1. Co to jest architektura komputera?

Architektura komputera to swego rodzaju „umowa”, jak komputer ma wyglądać „od strony programisty”:

- Jakie ma **zestawy instrukcji** (ISA – Instruction Set Architecture).
- W jaki sposób procesor adresuje pamięć (np. różne tryby adresowania).
- Jak wygląda współpraca z urządzeniami wejścia-wyjścia.

Organizacja komputera to natomiast **szczegóły sprzętowe**: jak fizycznie wykonano te instrukcje, jak zbudowano układy logiczne itd.

2. Trzy główne elementy komputera

Każdy komputer (od laptopa, przez smartfona, po superkomputer) ma:

1. **CPU (procesor)** – wykonuje rozkazy, liczy, podejmuje decyzje.
2. **Pamięć** – przechowuje dane i kod programów (np. RAM, pamięć podręczna, dyski).
3. **I/O** (wejście/wyjście) – klawiatura, mysz, drukarka, monitor i inne urządzenia peryferyjne.

Te elementy współpracują zwykle przez **magistrale** (szyny), czyli wspólne linie danych/adresów/sygnałów sterujących.

3. Model von Neumanna – podstawa w większości komputerów

1. **Wspólna pamięć** na dane i instrukcje.
2. Jedna główna magistrala do komunikacji z pamięcią.
3. Instrukcje i dane są traktowane (w dużej mierze) tak samo – procesor pobiera je, posługując się adresami pamięci.

Zaletą jest prostota, wadą – tzw. *von Neumann bottleneck* (wąskie gardło przy częstych operacjach na pamięci).

4. Architektura Harwardzka – alternatywa dla von Neumanna

1. **Oddzielne** pamięci (lub przynajmniej oddzielne magistrale) na instrukcje i dane.
2. Można **równocześnie** pobierać kod i dane, co bywa szybsze (np. w układach wbudowanych – mikrokontrolerach).
3. W komputerach PC spotyka się „mieszane" podejście (np. wspólna pamięć, ale oddzielny **cache** instrukcji i danych).

5. Cykl rozkazowy: „pobierz-dekoduj-wykonaj"

Każdy procesor realizuje instrukcje w rytmie:

1. **Fetch** – pobiera rozkaz z pamięci (adres rozkazu znajduje się w rejestrze PC – Program Counter).
2. **Decode** – sprawdza, co to za instrukcja (kod operacji, tryb adresowania).
3. **Execute** – wykonuje polecenie (np. dodawanie w ALU, skok, przesłanie danych).
4. **Write-back** (o ile potrzeba) – zapisuje wynik w rejestrze lub pamięci.

Ten cykl powtarza się dla każdej instrukcji.

6. CPU – co skrywa w środku?

6.1 Rejestry

- **Rejestry ogólnego przeznaczenia** (np. R0, R1, R2…) – do przechowywania tymczasowych danych w trakcie obliczeń (szybki dostęp!).
- **Rejestr licznika rozkazów (PC)** – wskazuje adres kolejnej instrukcji do wykonania.
- **Rejestr instrukcji (IR)** – przechowuje aktualnie dekodowany rozkaz.
- **Rejestr flag / statusowy (PSW)** – informacje o wynikach (czy wyszło zero, czy był przeniesiony bit, czy wystąpiła flaga przepełnienia itd.).

6.2 ALU (Arithmetic Logic Unit)

- Wykonuje operacje arytmetyczne (dodawanie, odejmowanie, mnożenie, dzielenie) i logiczne (AND, OR, NOT).

- Dostaje „polecenie” z Jednostki Sterującej (CU), którą operację ma zrobić.

6.3 Jednostka Sterująca (Control Unit, CU)

- „Dyrygent” całego CPU: pobiera instrukcje z pamięci, dekoduje je, steruje sygnałami do ALU i rejestrów.
- Odpowiada za to, by kolejne etapy cyklu rozkazowego przebiegały właściwie.

7. Magistrale (szyny)

Wewnątrz komputera zwykle mamy **trzy główne** typy magistral:

1. **Szyna danych** – przenosi rzeczywiste wartości danych między CPU, pamięcią i I/O.
2. **Szyna adresowa** – procesor wystawia tutaj adres komórki pamięci lub portu I/O, z którym chce pracować.
3. **Szyna sterująca** – przesyła sygnały typu „odczyt”, „zapis”, sygnały przerwań itd.

8. Sposoby komunikacji z urządzeniami – port-mapped I/O vs. memory-mapped I/O

1. **Port-mapped I/O**
   - Masz specjalne instrukcje (np. IN, OUT), które dotyczą portów wejścia/wyjścia.
   - Urządzenia I/O siedzą w „odrębnej” przestrzeni niż zwykła pamięć.
2. **Memory-mapped I/O**
   - Urządzenia I/O widoczne są pod **normalnymi** adresami pamięci.
   - Używasz LOAD, STORE (czy ekwiwalentów) do obsługi urządzeń, tak jakbyś czytał/zapisywał zwykłe komórki RAM.

9. Tryby adresowania – różne sposoby wskazania operandu

Wyobraź sobie, że szukasz książki w bibliotece. Możesz:

1. **Adresowanie bezpośrednie** – w instrukcji jest wprost numer półki (adres pamięci).

2. **Adresowanie pośrednie** – instrukcja mówi: „idź do kartki X, a tam masz dopiero właściwy adres".

3. **Adresowanie rejestrowe** – operand jest już w rejestrze (np. ADD R1, R2).

4. **Adresowanie natychmiastowe** – wartość jest „zaszyta" w instrukcji (np. ADD R1, #5).

5. **Adresowanie indeksowe** – instrukcja używa rejestru plus offset, przydatne przy tablicach (np. ADD R1, [R2 + 4]).

10. Architektury bezadresowa, jednoadresowa i dwuadresowa

1. **Bezadresowa (stosowa)**

   • Operujesz na stosie (push, pop, ADD działa na szczycie stosu).

   • Kod bywa dłuższy, bo dużo operacji push/pop.

2. **Jednoadresowa**

   • Typowy przykład to **akumulator**: ADD X oznacza „akumulator = akumulator + X".

   • Instrukcja ma jeden adres, bo drugi jest domyślny (akumulator).

3. **Dwuadresowa**

   • Przykład: ADD R1, R2 → R1 = R1 + R2.

   • Kod potrafi być krótszy, bo jedna instrukcja mówi, skąd brać dane i gdzie zapisać wynik.

11. RISC (Reduced Instruction Set Computer)

1. **Mały, prosty zestaw instrukcji** (każda „robi" raczej niewiele).

2. Operacje arytmetyczne zwykle tylko między rejestrami (load/store do pamięci – osobne rozkazy).

3. Przykłady: **MIPS**, **ARM**, **RISC-V**.

4. Zalety: łatwość budowy **potoku** (pipeline), szybkość wykonania pojedynczej instrukcji.

12. CISC (Complex Instruction Set Computer)

1. **Bogaty zestaw instrukcji** (niektóre rozkazy potrafią wykonywać złożone operacje, np. pobrać operand z pamięci, policzyć i zapisać do pamięci).
2. Architektury: **Intel x86**, AMD.
3. Kod źródłowy bywa krótszy w asemblerze (mniej instrukcji), ale sprzęt jest bardziej złożony (potokowanie trudniejsze).
4. W praktyce dzisiejsze x86 rozbijają duże instrukcje na mikrooperacje podobne do RISC.

13. Reprezentacja liczb całkowitych: Uzupełnienie do dwóch (U2)

U2 to najpopularniejsza metoda reprezentacji liczb ze znakiem, bo:

1. Mamy tylko **jedno zero** (nie ma „+0” i „-0”).
2. Dodawanie/odejmowanie działa tak samo dla liczb ujemnych i dodatnich (sprzęt się nie gubi).

Przykład (8 bitów):

- +5 to 00000101.
- -5 to 11111011 (czyli odwracamy bity +5 i dodajemy 1).

14. Inne reprezentacje liczb całkowitych

1. **Znak-moduł** (sign-magnitude): najstarszy bit jest znakiem, reszta to wartość bezwzględna. Mamy problem dwóch zer.
2. **Uzupełnienie do jedynki (U1)**: liczba ujemna = odwrócenie bitów liczby dodatniej. Też mamy dwa zera.
3. **Uzupełnienie do dwóch (U2)**: najpopularniejsze we współczesnych CPU.

15. Liczby zmiennoprzecinkowe i standard IEEE 754

1. Najczęściej spotkasz się z **formatem pojedynczej precyzji** (32 bity) i **podwójnej precyzji** (64 bity).

2. Składniki:

- **bit znaku** (0 dla dodatnich, 1 dla ujemnych),
- **mantysa** (część ułamkowa),
- **wykładnik z przesunięciem (bias)**, np. 127 w float, 1023 w double.

3.

16. Błędy arytmetyki zmiennoprzecinkowej

Ponieważ mantysa i wykładnik mają ograniczoną liczbę bitów:

- Część liczb nie jest reprezentowana dokładnie (np. 0,1 w systemie dwójkowym to w przybliżeniu).
- Mogą wystąpić zaokrąglenia, szczególnie przy dodawaniu małych liczb do bardzo dużych.
- To zjawisko trzeba mieć na uwadze w programach finansowych, naukowych itd.

17. Kodowanie znaków: ASCII, EBCDIC, Unicode

1. **ASCII** (7-bit) – stare, ale wciąż używane do podstawowych znaków łacińskich.
2. **EBCDIC** – kod 8-bitowy rodem z mainframe'ów IBM. Dziś rzadki poza tym środowiskiem.
3. **Unicode** – globalny standard dla wszystkich alfabetów (chiński, arabski, emoji). Popularne kodowania to **UTF-8**, **UTF-16**, **UTF-32**.

18. Hierarchia pamięci (memory hierarchy)

1. **Rejestry** – maleńkie, ale ultraszybkie, wewnątrz CPU.
2. **Cache L1, L2, L3** – pamięci podręczne o rosnącej wielkości i opóźnieniach.
3. **RAM (pamięć główna)** – sporo megabajtów/gigabajtów, wolniejsza niż cache.

4. **Magazyny masowe** (dyski SSD/HDD) – bardzo duża pojemność, ale jeszcze wolniejsze.

Zasada: w miarę, jak rośnie pojemność, rośnie też czas dostępu i spada cena za bajt.

19. Mechanizm cache: tag, index, offset

Przy **mapowaniu bezpośrednim** do pamięci podręcznej:

- Adres procesora dzielimy na **tag** (identyfikacja bloku), **index** (który wiersz cache) i **offset** (pozycja wewnątrz bloku).
- **Tag** sprawdzamy, by upewnić się, że blok z pamięci odpowiada właściwemu adresowi.
- Jeśli zawartość w cache zgadza się z adresem (hit) – pobieramy dane szybko; jeśli nie (miss) – trzeba sięgnąć do RAM.

20. Efektywny czas dostępu (ECD) do pamięci

ECD to uśredniony czas dostępu do danych, uwzględniający:

- Trafienia w cache (hit) i nietrafienia (miss).
- Czas dostępu do cache vs. czas dostępu do RAM.

Formuła często wygląda tak:
Poprawiamy ECD przez zwiększanie rozmiaru cache i/lub optymalizację kodu (lepsza lokalność danych).

21. Endianness: little endian vs. big endian

1. **Little endian** – najmłodszy bajt (LSB) zapisywany pod najniższym adresem. Np. procesory x86/Intel.
2. **Big endian** – najbardziej znaczący bajt (MSB) jest pod najmniejszym adresem. Spotykane w niektórych starszych architekturach, protokołach sieciowych.
3. Wymaga uwagi przy przesyłaniu danych między różnymi platformami.

22. Przerwania (interrupts) – reakcja na zdarzenia

1. **Maskowalne** – można je tymczasowo zablokować, jeśli nie chcemy przerwania w danej chwili (np. w krytycznej sekcji).
2. **Niemaskowalne (NMI)** – nie da się ich zignorować (np. poważny błąd sprzętu).
3. Po wystąpieniu przerwania CPU przerywa bieżący kod i skacze do procedury obsługi przerwania (ISR).

23. Wielopotokowość (pipeline)

Zamiast robić instrukcje **sekwencyjnie** w całości, dzielimy je na etapy (Fetch, Decode, Execute, Write-back). W jednym cyklu zegara różne instrukcje mogą być w różnych fazach, co zwiększa wydajność.

- **Konflikt danych** – kolejna instrukcja potrzebuje wyniku poprzedniej, który jeszcze się nie pojawił.
- **Konflikt kontroli** – skoki warunkowe (nie wiadomo, którą instrukcję pobierać dalej).
- **Konflikt zasobów** – dwie instrukcje naraz chcą tego samego sprzętu.

24. Rodzaje instrukcji: rejestr-rejestr, rejestr-pamięć, pamięć-pamięć

1. **Rejestr–rejestr** – ADD R1, R2, R3 (typowe dla RISC).
2. **Rejestr–pamięć** – ADD R1, [memory].
3. **Pamięć–pamięć** – ADD [A], [B] (charakterystyczne np. dla niektórych CISC).

Im więcej operacji na rejestrach, tym z reguły szybciej (bo rejestry są w CPU).

25. Architektura stosowa (bezadresowa)

- Wszystko dzieje się na stosie: wrzucasz (push) wartości, a instrukcja ADD bierze dwie wartości ze szczytu stosu, dodaje i odkłada wynik z powrotem na stos.
- Dobrze współpracuje z odwrotną notacją polską (RPN), np. 3 4 +.
- Wadą bywa większa liczba instrukcji do manipulowania stosem.

26. Pamięć wirtualna i stronicowanie

1. **Stronicowanie** – adresy wirtualne (widoczne dla programu) dzielimy na **strony** (pages), a w pamięci fizycznej (RAM) mamy **ramki** (frames).
2. Jeśli dana strona nie jest w RAM, następuje **page fault** i system ładuje ją z dysku (to wolne).
3. Dzięki temu każdy proces „myśli", że ma wielki ciągły blok pamięci, a OS dba o mapowanie.

27. TLB (Translation Lookaside Buffer)

- Specjalny rodzaj **cache** do **tłumaczenia** adresów wirtualnych na fizyczne.
- Gdy CPU odwołuje się do adresu wirtualnego, TLB sprawdza, czy ma wpis o tej stronie.
- Jeśli tak (TLB hit), tłumaczenie jest błyskawiczne. Jeśli nie (TLB miss), procesor musi sięgnąć do tablic stron w pamięci (wolniej).

28. Zasada równoważności sprzętu i oprogramowania

Można zaimplementować jakieś funkcje w **sprzęcie** (np. instrukcja kryptograficzna w CPU) lub w **oprogramowaniu** (np. algorytm szyfrujący w języku C). Zależy, co wolimy:

- Sprzęt bywa szybszy, ale mniej elastyczny.
- Software jest łatwiej zmieniać, ale może być wolniejszy.

29. Prawo Moora (i wspomnienie o Prawie Rocka)

1. **Prawo Moora** – liczba tranzystorów w układzie scalonym podwaja się mniej więcej co dwa lata. Przez dekady nakręcało to wzrost mocy CPU.
2. Powoli osiągamy granice fizyki i rosną koszty miniaturyzacji, stąd widać spowolnienie.
3. **Prawo Rocka** czasem przytaczane w kontekście kosztów i wydajności – mówi, że wraz ze wzrostem stopnia scalenia rosną też złożoność i koszty.

30. Przepełnienia i błędy obliczeniowe

1. **Przepełnienie liczb całkowitych**

   • Przy bez znaku (unsigned): wynik liczy się mod 2^n. np. 8-bit: 255 + 1 = 0.

   • Przy ze znakiem (signed) w U2: może zmienić znak (np. +127 + 1 = -128 w 8-bit).

2. **Błędy zmiennoprzecinkowe**

   • Mantysa i wykładnik mają skończony rozmiar → części liczb (np. 0,1) nie da się zapisać dokładnie.

   • Mogą wystąpić zaokrąglenia i utrata precyzji przy operacjach na bardzo dużych i bardzo małych liczbach.

Podsumowanie

W tym **30-punktowym** przeglądzie poznaliśmy:

- Podstawy **architektury komputera** (von Neumann, Harward, CPU, pamięć, I/O).
- Szczegóły **cyklu rozkazowego**, rejestrów, ALU, jednostki sterującej.
- **Tryby adresowania**, różnice w architekturach (RISC, CISC, bezadresowa, jednoadresowa, dwuadresowa).
- **Reprezentację danych** (U2, IEEE 754, Unicode, ASCII).
- **Hierarchię pamięci** (cache, RAM, dyski), potok (pipeline), pamięć wirtualną i TLB.
- **Przerwania**, **Prawo Moora**, zjawiska przepełnień.