

## Структура та програмна модель комп'ютера



©Олег Фаренюк,  
Creative Commons - Attribution Share Alike (CC BY-SA)

# Філософськи-практичний вступ

- Програмування – різновид інженерії<sup>1</sup>.
- Інженерна справа включає вирішення задач в рамках обмежень накладених як законами природи, так і наявними ресурсами та вимогами замовників.
- Зазвичай, існує декілька способів вирішити задачу, поміж яких варто прагнути знайти найкращий.
- Різні вимоги та обмеження практично завжди конфліктують між собою – маємо *інженерний конфлікт*, вирішення якого – знаходження *інженерного компромісу*.

Результатом цієї лекції буде каркас того, що ми вивчатимемо далі.  
Своєрідна розмальовка, яку потім заповнюватимемо. Будь ласка,  
будьте уважними!

---

<sup>1</sup>Хоча представники сфери ІТ іноді це й заперечують...

- Хто не бачив класичної картинки на тему, з чого складається комп'ютер?

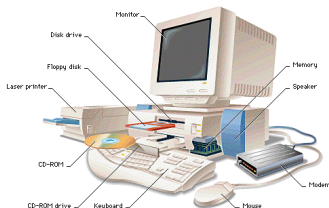


Fig 1.1 The hardware parts of a computer system

- На ній не зупинятимемося.
- Так само зараз не зупинятимемося на детальнішому вивченні окремих його частин та їх історії – поки важливим є лише загальне уявлення про структур та взаємодію компонент. Детальніше ці питання розглядатимуться в майбутньому<sup>2</sup>!

<sup>2</sup>Іронія – історія тієї чи іншої сфери, зазвичай, розглядається на початку підручника чи курсу. Тобто, тоді, коли вона ще не цікава переважній більшості користувачів та не може принести користі від аналізу в ретроспективі.

# Елементи комп'ютера

Для нас важливим буде усвідомлення існування наступних компонент:

- Центральний процесор<sup>3</sup> – Central processing unit, CPU
- Оперативна пам'ять<sup>4</sup> – Random Access Memory, RAM.
- Засоби вводу-виводу – Input/Output, I/O.

Поміж засобів вводу-виводу, виділимо – заради простоти, відмовившись від повноти розгляду чи ортогональності:

- Зовнішня енергонезалежна, (вторинна), часто – дискова, пам'ять<sup>5</sup>.
- Засоби інтерфейсу користувача та взаємодії із зовнішнім світом – клавіатура, мишка, відеокарта із дисплеєм, мережеві картки та модеми всіх технологій тощо. Надзвичайно різні пристрої із різними вимогами та своїми трюками реалізації.

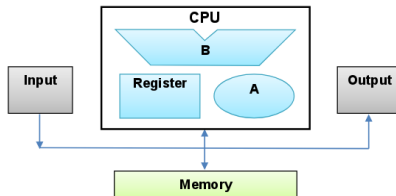
---

<sup>3</sup>Мікропроцесор – це інтегральна схема, яка реалізовує CPU. Зараз – майже синонім CPU.

<sup>4</sup>Іноді називають первинною чи основною пам'яттю.

<sup>5</sup>Всі розуміють, про що мова, але через зоопарк технологій, дати означення важко.

- CPU, RAM та різноманітні пристрої вводу-виводу з'єднані та обмінюються інформацією за допомогою **шин** (bus).
- Що таке шини, як вони функціонують, навіщо нам DMA<sup>6</sup> і т.д. – розглядатимемо в подальшому. Поки вважатимемо їх абстрактними каналами взаємодії та передачі інформації.
- Враховуючи різноманітність вимог та задач, шини дуже різні.



<sup>6</sup>І що це за букви, BTW.

# Різновиди шин

Грубо<sup>7</sup> виділимо наступні шини:

- шину пам'яті<sup>8</sup>,
- "внутрішні"<sup>9</sup> шини вводу-виводу, такі як PCI-Express<sup>10</sup> та SATA,
- зовнішні шини – для підключення різноманітної периферії, такі як USB, FireWire, legacy COM-порти тощо.
- ◇ Зазвичай, зовнішні шини, такі як USB чи й SATA, реалізуються спеціалізовані пристрої (хаби), що з'єднані із рештою комп'ютера PCIe чи іншою "внутрішньою" шиною.
- Набір схем, які реалізують взаємодію всіх цих компонент, називають чіпсетом (chipset).

---

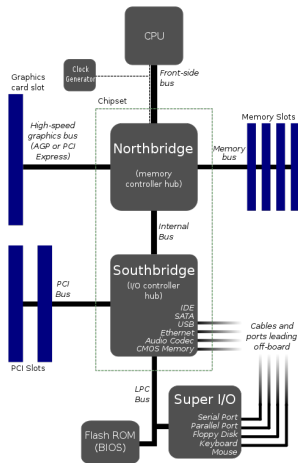
<sup>7</sup> Нагадую, поки ми не прагнемо до повноти розгляду чи ортогональності – нам слід зрозуміти загальну схему.

<sup>8</sup> Критичну щодо швидкодії, це накладає серйозні обмеження на неї.

<sup>9</sup> Це не термін! Лише умовна назва для наших потреб.

<sup>10</sup> Та PCI/AGP/ISA – швидкісні, як на свій час, хоча й повільніші за шини пам'яті і з іншими задачами.

# Материнська плата




- Через жорсткі вимоги до продуктивності, сучасні процесори більшість Northbridge "втягнули" в себе. Чіпсет, що виконує залишки його функцій та функції Southbridge, часто називають Platform Controller Hub (PCH).

- Кожен байт RAM має свій номер<sup>11</sup>.
- Власне, елементарну адресовану комірку пам'яті – таку, яка має адресу і до якої можна безпосередньо звернутися, називають байтом<sup>12</sup>.
- ◇ Існували різні розміри байтів, історично зафіксувалося значення 8 біт<sup>13</sup>. Воно достатньо зручне.
- Процесор звертається до пам'яті за допомогою **шини пам'яті**.
- ◇ Для цього він повідомляє пам'ять про операцію, яку він хоче здійснити – читання чи запис та номер байта, який його цікавить.

<sup>11</sup>Бувають нюанси. Наприклад, банки пам'яті – свого роду паралельні її світи. Звичною справою є те, що нумерація не неперервна. Фізично, пам'ять часто передаватиме багато байт за раз. Не всі процесори дозволяють незалежно звертатися до кожного байту, або такі звертання, хоча й можливі, є повільними. Подробиці розглядатимемо в майбутньому.

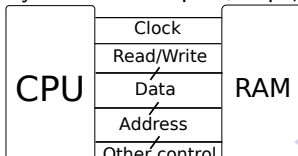
<sup>12</sup>Історично це була кількість біт, достатня для кодування одного символу.

<sup>13</sup>Ймовірно, завдяки популярності IBM System/360, яка використовувала цей розмір, потім підхоплений революцією мікропроцесорів – 8-бітними 8080 і т.д.] 



# Шина доступу до пам'яті

- Шина доступу до пам'яті складається із декількох частин, по одній для кожної задачі.
- Їх теж теж називають шинами:
  - ◇ шина адреси,
  - ◇ шина даних,
  - ◇ шина керування.
- Приклад операції читання:
  - ◇ процесор встановлює на шині адресу комірки пам'яті, яка його цікавить<sup>14</sup>.
  - ◇ Отримавши шиною керувальний сигнал, пам'ять виставляє на шині даних вміст відповідного байта.
  - ◇ Процесор читає його із шини даних.
- Для запису, процесор виставляє адресу даних, в яку слід записати, на шині адреси, адресу даних – на шині даних, після чого віддає команду пам'яті шиною керування. Пам'ять записує дані із шини даних у байт з номером, переданим шиною адрес.



## Взаємодія компонент комп'ютера – I/O

- Шини вводу-виводу зазвичай організовані трішки по іншому. Вони дуже різноманітні – деякі розглядатимемо в майбутньому.
- До такої шини часто під'єднано декілька пристроїв, тому перш, ніж почати обмін, слід домовитися, котра пара пристроїв зараз спілкуватиметься.
- Часто на шині є один або кілька<sup>15</sup> головних пристроїв (master, manager) та якась кількість підлеглих (slave, worker).
- Master ініціює обмін та керує ним. Зокрема, вирішує, із котрим пристроєм відбуватиметься взаємодія.
- Пристрої вводу-виводу значно різноманітніші, ніж RAM. Тому взаємодія із ним включає не тільки обмін даними, але й обмін **командами**. Наприклад: "перейди в режим вимірювання прискорення", "встанови вказану роздільну здатність", "передай три останніх виміри", "засни".

---

<sup>15</sup> | тоді їм ще доводиться між собою домовлятися – так-званий арбітраж шини.

## Взаємодія компонент комп'ютера – I/O, переривання та DMA

- Периферійні пристрої, зазвичай, багато повільніші за CPU. Крім того, вони функціують асинхронно відносно CPU. Активне очікування (busy loop) є дуже неефективним.
- Тому активно використовуються **апаратні переривання**. Пристрій на шині, slave, може повідомити процесор, що у нього новини – прибули дані, завершено операцію тощо.
- Тоді процесор, за першої нагоди, може прореагувати, не відволікаючись решту часу.
- Відволікати процесор лише щоб передати дані від пристрою в пам'ять чи на інший пристрій, може бути неефективним.
- Тому використовується Direct Memory Access – DMA. Процесор інструктує DMA контролер – коли прийдуть дані від такого-то пристрою (про що повідомить переривання) – класти їх туди. І займається своїми справами.

## Взаємодія компонент комп'ютера – memory mapped I/O

- Ввід-вивід процесор може здійснювати двома різними підходами.
- Перший – окремі машинні команди, призначенні для вводу-виводу.
- Другий – периферійні пристрої "шпигують" (слухають) шину адреси RAM, і для певних адрес, вважають, що це звертання до них, а не до RAM (остання в цей час мовчить).
- Такий підхід називають memory-mapped I/O – ввід-вивід ініціюється звертанням до певних комірок пам'яті. Наприклад: запис коду букви в байт номер B8000h для певної архітектури<sup>16</sup>. призводить до того, що вона з'являється в лівому верхньому кутку екрану. Тобто, це звертання, насправді, перехоплюється відеоадаптером
- Власне, за допомогою memory-mapped I/O, DMA може організовувати обмін даними між периферією, без залучення CPU.

---

<sup>16</sup> Мова про архітектуру IBM PC/AT – предка всіх сучасних не-Apple персоналок

- Розмова про апаратне забезпечення змушувала вживати багато нових слів – обійти це поки неможливо.
- Однак, говорячи про програмне забезпечення, можна почати із знайомого вам та рухатися вниз по рівнях абстракції.
- Почнемо із тривіальної задачі: прочитати файл із послідовністю дійсних чисел та вивести їх середнє значення і дисперсію:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2$$

- Будемо також вважати, що всі дані для обчислень ми хочемо завантажити у пам'ять – для ще якогось аналізу – обчислити дисперсію можна і без цього:

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2 = \dots = \left( \frac{1}{N} \sum_{i=1}^N x_i^2 \right) - \bar{x}^2$$

Мотив: для подальшого розгляду хотілося б мати привід читати в пам'ять не відомий наперед об'єм даних, а ускладнювати логіку коду не хотілося б.

- Програма на Python, яка це робить, може виглядати якось так:

```
import math

with open("data.txt", "r") as datafile :
    data = [float(x) for x in datafile.read().split()]
    average = sum(data)/len(data)

std_dev = math.sqrt(sum((x-average)**2 for x in data)/len(data))

print(average)
print(std_dev)
```

- Обробка помилок у цій програмі, як і у більшості нижче, рудиментарна, щоб не відволікати від суті, однак, все ж, присутня.

# Аналіз коду

- 1 Програма вважає, що дані знаходяться у файлі "data.txt".
- 2 При тому, файл цей має знаходитися в **поточній директорії**.
- 3 Перш ніж можна буде щось робити із файлом, слід повідомити операційній системі (**ОС**) про ваші наміри. Це – процес *відкриття файлу*. ОС при цьому перевіряє, чи такий файл існує та чи має дана програма права із ним працювати.
- 4 Запит на відкриття файлу програма здійснює за допомогою функції `open()`.
- 5 При цьому програма вибирає режим роботи із файлом. Тут – лише читання, про що свідчить аргумент "r".
- 6 Після завершення роботи із файлом, його слід *закрити* – повідомити ОС про це та дати їй можливість "підбити підсумки" – оновити розмір файлу, дату його зміни, звільнити внутрішні структури даних із ним пов'язані, розпочати процес очищення буферів тощо.

- 7 Для цього до файлового об'єкту Python можна застосувати метод `close()`. Однак, якщо в процесі роботи над файлом станеться виключення – до завершення процесу, що його відкрив, файл так і залишиться відкритим. Тому тут ми скористалися менеджером контексту `with ... as`, який автоматично викличе `close()`, не залежно від причин виходу з нього. Без менеджера контексту код виглядав би так:

```
datafile = open("data.txt", "r")
file_content = datafile.read()
.....
datafile.close()
```

- 8 Для маніпуляції із файловим об'єктом ми користуємося таким засобом Python як *посилання* (reference). Менеджер контексту чи безпосередній виклик `open()` створює відповідний об'єкт, (в процесі відбувається звертання до ОС і все таке) та повертає нам "ниточку" до нього – спосіб із ним взаємодіяти.



- 9 Наступна задача – прочитати весь вміст файлу та зберегти його в пам'яті процесу.
- 10 Ми вважаємо, що дані – послідовність чисел, збережено там у вигляді *тексту*, а числа розділяються так-званими пробільними символами<sup>17</sup> (whitespaces).
- 11 Для комп'ютера текстовий формат чисел незручний – слід виконати перетворення. В Python найзручніший тип дробових чисел, точніше, чисел із рухомою комою/крапкою називають float, в багатьох інших мовах – double.
- 12 Хорошою практикою програмування є виражати свої високорівневі наміри, а не детально описувати процес.
- 13 Враховуючи це все, читання повного вмісту файлу виглядатиме так:

```
data = [float(x) for x in datafile.read().split()]
```

Тут використано так-звану генерацію списків чи list comprehension. Читати цей, дещо дивний, код можна так: "прочитати весь файл, розбити по пробільних символах, перетворити кожен елемент у float і результат помістити у список".

<sup>17</sup> До них, крім власне пробілу, входять також табуляції, символ нового рядка (обидва під Windows, якщо ви розумієте, про що мова – якщо ні, нижче це теж розглядатиметься.)

- 14 Менш елегантний, ймовірно – повільніший, але більш безпосередній та звичний програмістам на багатьох інших мовах варіант виглядатиме так:

```
1 file_content = datafile.read()
2
3 data = []
4 for n in file_content.split():
5     data.append(float(n))
```

Поки що вручну поділ на елементи не розглядатимемо – все ще використано `split()`.

- 15 Тут, для списку `data`, як і для файлового об'єкту, створеного `open()`, Python бере на себе маніпуляцію пам'яттю. Як він це робить – одна із ключових тем цієї лекції.
- 16 Зауважте: при написанні програми ми вже неявно маємо на увазі, **що існує декілька різних видів пам'яті**: RAM та пам'ять для тривалого зберігання (HDD чи SSD), при тому, остання – повільніша.
- 17 Далі програма нарешті виконує арифметичні операції (над змінними в пам'яті у форматі `double`), для яких й була створена.

- 18 Знайшовши середнє значення та дисперсію, програма повинна якось продемонструвати результат. Для цього вона користується викликом функції `print()`.
- 19 `print()` виводить у так-званий `stdout` – *стандартний вивід*. (Зразу варто згадати концепцію Unix: "все є файлом").
- 20 По замовчуванню `stdout` є пов'язаний із консоллю – чорним<sup>18</sup> текстовим віконечком.
- 21 Нарешті, коли процес завершується, ОС повинна звільнити всі виділені для нього ресурси – пам'ять, файли тощо.

Хоча, сподіваюся, всі чи більшість цих етапів вам відомі та усвідомлені, подивитися на них під таким кутом важливо для подальшого розгляду.

---

<sup>18</sup> Колір залежить від вподобань користувача.

## Процеси і програми

- **Процес**<sup>19</sup> – одна із ключових абстракцій комп'ютерного світу.
- Детальніше вона розглядатиметься в курсі ОС.
- Зараз лише домовимося, що процес – копія програми в RAM, яка виконується.
- Іншими словами, програма – пасивна колекція команд, а процес здійснює фактичне виконання цих команд.
- На процес зручно дивитися як на контейнер ресурсів, пов'язаних із екземпляром програми, що виконується, разом із, мінімум, одним потоком виконання. Поміж цих ресурсів – виділена пам'ять, відкриті файли, завантажені динамічні бібліотеки, поточна директорія<sup>20</sup> і т.д. і т.п.
- При завершенні процесу, сучасні операційні системи здатні звільнити більшість виділених йому ресурсів<sup>21</sup>

<sup>19</sup>Н. Deitel у своїй книзі "Operation systems", 2nd ed., що термін "процес" вперше було вжито операційною системою Multics.

<sup>20</sup>Також називають "поточним шляхом" – це шлях у файловій системі, який використовується по замовчужанню для пошуку файлів, – якщо не явно вказано інший. В процесі виконання процес може змінювати свою поточну директорію.

<sup>21</sup>Раніше бувало й по іншому. Наприклад, популярна колись MS DOS часто не мала шансів самостійно звільнити виділену процесом так-звану Extended чи Expanded пам'ять. Винятки сучасних ОС, часто пов'язані із ресурсами відлагодження та профілювання програм – всілякі апаратні та програмні лічильники. Див., наприклад, підсистему Event Tracing for Windows (ETW).

# Трансляція

- Вище було розглянуто просту програму – інструкції комп'ютеру.
- Питання – хто саме її виконуватиме і як?
- Комп'ютер, напевне. Однак, *як відомо*<sup>22</sup>, працює із нулями та одиничками, а наша програма – якийсь текст, навіть із звичними англійськими словами.
- Тобто, щоб виконати цю програму, нам знадобиться *транслятор*.
- Транслятор "перекладає" програму із однієї *мови програмування* на іншу.
- Результат трансляції може бути або переданий ще одному транслятору або виконаний безпосередньо – якщо цільову мову трансляції розуміє апаратне забезпечення.

---

<sup>22</sup>Все ж, що це означає, розуміє помітно менша частина людей, ніж ті, що цю фразу чули. ►

# Бінарний код

- Найпростіший приклад<sup>23</sup>: запуск стандартної програми `ls` Unix-системи (Linux) чи `notepad.exe` – стандартного блокнот MS Windows.
- Обидві програми вже є в **бінарному коді**<sup>24</sup>, зрозумілому процесору, для якого вони призначені.
- Тобто, після необхідних перетворень, призначених перетворити програму – файл чи файли на диску, у процес – код, що виконується, які здійснює операційна система, **код безпосередньо виконується центральним процесором**.
- Процесор транслює двійкові (машинні) команди у якісь свої внутрішні дії, які, в кінцевому результаті, дають бажаний користувачам результат.
- Детальніше ми бінарний код розглядатимемо в майбутньому. Якщо коротко, кожна команда, яку розуміє процесор, має свій двійковий код. Набір команд процесора називають системою команд – `instruction set architecture (ISA)`.

<sup>23</sup> Для розгляду на обраному нами рівні абстракції та деталізації.

<sup>24</sup> Детальніше, що це значить, вивчатимемо в подальшому.

# Інтерпретовані мови

- Python демонструє інший підхід до трансляції.
- Заради однозначності, вважатимемо, що використовується транслятором CPython від <https://www.python.org>.
- Це – так-званий *інтерпретатор*. Він безпосередньо виконує код програми на Python, рядок за рядком.
- Сам *інтерпретатор* CPython – бінарна програма, яка виконується на центральному процесорі.

# Компілятори та асемблери

- Питання – звідки беруться двійкові програми? Їх створює інший різновид трансляторів – *компілятори*.
- Виділяють також спеціальний клас таких трансляторів – асемблери (Assembler), які транслюють із спеціальної (мнемонічної) форми представлення машинного коду (Assembly language) у, власне, машинний код.
- Важливим інструментом є, також, редактор зв'язків чи лінкер, який об'єднує окремі частини програми в єдиний виконавчий модуль. Детальніше його діяльність розглядається на практичних.



# Приклад коду на асемблері

- Програмування в машинних кодах:

```

10110100 00001001 10111010
00001001 00000001 11001101
00100001 11001101 00100000
01001000 01100101 01101100
01101100 01101111 00100000
01110111 01101111 01110010
01101100 01100100 00100001
00100100

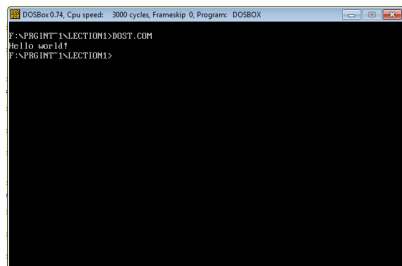
```

Або так:

```

B4 09 BA 09 01 CD 21 CD 20
48 65 6C 6C 6F 20 77 6F 72
6C 64 21 24

```



```

org    100h
use16

```

```

mov    ah,09h    ; B4 09
mov    dx,hello  ; BA 09 01
int     21h      ; CD 21

int     20h      ; CD 20

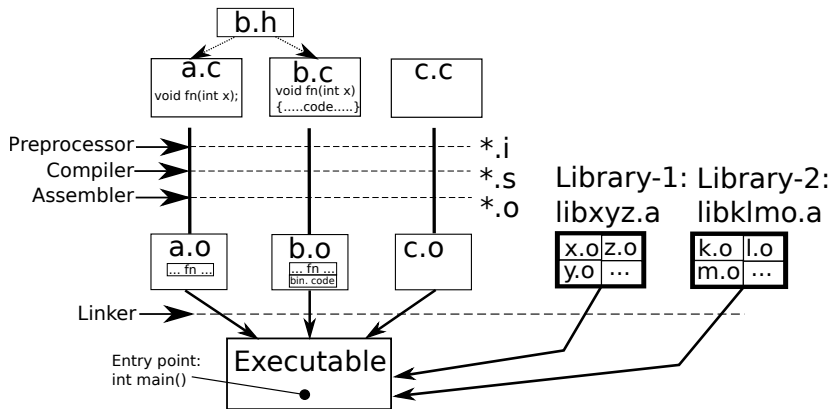
```

```

hello db    'Hello world!$'
; 48 65 6C 6C 6F 20 77 6F 72 6C
; 64 21 24

```

## Роздільна компіляція



- Розширення – ті, що, за замовчуванням, використовує GCC.
- Проміжні файли, в нормі, не зберігаються. Щоб вони залишилися, слід скористатися ключем `-save-temps`.
- За замовчуванням, використовується AT&T стиль асемблера. Перекопати використовувати стиль Intel: `-masm=intel`. Для кращого сприйняття варто також додати `-fverbose-asm`.

# Байт-код та віртуальна машина Java

- Програму на Java, зазвичай, компілюють.
- Однак, у так-званий **байткод** (bytecode) – машинний код віртуальної машини Java (**JVM**).
- Віртуальна машина Java цей код виконує, інтерпретуючи. Зроблено так заради переносимості<sup>25</sup>.

---

<sup>25</sup>Слоганом Java є "Write once, run anywhere" – в ідеальному варіанті програма пишеться раз, компілюється раз, а виконується на віртуальній машині, яку, звичайно, треба написати для кожної платформи, але це робиться лише раз та й багато коду між платформами спільного.

## Чимдалі в ліс...

- Крім CPython існують й інші реалізації цієї мови, наприклад Jython<sup>26</sup> та IronPython<sup>27</sup>.
- Перший із них – реалізація Python поверх віртуальної машини Java, другий – поверх "альтер-его" Java від Microsoft<sup>28</sup> – поверх Common Language Runtime (CLR)<sup>29</sup>, віртуальної машини .NET.
- Вони обоє дозволяють легко використовувати у коді на Python об'єкти, створені для відповідних віртуальних машин.
- Також у цих реалізаціях відсутній GIL – Global Interpreter Lock.

### Спільним для них є:

- Кожна із них існує і у вигляді інтерпретатора і у вигляді компілятора.
- У варіанті інтерпретатора код на Python виконується байткодом, скажімо, Java, у відповідній віртуальній машині, який, у свою чергу, віртуальною машиною виконується на апаратному забезпеченні. Хоча, бувають нюанси – див. далі.
- У варіанті компіляції код компілюється в байткод.

<sup>26</sup>На жаль, проект заморожений – до Python3 так і не був оновлений, станом на момент написання. Див. також <http://www.jython.org>.

<sup>27</sup>Як і Jython – закинутий на версії 2.7. Див. <http://ironpython.net>.

<sup>28</sup>Повчальною є історія спроб Microsoft завести собі свою Java та народження .NET.

<sup>29</sup>Точніше, поверх Dynamic Language Runtime, яка, у свою чергу – поверх CLR.

## Чимдалі в ліс...

- Ще одним прикладом є GraalVM<sup>30</sup> – віртуальна машина Java, реалізована на Java.
- На додачу до JVM мов, підтримує JavaScript (Node.js), Python, Ruby, R та має SDK для додавання нових.
- Вміє запускати код мовами програмування типу C чи C++, для яких, зазвичай, генерується нативний код<sup>31</sup>.
- ◇ Для цього потребує clang – транслює LLVM bitcode<sup>32</sup> у байткод Java.
- Дозволяє комбінувати програми, написані різними мовами – аналог C calling conventions, за допомогою яких спілкується нативний код сучасних ОС.

---

<sup>30</sup> Детальніше див. <https://habr.com/ru/company/haulmont/blog/433432/> і, скажімо, <https://github.com/graalvm/graalpython>.

<sup>31</sup> Бінарний (об'єктний) код конкретної платформи.

<sup>32</sup> Проміжний результат компіляції clang-ом.

# JIT

- Крім компіляції та інтерпретації, існує багато проміжних варіантів.
- Наприклад, багато інтерпретаторів (включаючи JVM) за першого виконання можуть *компілювати* код у машинне представлення цільової платформи, і вже потім виконувати його – з тією ж ефективністю, що й "рідний", *нативний* (native) код платформи – **just-in-time compilation** (JIT).
- Сучасна JVM виконує ряд інших важливих трюків-оптимізацій, наприклад, компілює байт-код із врахуванням даних про те, як він виконувався. Див. також profile-guided optimization.
- Хороший цикл статей про JIT:  
<https://carolchen.me/blog/jits-intro/>,  
<https://carolchen.me/blog/jits-impls/> (російською:  
<https://habr.com/ru/company/mailru/blog/513290/>).

# Оптимізація

- Компілятор часто виконує якісь перетворення програми, щоб зробити її швидшою. Подробиці обговорюватимуться на практичних.
- Поняття Undefined behaviour існує, в значній мірі, щоб спростити оптимізацію.
- Link-time optimization – назва технології оптимізації **цілої** програми – на противагу оптимізації кожного об'єктного файлу, без інформації про решту проекту.

# Інтерпретація vs компіляція – 1

- Приклад із Jython/IronPython ілюструє важливу ідею.
- Ми звикли вживати слова "інтерпретована мова програмування" чи "компільована мова програмування", однак, вибір компілювати чи інтерпретувати робить розробник транслятора.
- Мови, які ми звикли сприймати як компільовані, мають свої інтерпретатори (включаючи C<sup>33</sup>, C++<sup>34</sup> та FORTRAN), а для інтерпретованих мов часто існують компілятори.
- Цікавий приклад – езотерична мова Befugne, із двомірним кодом, розроблена, щоб максимально ускладнити компіляцію. Компілятор таки існує.

---

<sup>33</sup>Наприклад, CINT – <http://www.hanno.jp/gotom/Cint.html>, частково підтримує C++.

<sup>34</sup>Наприклад, cling – <https://root.cern.ch/cling>. Ще один схожий проект – Ch, <http://www.softintegration.com>.



## Інтерпретація vs компіляція – 2

- Певна відмінність існує між компільованими та інтерпретованими мовами, все ж, існує.
- Творці інтерпретованих мов, зазвичай, зосереджуються на простоті написання коду нею, можливості його легко запускати (без перекомпіляції) тощо – більше турботи про технічні аспекти (наприклад, типи змінних) беручи на себе, ціною втрати продуктивності.
- Розробники компільованих мов – на продуктивності програм, написаних відповідною мовою.
- Класичний приклад тут Python – писати на ньому програми просто та зручно, однак, ефективність, в порівнянні із аналогічним кодом на C++, нижча на порядок-півтора.
- Щоправда, байткоду Java це не дуже стосується. JVM є інтерпретатором, але вона інтерпретує машинну мову, розроблену спеціально для того, щоб бути швидкою та універсальною, а не зрозумілою та простою для написання людиною<sup>35</sup>.

---

<sup>35</sup>Тому втрати швидкодії мінімальні – на сучасних JVM код повільніший в 1.1-2 рази, а за певних умов (хоча, про них більше говорять, ніж зустрічають на практиці), може бути швидшим за код на C/C++, завдяки динамічній перекомпіляції із оптимізацією під конкретні потоки даних, з якими працює код. Правда, трапляються ситуації, коли код повільніший в 5 і більше раз.

# Мікроархітектура

- На центральному процесорі історія із трансляцією не закінчується.
- Часто команди, що поступають в процесора та належать до його системи команд (так-званої Instruction set architecture – ISA<sup>36</sup>) не виконуються безпосередньо а інтерпретуються всередині процесора, так-званим *мікрокодом*, із використанням своєї системи *мікрокоманд*, *мікроасемблером* тощо.

---

<sup>36</sup>Не плутати із шиною ISA (Industry Standard Architecture), популярною в 80-х та на початку 90-х!

# Емуляція

- На мікроархітектурі історія теж не закінчується.
- Код – операційна система, разом із прикладними програмами, може виконуватися в *емуляторі*, який, у свою чергу, виконується на зовсім іншій платформі.
- Емулятор – програма (чи, рідше, пристрій), яка дозволяє одній машині поводитися як іншій, часто – не пов'язаній.
- При чому, у віртуальній машині може бути запущена нова віртуальна машина, і так можна продовжувати, поки не вичерпається пам'ять або можливості емуляторів.
- Звичайно, практично такі матрьошки використовуються не часто, але враховуючи поточну популярність хмарних обчислень – трапляються. А ще недавно дана цитата була жартом (<https://bash.im/quote/59325>):

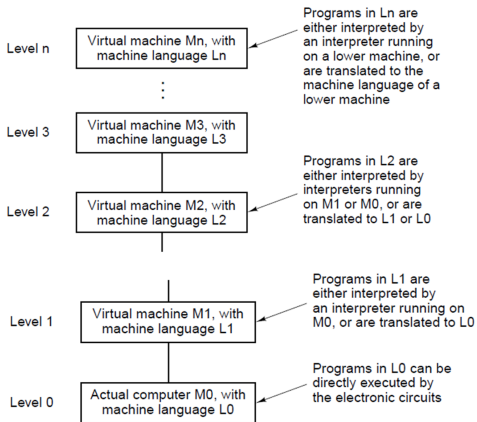
*"Флудити то флудити! :) StarControl2, запущений в DosBox, під X в Debian, який запущений в VMWare, яка – в WinXP. До кого звертатися з приводу звуку, що не працює?"*

*"До санітарів"*

# Віртуалізація

- Суміжним поняттям є віртуалізація.
- Однією із найважливіших функцій операційної системи, критична для успіху самої ідеї операційної системи, є *віртуалізація* периферійних пристроїв – вона дозволяє кожній програмі вважати, що та одноособово працює із процесором, пам'яттю, периферійними пристроями, такими як диски, клавіатура та дисплей.
- Тут зосередимося на вужчому застосуванні цього поняття.
- Виконання програми для PDP-11 на системі із процесором сімейства x86 вимагатиме повної емуляції.
- Однак, повна емуляція програми для x86 на ньому ж, яка є доволі затратною, ще й не дуже потрібна.
- Більшість команд головний процесор може виконувати безпосередньо. Команди ж, які можуть вплинути небажано на гостьову систему, звертання до периферії тощо, перехоплюються віртуалізатором і для гостьової системи робиться вигляд, що ця команда виконала свою роботу.
- Популярна зараз архітектура x86 – жахлива з точки зору віртуалізації.

- Таким чином, наслідуючи Таненбаума, виконання програми на комп'ютері можна зобразити так:



- В різні моменти важливим є різний рівень деталізації – ми з тим ще стикатимемося.

- Одна із найскладніших проблем, що стоять перед С та С++ – різноманітність платформ, на яких вона використовується.
- Хоча зараз 8-бітний байт – норма<sup>37</sup>, а цілі від'ємні числа представляються у доповнювальному коді<sup>38</sup>, так було не завжди.
- Називаючи байтом мінімальну адресовану комірку пам'яті<sup>39</sup>, згідно книги Таненбаума "Structured Computer Organization":

Computer	Bits/cell
Burroughs B1700	1
IBM PC	8
DEC PDP-8	12
IBM 1130	16
DEC PDP-15	18
XDS 940	24
Electrologica X8	27
XDS Sigma 9	32
Honeywell 6180	36
CDC 3600	48
CDC Cyber	60

<sup>37</sup>Тривалі пошуки систем із байтом відмінної довжини маже не дали результатів – хіба туманні згадки про якісь пропріетарні DSP.

<sup>38</sup>Знову ж, винятків не вдалося знайти, але є надія на екзотичні DSP.

<sup>39</sup>Як завжди, є багато нюансів і екзотичних винятків, але, в цілому, поняття добре означене.

- Те ж стосується і машинного слова – поняття ніби просте, базове, але довга історія розвитку заплутала і його.
- Ідейно, це – комірка пам'яті, найбільш зручна для CPU. Зазвичай, під цим мається на увазі, що її розмір рівний розміру регістрів CPU.
- Давні архітектури використовували різні хитрі трюки – як для оптимізації, так і через обмеження технологій чи не повного усвідомлення наслідків<sup>40</sup>. Це породило безліч винятків. Іноді і зараз трапляються експерименти із екзотичними підходами.
- Розвиток конкретної архітектури, із дотриманням зворотної сумісності, породив дуже хитрозакручених монстрів. Приклад – на наступному слайді.
- Див. також [https://en.wikipedia.org/wiki/Word\\_\(computer\\_architecture\)#Table\\_of\\_word\\_sizes](https://en.wikipedia.org/wiki/Word_(computer_architecture)#Table_of_word_sizes) – зразу можна згадати, що були ще й десяткові машини<sup>41</sup>.

---

<sup>40</sup>Остання причина є важливою значно рідше, ніж може здаватися – інженери кожної епохи змушені працювати, враховуючи її технологічні обмеження, не очевидні їх наступникам.

<sup>41</sup>Не знаю, чи на них використовувалося С, але якісь рудименти підготовки для цього в стандартах трапляються, див, наприклад, члени шаблону `std::numeric_limits`.

## Приклад – історія машинного слова x86

- З'явилася дана архітектура (ISA), як 16-бітна – процесори 8086/8088.
- При тому, на рівні асемблера, сумісна із 8-бітними попередниками – 8008 та 8080. (Див. також додаток.)
- Мала 16-бітні регістри, тому, природно, (машинним) словом, word, назвали комірку, розміром 16 біт, 2 байти.
- Потім, з приходом 80386, архітектура стала 32-бітною – зберігаючи сумісність із 16-бітним кодом.
- Тому, хоча регістри в 32-бітному режимі – 32-бітні, терміном "слово" називають 16-бітну комірку, а 32-бітну – подвійним словом, dword.
- З приходом 64-бітного розширення<sup>42</sup>, стало ще веселіше. Регістри 64-бітні, це "четвірне" слово, qword.
- Однак, хоча регістри 64-бітні, зроблено було це не так заради цілих чисел, як заради вказівників. Тому базове ціле число (int) залишилося 32-бітним. А архітектура використовує дещо неочевидні для невтаємничених правилами маніпуляції верхньою частиною регістрів.

---

<sup>42</sup>x86-64, не плутати із вже майже мертвою IA64.



- Детальніше див. [https://en.wikipedia.org/wiki/64-bit\\_computing#64-bit\\_data\\_models](https://en.wikipedia.org/wiki/64-bit_computing#64-bit_data_models) – LLP64/LP64:

Data model ↕	short (integer) ↕	int ↕	long (integer) ↕	long long ↕	pointers, size_t ↕	Sample operating systems ↕
LLP64, IL32P64	16	32	32	64	64	Microsoft Windows (x86-64 and IA-64) using Visual C++; and MinGW
LP64, I32LP64	16	32	64	64	64	Most Unix and Unix-like systems, e.g., Solaris, Linux, BSD, macOS. Windows when using Cygwin; z/OS
ILP64	16	64	64	64	64	HAL Computer Systems port of Solaris to the SPARC64
SILP64	64	64	64	64	64	Classic UNICOS <sup>[41]</sup> (versus UNICOS/mp, etc.)

## Продовжуючи із архітектурними відмінностями

- Розмір байта вже стабілізувався на 8 біт. Цього вимагає і POSIX і Windows. Доповнювальний код теж всезагальний. Винятки можуть бути поміж екзотичних DSP.
- Фактичний розмір машинного слова теж доволі обмежений. Зазвичай це 32 біти, рідше – 16 біт, для вказівників – 64 біти. (Винятки бувають, але рідко).
- Однак, пам'ять в системі може бути різних видів. Пам'ять коду, пам'ять даних, пам'ять на читання-запис та лише на читання і т.д. і т.п.
- Однак, CPU та підсистема пам'яті повинні мати право переставляти операції звертання. Також, хотілося б надати це право компілятору.
- Однак, обчислювачів у системі може бути багато.
- Однак, їх логіка узгодження їх ієрархій пам'яті може бути найрізнораманітнішою.
- І т.д. і т.п.

- Тому, стандарти С і С++ часто дуже неконкретні у низькорівневих речах.
- Причина: з одного боку, не змушувати компілятор робити щось неприродне для апаратного забезпечення, ціною втрати продуктивності, з іншого – залишати йому простір для оптимізації.
- Ціна – головна біль для програмістів.

# Модель машини C/C++ – 1

- Моделі машини C і C++ практично ідентичні<sup>43</sup>.
- Пам'ять – пронумерована послідовність байт. Кожен байт має унікальний номер, номери ці – впорядковані, можна говорити про попередні та наступні байти.
- Однак, нумерація може бути не неперервною – із проміжками. Конкретне представлення номерів байт (представлення вказівника) не фіксується, хоча, в цілому, допускається<sup>44</sup> конвертування у цілі типи достатнього розміру (`intptr_t`, `uintptr_t`) і назад.
- Пам'ять може бути різних видів – може існувати кілька "паралельних" адресних просторів.
- Фундаментальні типи безпосередньо відображаються на комірки пам'яті відповідних розмірів – без всіляких посередників.
- Компілятор має велику свободу щодо переставляння різних операцій, доти, поки кінцевий результат не відрізнятиметься.

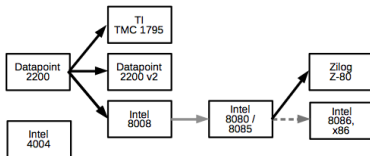
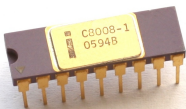
<sup>43</sup> Однак, на загал, це єдине місце де є сенс згадувати C і C++ разом – у всьому іншому, крім базового "синтаксичного ядра" це абсолютно різні мови.

<sup>44</sup> Але вважається поганою практикою! Виняток – хитрі оптимізації, зокрема – необхідні для реалізації lock-free структур даних. Причина – над вказівниками побітові операції безпосередньо виконувати не можна.

- Зокрема, важливим є поняття sequence point (точка перебігу) – точки, де всі попередні операції із побічними ефектами вже відбулися, а всі наступні – ще не розпочалися. компілятор не має права переставляти операції із побічними ефектами відносно них.
- C++11 його деталізує, щоб врахувати паралельні можливості сучасних машин: два вирази можуть бути впорядкованими – перший строго йде після другого (sequenced before/sequenced after); із невідомим впорядкуванням – спочатку один із них виконується, потім інший, але котрий перший – не визначено (indeterminately sequenced); виконання не впорядковане (unsequenced).
- Sequence point в С: оператор коми, логічні оператори && та ||, умова тернарного оператора, в кінці виразів, перед входом у функцію, після return із функції і ще ряд прикладів.
- Див. також "Abstraction and the C++ machine model" by B. Stroustrup (<http://www.stroustrup.com/abstraction-and-machine.pdf>), [https://en.wikipedia.org/wiki/Sequence\\_point](https://en.wikipedia.org/wiki/Sequence_point) та, скажімо, <https://blogs.msdn.microsoft.com/larryosterman/2007/05/16/the-c-abstract-machine/>.

- Надалі ми орієнтуватимемося на модель пам'яті та архітектури x86 та ARM, іноді згадуючи AVR8.

- Intel 8008 – далекий, не до кінця рідний, предок сучасних x86:
- Один із перших **мікропроцесорів** – 1972, (першим був не пов'язаний Intel 4004 – 1971).



- Являє собою інтегровану реалізацію центрального процесора "програмованого терміналу" CDC Datapoint 2200:



- <http://www.rogerarrick.com/osiris/burgett.txt> – Інтерв'ю з програмістом Intel, який розробляв інструменти розробки під

Цитата із казки "Чарівна круговерть" Євгена Наумова, яка гарно описує світанок комп'ютерного світу:

- Було це чи не було, казка це чи бувальщина, тільки бачив я все на власні очі, - почав Дід Драндулет. - Коли з'явилися перші автомобілі, вони самі ще не знали, якими їм бути. І небачені чудовиська запрудили шляхи.

Рогаті автомашини. Їжаки-автомобілі, в яких кузови були всіяні гострими сталевими голками. Гнучкі й довгі машини-гадюки. Круглі та тугі, ніби м'яч, грузовики. Вони могли підібгати колеса й котитися бездоріжжям. Автострибуни. Були навіть автомобілі, які могли розсипатися на безліч маленьких автомобільчиків, пролізти через будь-яку шпаринку, а потім з'єднатися і знову їхати далі.

Були машини з вісьмома колесами - чотири внизу, стільки ж зверху. Коли вони перевертались, то могли спокійнісінько їхати далі догори дном.

Зустрічалися триколісні та двоколісні. А одного разу з'явився автомобіль на одному колесі! Чудернацький вигляд він мав - довгий і тонкий, мов телеграфний стовп.

- Як же він їздив на одному колесі? - здивувався Малько-Ванько.

- Ще й як їздив! Автостоп - так його стали називати - хвацько носився не тільки по дорогах, а й по стежках, бордюрах і навіть... по карнизах будинків.

- А по карнизах навіщо? Хіба була потреба?

- Потреба була! Тому що - і це було найголовнішим - автомобілі не тільки не знали, який вигляд вони мали, а й не відали, як їздити. Кожен мчав, куди хотів і як хотів. Плутанина, безладдя! І була це країна Безладна Круговерть. На дорогах весь час траплялися сутички, аварії, чвари, скандали, бійки. То їжак штрикне когось своїми колючками, то Рогач потовче, то Автогадюка задушить у сталевих обіймах... На дорогах затори, пробки! А машин ставало дедалі більше.



