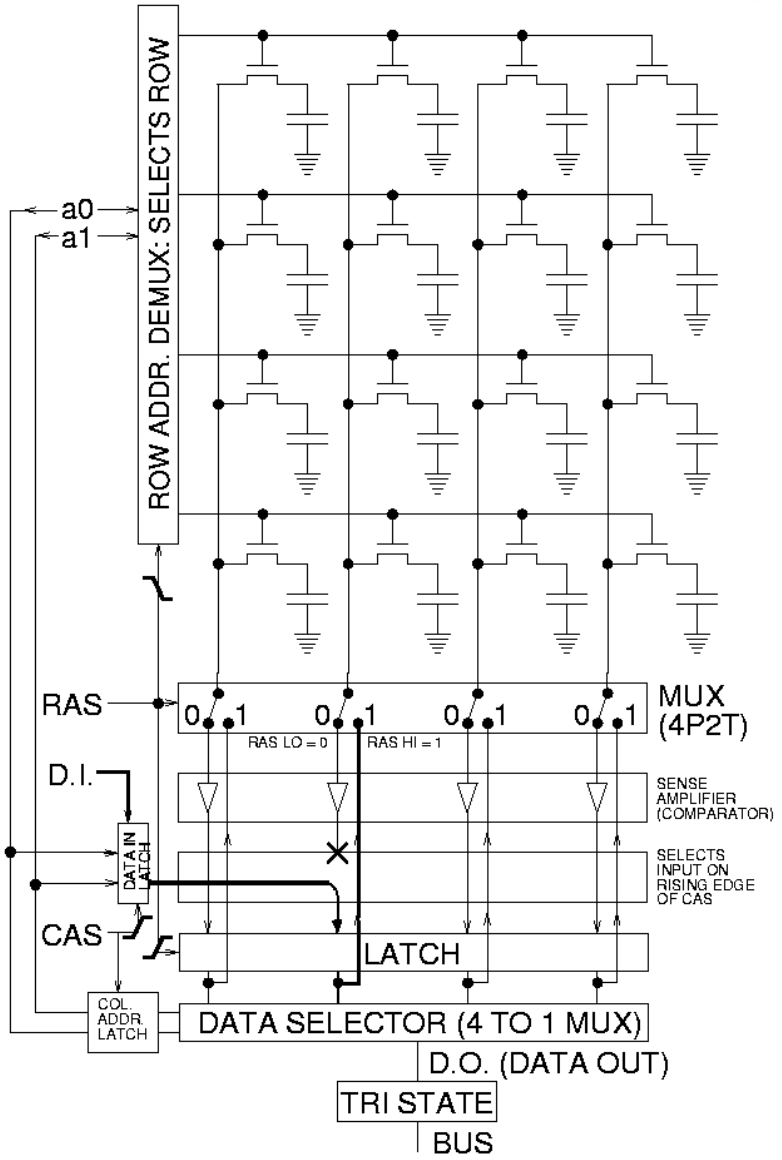
Co to jest RAM i jak to działa

- czytanie …



Co to jest RAM i jak to działa

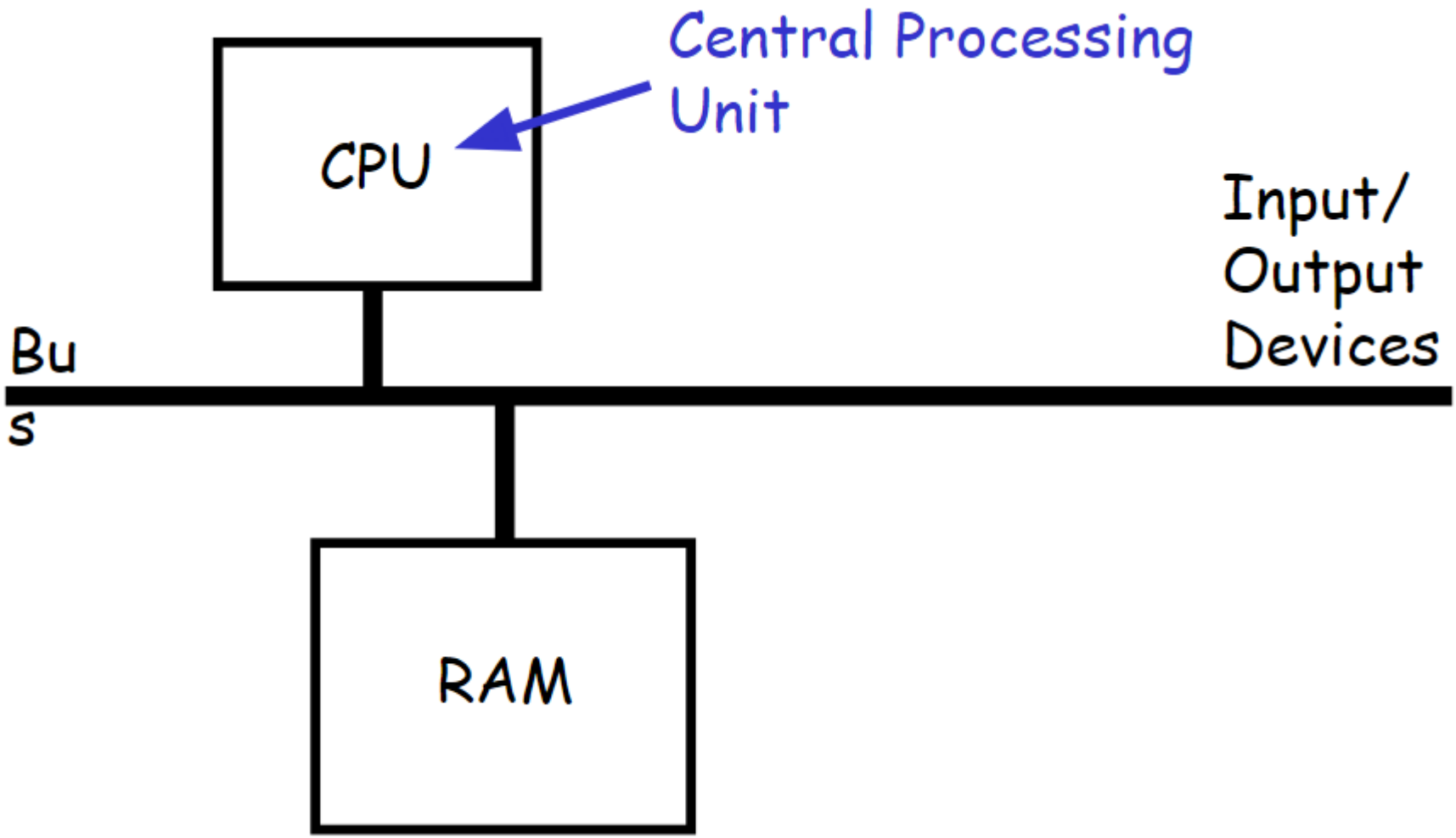
- wpisywanie …



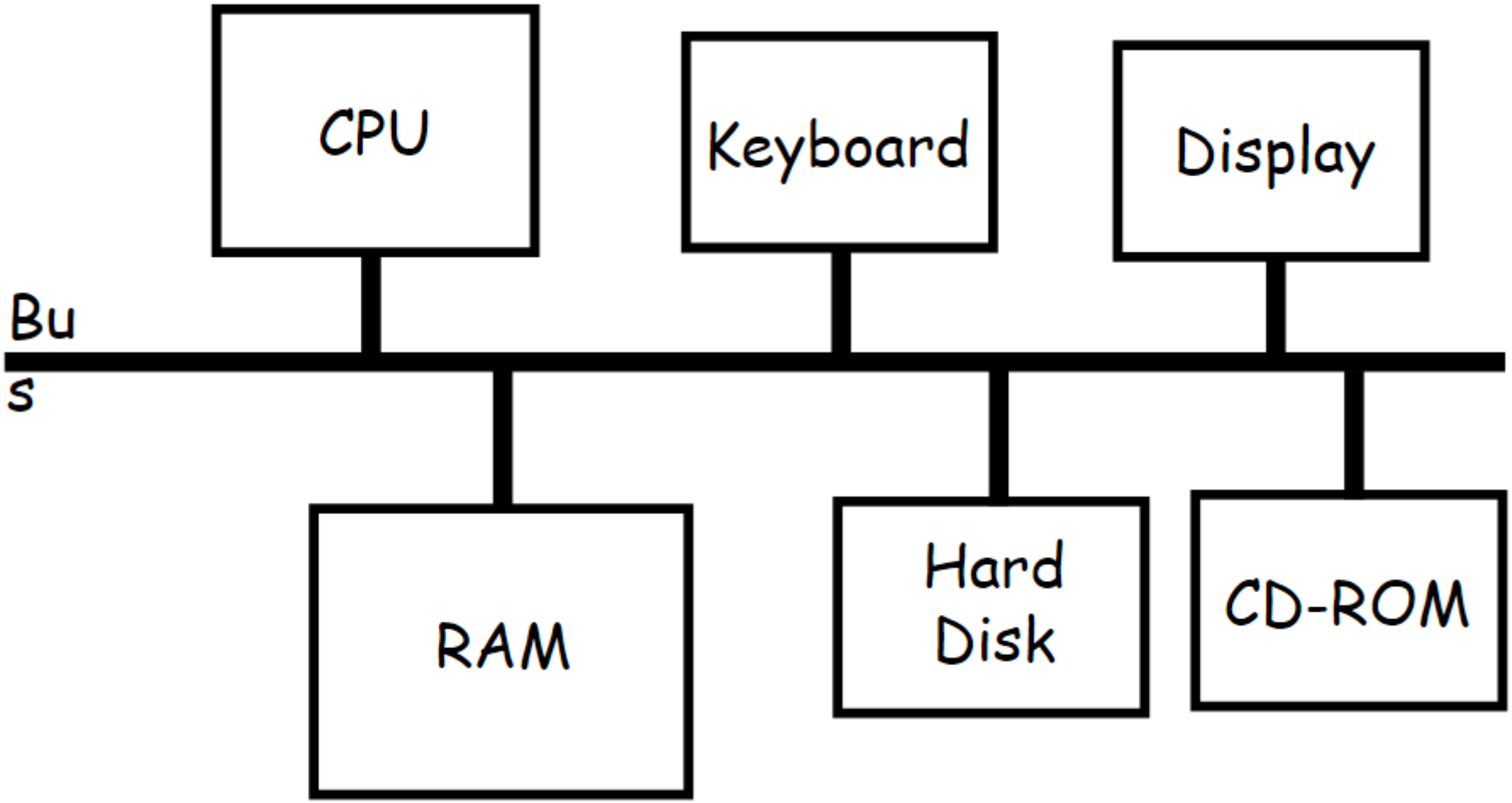
RAM (cd.)

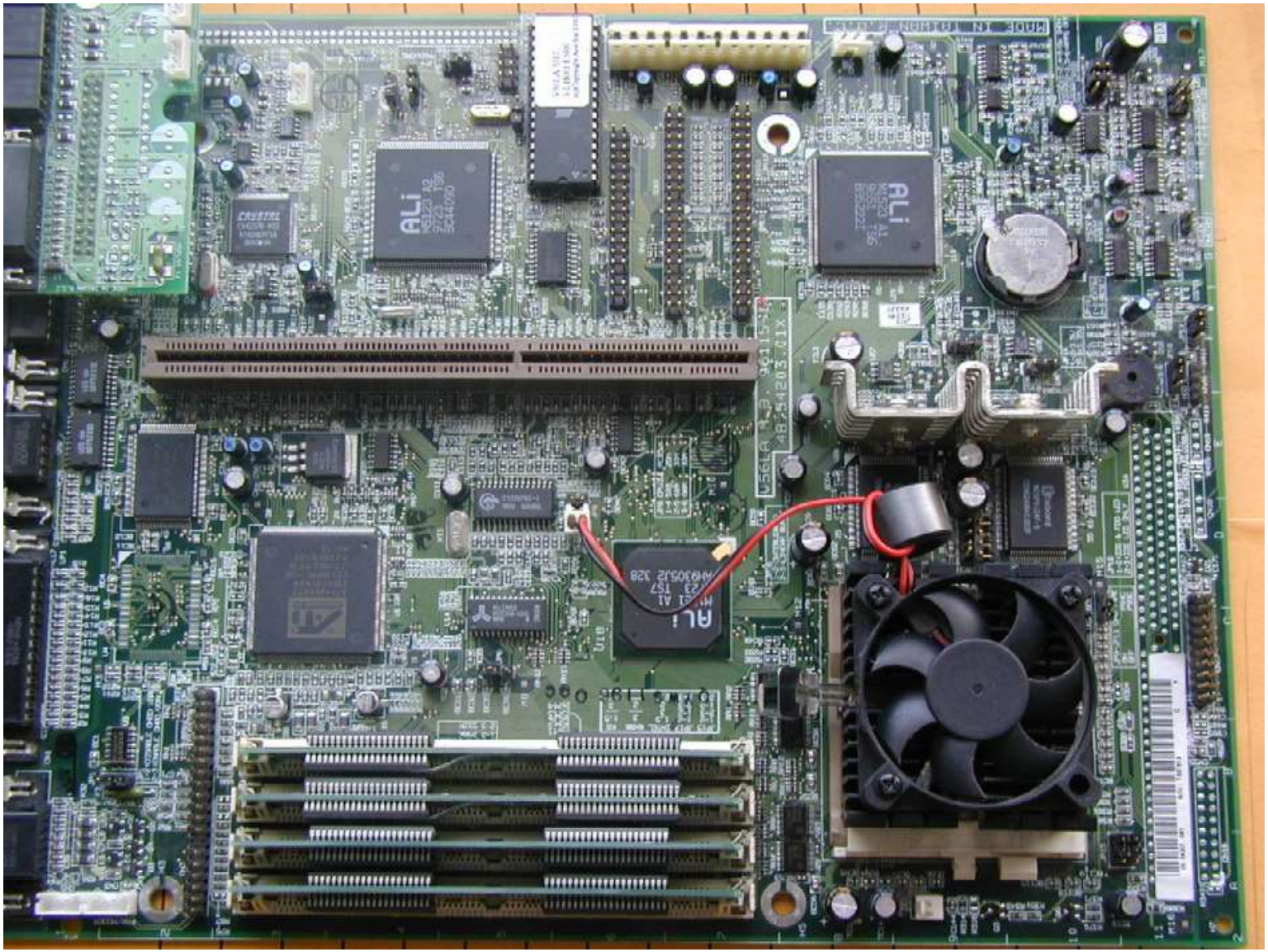
- Zazwyczaj jak mówimy o pamięci komputera, to mamy na myśli RAM (ale jest jeszcze HD – Hard Drive)
- Program jest zapisany w pamięci RAM, co znaczy, że sekwencja instrukcji jest kodowana jako bajty w kolejnych komórkach pamięci RAM, zaczynając od ustalonego adresu
- Dane (wartości zmiennych) są zapisywane gdzie indziej w RAM, i nie koniecznie w kolejnych komórkach (zależy to od systemu operacyjnego)
- Instrukcje oraz dane są dostępne w RAM poprzez swoje adresy

Computer Architecture



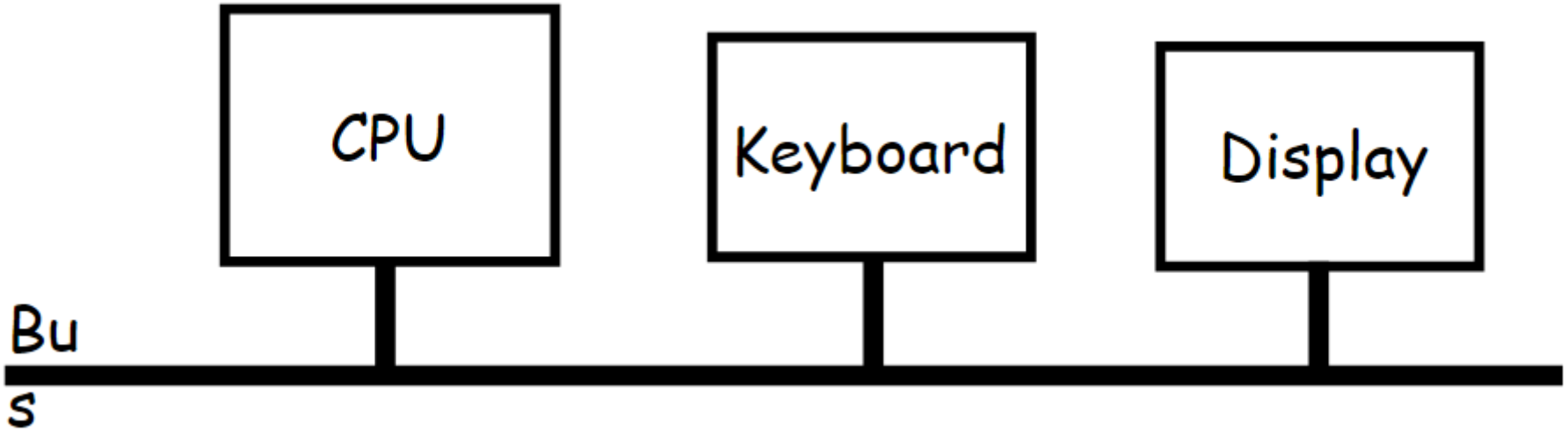
Computer Architecture



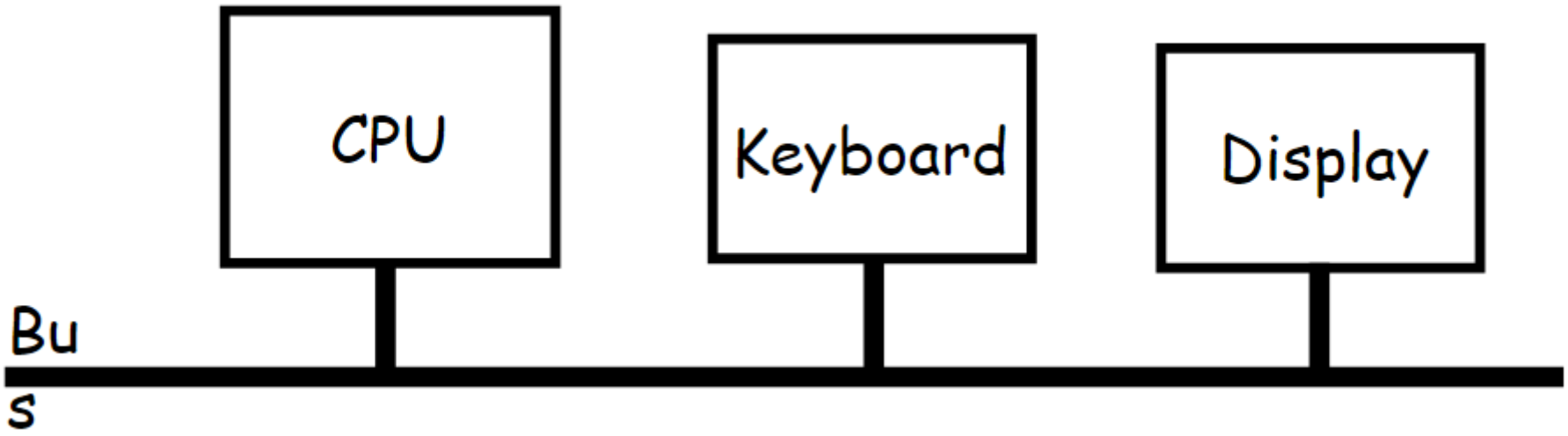

ALi
ALi
ALi

The Bus (magistrala lub szyna)

- co to jest?
- prosty i szybki sposób do komunikacji, tj. przesyłania bajtów pomiędzy urządzeniami
- taka “autostrada” do przesyłania informacji
- a w zasadzie jako “pojemnik” dostępny dla wielu urządzeń

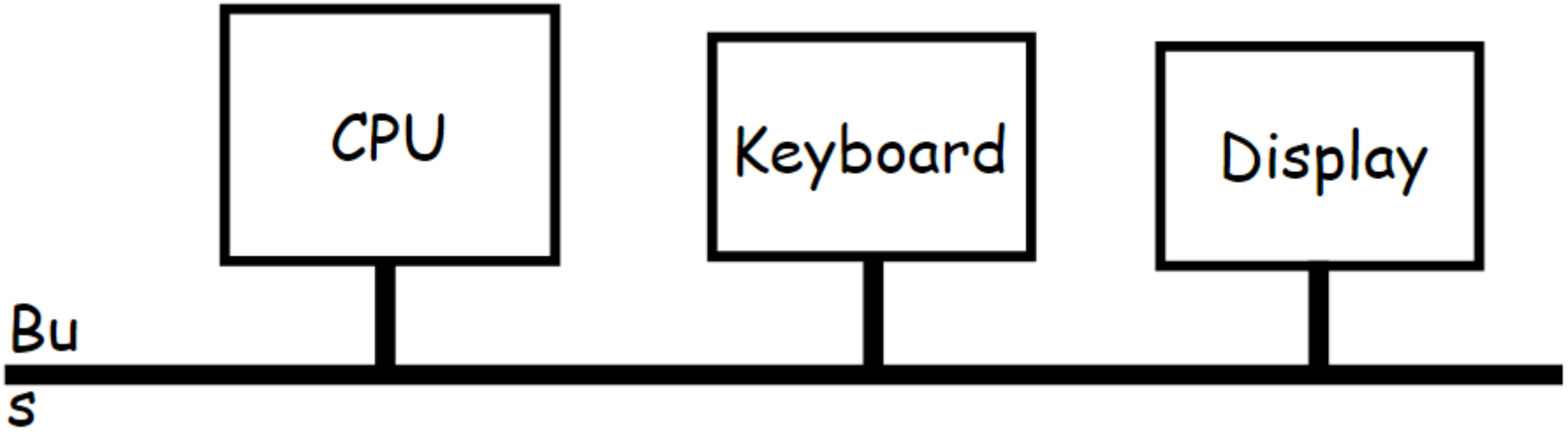


The Bus



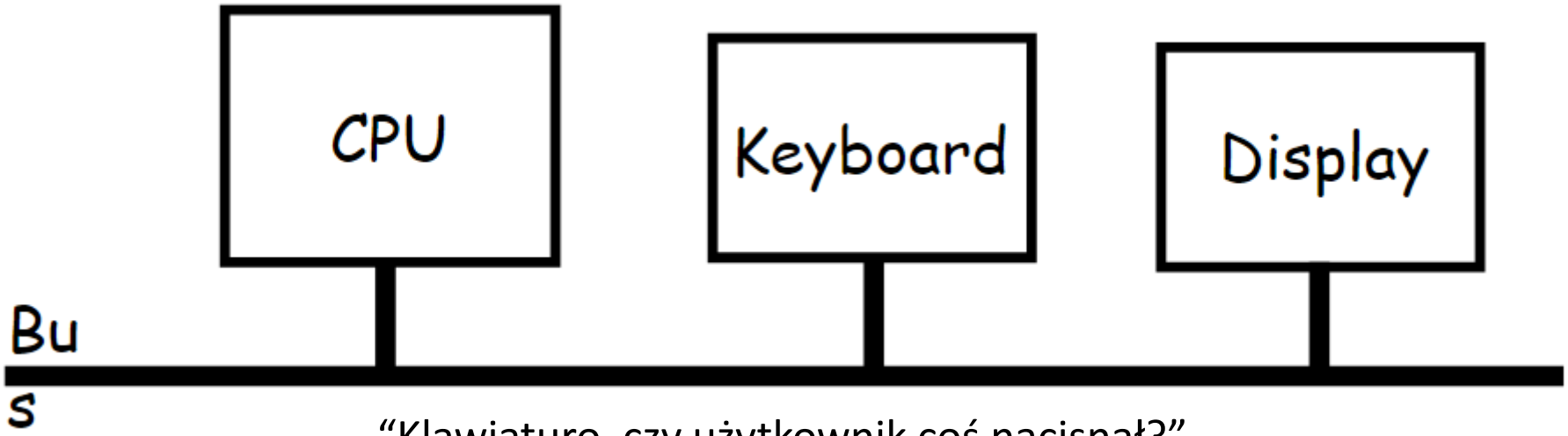
Magistrala

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



Magistrala

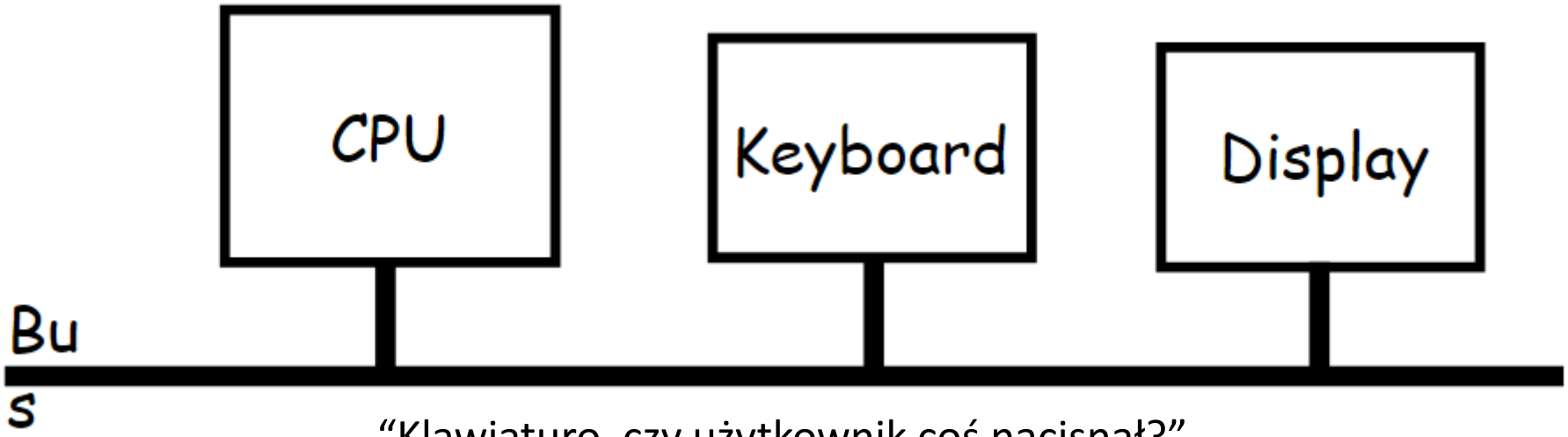
- CPU wstawia na magistrale zapytanie “Klawiaturo, czy użytkownik coś nacisnął?”



“Klawiaturo, czy użytkownik coś nacisnął?”

Magistrala

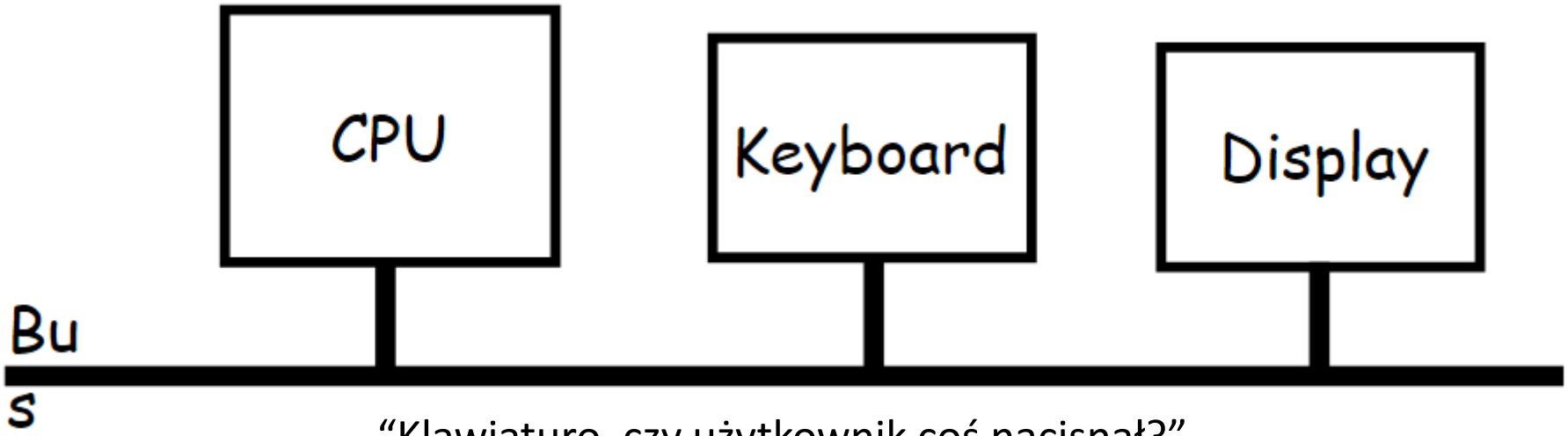
- Każde urządzenie bez przerwy sprawdza co jest na magistrali, i czy to jego dotyczy



“Klawiaturo, czy użytkownik coś nacisnął?”

Magistrala

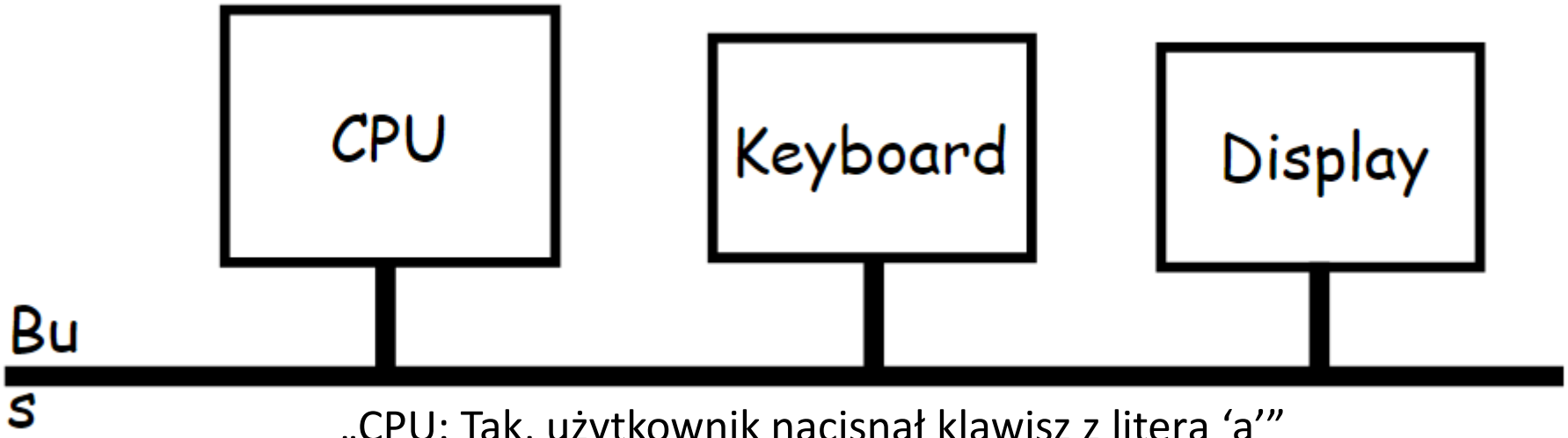
- Klawiatur czyta co jest aktualnie na magistrali i zauważa, że to dotyczy jej. Pozostałe urządzenia ignorują te wiadomość, bo ich nie dotyczy



“Klawiaturo, czy użytkownik coś nacisnął?”

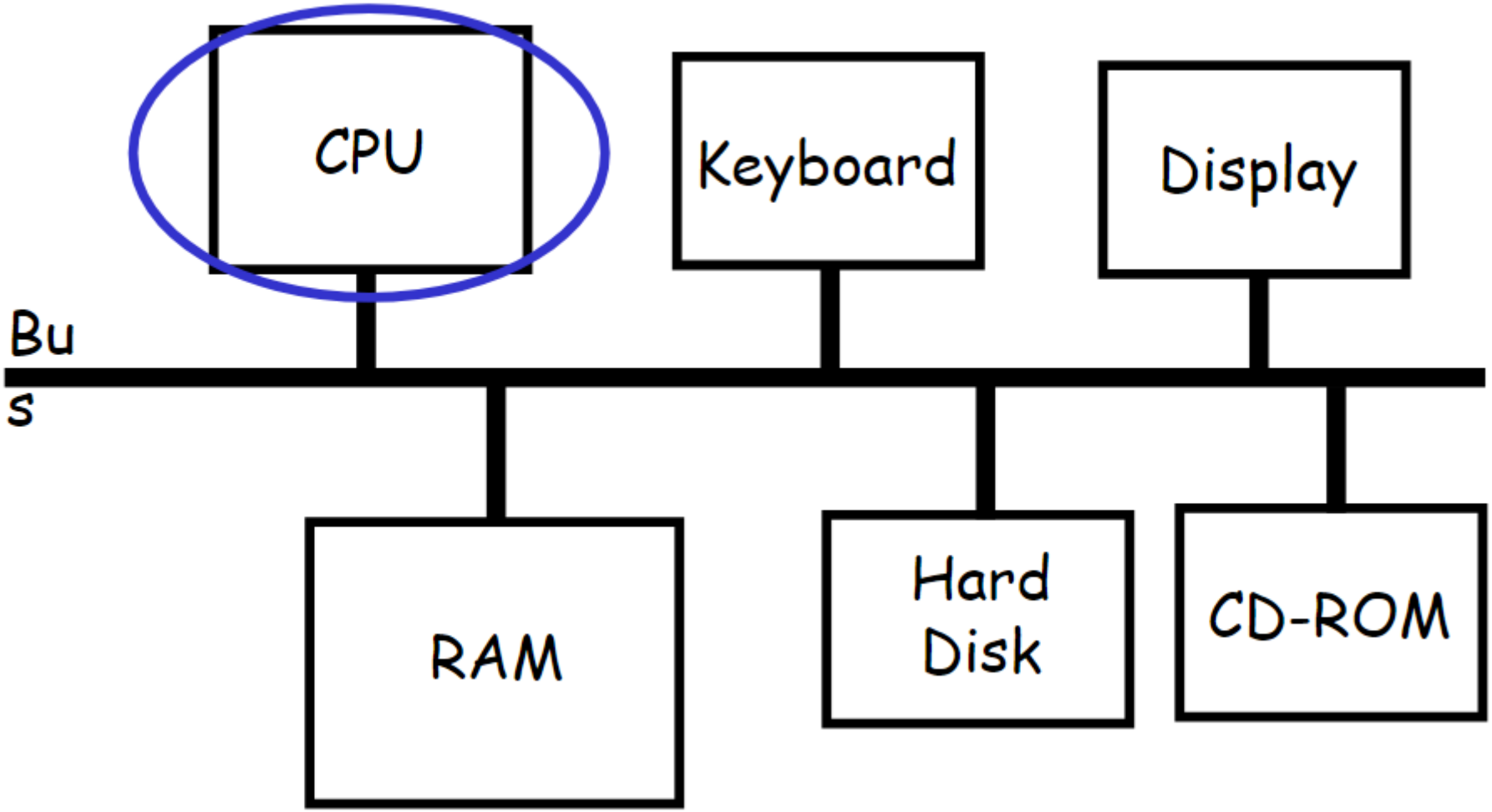
Magistrala

- CPU czyta co jest na magistrali i w ten sposób dostaje odpowiedź od Klawiatury



„CPU: Tak, użytkownik nacisnął klawisz z literą ‘a’”

Architektura komputera



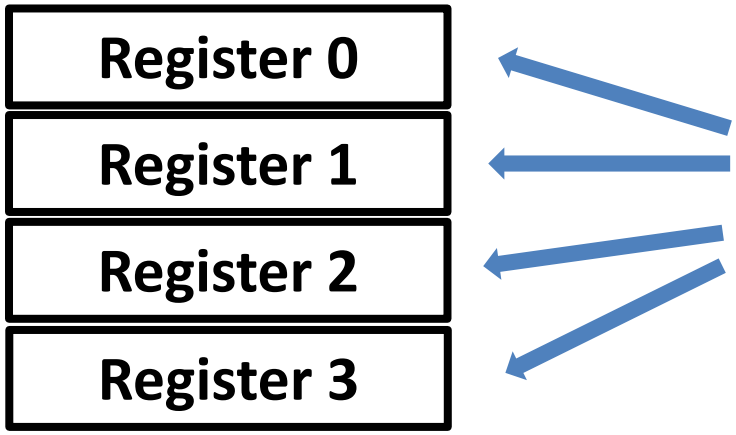
CPU

- CPU jest „mózgiem” komputera
- To tutaj wykonywane są kolejne instrukcje programu

- Zobaczmy co jest w środku

CPU (cd.)

Memory Registers

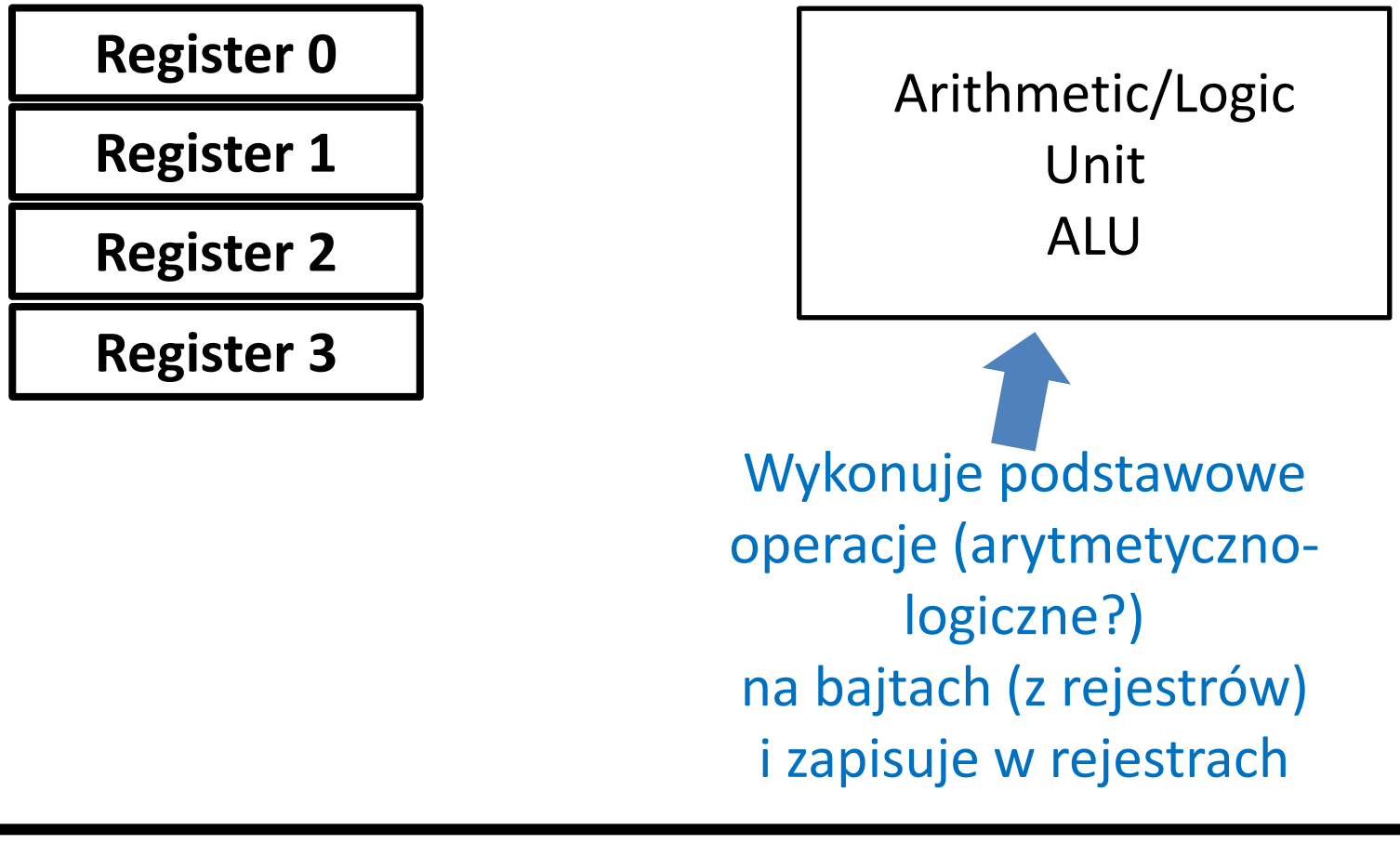


Pamięć tymczasowa.
komputer “ładuje” dane z RAM (lub z innych urządzeń) do rejestrów, wykonuje operacje na tych danych a następnie zapisuje wyniki do RAM lub wysyła do urządzeń.

Z poprzedniego przykładu: “czytaj wartość zmiennej A z pamięci; czytaj wartość zmiennej B; wykonaj A+B; wynik zapisz w zmiennej.” Czytanie odbywa się do rejestrów. Operacja dodawania jest wykonana na rejestrach; wynik jest też zapisany w rejestrze.

CPU (cd.)

Memory Registers



CPU (cd.)

Memory Registers

Register 0

Register 1

Register 2

Register 3

Arithmetic/Logic
Unit
ALU

Instruction Register

Tutaj jest aktualna instrukcja programu do wykonania

CPU (cd.)

Memory Registers

Register 0

Register 1

Register 2

Register 3

Arithmetic/Logic Unit
ALU

Instruction Register

Instr. Pointer (IP)

Tutaj jest przechowywany adres (w RAM) aktualnej instrukcji wykonywanego programu

CPU (cd.)

Memory Registers

Register 0

Register 1

Register 2

Register 3

Arithmetic/Logic Unit
ALU

Instruction Register

Instr. Pointer (IP)

Control Unit
(State Machine)

The Control Unit (jednostka kontrolna)

- Działaniem procesora (CPU) steruje Control Unit

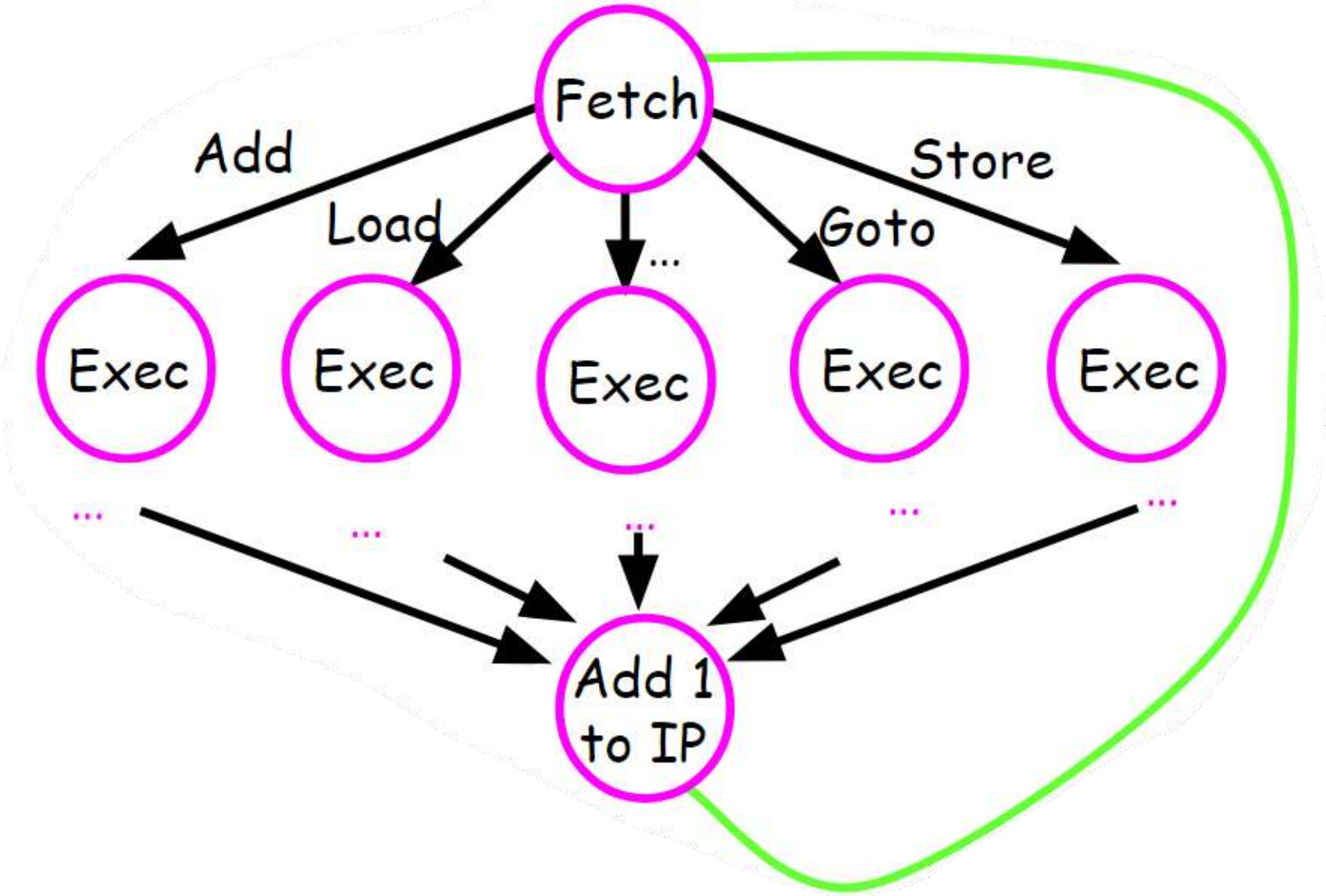
- Na czym to polega?

The Control Unit

- Struktura Control Unit State Machine jest prosta i składa się z następujących 3 kroków:

1) Fetch: pobierz z RAM instrukcje, której adres jest zapisany w rejestrze IP, i zapisz ją w Instruction Register (IR)
2) Execute: wykonaj aktualną instrukcję z IR; instrukcji jest niewiele i są proste
3) Powtarzaj: dodaj 1 do adresu w rejestrze IP (jeśli jest to instrukcja skoku, to może być inne IP) i przejdź do kroku 1

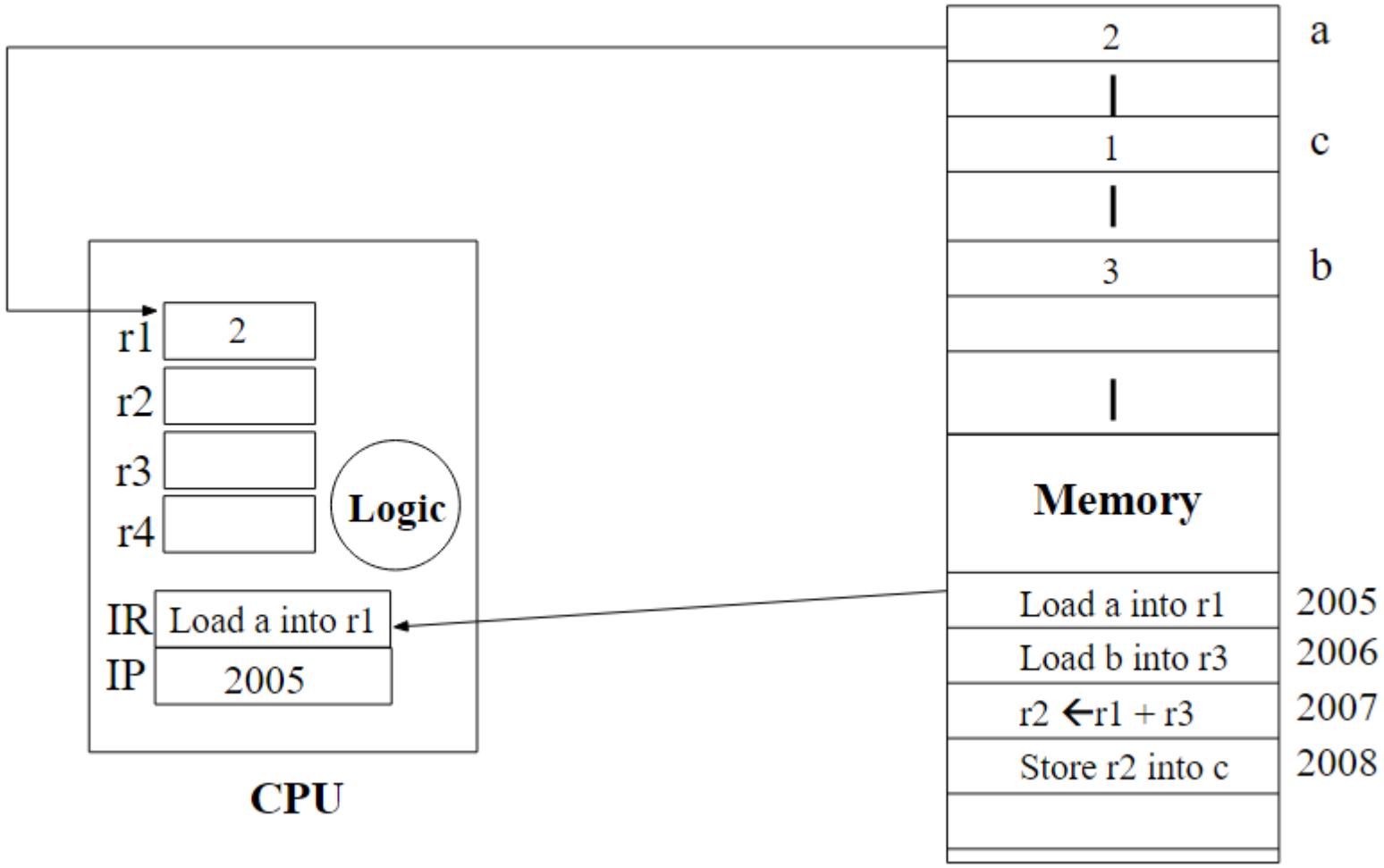
The Control Unit jako State Machine (automat)



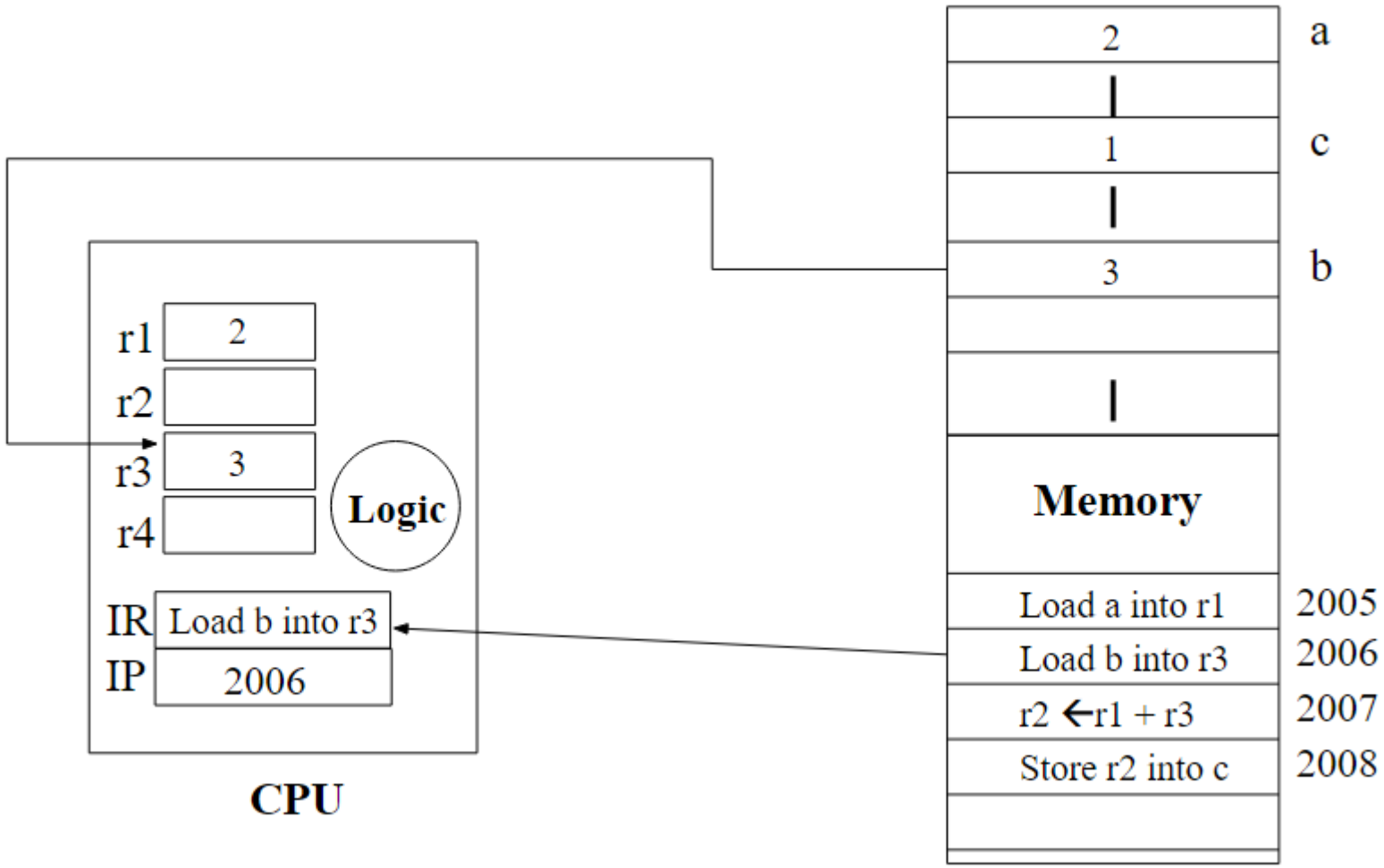
Prosty Program

- Dodaj wartości zmiennych a oraz b.
- Wynik zapisz na zmiennej c, tj, c ◄ a+b
- sekwencja instrukcji (program):
  - zapisz wartość zmiennej a do rejestru r1
  - zapisz wartość zmiennej b do rejestru r3
  - wykonaj r2 ◄ r1 + r3
  - zapisz wartość z rejestru r2 do zmiennej c

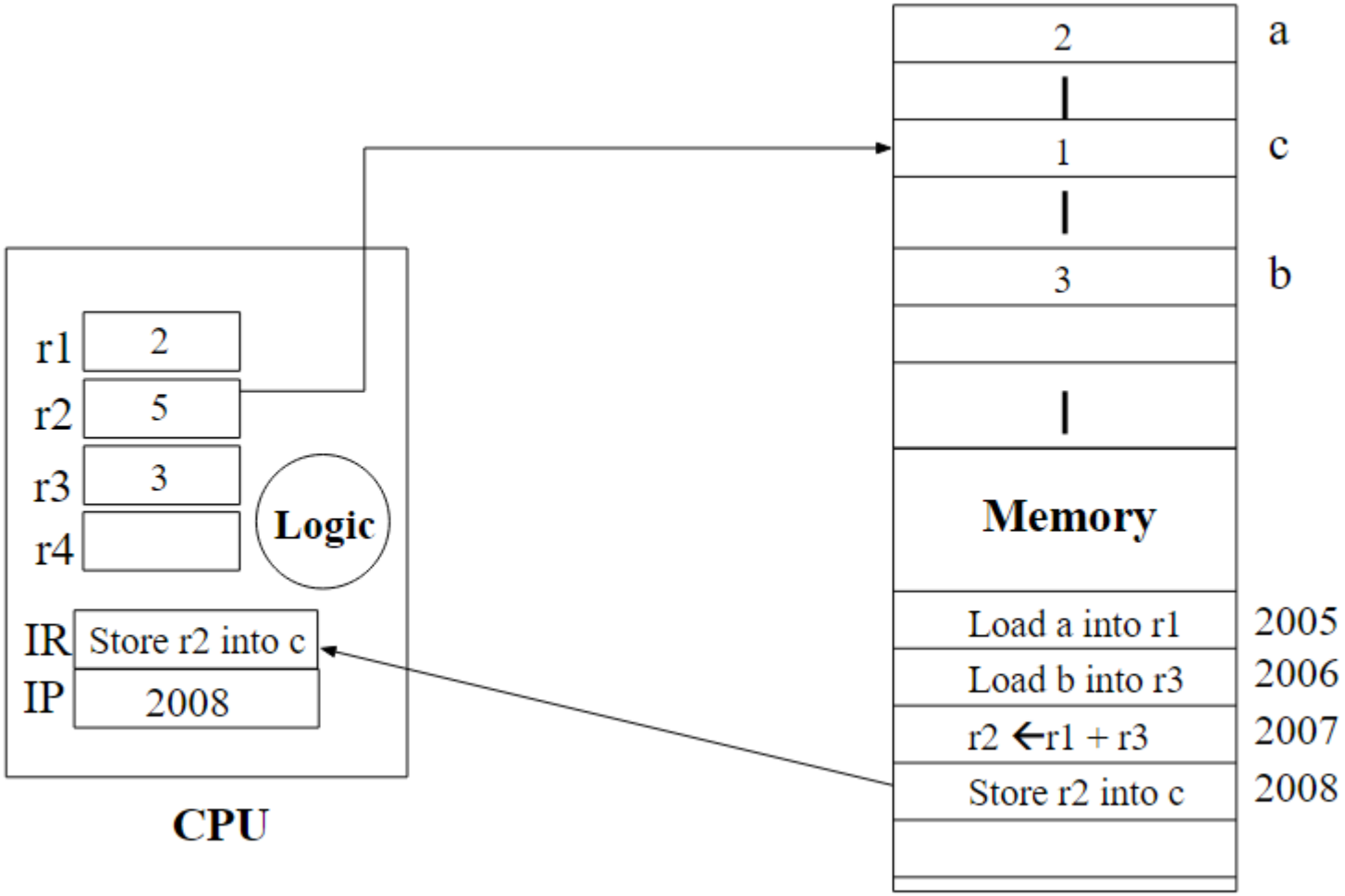
Wykonanie Programu



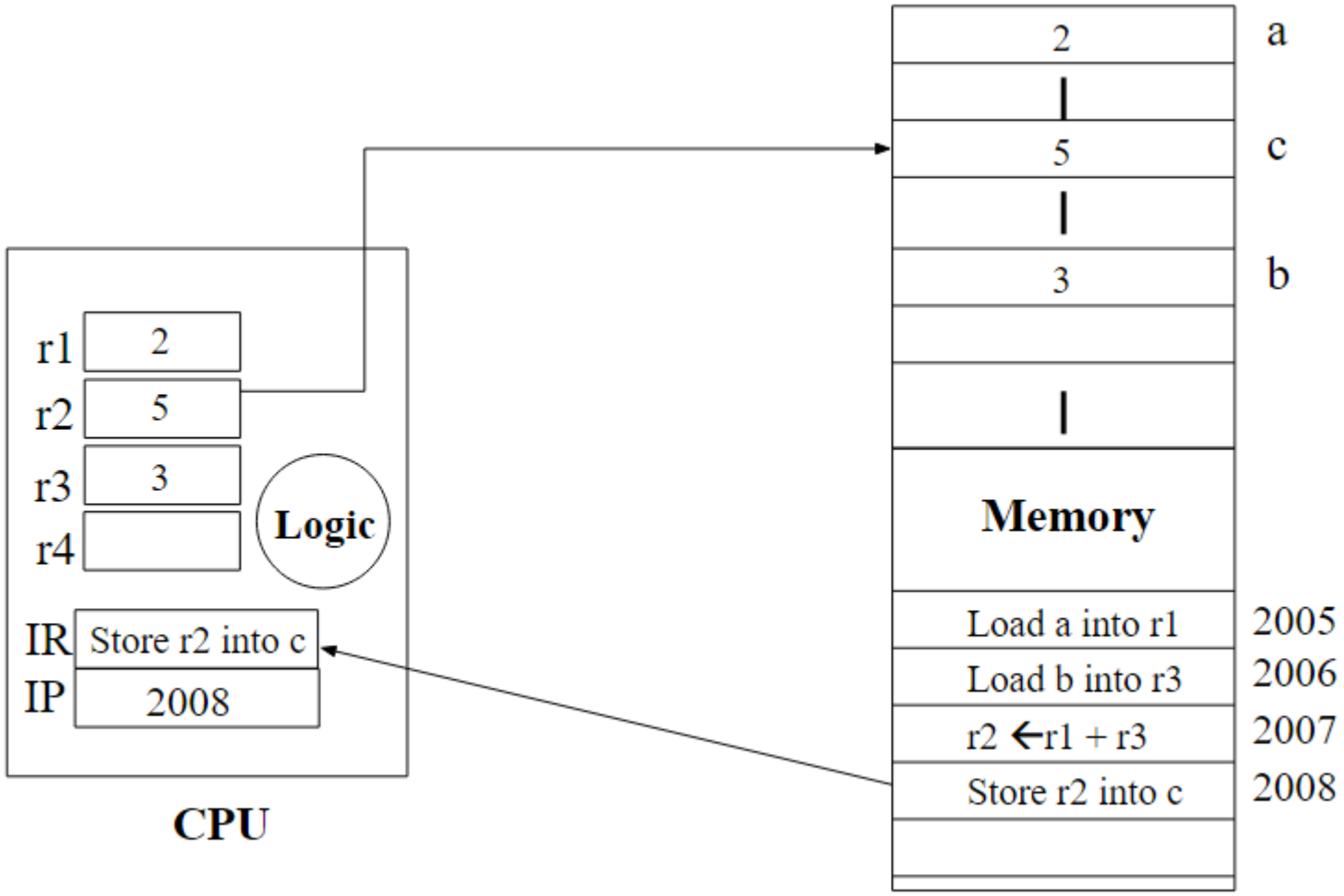
Wykonanie Programu



Wykonanie Programu

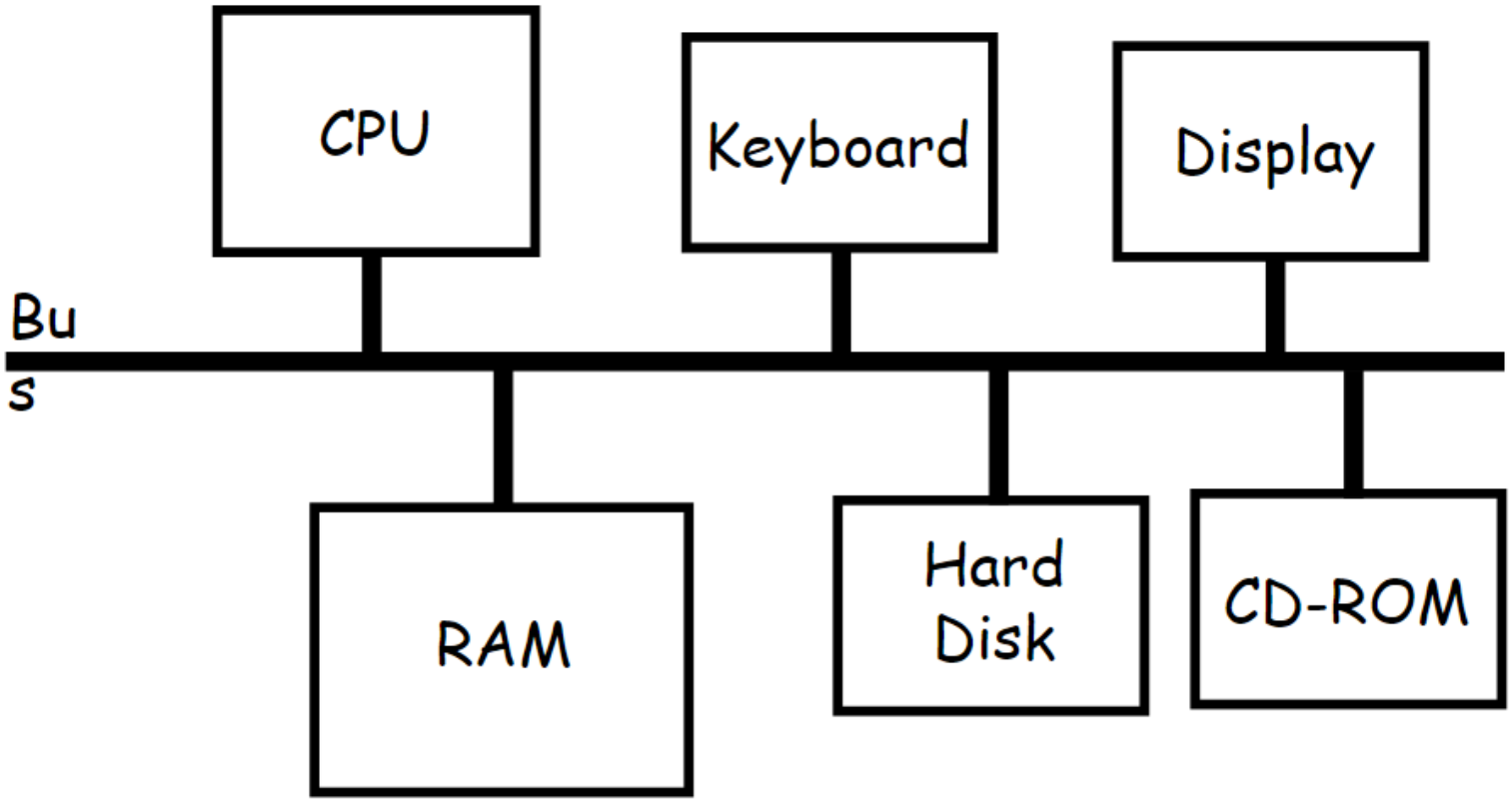


Wykonanie Programu



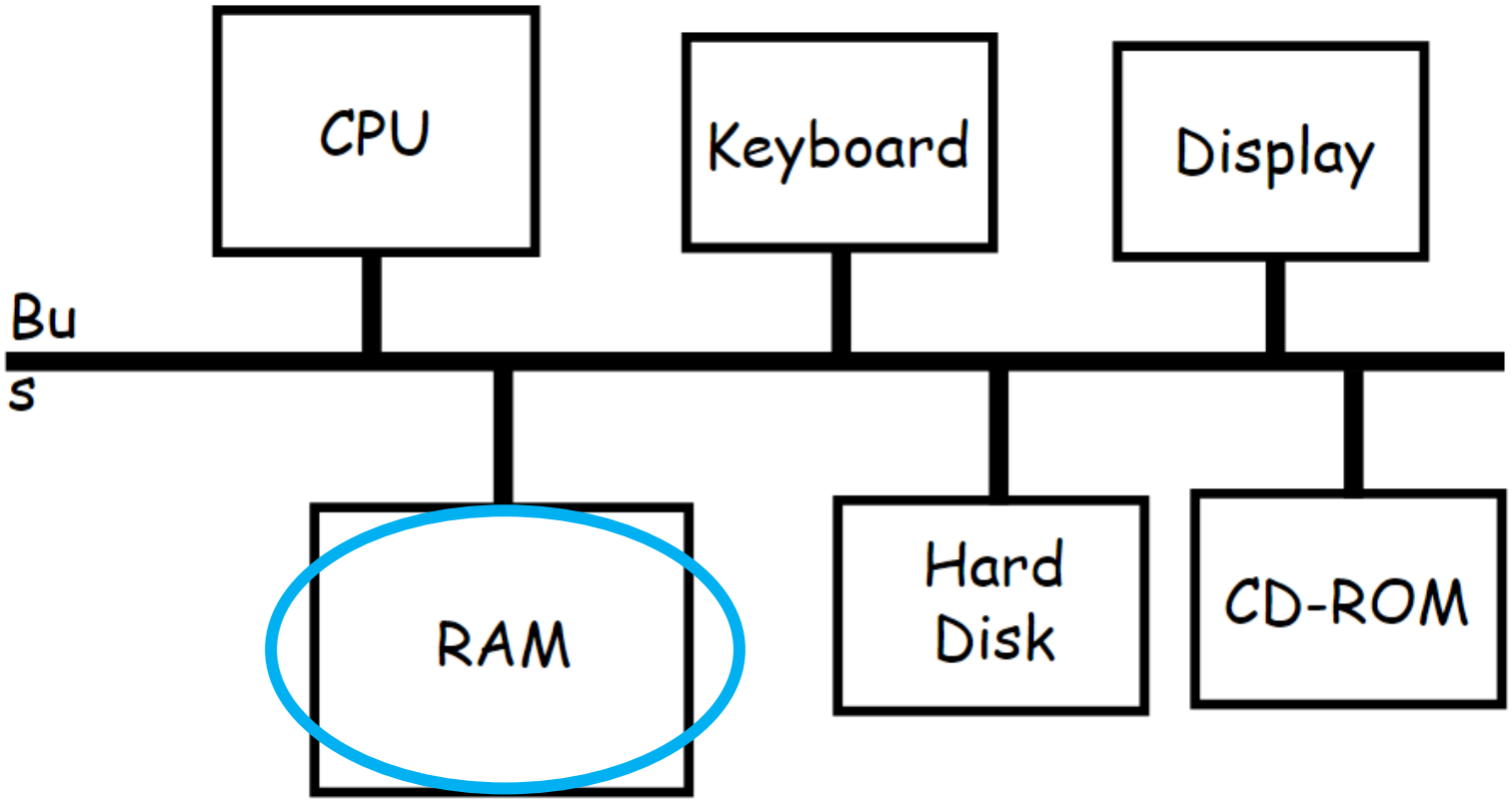
Poskładajmy te elementy

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



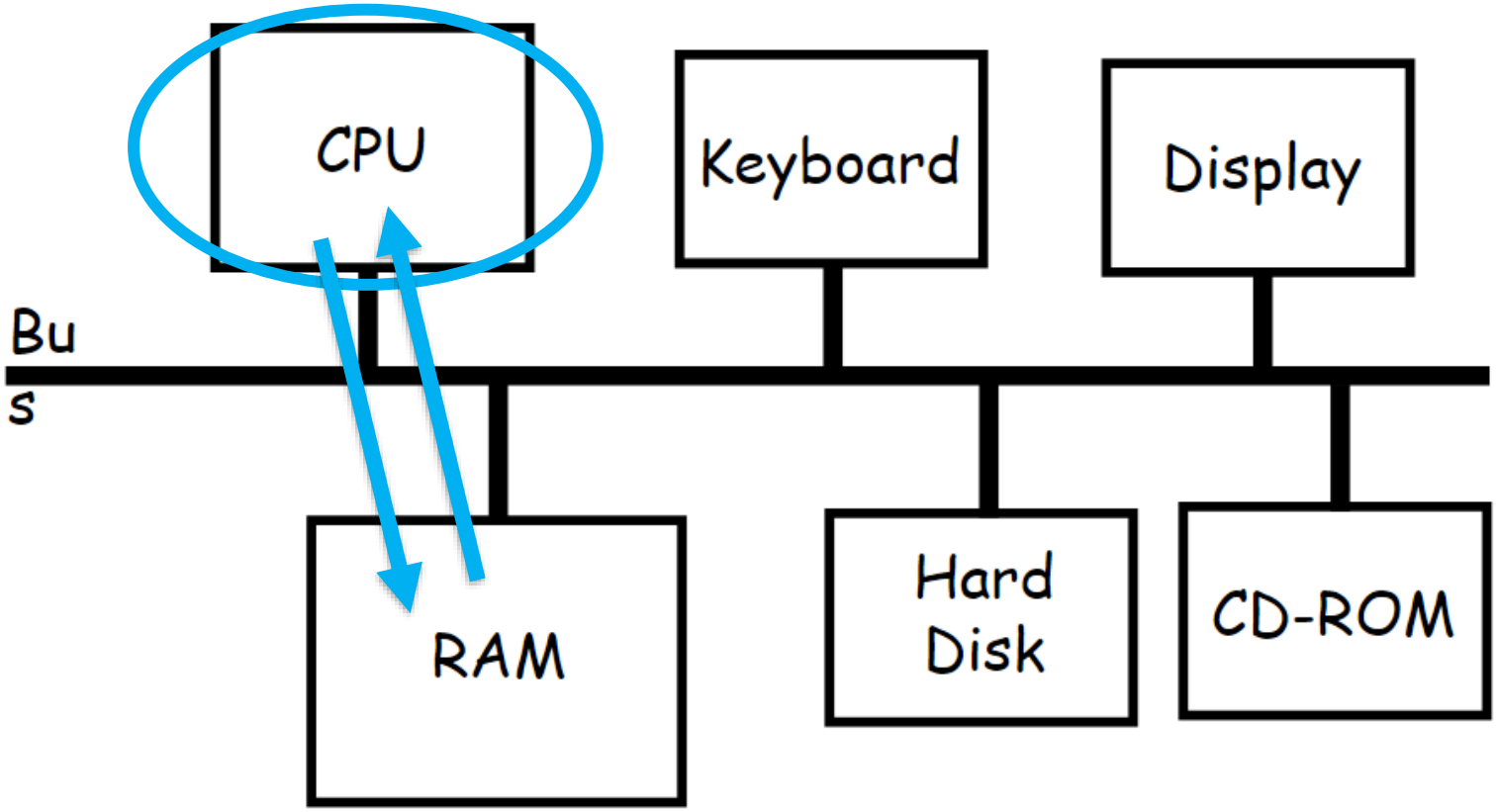
Poskładajmy te elementy

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



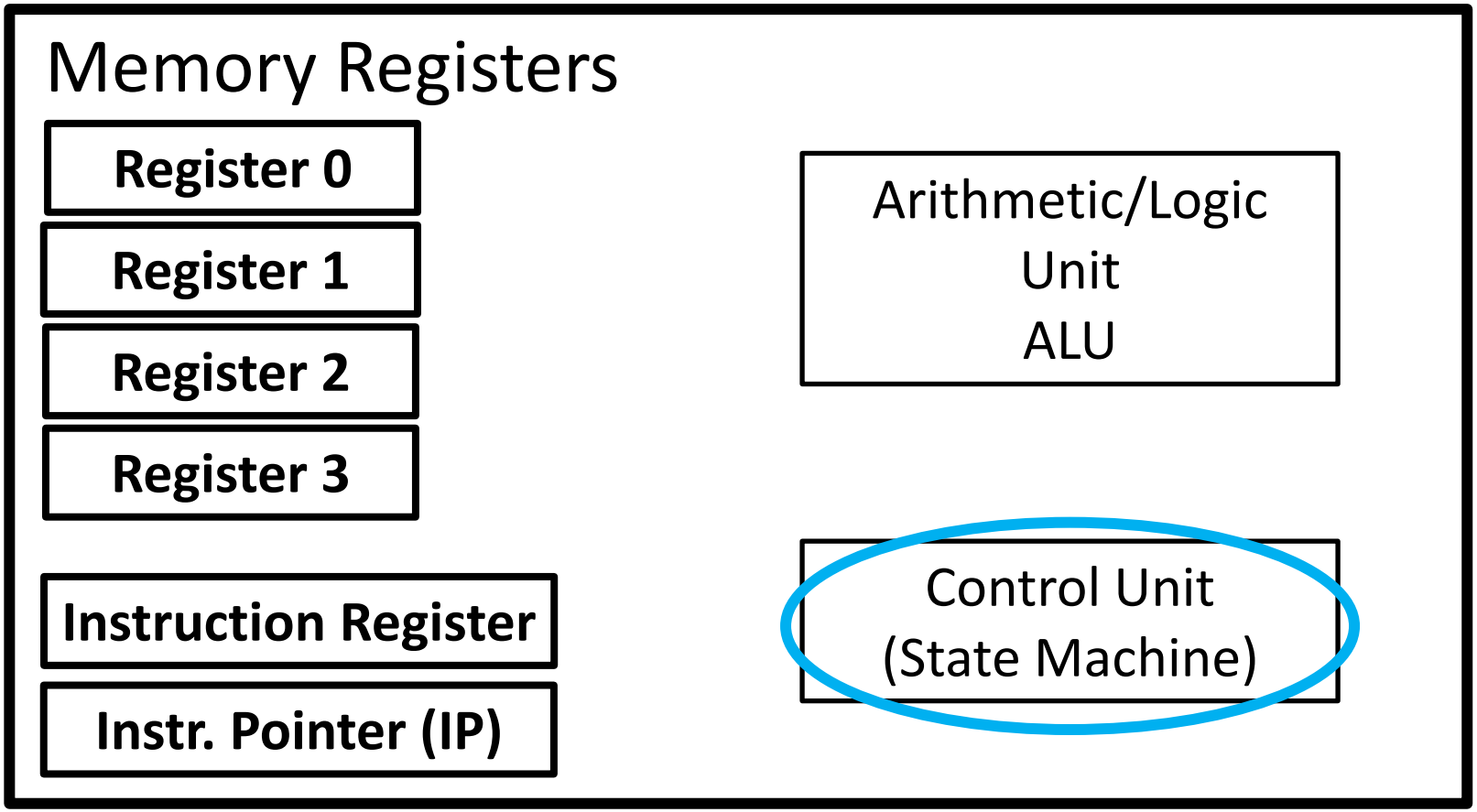
Poskładajmy te elementy

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



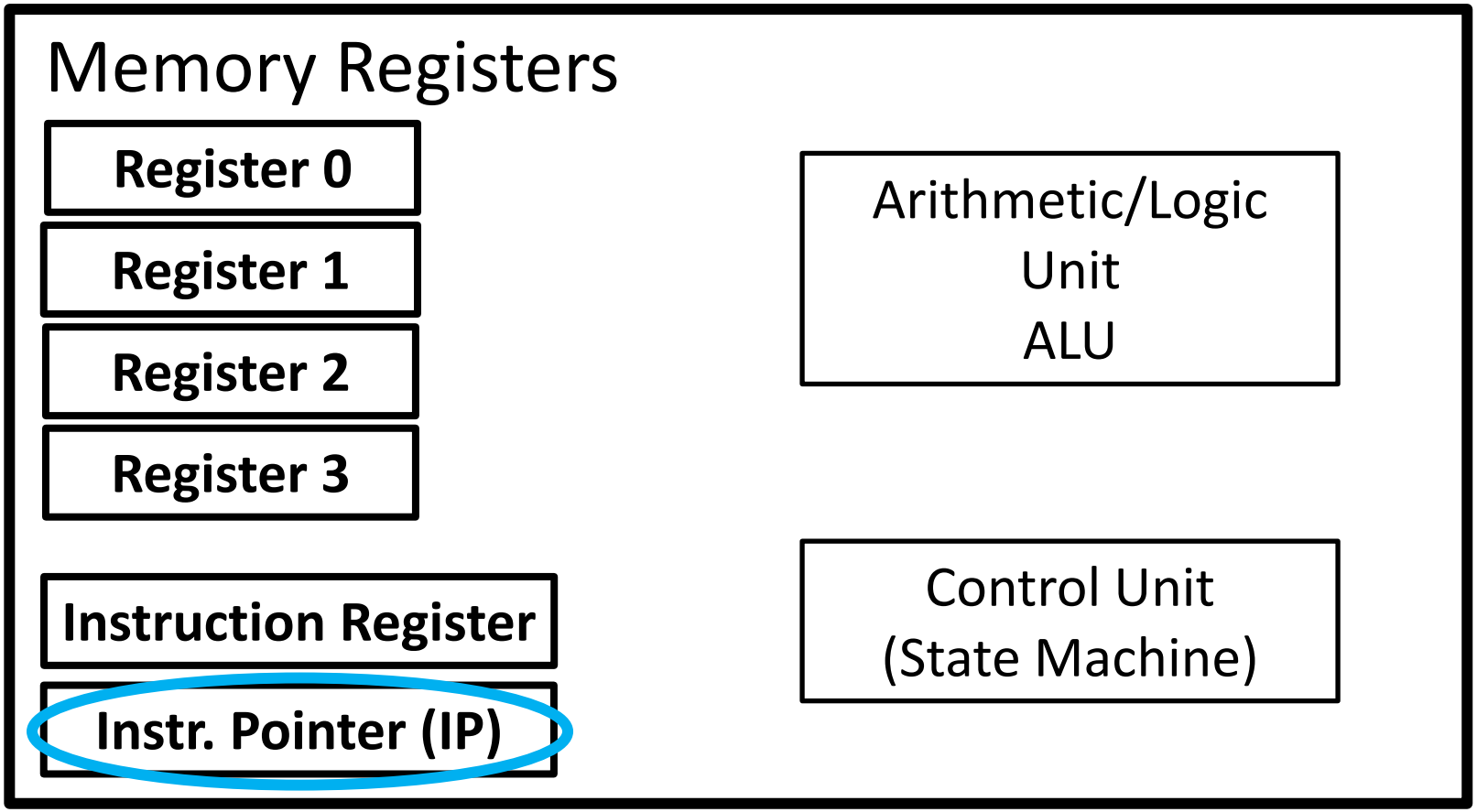
Poskładajmy te elementy

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



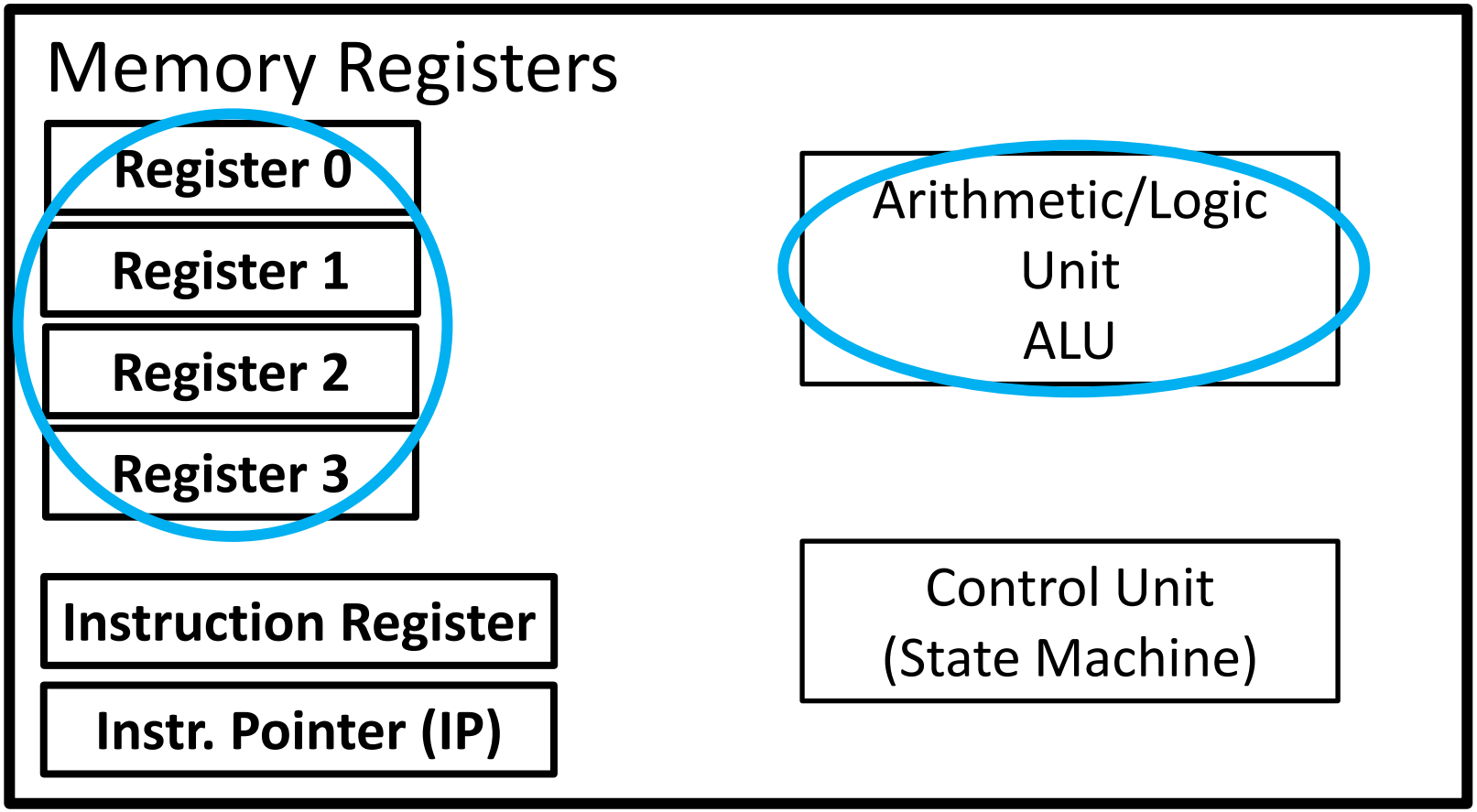
Poskładajmy te elementy

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



Poskładajmy te elementy

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



Programowanie

Programy i instrukcje

- Program składa się z sekwencji instrukcji
- CPU wykonuje jedną instrukcje w każdym cyklu zegarowym
- współczesne procesory wielordzeniowe wykonują trochę więcej, ale na razie nie jest to istotne
- Poziomy programowania:
- Najniższy poziom: Machine language
- Pośredni poziom: Assembly language
- Do tworzenia aplikacji : High-level programming language

Programy i instrukcje

W językach programowania wysokiego poziomu każda instrukcja jest dekodowana na wiele instrukcji niskiego poziomu

- np. c ← a + b jest kompilowana do

Load a into r1
Load b into r3
r2 ◄ r1 + r3
Store r2 into

Assembly language: specyfikuje instrukcje niskiego poziomu do postaci mnemonicznej

- np. Load r1, a

Machine language: instrukcje są ciągami bitów

- np. 1101101000001110011

Machine/Assembly Language

- Załóżmy, że mamy maszynę, która wykonuje instrukcje
- Zasadnicze Pytanie:
- Jakie instrukcje?
- Jak te instrukcje maja się do sprzętu (computer hardware)?

Complex vs Simple Instructions

- Procesory, które używają złożonego zestawu instrukcji:
  - CISC = Complex Instruction Set Computer
  - 20 lat temu, prawie wszystkie procesory były CISC
- W latach 80-tych XX wieku wprowadzony został zestaw RISC:
  - RISC = Reduced Instruction Set Computer

Complex vs Simple Instructions

- RISC = Reduced Instruction Set Computer
  - mniej instrukcji
  - ale prostszych i łatwiejszych do zaprojektowania CPU
  - wystarczających, żeby z nich złożyć wszystko co potrzeba, tj. niezbędne złożone instrukcje

- Okazało się, że, dla wieku ważnych aplikacji, procesory oparte na RISC są bardziej wydajne niż te oparte na Complex Instruction Set Computer (CISC)

Complex vs Simple Instructions

- Tym niemniej, np. Pentium był jeszcze oparty na CISC!
- Dlaczego?: kompatybilność ze starym oprogramowaniem, w tym systemami operacyjnymi
- Nowe rodzaje aplikacji (multimedia) są bardziej wydajne, jeśli są wykonywane na specjalnych pół-autonomicznych układach, np. karty graficzne
- Obecnie technologie informacyjne są zbyt złożone, żeby ograniczać się tylko do RISC versus CISC

Typowe instrukcje w asemblerze

1) “Load” – skopiuj zawartość komórki RAM i wstaw do jednego z rejestrów

2) “Load Direct” – wstaw słowo (np. bajt) do jednego z rejestrów

3) “Store” - zapisz słowo z rejestru do pamięci RAM

4) “Add” - dodaj zawartości dwóch rejestrów a wynik wstaw do trzeciego rejestru itp.

Typowe instrukcje w asemblerze

5) “Compare” - porównaj czy wartość w jednym rejestrze jest większa niż w drugim rejestrze. Jeśli tak, to wstaw do Rejestru r0 wartość “0”, jeśli nie to wstaw wartość “1”.

6) “Jump” - Jeśli wartość w Rejestrze r0 jest “0”, to zmień wartość w rejestrze IP (Instruction Pointer) na wartość, która jest aktualnie w ustalonym rejestrze

7) “Branch” - Jeśli wartość w rejestrze r1 jest większa niż w rejestrze r2, to wstaw do rejestru IP ustaloną wartość

Machine Languages

- Różne rodzaje procesorów posiadają różne zestawy instrukcji
  - Pentium family / Celeron / Xeon / AMD K6 / Cyrix … (Intel x86 family)
  - PowerPC (Mac)
  - DragonBall (Palm Pilot)
  - StrongARM/MIPS (WinCE)
  - Wiele innych (specjalizowanych i uniwersalnych)
- Instrukcje są różnie kodowane w assembly/machine languages

Prosty przykład CPU

- Uproszczony CPU i odpowiadający mu Machine Language
- Nasz CPU posiada:
  - 8 Rejestrów – każdy po 16 bitów (2 bajty)

Prosty przykład CPU

- Uproszczony CPU i odpowiadający mu Machine Language
- Nasz CPU posiada:
  - 8 Rejestrów – każdy po 16 bitów (2 bajty)
- RAM:
  - Można czytać i wpisywać (loads and stores) w blokach po 16 bitów (2 bajtów)
  - potrzebne jest 28 = 256 adresów
  - Wielkość tej pamięci to 256*2 = 512 bajtów

Zapisywanie w pamięci

- Będziemy operować 16-bitowymi ciągami
- np. 0110110010100101

- Trudno zapamiętać takie ciągi, więc
  - używać będziemy szestnaskowego (heksadecymalnego) systemu do kodowania danych oraz instrukcji

Hexadecimal

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

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

Hexadecimal

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

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

Hexadecimal

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

np.

0110 1100 1010 0101

6 C A 5

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

Machine “Core”

- Wszystko jest binarne (hex)

Registers

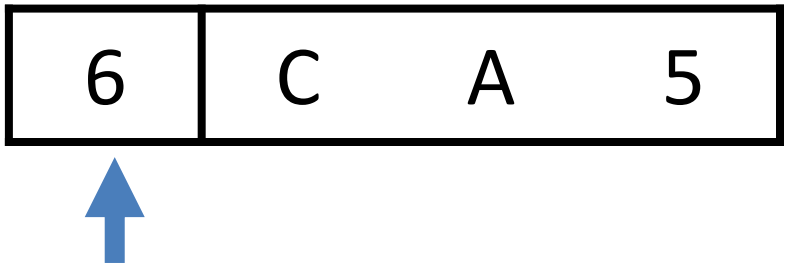
```
R0: 0000
R1: 0CA8
R2: A9DB
R3: 0705
R4: 1011
R5: 90A0
R6: 0807
R7: 00A0
```

Memory

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

Kodowanie Instrukcji

- Maszynowe instrukcje to również liczby 16 bitowe
- Dzielimy te bity na grupy:



- Pierwsze 4 bity są nazwane Op-Code (kody operacji) :
  - wskazują na typ instrukcji
  - Ile jest możliwych typów na 4 bitach?

Instrukcje

- 16 możliwych Op-Codes (kodów operacji):
- Zapoznamy się tylko z niektórymi…

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

Halt Instruction

- Opcode 0: Halt
- Instrukcja stopu, tj. zatrzymania działania
- pozostałe 12 bity są ignorowane, np.

0	0	0	0
0	F	F	F
0	9	A	C

Dają taki sam efekt

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

Data Instructions

- Opcode B: Load Direct / Address
- Wstawia do Rejestru ustaloną wartość (Specified Value)
- kod:

0	F	F	F

Efekt: w Rejestrze A jest ta wartość

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

Data Instructions

- Opcode B: Load Direct / Address
- Wstawia do Rejestru ustaloną wartość (Specified Value)
- kod:

B	regA	value (8 bits)

Efekt: w Rejestrze A jest ta wartość

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

Arithmetic/Logic Instructions

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

1	regA	regB	regC

Efekt: R wartość w rejestrze A wynosi regB + regC

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

Dodawanie liczb

- Prosty program

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

Algorytm:

Na początku ‘licznik’ R1 ustawiamy na 1.

Dodajemy ‘licznik’ R1 do ‘wynik’ R2 i większamy za każdym razem ‘licznik’ R1 o 1

Jeśli R1=6, to halt (kończymy)

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

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

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

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

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

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

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

Arithmetic/Logic Instructions

- Opcode 2: Subtract (odejmij)
- Podobne do Add…
- kod:

2	regA	regB	regC

Efekt: Register A ma wartość
Register B - Register C

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

Control Instructions

- Opcode 6: Jump if Positive
- Jump (skocz) do komórki RAM pod adres jeśli Register > 0
- kod:

6	regA	address (8 bits)

Efekt: Register A ma wartość
Register B - Register C

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

Co zrobić żeby uprościć program

- Te same instrukcje sześciokrotnie w naszym prostym programie dodającym liczby od 1 do 15 (hex).
  - A jeśli w programie jest dodawanie liczb od 1 15000?
  - Jak sobie wtedy poradzić?
- Rozwiązanie: pętle (Loops)
  - Za pomocą instrukcji jump (lub branch)
  - Warunkowo jest zmieniana wartość rejestru IP do wcześniejszej instrukcji w programie

Dodawanie liczb- jeszcze raz

- Użyj petli Loop, z pomocą instrukcji jump po to żeby policzyć 1 + 2 + 3 + 4 + 5 + … + N

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

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

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

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

- Uwaga: zmniejszamy zamiast zwiększać dodawane liczby

Instrukcje kontrolne

- Opcode 7: Jump and count (backwards)
- Skocz do instrukcji pod adresem jeśli regA > 0,
- AND zmniejsz wartość Register o 1
- kod:

7	regA	address (8 bits)

Efekt: Jeśli Register A > 0,
Go To (przejdź) do instrukcji pod adresem oraz A := A - 1

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

Dodawanie liczb - jeszcze jeden raz

- Inna pętla Loop za pomocą instrukcji Jump & Count dla

  1 + 2 + 3 + 4 + 5 + … + N

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

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

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

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

Nie jest potrzebny rejestr R0 do przechowywania wartości (zmniejszanej/ zwiększanej o 1). To co trzeba jest już w instrukcji Jump&Count

Instrukcje do operowania na RAM

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

7	regA	address (8 bits)

Efekt: zapisz z RAM do Register A

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

Instrukcje do operowania na RAM

- Opcode A: Store (do pamięci)
- Zapisz wartość Register do RAM
- kod:

A	regA	address (8 bits)

Efekt: zapisz wartość
Register A do RAM pod adres

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

Instrukcje kontrolne

- Opcode 9: Load (z pamięci)
- Skopiuj z RAM do Register
- kod:

9	regA	regB	regC

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

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

Instrukcje kontrolne

- Jak CPU rozróżnia te dwie instrukcje ?

9	regA	address (8 bits)

9	regA	regB	regC

- CPU ma 8 Rrejestrów i potrzebuje, tylko 3 bitów do zakodowania Register A
- Pozostałe bity są użyte żeby wskazać na typ LOAD

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

Hacks

- To jest tzw. „hack” - hak!
- Haki mogą być niebezpieczne! (nie wszystkie). Ogólnie, powinny być unikane w programowaniu
- Pozwalają na włamania, tj. dostęp do zastrzeżonych danych w pamięci
- Niestety, są one powszechne
- Niektórzy programiści (hakerzy) lubią haki
- Haki są główna przyczyną powstawania „pluskwy” w kodzie!

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

Wirusy!

- Program modyfikuje (replikuje) sam siebie!

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

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

nadpisywane są kolejno linie 15, 14, 13, 12, 11, 10 na to co jest pod adresami 0F, 0E, 0D, 0C, 0B, 0A

Wirusy!

- Program modyfikuje (replikuje) sam siebie!

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

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

nadpisywane są kolejno linie 15, 14, 13, 12, 11, 10 na to co jest pod adresami 0F, 0E, 0D, 0C, 0B, 0A

Wirusy!

- Program modyfikuje (replikuje) sam siebie!

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

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

nadpisywane są kolejno linie 15, 14, 13, 12, 11, 10 na to co jest pod adresami 0F, 0E, 0D, 0C, 0B, 0A

Wirusy!

- Program modyfikuje (replikuje) sam siebie!

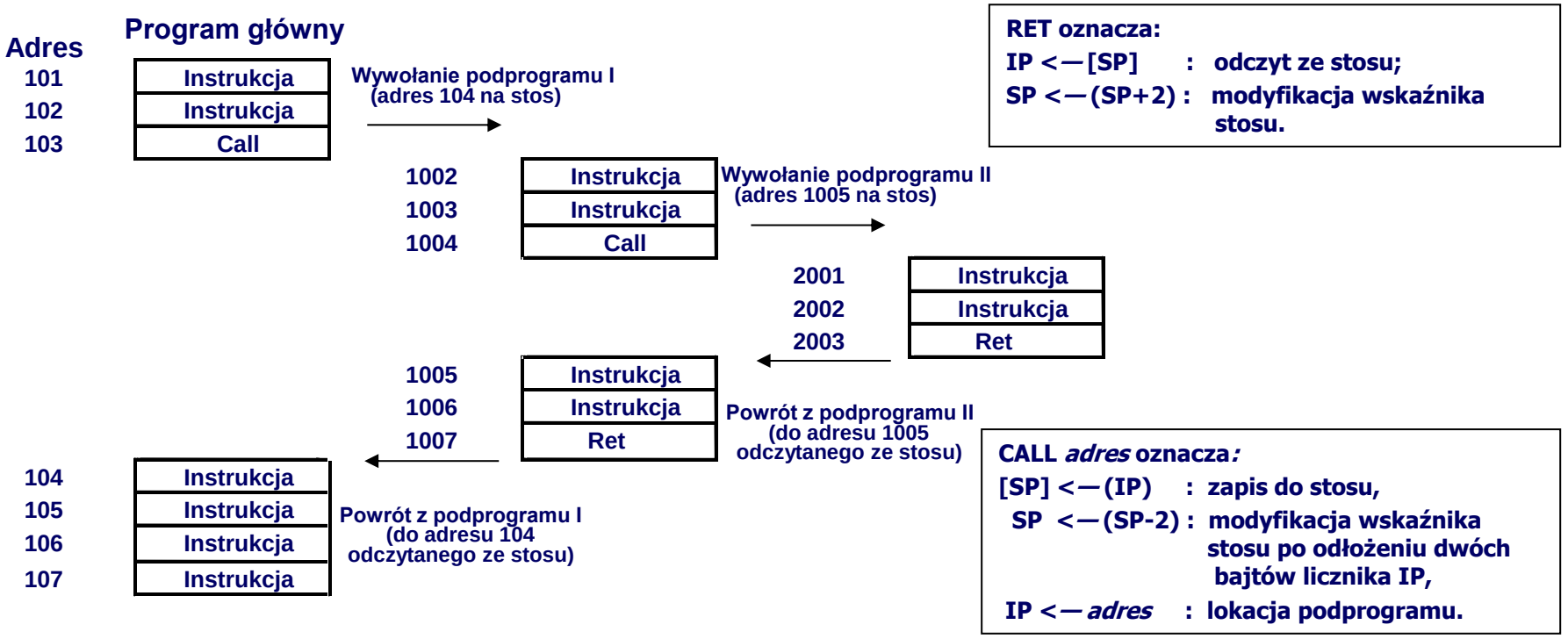
```
0A: Load R1 □  0005            (długość kodu do replikacji -1)
0B: Load R2 □  0010            (koniec tego kodu +1)
0C: Load R3 □  000A            (początek tego kodu)

0D: Load R0 □  Address[R3+R1]  (wstaw instrukcję o adresie 0F do R0)
(w następnych krokach będą to poprzednie)
0E: Store Address[R2+R1] □  R0 (zapisz tę instrukcję pod adres 15)
(następne instrukcje zapisuj o 1 wyżej)
0F: If (R1>0), Jump to 0D (jeśli w R1 nie ma zera, to skocz do 0D
                and decrease R1        i zmniejsz R1 o 1)
10: Load R1 □  0005
11: Load R2 □  000F
12: Load R3 □  000A
13: Load R0 □  Address[R3+R1]
14: Store Address[R2+R1] □  R0
15: If (R1>0), Jump to 0D and decrease R1
...  ____________________________________________________________
..: . . . .                    (dalsza część kodu)
..: . . . .
```

nadpisywane są kolejno linie 15, 14, 13, 12, 11, 10 na to co jest pod adresami 0F, 0E, 0D, 0C, 0B, 0A

STOS

- Stosem nazywamy wyróżniony obszar w pamięci używany wg reguł:
  - informacje zapisane są na stos do kolejnych komórek (pod kolejnymi adresami), przy czym żadnego adresu nie wolno pominąć
  - odczytujemy informacje w kolejności odwrotnej do ich zapisu
  - informacje odczytujemy z ostatnio zapełnionej komórki, natomiast zapisujemy do pierwszej wolnej
- Czyli obowiązuje reguła LIFO - ostatni wchodzi pierwszy wychodzi



Rejestr IP i rejestr FLAGOWY

- IP (wskaźnik rozkazów)- rejestr ten zawiera zawsze offset pamięci, w którym zawarty jest następny rozkaz do wykonania. Bazowy adres segmentu kodu zawarty jest w rejestrze CS. Stąd pełny adres logiczny wykonywanego rozkazu wskazywany jest parą rejestrów CS:IP.
- Oznacza to, że jeśli wykonywany jest rozkaz to wskaźnik rozkazów ustawiany jest do następnego adresu pamięci, pod którym znajduje się rozkaz do wykonania. Wyjątek stanowią rozkazy wywołania i skoku.
- FLAGS (rejestr znaczników, rejestr flagowy)- rejestr ten jest zbiorem poszczególnych bitów kontrolnych (znaczników), które wskazują wystąpienie określonego stanu procesora.

Znaczniki stanu	**=1**	**=0**
OF - flaga nadmiaru (przepełnienia)	**OV (over)**	**NV (not over)**
SF - flaga znaku	**NG (negative)**	**PL (plus)**
ZF - flaga zera	**ZR (zero)**	**NZ (not zero)**
PF - flaga parzystości	**PE (parity even)**	**PO (parity odd)**
Znaczniki kontrolne		
TF - flaga pracy krokowej		

Podsumowanie

- Był to opis (zgrubny) typowego języka maszynowego i jego instrukcji
  - Halt Instruction
  - Data Instructions
  - Arithmetic/Logic Instructions
  - Control Instructions
  - Memory Instructions

Podsumowanie (cd.)

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

Historia języków programowania

- Fortran (1954) John Backus IBM, do obliczeń naukowych (głównie w Fizyce) ale też atomowych i termojądrowych
- Cobol (1959) (akronim od ang. common business-oriented language) – wysokopoziomowy język programowania stworzony i używany do tworzenia aplikacji biznesowych. COBOL jest językiem imperatywnym, proceduralnym, oraz od 2002 roku, obiektowym.
- Algol (1958) bardziej uniwersalny niż Fortran
  - nadal są skoki w formie „go to”
- Lisp (1958) programowanie na listach
- A Programming Language (APL) – język programowania wysokiego poziomu, znany ze swojej zwięzłości i możliwości generowania macierzy. Opracował go w połowie lat 60. Kenneth E. Iverson

Historia języków programowania

- Algol + Fortran → PL/1 (1964)
- Basic (1964) dla każdego
- Simula (1967) + Algol → Smalltalk (1969) – obiektowy
- BCPL → B → C (1971)
- Algol → Pascal (1971) → Modula 1,2,3,

Historia języków programowania

- C++ (1983) = C z obiektami
  - C jest nadal używany
- Awk (1978) → Perl (1987)
  - Web programming language
- Java (1991)
  - Web applets
- Visual Basic(1991) macros and programs
  - Core of Microsoft systems

Jaki powinien być dobry jezyk programowania?

- Łatwy do kodowania
- Chroni przed popełnianiem błędów
- Obsługuje debugowanie, gdy jest to potrzebne
- Ma wszechstronny zestaw narzędzi

Big number bug (błąd dużej liczby)

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

Pentium II bug

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

Kropka zamiast przecinka

- NASA Mariner 1 , Venus probe (1962)
- Miał to być pierwszy amerykański statek kosmicznym, który by odwiedził inną planetę. Podczas startu 22 lipca 1962 po czterech minutach zachowywał się nieregularnie.
- Został zniszczony poprzez wewnetrzną procedurę.
- Przyczyna: kropka zamiast przecinka w pętli DO w kodzie FORTRAN-owym

Błąd kontroli przepływu

- AT&T usługa sieciowa dalekiego zasięgu nie działa przez dziewięć godzin (zła instrukcja BREAK w kodzie C)

- 15 stycznia 1990 r .: 70 milionów ze 138 milionów klientów w USA straciło usługi. Koszt ATT wynosił od 75 do 100 milionów USD (plus utrata reputacji).

Błąd zarządzania strukturą danych

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

Meltdown (podatność na zagrożenia)

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

Podsumowanie

- Programowanie jest trudne
- Programiści muszą:
  - dokładnie zrozumieć zadanie
  - przewidzieć wszystkie możliwości
  - pisać dobry (?) kod
  - przewidzieć, co mogą zrobić użytkownicy (trudne)
- Języki (i frameworki) programowania pozwalają używać narzędzi do budowania kodu . Tam też moga być błędy
- Koszt błędu może być bardzo duży
- Nie istnieje prawo Moore'a dotyczące oprogramowania.

Dziękuję za uwagę!

Slajdy na podstawie wykładów prof. Stanisława Ambroszkewicza