# EMBEDDED

## л.13
## Ефективний код

Палій Святослав

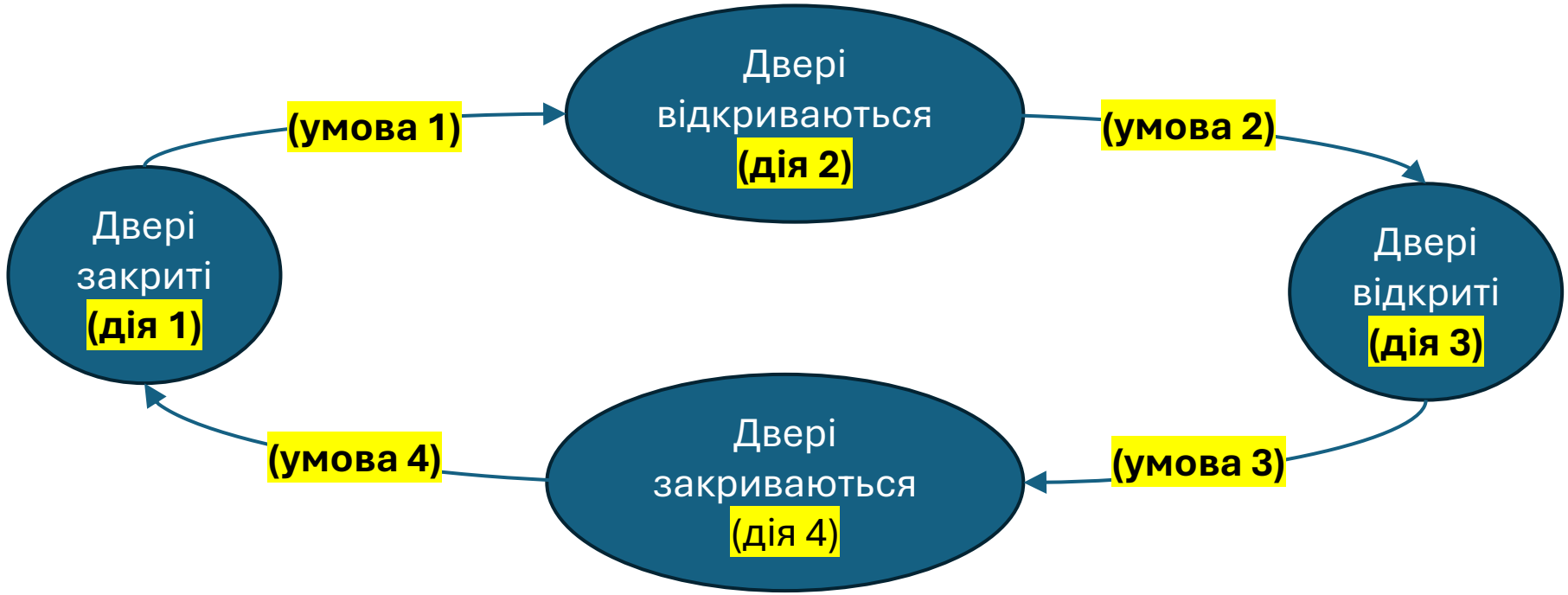
Український Католицький Університет
APPLIED SCIENCES FACULTY

# Перше правило ефективності

Для того щоб код мав шанси бути правильний та підтримуваний він повинен бути зрозумілим.

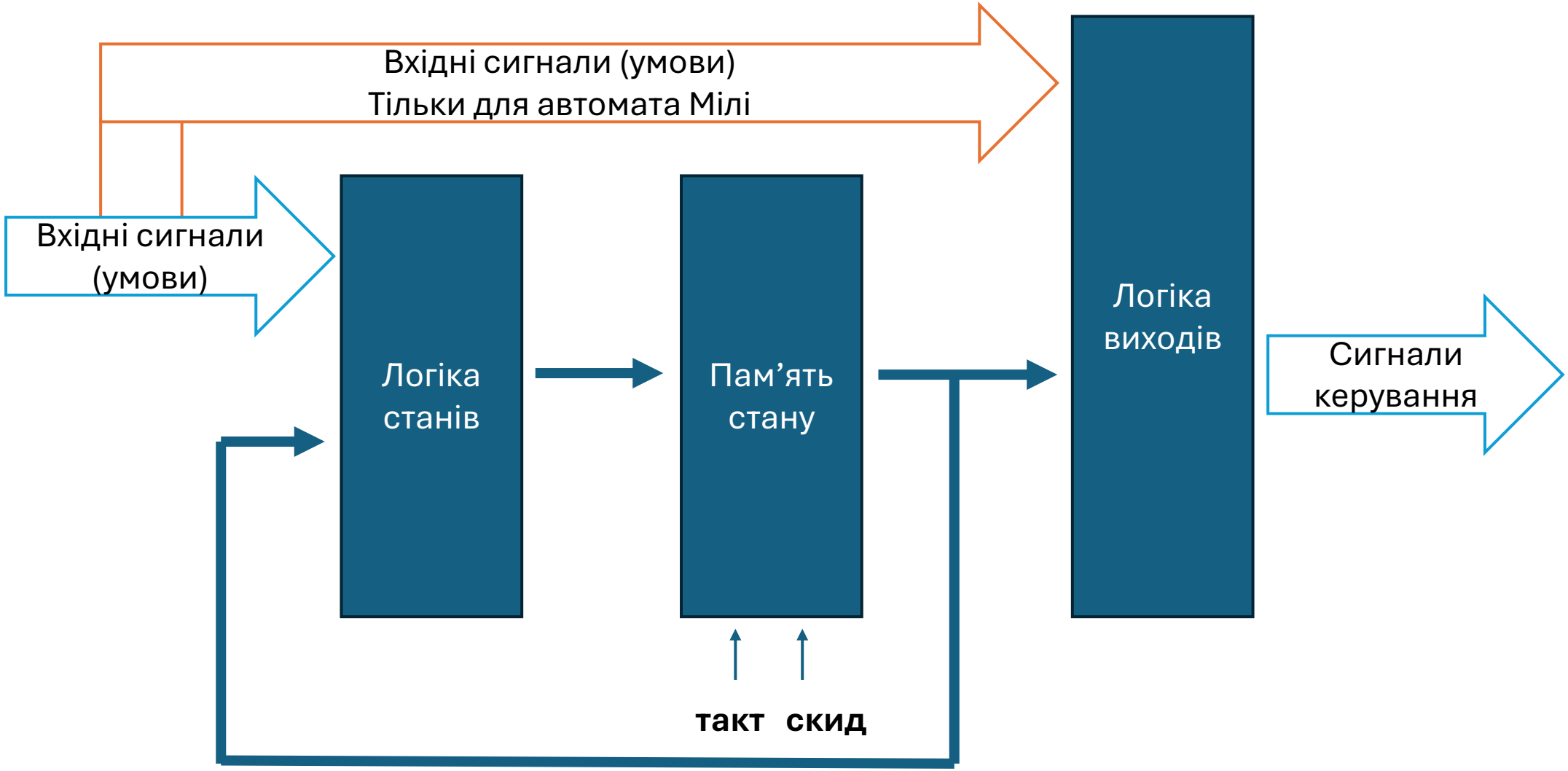
Оптимізація не повинна приносити в жертву читабельність чи правильність коду.

# Формальні підходи, автомат станів




**Автомат Мура**

- **Вихідні сигнали**: Залежать від поточного стану автомата.
- **Стан**: Залежить від попереднього стану та умови.
- **Переваги**: Простота реалізації та передбачуваність вихідних сигналів, оскільки вони залежать тільки від стану.
- **Недоліки**: Можлива затримка у реакції на вхідні сигнали, оскільки вихід змінюється тільки після переходу до нового стану.




**Автомат Мілі**

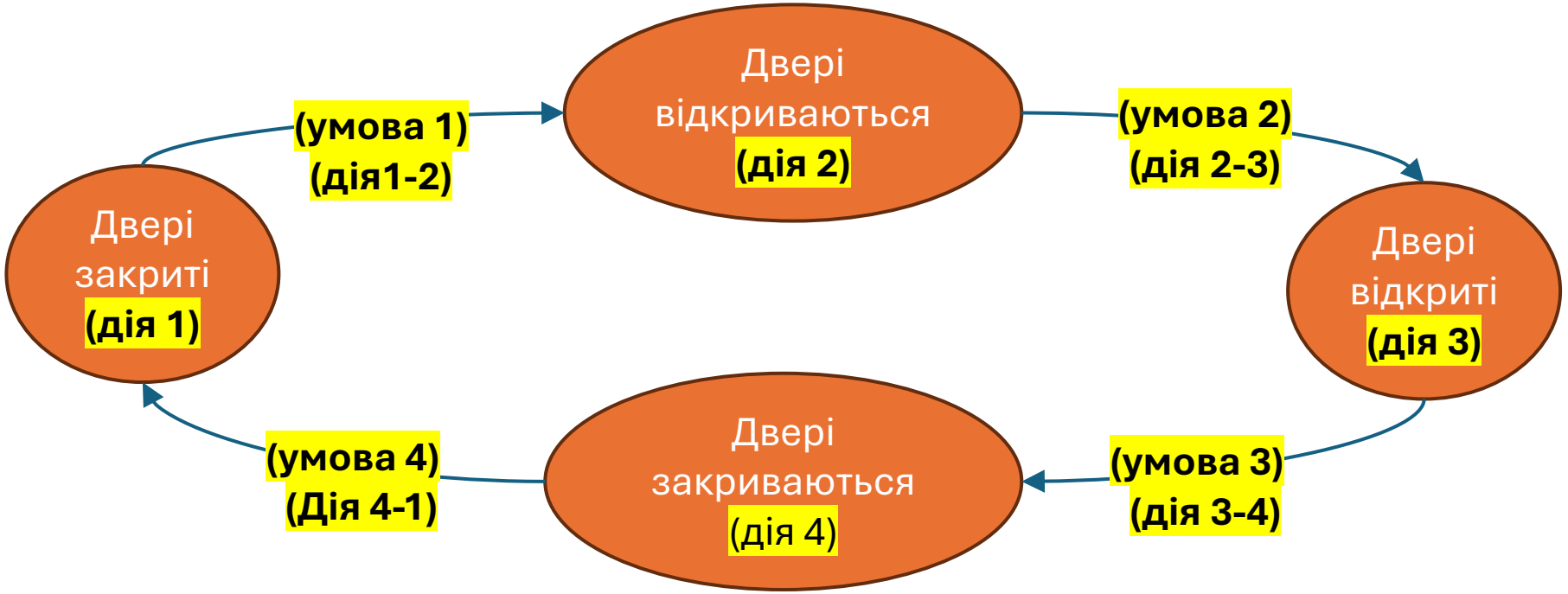
•**Вихідні сигнали**: Залежать від поточного стану та вхідного сигналу.

•**Стан**: Залежить від попереднього стану та умови.

•**Переваги**: Швидка реакція на вхідні сигнали, оскільки вихід може змінюватися на переході або й без переходу до нового стану.

•**Недоліки**: Складніша реалізація (особливо апаратна, не програмна), оскільки вихідні сигнали залежать від комбінації стану та вхідного сигналу.

# Формальні підходи, автомат станів Мілі

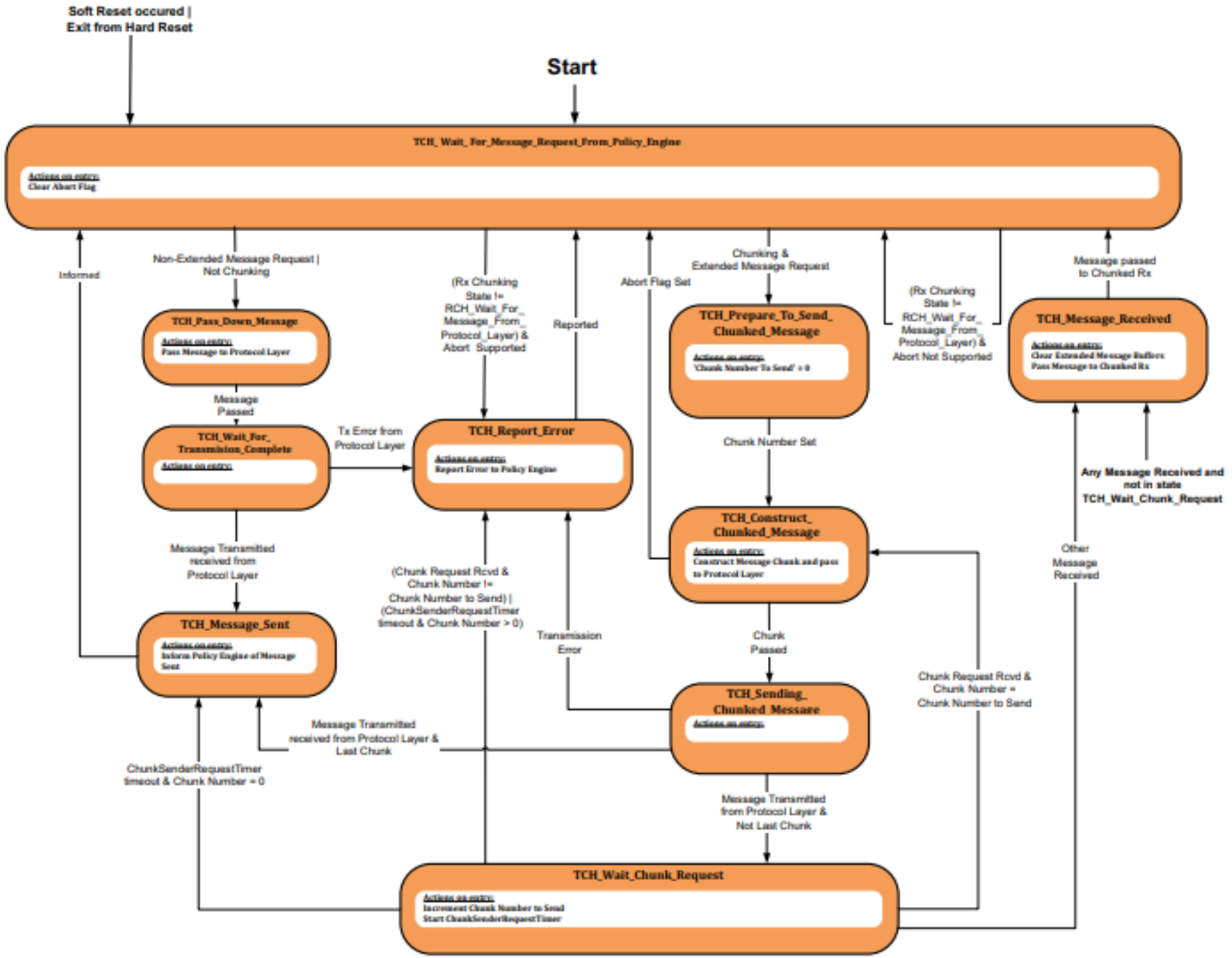


# Приклад, одна з USB-PD машин станів

**Джерело:**
Universal Serial Bus Power Delivery Specification

Revision: 3.2
Version: 1.1
Release Date: 2024-10

Figure 6.61 Chunked Tx State Diagram



```c
#include <stdio.h>

typedef enum {
    STATE_CLOSED,
    STATE_OPENING,
    STATE_OPEN,
    STATE_CLOSING
} State;

State currentState = STATE_CLOSED;

void handleClosedState() {
    printf("Двері закриті.\n");
}

void handleOpeningState() {
    printf("Двері відкриваються.\n");
}

void handleOpenState() {
    printf("Двері відкриті.\n");
}
```

По суті є прикладом автомата Мура

---

І ще одне: не бійтеся писати функції котрі викликаються один раз

--

default: обов'язковий !

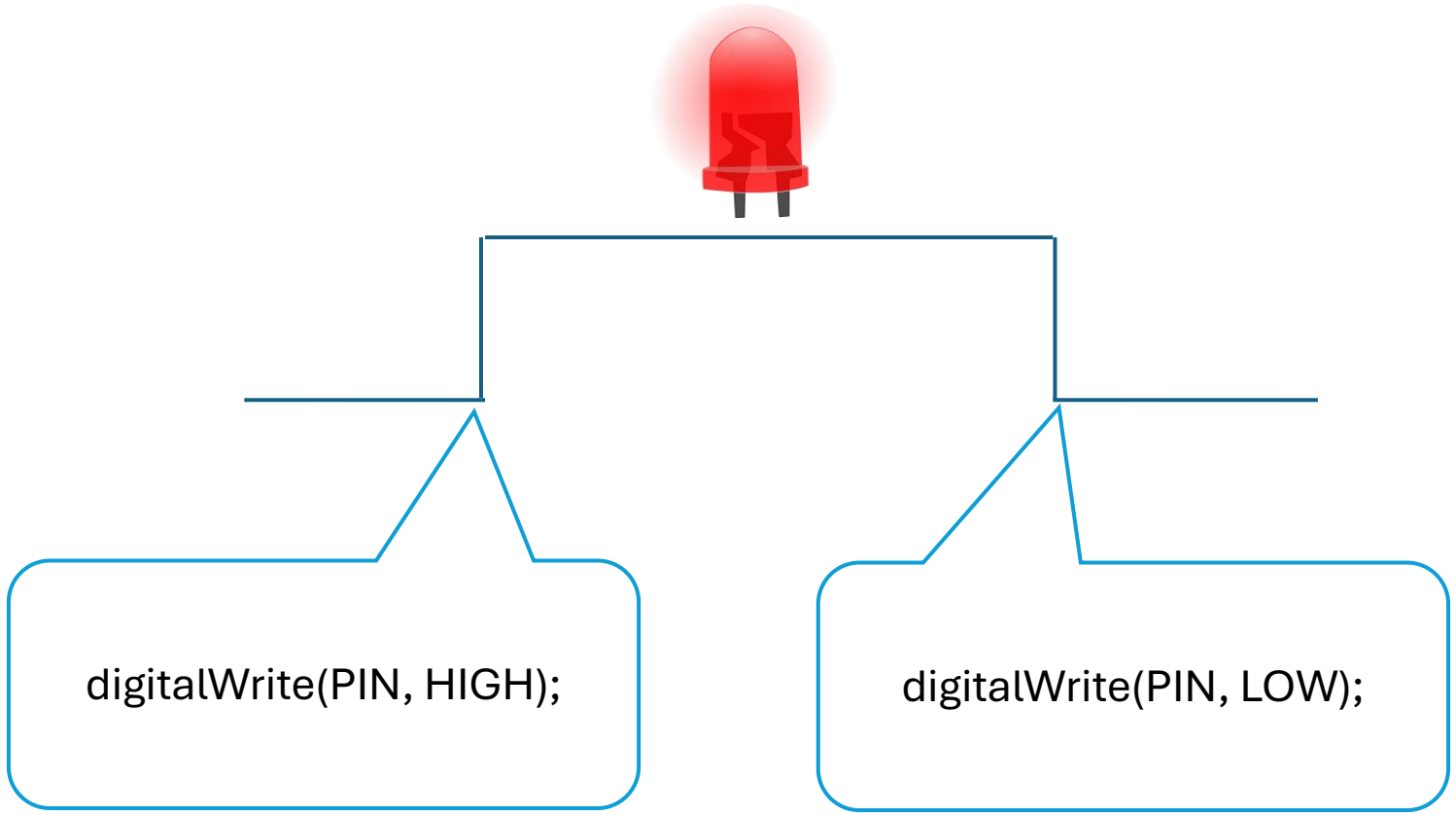
```c
void updateState() {
    switch (currentState) {
        case STATE_CLOSED:
            handleClosedState();
            // Логіка переходу до наступного стану
            if (Умова_1)
                currentState = STATE_OPENING;
            break;
        case STATE_OPENING:
            handleOpeningState();
            // Логіка переходу до наступного стану
            if (Умова_2)
                currentState = STATE_OPEN;
            break;
        case STATE_OPEN:
            handleOpenState();
            // Логіка переходу до наступного стану
            if (Умова_3)
                currentState = STATE_CLOSED;
            break;
        case STATE_CLOSING:
            // Логіка переходу до наступного стану
            if (Умова_4)
                currentState = STATE_CLOSED;
            break;
        default:
            printf("Невідомий стан.\n");
            break;
    }
}

int main() {
    for (;;) {
        updateState();
    }
}
```

```c
#include <stdio.h>

typedef enum {
    STATE_CLOSED,
    STATE_OPENING,
    STATE_OPEN,
    STATE_CLOSING
} State;

State currentState = STATE_CLOSED;

void handleClosedState() {
    printf("Двері закриті.\n");
}

void handleOpeningState() {
    printf("Двері відкриваються.\n");
}

void handleOnOpening() {
    // однократна дія при відкриванні дверей
}

void handleOpenState() {
    printf("Двері відкриті.\n");
}

void handleOnClosing() {
    // однократна дія при закриванні дверей
}
```

автомат Мілі

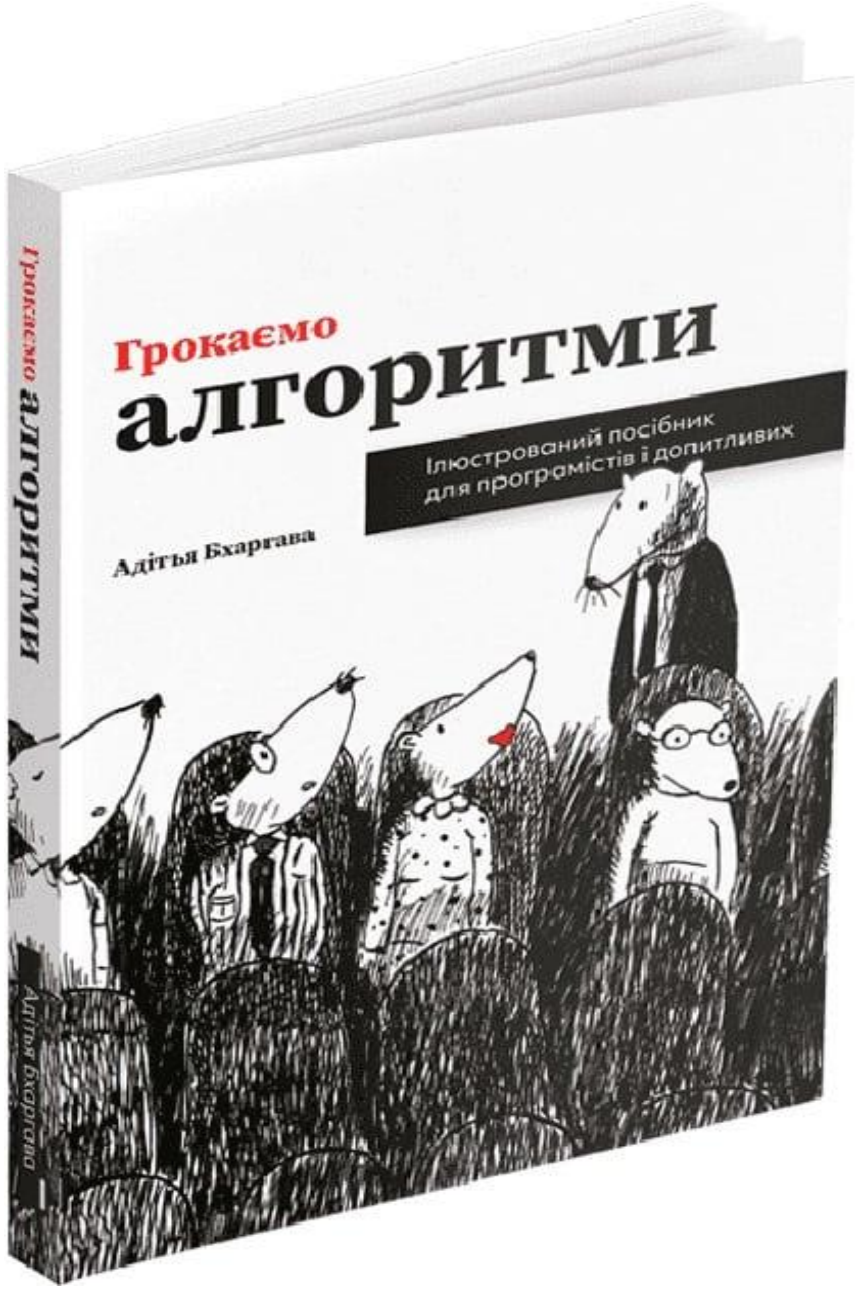
```c
void updateState() {
    switch (currentState) {
        case STATE_CLOSED:
            handleClosedState();
            // Логіка переходу до наступного стану
            if (Умова_1)
                currentState = STATE_OPENING;
            break;
        case STATE_OPENING:
            handleOpeningState();
            // Логіка переходу до наступного стану
            if (Умова_2) {
                handleOnOpening();
                currentState = STATE_OPEN;
            }
            break;
        case STATE_OPEN:
            handleOpenState();
            // Логіка переходу до наступного стану
            if (Умова_3)
                currentState = STATE_CLOSED;
            break;
            case STATE_=CLOSING:
            // Логіка переходу до наступного стану
            if (Умова_4) {
                handleOnClosing();
                currentState = STATE_CLOSED;
            }
            break;
        default:
            printf("Невідомий стан.\n");
            break;
    }
}

...
```

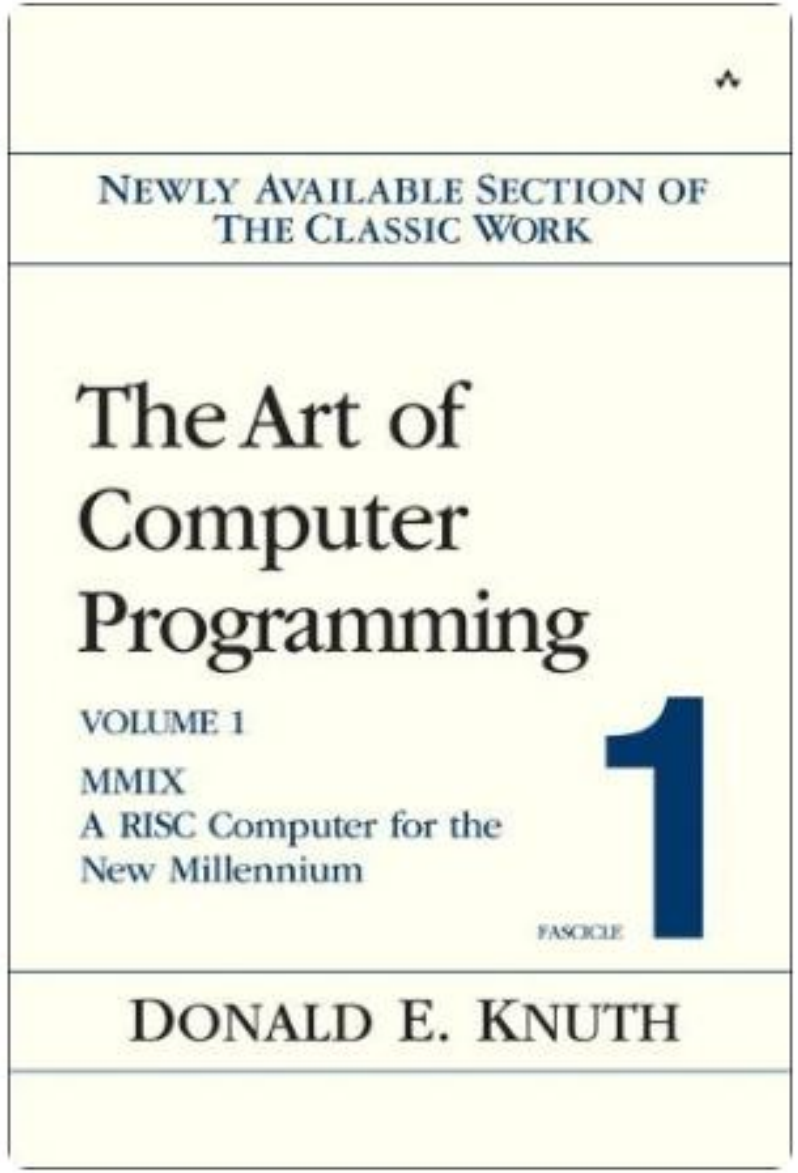
Тут якраз слід зауважити, що стан котрий на апаратному рівні триває в часі на програмному зазвичай ініціюється одномоментною дією.



# Друге правило ефективності

Потрібно вибирати ефективні алгоритми


Грокаємо алгоритми
Грокаємо
алгоритми
Ілюстрований посібник
для програмістів і допитливих
Адітья Бхаргава
Адітья Бхаргава


NEWLY AVAILABLE SECTION OF
THE CLASSIC WORK
The Art of
Computer
Programming
VOLUME 1
MMIX
A RISC Computer for the
New Millennium
FASCICLE
1
DONALD E. KNUTH

# Нотація Big O

Нотація Big O використовується для класифікації алгоритмів за тим, як їх час виконання або вимоги до пам'яті зростають зі збільшенням розміру вхідних даних. Вона допомагає програмістам оцінити ефективність алгоритмів і вибрати найкращий варіант для конкретної задачі

### Приклади Нотації Big O

- **O(1)**: Константна складність. Алгоритм виконується за постійний час, незалежно від розміру вхідних даних.
- **O(n)**: Лінійна складність. Час виконання алгоритму зростає пропорційно до розміру вхідних даних.
- **O($n^2$)**: Квадратична складність. Час виконання алгоритму зростає пропорційно до квадрату розміру вхідних даних.
- **O(log n)**: Логарифмічна складність. Час виконання алгоритму зростає логарифмічно зі збільшенням розміру вхідних даних.

### Застосування Нотації Big O

Нотація Big O широко використовується в комп'ютерних науках для аналізу алгоритмів. Вона допомагає визначити найгірший випадок виконання алгоритму, що є важливим для оцінки його ефективності та надійності

**Порівняння алгоритмів**: Використовуючи нотацію Big O, можна порівнювати різні алгоритми за їх асимптотичною складністю. Наприклад, алгоритм з O(n) буде більш ефективним для великих вхідних даних, ніж алгоритм з O(n^2).

**Оптимізація**: Знання асимптотичної складності алгоритмів дозволяє програмістам оптимізувати код і вибирати найбільш ефективні методи для вирішення задач.

### Висновок

Нотація Big O є важливим інструментом для аналізу та порівняння алгоритмів. Вона допомагає програмістам оцінити ефективність алгоритмів і вибрати найкращий варіант для конкретної задачі. Розуміння нотації Big O є необхідним для успішного програмування та розробки ефективних алгоритмів

# Нотація Big O - приклади

- Пошук найбільшого чи найменшого числа в непосортованому масиві має лінійну складність **O(n)**

```
for (int i=0; i<ARRAY_SIZE; i++)
{ ... }
```

- Бульбашкове сортування має кватратичну складність **O($n^2$)**

```
for (int i=0; i<ARRAY_SIZE; i++)
  for (int j=0; j<ARRAY_SIZE; j++)
    { ... }
```

- Швидке сортування має лінеаритмічну складність **O(n*log(n) = log(n!))**

- Двійковий пошук в сортованому списку має логарифмічну складність **O(log(n))**

```
int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1;  // Елемент не знайдено
}
```

Зауважте також можливість сортувати дані один раз, а потім використовувати переваги сортованих даних багато разів. Варто розглядати можливість вбудувати сортування даних відразу в процедуру їх накопичення.

# Угорський метод

**Угорський метод** — це алгоритм, який використовується для вирішення задач оптимізації, зокрема задачі призначення. Він дозволяє мінімізувати витрати або максимізувати ефективність при розподілі ресурсів чи завдань між агентами. Цей метод був розроблений угорським математиком Гарольдом Куном у 1955 році і вдосконалений Джеймсом Мункресом у 1957 році, тому його також називають алгоритмом Куна-Мункреса

**Основи угорського методу**
Угорський метод базується на теорії графів і лінійному програмуванні. Він використовує матрицю витрат, де кожен елемент представляє витрати на виконання певного завдання певним агентом. Метою є знайти оптимальний розподіл завдань, який мінімізує загальні витрати.

**Етапи угорського методу**:
**1.Побудова матриці витрат**: Створюється матриця, де рядки представляють агентів, а стовпці — завдання. Кожен елемент матриці містить витрати на виконання завдання агентом.
**2.Редукція матриці**: Від кожного елемента матриці віднімається мінімальне значення в його рядку і стовпці, щоб отримати нову матрицю.
**3.Побудова графа рівності**: Створюється граф, де ребра з'єднують агента і завдання з нульовими витратами.
**4.Призначення завдань**: Завдання призначаються агентам таким чином, щоб мінімізувати загальні витрати.

**Застосування угорського методу**
Угорський метод широко використовується в різних галузях, включаючи логістику, виробництво, управління проектами та навіть в обробці зображень. Ось кілька прикладів його застосування:
**1.Розподіл завдань**: Угорський метод допомагає оптимально розподілити завдання між працівниками, мінімізуючи витрати на виконання кожного завдання.
**2.Оптимізація логістики**: Використовується для оптимізації розподілу вантажів між транспортними засобами, щоб мінімізувати витрати на перевезення.
**3.Обробка зображень**: Угорський метод застосовується для зіставлення об'єктів на зображеннях, що дозволяє покращити точність розпізнавання.

# Cordic алгоритми

**CORDIC** (Coordinate Rotation Digital Computer) — це ітераційний метод, який використовує обертання для обчислення широкого спектру елементарних функцій, таких як тригонометричні функції (sin, cos, tan), гіперболічні функції, логарифми та квадратні корені

Цей алгоритм був розроблений Джеком Вольдом у 1959 році для використання в цифрових обчислювальних системах

**Основи алгоритму CORDIC**

CORDIC базується на ідеї обертання векторів у площині. Він використовує лише прості операції додавання, віднімання та зсуву, що робить його дуже ефективним для реалізації на апаратному рівні, особливо на малопотужних пристроях, таких як мікроконтролери та програмовані логічні інтегральні схеми (ПЛІС)

**Основні етапи алгоритму CORDIC**:
**1.Ініціалізація**: Встановлення початкових значень векторів.
**2.Ітерації**: Виконання серії обертань, кожне з яких наближає вектор до бажаного результату.
**3.Корекція**: Масштабування результату для отримання точного значення.

**Застосування алгоритму CORDIC**

CORDIC широко використовується в різних галузях, включаючи цифрову обробку сигналів, машинну графіку, навігацію та управління рухом
CORDIC дозволяє обчислювати значення sin, cos, tan без використання операцій з плаваючою точкою, що робить його ідеальним для вбудованих систем.
Обчислення sinh, cosh, tanh також може бути виконано за допомогою CORDIC.
CORDIC може бути використаний для обчислення логарифмів та квадратних коренів, що робить його універсальним інструментом для математичних обчислень.

# Жадібні алгоритми

**Жадібні алгоритми** (Greedy algorithms) — це прості та прямолінійні евристичні алгоритми, які приймають найкраще рішення на кожному етапі, виходячи з наявних даних, не зважаючи на можливі наслідки. Вони сподіваються, що врешті-решт отримають оптимальний розв'язок. Жадібні алгоритми легкі в реалізації і часто дуже ефективні за часом виконання, хоча не завжди можуть забезпечити оптимальний розв'язок для всіх задач

**Принцип жадібного вибору**

До оптимізаційної задачі можна застосувати принцип жадібного вибору, якщо послідовність локально оптимальних виборів дає глобально-оптимальний розв'язок.

Наприклад, у задачі комівояжера жадібний алгоритм може вибирати найближче з невідвіданих міст на кожному етапі.

**Обмеження жадібних алгоритмів**

Жадібні алгоритми можуть бути неефективними для задач, які не мають властивості оптимальної підструктури. Наприклад, у задачі комівояжера жадібний алгоритм може видати найгірший з можливих розв'язків. Вони також можуть бути «короткозорими» і «невідновлюваними», тобто не переглядають попередні вибори для здійснення наступного.

.

```c
typedef struct {
    int weight;
    int value;
} Item;


int knapsack(Item items[], int n, int W) {
    int totalValue = 0;
    for (int i = 0; i < n; i++) {
        if (items[i].weight <= W) {
            W -= items[i].weight;
            totalValue += items[i].value;
        }
    }
    return totalValue;
}
```

# Третє правило ефективності

Потрібно враховувати особливості архітектури та типів даних

# Розрядність даних

```
unsigned long startTime;
unsigned long endTime;

void setup() {
  int a, b, c;

  Serial.begin(9600);
  startTime = millis();  // Початок вимірювання часу
  a = 0;
  b = 2;
  c = 1;
  for (int outer_loop2 = 0; outer_loop2 < 1000; outer_loop2++)
  {
    a = 0;
    for (int i = 0; i < 255; i++) {
      a = a + b - c;
//      if (a > 100) a -= 100;
      b++;
      c++;
    }
  }
  endTime = millis();  // Кінець вимірювання часу
  Serial.print("Час виконання циклу: ");
  Serial.print(endTime - startTime);
  Serial.print(" мс ");
  Serial.print("Значення a: ");
  Serial.println(a);
}

void loop() {
  // Нічого не робимо в loop
}
```

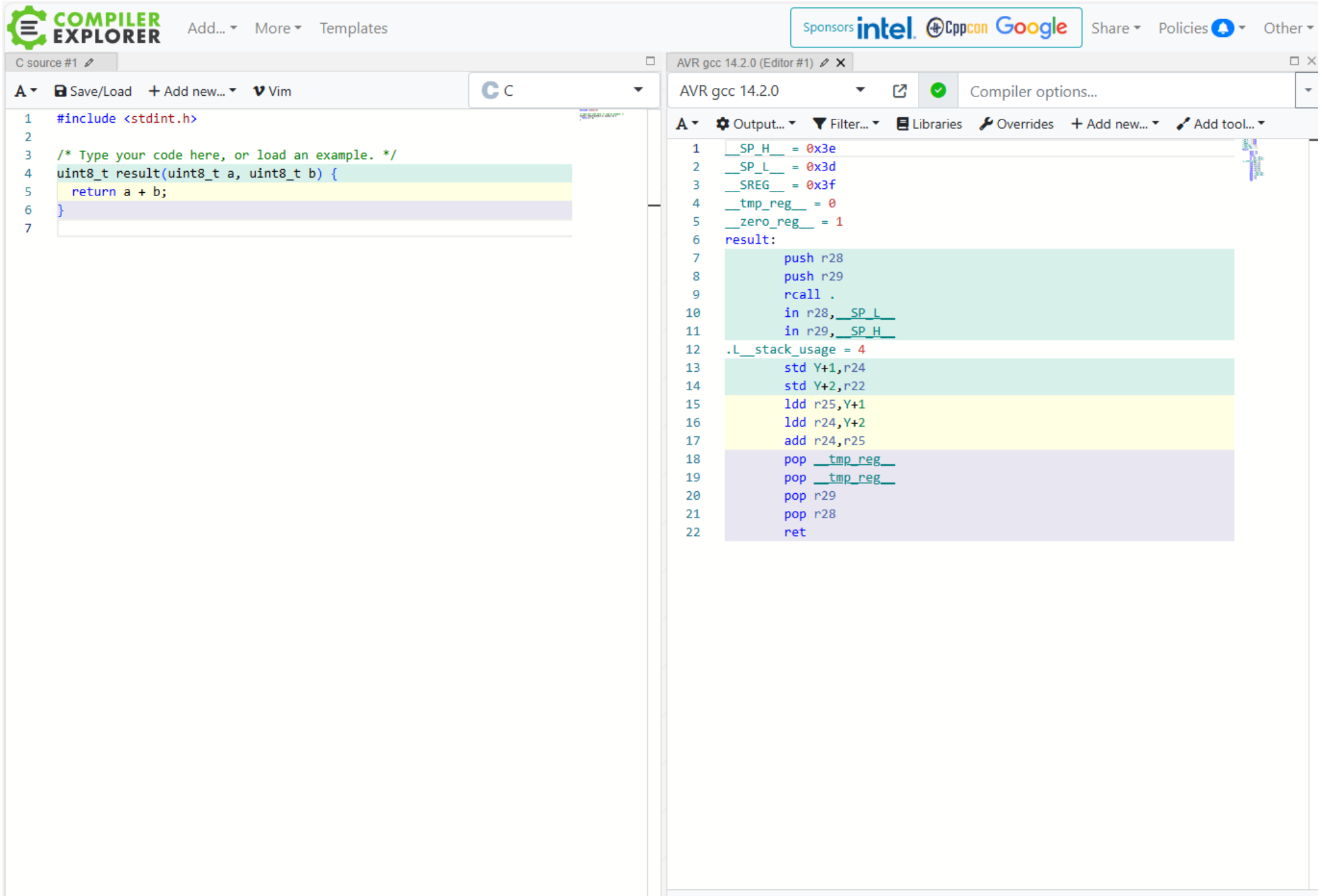
```ino
unsigned long startTime;
unsigned long endTime;

void setup() {
  int a, b, c;

  Serial.begin(9600);
  startTime = millis();  // Початок вимір
  a = 0;
  b = 2;
  c = 1;
  for (int outer_loop2 = 0; outer_loop2 < 1000; outer_loop2++)
  {
    a = 0;
    for (int i = 0; i < 255; i++) {
      a = a + b - c;
      if (a > 100) a -= 100;
      b++;
      c++;
    }
  }
  endTime = millis();  // Кінець вимірювання часу
  Serial.print("Час виконання циклу: ");
  Serial.print(endTime - startTime);
  Serial.print(" мс ");
  Serial.print("Значення a: ");
  Serial.println(a);
}

void loop() {
  // Нічого не робимо в loop
}
```

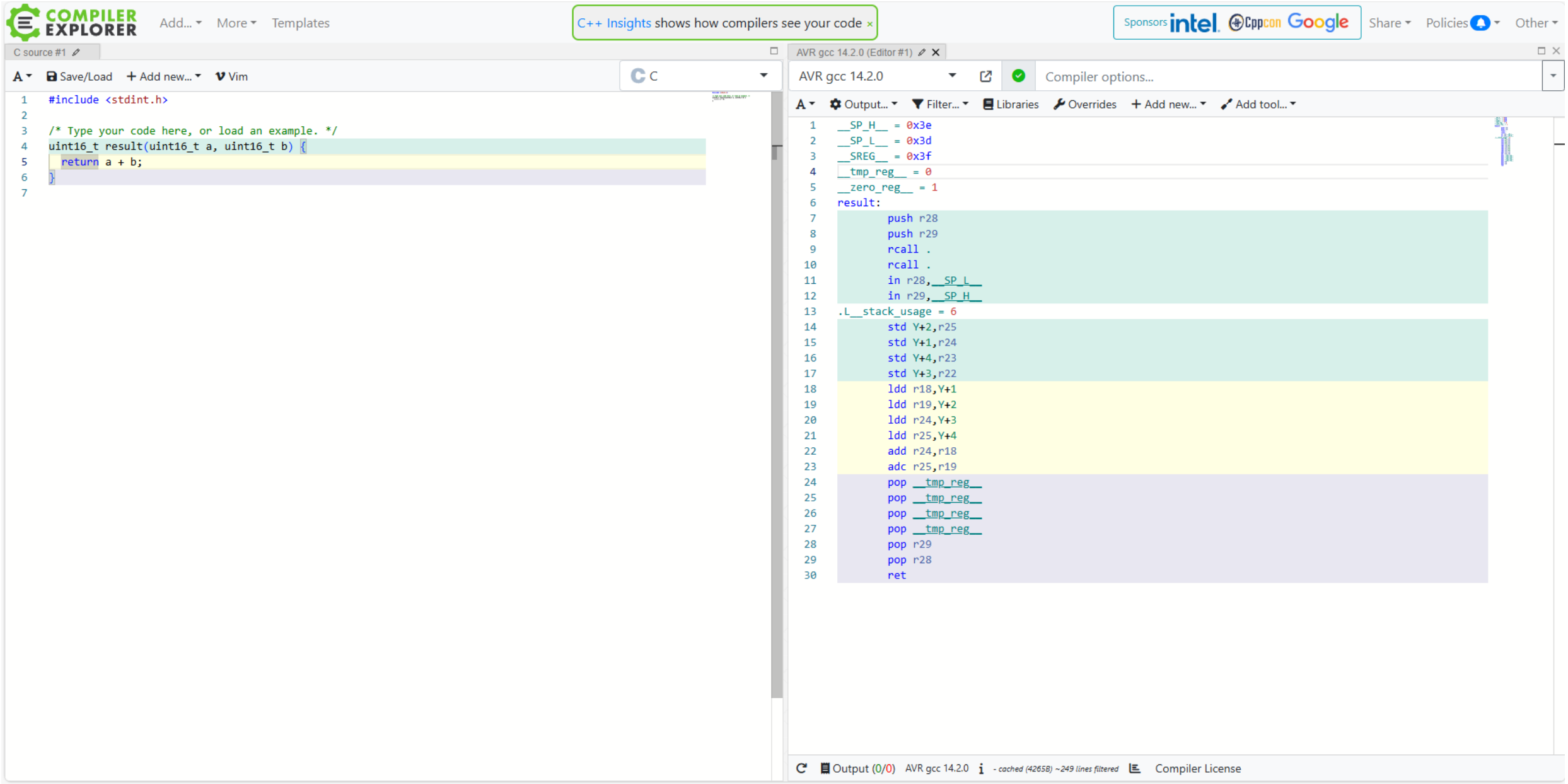
Тут слід практично подивитися як типи даних впливають на швидкодію.

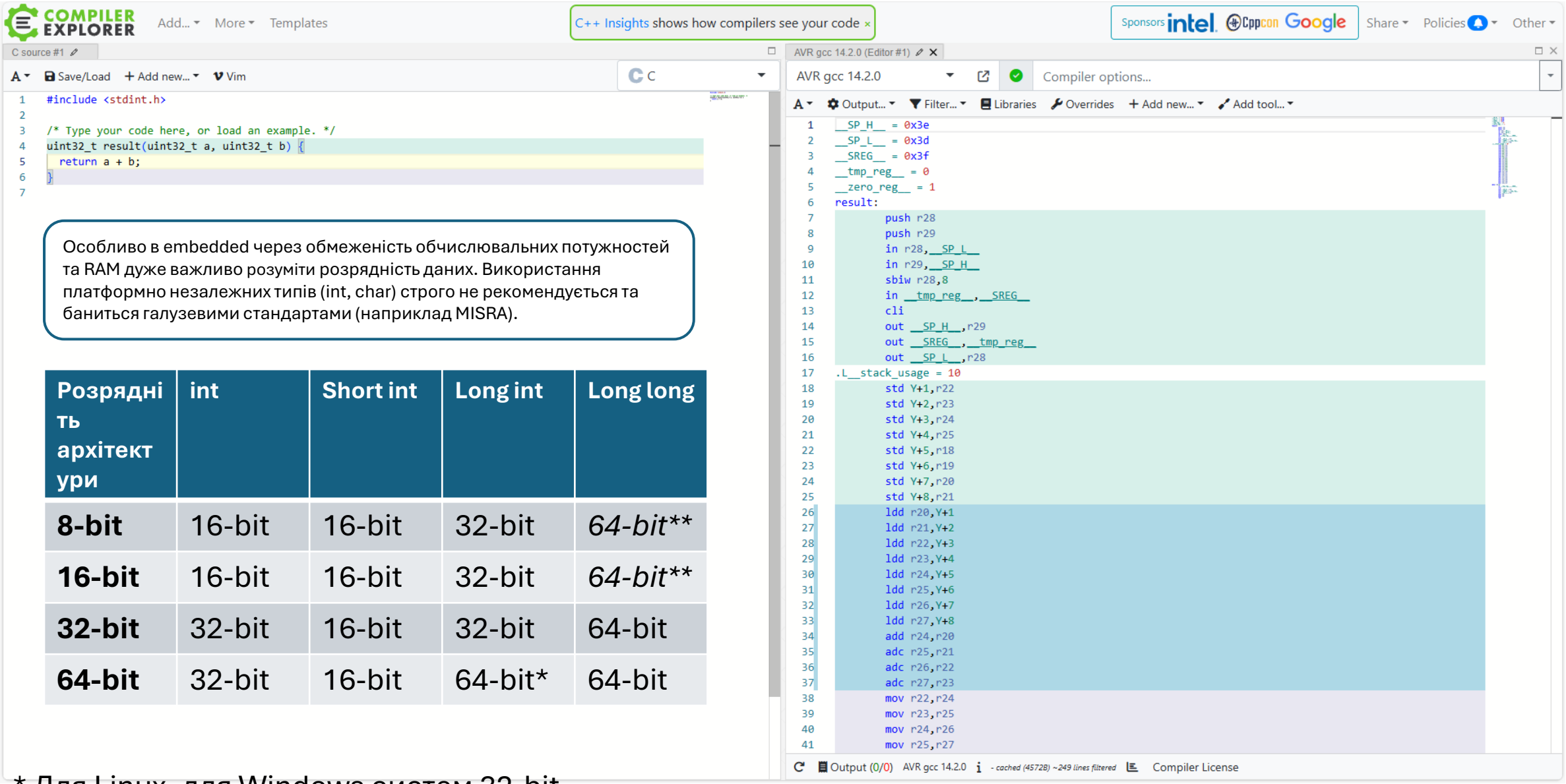
Можна ще глянути як міняє **volatile**

Без цієї стрічки компілято сам здатний порахувати результат наперед

Очевидно що найшвидше будуть виконуватися операції в «нативному» типі даних.


COMPILER EXPLORER
Add...
More
Templates
Sponsors intel Cppcon Google
Share
Policies
Other
C source #1
Save/Load
Add new...
Vim
C
#include <stdint.h>
/* Type your code here, or load an example. */
uint8_t result(uint8_t a, uint8_t b) {
return a + b;
}
AVR gcc 14.2.0 (Editor #1)
AVR gcc 14.2.0
Compiler options...
Output...
Filter...
Libraries
Overrides
Add new...
Add tool...
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__SREG__ = 0x3f
__tmp_reg__ = 0
__zero_reg__ = 1
result:
push r28
push r29
rcall .
in r28,__SP_L__
in r29,__SP_H__
.L__stack_usage = 4
std Y+1,r24
std Y+2,r22
ldd r25,Y+1
ldd r24,Y+2
add r24,r25
pop __tmp_reg__
pop __tmp_reg__
pop r29
pop r28
ret


COMPILER EXPLORER
Add...
More
Templates
C++ Insights shows how compilers see your code
Sponsors intel CppCon Google
Share
Policies
Other
C source #1
Save/Load
Add new...
Vim
C
#include <stdint.h>
/* Type your code here, or load an example. */
uint16_t result(uint16_t a, uint16_t b) {
return a + b;
}
AVR gcc 14.2.0 (Editor #1)
AVR gcc 14.2.0
Compiler options...
Output...
Filter...
Libraries
Overrides
Add new...
Add tool...
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__SREG__ = 0x3f
__tmp_reg__ = 0
__zero_reg__ = 1
result:
push r28
push r29
rcall .
rcall .
in r28,__SP_L__
in r29,__SP_H__
.L__stack_usage = 6
std Y+2,r25
std Y+1,r24
std Y+4,r23
std Y+3,r22
ldd r18,Y+1
ldd r19,Y+2
ldd r24,Y+3
ldd r25,Y+4
add r24,r18
adc r25,r19
pop __tmp_reg__
pop __tmp_reg__
pop __tmp_reg__
pop __tmp_reg__
pop r29
pop r28
ret
Output (0/0)
AVR gcc 14.2.0
- cached (42658) ~249 lines filtered
Compiler License




Особливо в embedded через обмеженість обчислювальних потужностей та RAM дуже важливо розуміти розрядність даних. Використання платформно незалежних типів (int, char) строго не рекомендується та баниться галузевими стандартами (наприклад MISRA).

| Розрядність архітектури | int | Short int | Long int | Long long |
|---|---|---|---|---|
| **8-bit** | 16-bit | 16-bit | 32-bit | *64-bit*** |
| **16-bit** | 16-bit | 16-bit | 32-bit | *64-bit*** |
| **32-bit** | 32-bit | 16-bit | 32-bit | 64-bit |
| **64-bit** | 32-bit | 16-bit | 64-bit* | 64-bit |

* Для Linux, для Windows систем 32-bit

** часто просто не підтримуються

# Знаковий мікс

```c
/******************************************************************************

                              Online C Compiler.
                Code, Compile, Run and Debug C program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/
#include <stdio.h>
#include <stdint.h>

int main() {
    char a = 251;
    unsigned char b = a;

    printf("a = %X", a);
    printf("\nb = %X", b);

    if (a == b)
        printf("\n Same");
    else
        printf("\n Not Same");

    return 0;
}
```

```c
/******************************************************************************

                              Online C Compiler.
                  Code, Compile, Run and Debug C program online.
Write your code in this editor and press "Run" button to compile and execute it.

*******************************************************************************/
#include <stdio.h>
#include <stdint.h>

int main() {
    char a = 251;
    unsigned char b = a;

    printf("a = %X", a);
    printf("\nb = %X", b);

    if (a == b)
        printf("\n Same");
    else
        printf("\n Not Same");

    return 0;
}
```

```
a = FFFFFFFB
b = FB
 Not Same

...Program finished with exit code 0
Press ENTER to exit console.
```

# І ще трохи прикольних алгоритмів

Використання побітових операцій може значно прискорити виконання арифметичних операцій. Наприклад, множення на степені 2 можна замінити зсувом вліво, а ділення — зсувом вправо.

```
int multiplyByTwo(int x) {
    return x << 1;  // Зсув вліво на один біт
}

   int divideByTwo(int x) {
    return x >> 1;  // Зсув вправо на один біт
}
```

Перевірка парності числа може бути виконана за допомогою побітової операції AND.

```
bool isEven(int x) {
    return (x & 1) == 0;  // Якщо останній біт - 0, число парне
}
```

Обмін значень двох змінних можна виконати без використання додаткової змінної за допомогою побітових операцій XOR.

```
void swap(int *a, int *b) {
    *a = *a ^ *b;
    *b = *a ^ *b;
    *a = *a ^ *b;
}
```

Алгоритм Евкліда для знаходження найбільшого спільного дільника двох чисел є дуже ефективним.

```
int gcd(int a, int b) {
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}
```

Замість використання умовних операторів можна використовувати побітові операції для знаходження мінімального та максимального значення.

```
int min(int a, int b) {
    return b ^ ((a ^ b) & -(a < b));
}

int max(int a, int b) {
    return a ^ ((a ^ b) & -(a < b));
}
```

# Ділення двох чисел з отриманням округленого результату.

```
int divideAndRound(int numerator, int denominator) {
    if (denominator == 0) {
        printf("Помилка: ділення на нуль.\n");
        return -1;  // Повертаємо -1 у випадку помилки
    }
    return (numerator + denominator / 2) / denominator;
}
```

# Щиро дякую!