Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

ЗВІТ

з лабораторних робіт №5-6
з дисципліни ”Програмне забезпечення комп’ютерних систем”

Тема: Паралельні комп’ютерні системи

Варіант №5 ***(81)***

Виконав:
Студент 1 курсу, групи ІМ-51мн
Ковальов Олександр

Перевірила:
к.т.н., Русанова Ольга Веніаминівна

Дата здачі: 2.12.2025

КИЇВ – 2025

Мета роботи. Визначення часу виконання арифметичного виразу в заданій ПКС з урахуванням кількості процесорів. Вибір оптимального способу розпаралелювання АВ при його виконанні на заданій ПКС з урахуванням її параметрів.

Вхідні дані: Дерево паралельної форми арифметичного виразу (результат виконання лабораторної роботи №2) та множина дерев (графів) паралельної форми, отриманих у результаті виконання лабораторних робіт №2-4. Крім того, кожний студент узгоджує з викладачем тип ПКС та її параметри, такі як кількість процесорів (шарів конвейєра) і час виконання різних алгебраїчних операцій. За варіантом (*81*) визначена **векторна система** з 1 суматором, 1 блоком віднімання, 1 блоком множення та 1 блоком ділення.

Завдання: Розробити програмну модель роботи заданої паралельної комп'ютерної системи з урахуванням кількості процесорів (шарів) та часу виконання різних алгебраїчних операцій. При виконанні цього завдання студент може використовувати будь-яку мову програмування. Продемонструвати роботу моделі паралельної системи при виконанні вхідного арифметичного виразу. Визначити час виконання АВ, коефіцієнт прискорення та ефективність роботи системи. З одержаної множини графів паралельних форм вибрати оптимальний(-і) для реалізації в ПКС заданої архітектури. Провести дослідження за направленим пошуком графів АВ, адекватних заданій архітектурі (графу) системи. Для виконання поставлених завдань необхідно промоделювати виконання усіх еквівалентних графів паралельних форм АВ на заданій ПКС. Визначити час їх виконання, коефіцієнти прискорення та ефективності роботи системи. Занести отримані результати у таблицю та вибрати оптимальну форму АВ.

Хід роботи.

В рамках виконання роботи було розроблено та інтегровано підсистему симуляції паралельної комп'ютерної системи (ПКС) векторного типу для аналізу ефективності виконання арифметичних виразів.

Робота розпочалася з реалізації модуля `pcs` (Parallel Computer System), який відповідає за моделювання апаратної частини. Враховуючи отримане завдання, було сконфігуровано модель векторної системи з наступними характеристиками ресурсів: 1 суматор, 1 блок віднімання, 1 блок множення та 1 блок ділення. Для кожної операції встановлено відповідний час виконання у тактах (Ticks) – додавання 1 такт, віднімання 1 так, множення 2 такти, ділення 4 такти.

Ключовим етапом реалізації стала розробка симулятора на базі алгоритму List Scheduling (планування списків). Оскільки абстрактне синтаксичне дерево (AST), отримане в попередніх роботах, є рекурсивною структурою, непристосованою для прямого планування, першим кроком симуляції є конвертація AST у спрямований ациклічний граф завдань (Task Graph). Функція `flatten_ast` рекурсивно обходить дерево та формує плоский список об'єктів `Task`. Кожне завдання отримує унікальний ідентифікатор, тип операції (що відповідає типу функціонального блоку процесора) та список ідентифікаторів завдань-залежностей, які мають бути завершені перед початком поточного. Листові вузли дерева (числа та змінні) обробляються як операції типу `Load` з нульовим часом виконання, що імітує миттєву доступність операндів з пам'яті.

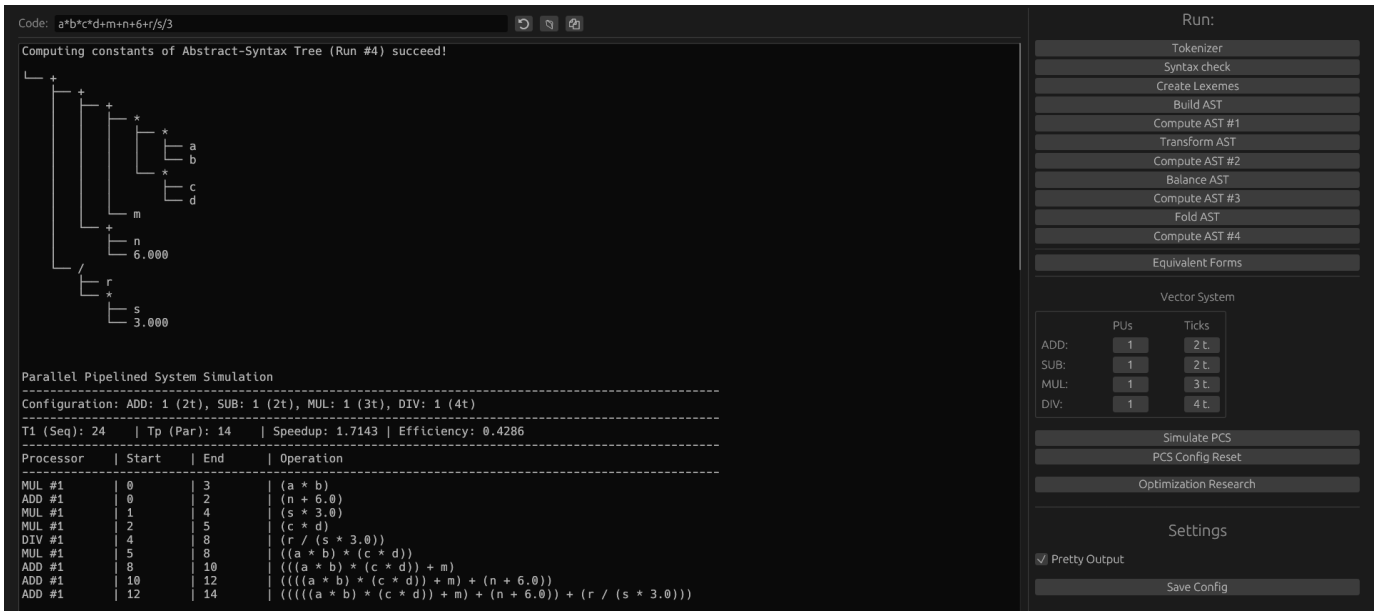
Сам процес симуляції реалізовано як потактовий цикл (tick-based simulation). На кожному такті алгоритм виконує наступні дії:

1. Перевіряє завершення активних завдань та звільняє відповідні ресурси процесора.
2. Формує чергу готових до виконання завдань (`ready_queue`), відбираючи ті вузли графа, для яких задоволені всі залежності по даних (всі попередні операції завершено).
3. Сортує чергу готових завдань за евристикою "найдовший час виконання першим" (Longest Processing Time first) для оптимізації щільності розкладу.
4. Намагається призначити завдання з черги на вільні функціональні блоки. Якщо відповідний ресурс (наприклад, блок множення) вільний, завдання запускається, а ресурс блокується на кількість тактів, що відповідає вартості операції. В іншому випадку завдання залишається в черзі до наступного такту.

Після завершення симуляції розраховуються ключові метрики ефективності:

- **Послідовний час –** T_1 : сума часів виконання всіх арифметичних операцій в однопроцесорному режимі.
- **Паралельний час –** T_p : час закінчення останнього завдання в симульованому розкладі.
- **Коефіцієнт прискорення –** S: відношення $T_1 \div T_p$.

- **Ефективність –** E: відношення S до загальної кількості функціональних блоків.

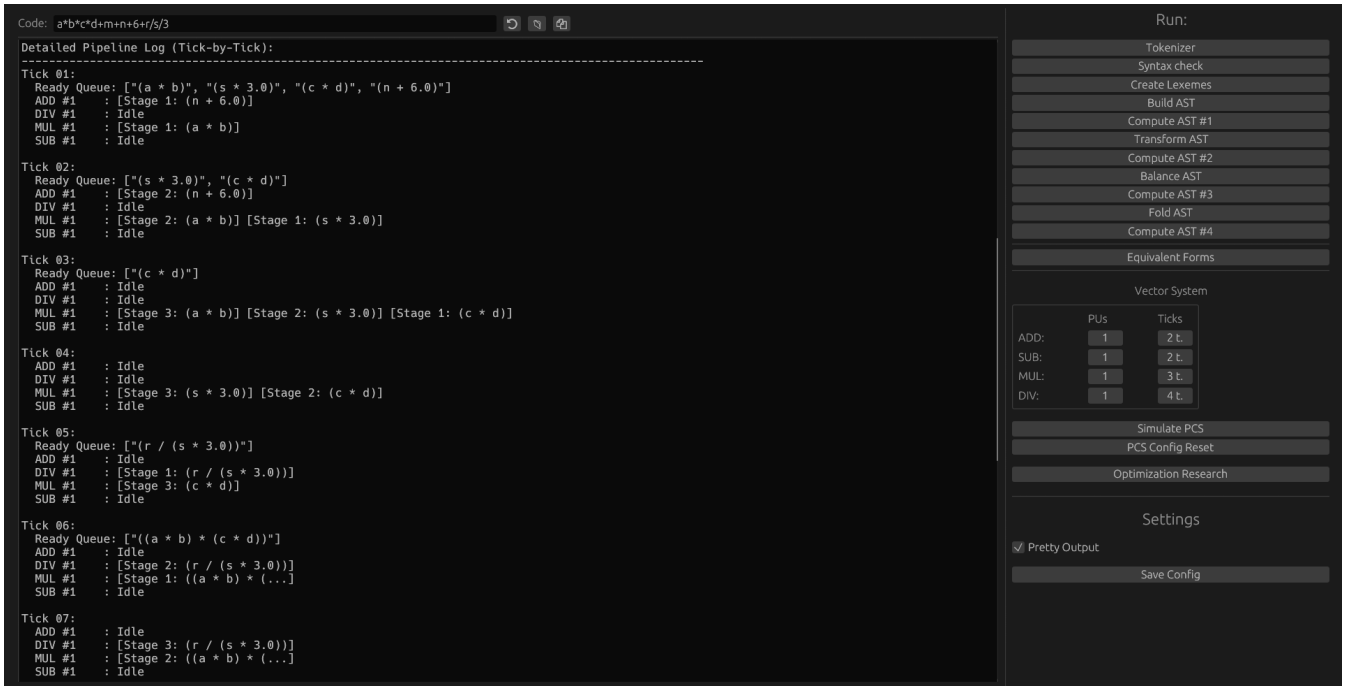


Згодом функціонал системи було суттєво розширено шляхом впровадження підтримки **конвеєрного виконання операцій (pipelining)**. У попередній реалізації функціональний блок вважався зайнятим протягом усього часу виконання операції (latency). В оновленій моделі було змінено логіку керування ресурсами: тепер функціональні блоки здатні приймати нові вхідні дані на кожному такті (пропускна здатність 1 операція/такт), навіть якщо обробка попередньої операції ще триває на внутрішніх стадіях конвеєра. Це дозволило реалізувати часове перекриття виконання операцій (instruction overlap), що значно зменшує загальний час виконання програми (T_p) за наявності незалежних даних.

Для ефективного використання конвеєра було модифіковано алгоритм планування. Під час лінеаризації AST у граф завдань тепер додатково обчислюється **ранг (або шар) кожного вузла**. Ранг визначається як відстань від поточного вузла до найглибшого листка піддерева. Це дозволило застосувати нову евристику сортування черги `ready_queue`: пріоритет надається завданням із меншим рангом (ближчим до листків), а за рівності рангів — операціям із більшою тривалістю виконання. Такий підхід забезпечує своєчасну підготовку операндів для вищих рівнів дерева виразу.

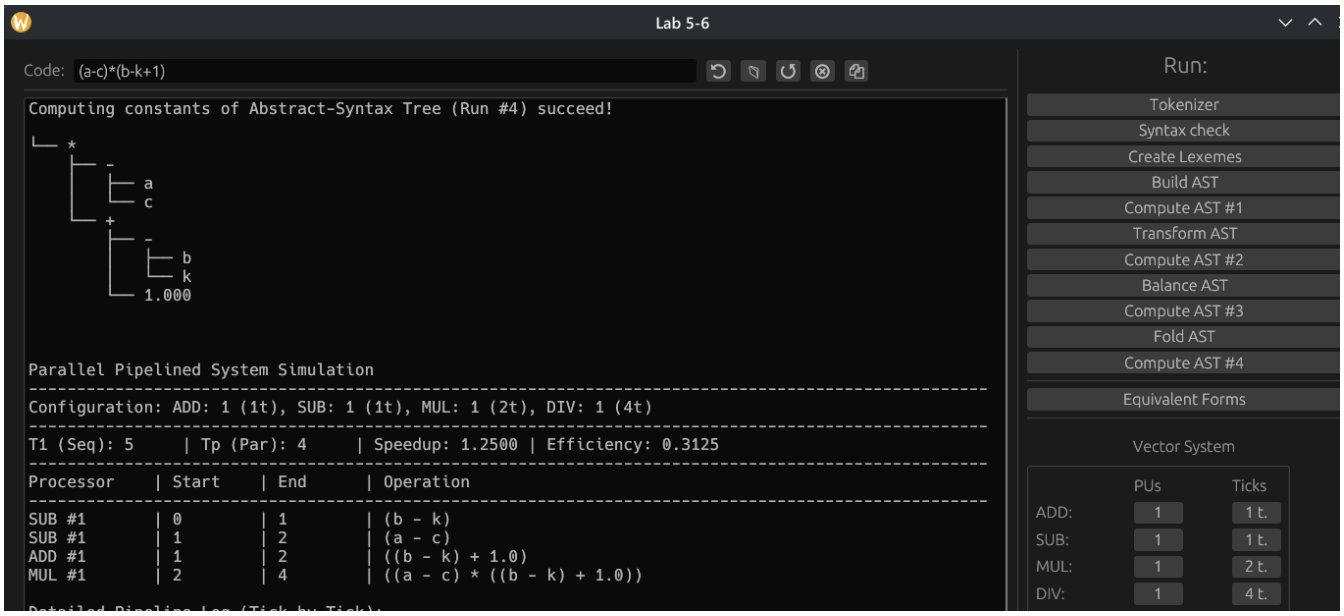
Також було вдосконалено систему моніторингу та звітування. Замість абстрактних назв типів операцій, симулятор тепер формує та відстежує конкретні рядкові представлення підвиразів (наприклад, $(a + b)$ або $T_1 * T_2$). У логах потактової симуляції реалізовано відображення внутрішнього стану конвеєрів:

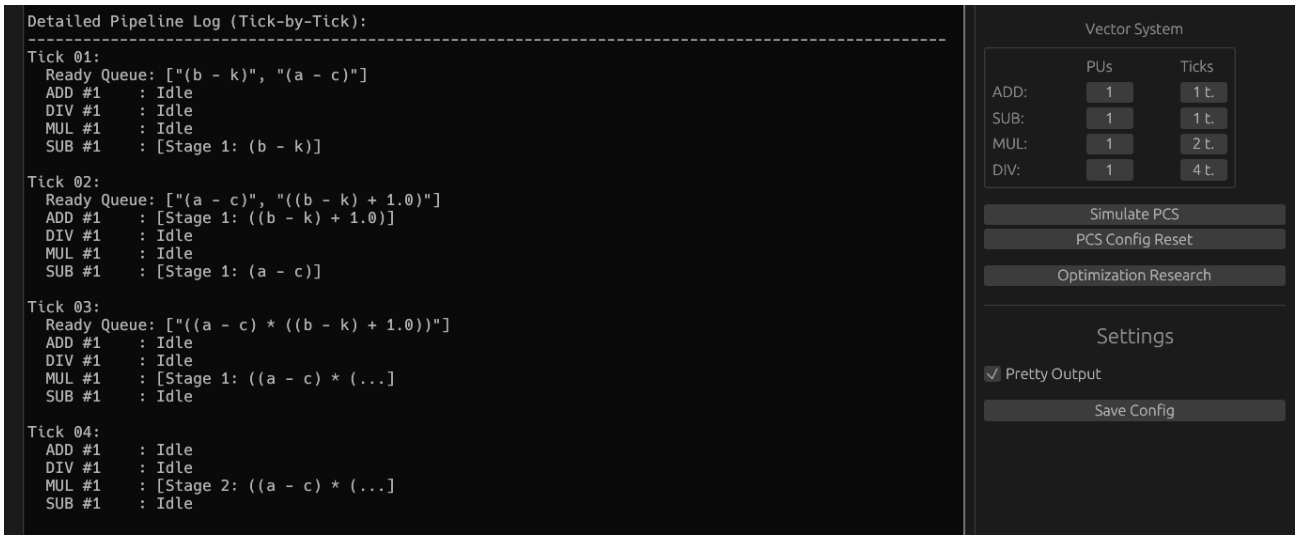
для кожного активного блоку виводиться інформація про те, яка саме операція виконується та на якій стадії (Stage) вона знаходиться (наприклад, `[Stage 2: (a+b)]`). Це забезпечує повну прозорість процесу обчислень та дозволяє візуально верифікувати коректність завантаження конвеєрних блоків.



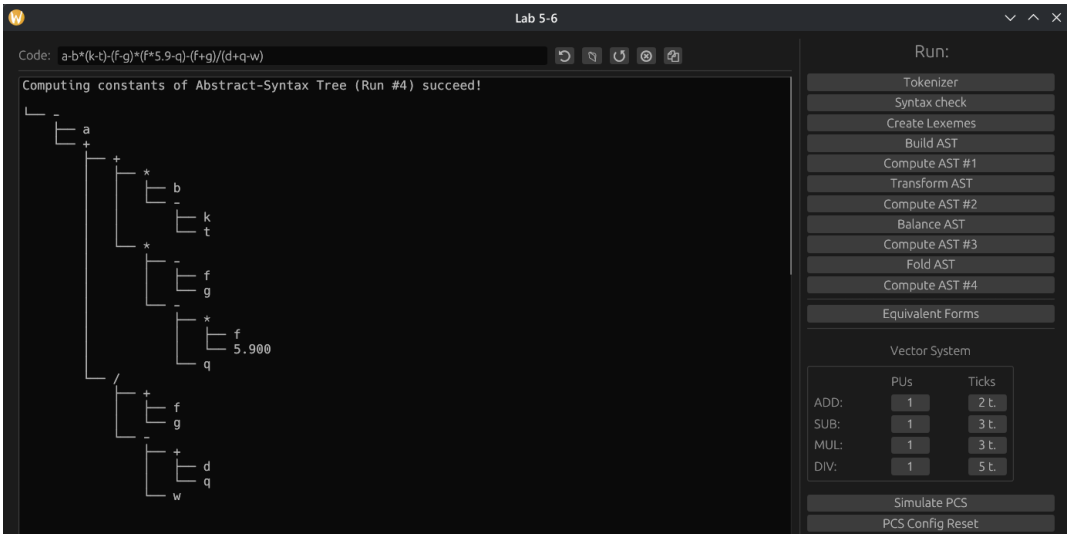
Додаткові приклади виразів.

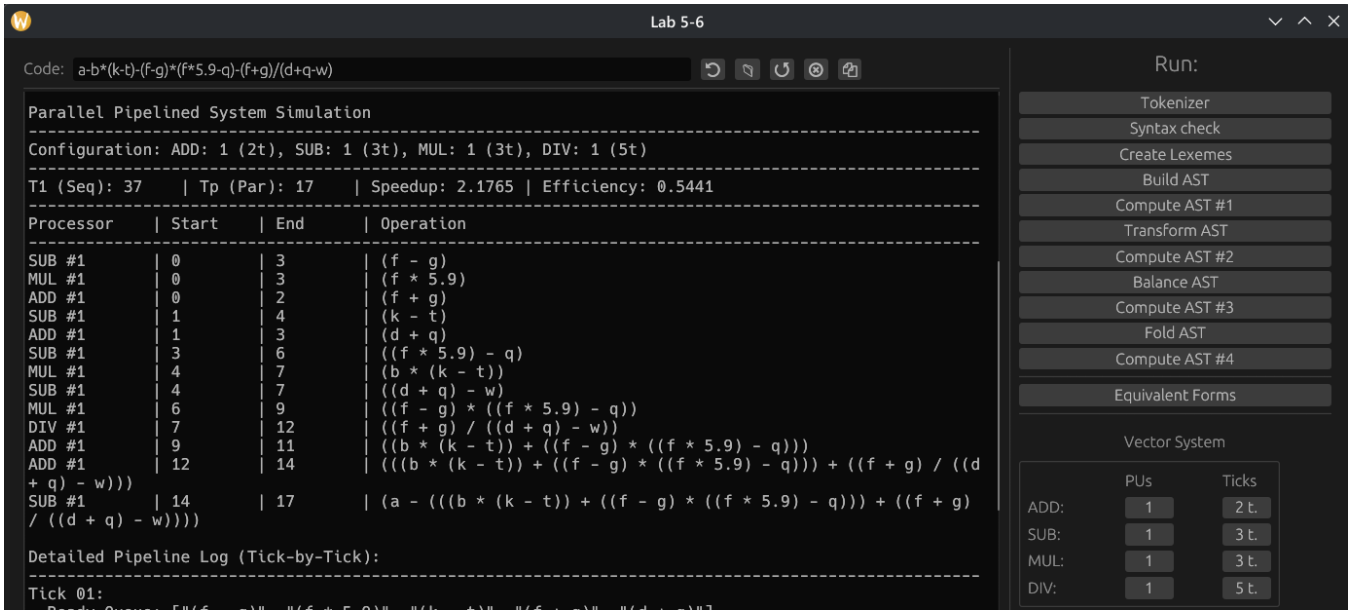
Приклад 1:

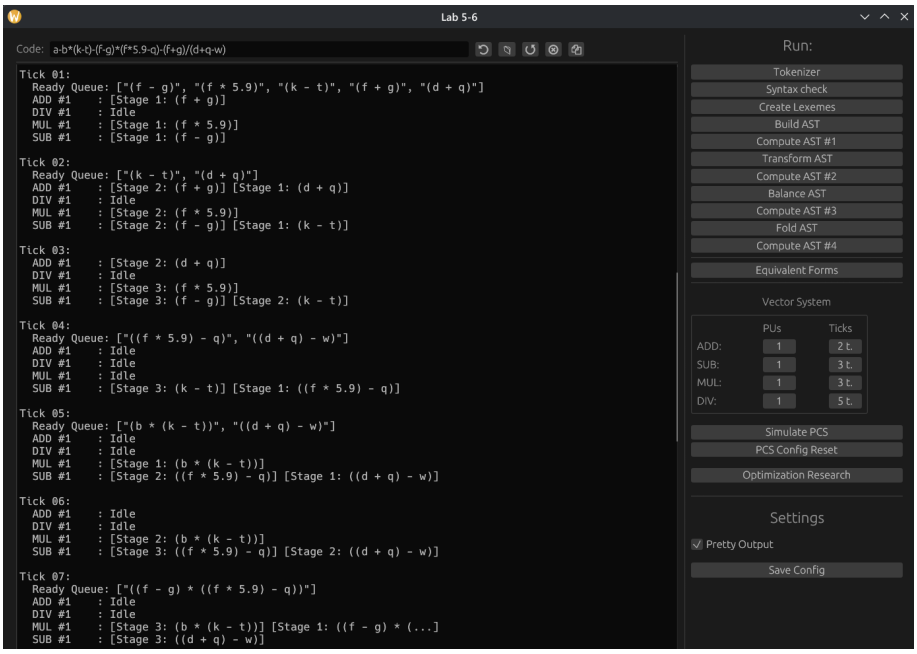


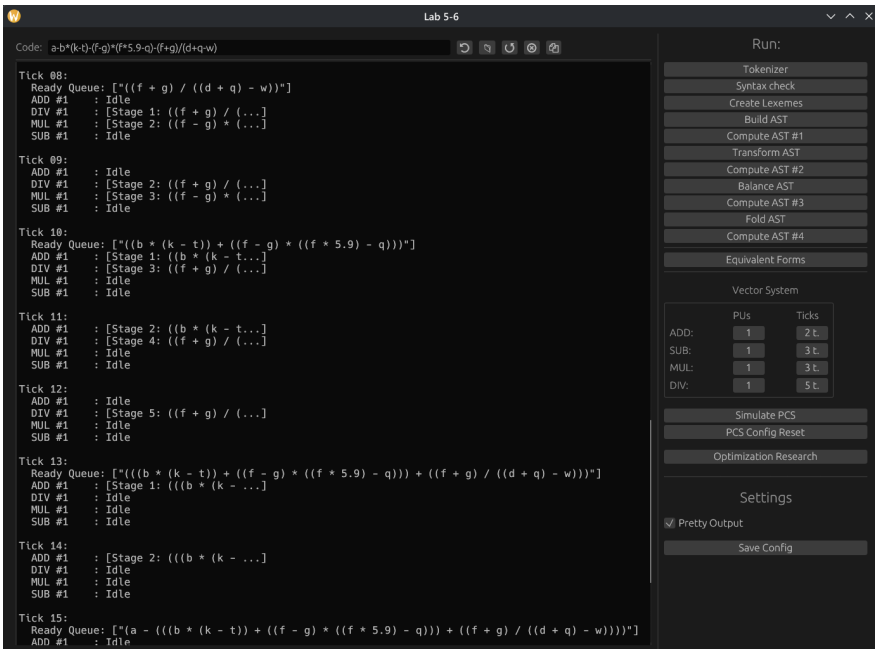


Приклад 2:




Lab 5-6
Code: a-b*(k-t)-(f-g)*(f*5.9-q)-(f+g)/(d+q-w)
Parallel Pipelined System Simulation
Configuration: ADD: 1 (2t), SUB: 1 (3t), MUL: 1 (3t), DIV: 1 (5t)
T1 (Seq): 37 | Tp (Par): 17 | Speedup: 2.1765 | Efficiency: 0.5441
Processor | Start | End | Operation
SUB #1 | 0 | 3 | (f - g)
MUL #1 | 0 | 3 | (f * 5.9)
ADD #1 | 0 | 2 | (f + g)
SUB #1 | 1 | 4 | (k - t)
ADD #1 | 1 | 3 | (d + q)
SUB #1 | 3 | 6 | ((f * 5.9) - q)
MUL #1 | 4 | 7 | (b * (k - t))
SUB #1 | 4 | 7 | ((d + q) - w)
MUL #1 | 6 | 9 | ((f - g) * ((f * 5.9) - q))
DIV #1 | 7 | 12 | ((f + g) / ((d + q) - w))
ADD #1 | 9 | 11 | ((b * (k - t)) + ((f - g) * ((f * 5.9) - q)))
ADD #1 | 12 | 14 | (((b * (k - t)) + ((f - g) * ((f * 5.9) - q))) + ((f + g) / ((d + q) - w)))
SUB #1 | 14 | 17 | (a - (((b * (k - t)) + ((f - g) * ((f * 5.9) - q))) + ((f + g) / ((d + q) - w))))
Detailed Pipeline Log (Tick-by-Tick):
Tick 01:
Run:
Tokenizer
Syntax check
Create Lexemes
Build AST
Compute AST #1
Transform AST
Compute AST #2
Balance AST
Compute AST #3
Fold AST
Compute AST #4
Equivalent Forms
Vector System
PUs
Ticks
ADD: 1 2 t.
SUB: 1 3 t.
MUL: 1 3 t.
DIV: 1 5 t.


Lab 5-6
Code: a-b*(k-t)-(f-g)*(f*5.9-q)-(f+g)/(d+q-w)
Tick 01:
Ready Queue: ["(f - g)", "(f * 5.9)", "(k - t)", "(f + g)", "(d + q)"]
ADD #1 : [Stage 1: (f + g)]
DIV #1 : Idle
MUL #1 : [Stage 1: (f * 5.9)]
SUB #1 : [Stage 1: (f - g)]
Tick 02:
Ready Queue: ["(k - t)", "(d + q)"]
ADD #1 : [Stage 2: (f + g)] [Stage 1: (d + q)]
DIV #1 : Idle
MUL #1 : [Stage 2: (f * 5.9)]
SUB #1 : [Stage 2: (f - g)] [Stage 1: (k - t)]
Tick 03:
ADD #1 : [Stage 2: (d + q)]
DIV #1 : Idle
MUL #1 : [Stage 3: (f * 5.9)]
SUB #1 : [Stage 3: (f - g)] [Stage 2: (k - t)]
Tick 04:
Ready Queue: ["((f * 5.9) - q)", "((d + q) - w)"]
ADD #1 : Idle
DIV #1 : Idle
MUL #1 : Idle
SUB #1 : [Stage 3: (k - t)] [Stage 1: ((f * 5.9) - q)]
Tick 05:
Ready Queue: ["(b * (k - t))", "((d + q) - w)"]
ADD #1 : Idle
DIV #1 : Idle
MUL #1 : [Stage 1: (b * (k - t))]
SUB #1 : [Stage 2: ((f * 5.9) - q)] [Stage 1: ((d + q) - w)]
Tick 06:
ADD #1 : Idle
DIV #1 : Idle
MUL #1 : [Stage 2: (b * (k - t))]
SUB #1 : [Stage 3: ((f * 5.9) - q)] [Stage 2: ((d + q) - w)]
Tick 07:
Ready Queue: ["((f - g) * ((f * 5.9) - q))"]
ADD #1 : Idle
DIV #1 : Idle
MUL #1 : [Stage 3: (b * (k - t))] [Stage 1: ((f - g) * (...]
SUB #1 : [Stage 3: ((d + q) - w)]
Run:
Tokenizer
Syntax check
Create Lexemes
Build AST
Compute AST #1
Transform AST
Compute AST #2
Balance AST
Compute AST #3
Fold AST
Compute AST #4
Equivalent Forms
Vector System
PUs
Ticks
ADD: 1 2 t.
SUB: 1 3 t.
MUL: 1 3 t.
DIV: 1 5 t.
Simulate PCS
PCS Config Reset
Optimization Research
Settings
Pretty Output
Save Config

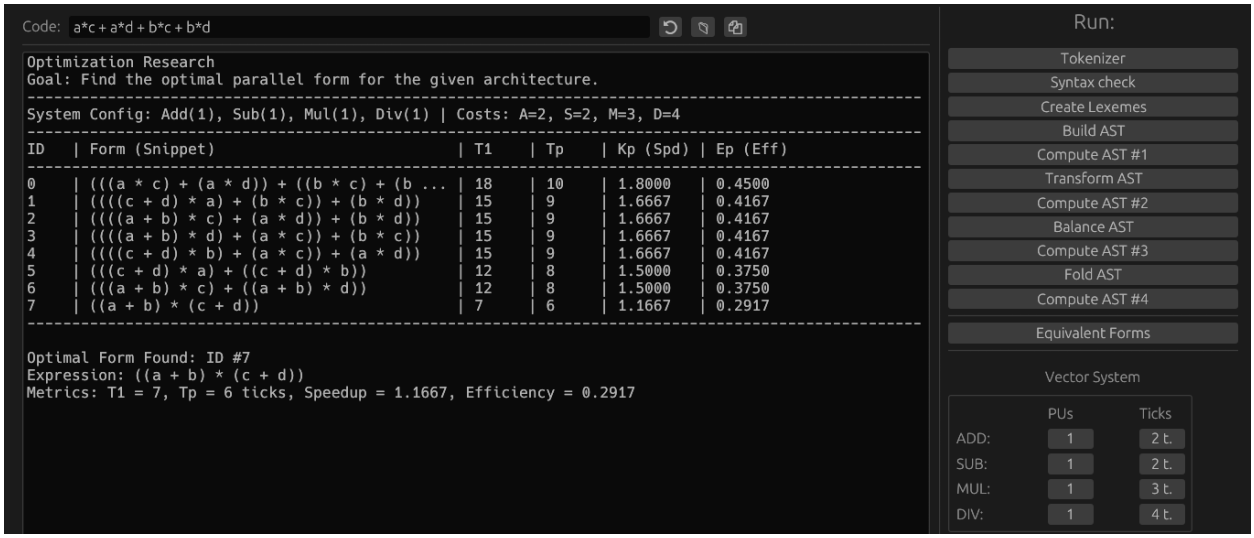

Lab 5-6
Code: a-b*(k-t)-(f-g)*(f*5.9-q)-(f+g)/(d+q-w)
Tick 08:
Ready Queue: ["((f + g) / ((d + q) - w))"]
ADD #1 : Idle
DIV #1 : [Stage 1: ((f + g) / (...]
MUL #1 : [Stage 2: ((f - g) * (...]
SUB #1 : Idle
Tick 09:
ADD #1 : Idle
DIV #1 : [Stage 2: ((f + g) / (...]
MUL #1 : [Stage 3: ((f - g) * (...]
SUB #1 : Idle
Tick 10:
Ready Queue: ["((b * (k - t)) + ((f - g) * ((f * 5.9) - q)))"]
ADD #1 : [Stage 1: ((b * (k - t...]
DIV #1 : [Stage 3: ((f + g) / (...]
MUL #1 : Idle
SUB #1 : Idle
Tick 11:
ADD #1 : [Stage 2: ((b * (k - t...]
DIV #1 : [Stage 4: ((f + g) / (...]
MUL #1 : Idle
SUB #1 : Idle
Tick 12:
ADD #1 : Idle
DIV #1 : [Stage 5: ((f + g) / (...]
MUL #1 : Idle
SUB #1 : Idle
Tick 13:
Ready Queue: ["(((b * (k - t)) + ((f - g) * ((f * 5.9) - q))) + ((f + g) / ((d + q) - w)))"]
ADD #1 : [Stage 1: (((b * (k - ...]
DIV #1 : Idle
MUL #1 : Idle
SUB #1 : Idle
Tick 14:
ADD #1 : [Stage 2: (((b * (k - ...]
DIV #1 : Idle
MUL #1 : Idle
SUB #1 : Idle
Tick 15:
Ready Queue: ["(a - (((b * (k - t)) + ((f - g) * ((f * 5.9) - q))) + ((f + g) / ((d + q) - w))))"]
ADD #1 : Idle
Run:
Tokenizer
Syntax check
Create Lexemes
Build AST
Compute AST #1
Transform AST
Compute AST #2
Balance AST
Compute AST #3
Fold AST
Compute AST #4
Equivalent Forms
Vector System
PUs
Ticks
ADD: 1 2 t.
SUB: 1 3 t.
MUL: 1 3 t.
DIV: 1 5 t.
Simulate PCS
PCS Config Reset
Optimization Research
Settings
Pretty Output
Save Config

Для виконання завдань Лабораторної роботи №6 було реалізовано модуль `research`, який автоматизує пошук оптимальної форми виразу. Цей модуль використовує напрацювання лабораторних робіт №3-4 (генерацію еквівалентних форм) у поєднанні з симулятором з лабораторної №5. Алгоритм дослідження працює наступним чином:

1. Отримує базове оптимізоване дерево виразу.
2. Генерує повну множину алгебраїчно еквівалентних форм, застосовуючи асоціативний та дистрибутивний закони.
3. Для кожної згенерованої форми запускається окремий екземпляр симулятора `VectorSystemSimulator` із заданими параметрами архітектури.
4. Збираються результати всіх симуляцій (T_p , S, E) для кожної форми.

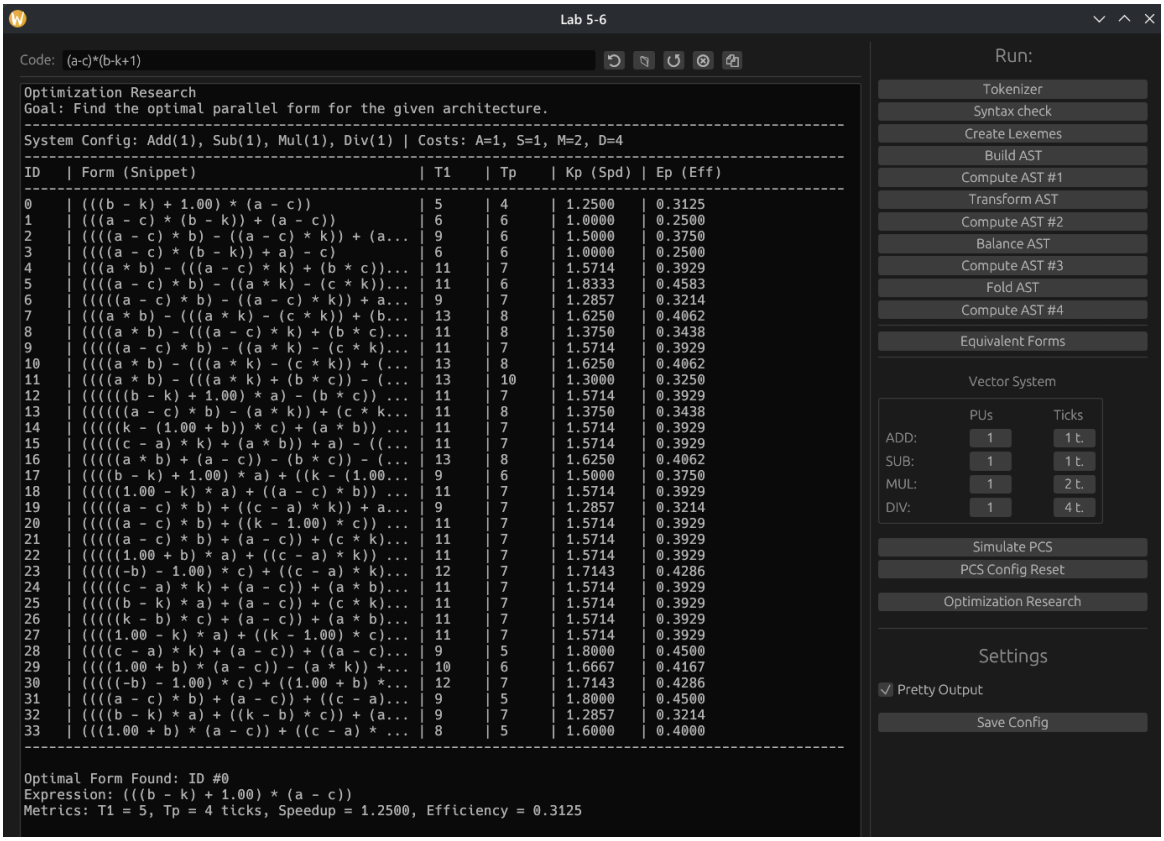
Результати дослідження виводяться у вигляді зведеної таблиці, де для кожної форми зазначено її текстове представлення та розраховані метрики. Система автоматично визначає оптимальну форму за критерієм мінімального часу виконання (T_p). У випадку, коли декілька форм мають однаковий мінімальний час, перевага надається тій, що має меншу сумарну обчислювальну складність (T_1), що дозволяє обрати найбільш енергоефективний варіант серед найшвидших.

Для візуалізації результатів у графічний інтерфейс додано компоненти, що відображають детальну діаграму виконання по тактах, показуючи стан кожного функціонального блоку та черги готових завдань у будь-який момент часу.

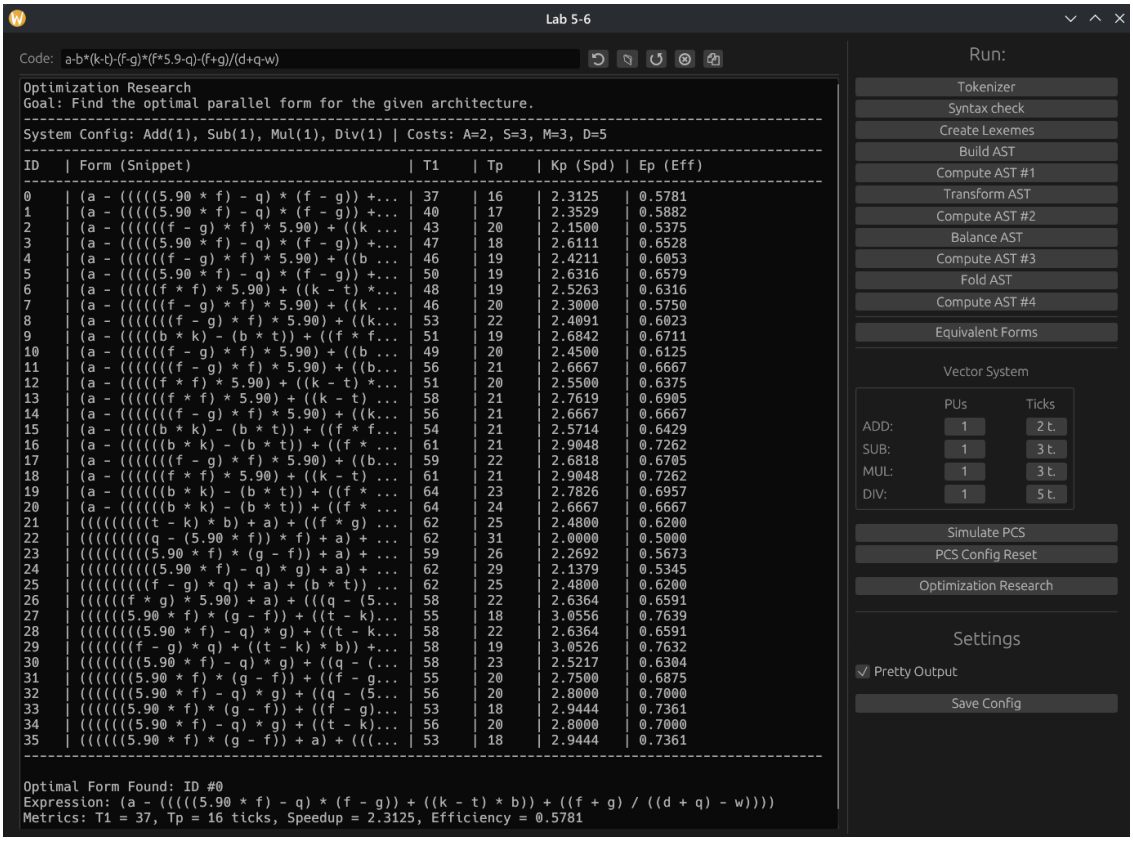


Додаткові приклади виразів.

Приклад 1:



Приклад 2:



Висновок.

У ході виконання лабораторної роботи було успішно спроектовано та програмно реалізовано систему симуляції роботи паралельної комп'ютерної системи (ПКС) векторного типу. Основною метою роботи було дослідження ефективності виконання арифметичних виразів на архітектурі з обмеженою кількістю функціональних блоків (суматорів, помножувачів тощо) та різною тривалістю виконання операцій.

В результаті розробки було створено комплексне програмне забезпечення мовою Rust, яке вирішує задачу відображення високорівневого абстрактного синтаксичного дерева (AST) на апаратні ресурси системи. Ключовим досягненням роботи стала відмова від спрощеної блокуючої моделі виконання на користь **повноцінної конвеєрної (pipelined) архітектури**.

Реалізований алгоритм симуляції враховує дві ключові часові характеристики функціональних блоків: затримка (`latency` – час проходження операції через усі стадії блоку) та пропускну здатність. Це дозволило моделювати ситуацію, коли функціональний блок (наприклад, помножувач із затримкою 2 такти) здатний приймати нові вхідні дані на кожному такті, не чекаючи завершення попередньої операції. Такий підхід забезпечив можливість часового перекриття команд (`instruction overlap`), що суттєво підвищило теоретичну швидкодію системи та коефіцієнт прискорення (S).

Для забезпечення коректного та ефективного планування завдань у конвеєрному режимі було вдосконалено алгоритм List Scheduling. Зокрема, впроваджено систему **ранжування вузлів (layering)**. Кожній підоперації у графі завдань присвоюється ранг, що відповідає її віддаленості від листових вузлів. Це дозволило застосувати евристику планування, яка надає пріоритет операціям нижчих рівнів, забезпечуючи своєчасну підготовку даних для наступних стадій обчислень та мінімізуючи простої конвеєра через залежності по даних.

Важливою частиною роботи стала реалізація модуля автоматизованого дослідження (`Researcher`). Цей модуль генерує множину алгебраїчно еквівалентних форм вхідного виразу (використовуючи асоціативний та дистрибутивний закони) та проводить симуляцію кожної з них на заданій конфігурації ПКС. Порівняльний аналіз отриманих метрик (T_1, T_p, S, E) дозволив на практиці підтвердити, що різні форми запису одного й того ж математичного виразу можуть мати суттєво різний час виконання на паралельній архітектурі. Система автоматично визначає форму, що забезпечує мінімальний час виконання (T_p) та найвищу ефективність використання ресурсів.

Для верифікації коректності роботи планувальника було розроблено детальну систему логування. Вихідні дані симуляції містять потактову діаграму стану кожного функціонального блоку, де відображається не просто факт зайнятості пристрою, а конкретна операція (у символьному вигляді, наприклад, `a+b`) та стадія її виконання всередині конвеєра.

Таким чином, розроблена система дозволяє не лише моделювати роботу векторної ПКС, але й виконувати автоматичну оптимізацію програмного коду під конкретну апаратну архітектуру, що є критично важливим завданням у проектуванні високопродуктивних обчислювальних систем.

Лістинг.

pcs/vector.rs

```rust
use crate::compiler::ast::tree::{
  AbstractSyntaxTree, AstNode, BinaryOperationKind, UnaryOperationKind,
};
use crate::compiler::pcs::{SystemConfiguration, TimeConfiguration};
use crate::compiler::reports::Reporter;
use crate::utils::StringBuffer;
use std::collections::{HashMap, HashSet};

#[derive(Debug, Clone, Copy, PartialEq, Eq, Hash)]
pub enum OperationType {
  Add,
  Sub,
  Mul,
  Div,
  Load, // For variables and numbers (immediate execution)
}

impl OperationType {
  fn execution_time(&self, time_config: &TimeConfiguration) -> usize {
    match self {
      Self::Add => time_config.add,
      Self::Sub => time_config.sub,
      Self::Mul => time_config.mul,
      Self::Div => time_config.div,
      Self::Load => 0,
    }
  }

  fn from_binary(kind: &BinaryOperationKind) -> Self {
    match kind {
      BinaryOperationKind::Plus => Self::Add,
      BinaryOperationKind::Minus => Self::Sub,
      BinaryOperationKind::Multiply => Self::Mul,
      BinaryOperationKind::Divide => Self::Div,
      _ => OperationType::Load, // Logical ops are treated as immediate/loads for this lab
    }
  }
}

impl std::fmt::Display for OperationType {
  fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
    let str = match self {
      Self::Add => "ADD_BLOCK",
      Self::Sub => "SUB_BLOCK",
      Self::Mul => "MUL_BLOCK",
      Self::Div => "DIV_BLOCK",
      Self::Load => "LOAD",
```

```
        };
        write!(f, "{}", str)
    }
}

#[derive(Debug, Clone)]
struct Task {
    id: usize,
    operation_type: OperationType,
    dependencies: Vec<usize>,
    display_name: String,
}

#[derive(Debug, Clone)]
pub struct ScheduledTask {
    pub task_id: usize,
    pub name: String,
    pub start_time: usize,
    pub end_time: usize,
    pub processor: OperationType,
}

#[derive(Debug, Clone)]
pub struct TickLog {
    pub tick: usize,
    // Map: Processor Name -> Task Name (or "Idle")
    pub processor_states: HashMap<String, String>,
    pub ready_queue: Vec<String>,
}

pub struct SimulationResult {
    pub schedule: Vec<ScheduledTask>,
    pub tick_logs: Vec<TickLog>,
    pub t1: usize,
    pub tp: usize,
    pub speedup: f64,
    pub efficiency: f64,

    pub configuration: SystemConfiguration,
}

pub struct VectorSystemSimulator<'a> {
    ast: &'a AbstractSyntaxTree,
    configuration: &'a SystemConfiguration,
}

impl<'a> VectorSystemSimulator<'a> {
    pub fn new(
    ast: &'a AbstractSyntaxTree, configuration: &'a SystemConfiguration,
    ) -> Self {
```

```rust
        Self { ast, configuration }
    }

    pub fn simulate(&self) -> SimulationResult {
        // Convert AST to Task Graph
        let (tasks, _) = Self::flatten_ast(self.ast);

        // Simulate execution
        let (schedule, tick_logs) = self.run_list_scheduling(tasks.clone());

        // Calculate metrics
        // Tp = End time of the last task
        let tp = schedule.iter().map(|t| t.end_time).max().unwrap_or(0);

        // T1 = Sum of execution times of all computational tasks (excluding Loads)
        let t1: usize = schedule.iter().map(|t| t.end_time - t.start_time).sum();

        let total_processors = self.configuration.processors.total();

        // Avoid division by zero
        let speedup = if tp > 0 { t1 as f64 / tp as f64 } else { 0.0 };
        let efficiency = if total_processors > 0 {
            speedup / total_processors as f64
        } else {
            0.0
        };

        SimulationResult {
            schedule,
            tick_logs,
            t1,
            tp,
            speedup,
            efficiency,

            configuration: self.configuration.clone(),
        }
    }

    /// Converts the recursive AST into a flat HashMap of Tasks with dependencies.
    fn flatten_ast(ast: &AbstractSyntaxTree) -> (HashMap<usize, Task>, usize) {
        let mut tasks = HashMap::new();
        let mut id_counter = 0;
        let root_id = Self::traverse_node(&ast.peek, &mut tasks, &mut id_counter);
        (tasks, root_id)
    }

    fn traverse_node(
    node: &AstNode, tasks: &mut HashMap<usize, Task>, counter: &mut usize,
    ) -> usize {
```

```
        let current_id = *counter;
        *counter += 1;

        match node {
          AstNode::BinaryOperation {
            operation,
            left,
            right,
          } => {
            let left_id = Self::traverse_node(left, tasks, counter);
            let right_id = Self::traverse_node(right, tasks, counter);

            tasks.insert(
            current_id,
            Task {
              id: current_id,
              operation_type: OperationType::from_binary(operation),
              dependencies: vec![left_id, right_id],
              display_name: operation.to_string(),
            },
            );
          },
          AstNode::UnaryOperation {
            operation,
            expression,
          } => {
            let child_id = Self::traverse_node(expression, tasks, counter);

            // Map Unary Minus to Subtraction block (0 - expr)
            let operation_type = match operation {
              UnaryOperationKind::Minus => OperationType::Sub,
              _ => OperationType::Load, // Negation '!' treated as instant
            };

            tasks.insert(
            current_id,
            Task {
              id: current_id,
              operation_type,
              dependencies: vec![child_id],
              display_name: format!("Unary{}", operation),
            },
            );
          },
          AstNode::Number(n) => {
            tasks.insert(
            current_id,
            Task {
              id: current_id,
              operation_type: OperationType::Load,
```

```
            dependencies: vec![],
            display_name: format!("{:.1}", n),
          },
          );
        },
        AstNode::Identifier(s) => {
          tasks.insert(
          current_id,
          Task {
            id: current_id,
            operation_type: OperationType::Load,
            dependencies: vec![],
            display_name: s.clone(),
          },
          );
        },
        // Function calls and Array access treated as Load (black box)
        AstNode::FunctionCall { name, .. } => {
          tasks.insert(
          current_id,
          Task {
            id: current_id,
            operation_type: OperationType::Load,
            dependencies: vec![], // Simplifying: arguments handled inside but treated as one unit
  here
            display_name: format!("{}()", name),
          },
          );
        },
        AstNode::ArrayAccess { identifier, .. } => {
          tasks.insert(
          current_id,
          Task {
            id: current_id,
            operation_type: OperationType::Load,
            dependencies: vec![],
            display_name: format!("{}[..]", identifier),
          },
          );
        },
        _ => {
          tasks.insert(
          current_id,
          Task {
            id: current_id,
            operation_type: OperationType::Load,
            dependencies: vec![],
            display_name: "?".to_string(),
          },
          );
```

```
        },
      }
      current_id
    }

    /// List Scheduling Algorithm
    fn run_list_scheduling(
    &self, tasks: HashMap<usize, Task>,
    ) -> (Vec<ScheduledTask>, Vec<TickLog>) {
      let mut final_schedule = Vec::new();
      let mut tick_logs = Vec::new();

      // Initialize Processor states: Map<OpType, Vec<BusyUntilTick>>
      // The Vec represents the pool of processors of that type.
      let mut processors: HashMap<OperationType, Vec<usize>> = HashMap::new();
      processors.insert(
      OperationType::Add,
      vec![0; self.configuration.processors.add],
      );
      processors.insert(
      OperationType::Sub,
      vec![0; self.configuration.processors.sub],
      );
      processors.insert(
      OperationType::Mul,
      vec![0; self.configuration.processors.mul],
      );
      processors.insert(
      OperationType::Div,
      vec![0; self.configuration.processors.div],
      );

      // "Load" operations don't need processors, they are instant.

      // Task states
      let mut task_finish_time: HashMap<usize, usize> = HashMap::new();
      let mut completed_tasks: HashSet<usize> = HashSet::new();

      // Pre-process "Load" tasks (variables/constants) as completed at T=0
      for task in tasks.values() {
        if task.operation_type == OperationType::Load {
          task_finish_time.insert(task.id, 0);
          completed_tasks.insert(task.id);
        }
      }

      let mut current_tick = 0;
      let mut active_tasks: HashMap<usize, (usize, OperationType)> = HashMap::new(); // TaskId -> (
  EndTime, ProcType)

```

```
        // Main Loop
        loop {
          // A. Check for completed tasks in this tick
          let mut just_finished = Vec::new();
          // We need to collect keys to remove to avoid borrowing issues
          let finished_ids: Vec<usize> = active_tasks
          .iter()
          .filter(|(_, (end_time, _))| *end_time <= current_tick)
          .map(|(id, _)| *id)
          .collect();

          for id in finished_ids {
            active_tasks.remove(&id);
            completed_tasks.insert(id);
            just_finished.push(id);
          }

          // If all tasks are done, break
          if completed_tasks.len() == tasks.len() {
            break;
          }

          // B. Find Ready Tasks
          let mut ready_queue: Vec<&Task> = tasks.values()
          .filter(|t| !completed_tasks.contains(&t.id)) // Not completed
          .filter(|t| !active_tasks.contains_key(&t.id)) // Not currently running
          .filter(|t| {
            // All dependencies must be completed
            t.dependencies.iter().all(|dep| completed_tasks.contains(dep))
          })
          .collect();

          // Heuristic: Sort by operation type cost (Longest Processing Time first) or just ID
          ready_queue.sort_by(|a, b| {
            let time_a = a.operation_type.execution_time(&self.configuration.time);
            let time_b = b.operation_type.execution_time(&self.configuration.time);
            if time_a != time_b {
              time_b.cmp(&time_a) // Higher cost first
            } else {
              a.id.cmp(&b.id)
            }
          });

          // Log snapshot preparation
          let mut current_tick_log = TickLog {
            tick: current_tick,
            processor_states: HashMap::new(),
            ready_queue: ready_queue.iter().map(|t| t.display_name.clone()).collect(),
          };

```

```
        // C. Assign Ready Tasks to Free Processors
        for task in ready_queue {
          if let Some(proc_pool) = processors.get_mut(&task.operation_type) {
            // Find a processor that is free at current_tick
            if let Some(proc_idx) = proc_pool
            .iter()
            .position(|&busy_until| busy_until <= current_tick)
            {
              // Schedule!
              let duration =
              task.operation_type.execution_time(&self.configuration.time);
              let start = current_tick;
              let end = start + duration;

              // Mark processor busy
              proc_pool[proc_idx] = end;

              // Add to schedule
              final_schedule.push(ScheduledTask {
                task_id: task.id,
                name: task.display_name.clone(),
                start_time: start,
                end_time: end,
                processor: task.operation_type,
              });

              // Add to active tasks
              active_tasks.insert(task.id, (end, task.operation_type));

              // We also record finish time for dependency checking later
              task_finish_time.insert(task.id, end);
            }
          }
        }

        // D. Populate Tick Log with Processor Status
        // Helper to find what task is running on a specific processor type
        // Note: This simple model assumes 1 processor of each type.
        // If COUNT > 1, we would need to track which specific processor index is used.
        for operation_type in processors.keys() {
          if operation_type.eq(&OperationType::Load) {
            continue;
          }

          let op_name = operation_type.to_string();
          let mut status = "Idle".to_string();

          // Find if any active task matches this op_type
          // Since we have 1 proc per type, we just check if there is ANY active task of this type
          if let Some((id, _)) =
```

```
            active_tasks.iter().find(|(_, (_, t))| t == operation_type)
            && let Some(task) = tasks.get(id)
            {
              status = format!("Processing '{}'", task.display_name);
            }

            current_tick_log.processor_states.insert(op_name, status);
          }
          tick_logs.push(current_tick_log);

          // E. Advance Time
          current_tick += 1;

          // Safety break
          if current_tick > 10000 {
            break;
          }
        }

        (final_schedule, tick_logs)
      }
    }

    impl Reporter {
      pub fn pcs_simulation(&self, result: &SimulationResult) -> String {
        let mut buffer = StringBuffer::default();

        buffer.add_line("Parallel Computer System Simulation Results:".to_string());
        buffer.add_line("-".repeat(60));
        buffer.add_line(format!(
        "Configuration: Add({}), Sub({}), Mul({}), Div({})",
        result.configuration.processors.add,
        result.configuration.processors.sub,
        result.configuration.processors.mul,
        result.configuration.processors.div,
        ));
        buffer.add_line(format!(
        "Costs (Ticks): Add={}, Sub={}, Mul={}, Div={}",
        result.configuration.time.add,
        result.configuration.time.sub,
        result.configuration.time.mul,
        result.configuration.time.div,
        ));
        buffer.add_line("-".repeat(60));

        // Metrics
        buffer.add_line(format!("Sequential Time (T1): {}", result.t1));
        buffer.add_line(format!("Parallel Time (Tp):   {}", result.tp));
        buffer.add_line(format!("Speedup (Ky):         {:.4}", result.speedup));
        buffer.add_line(format!("Efficiency (E):       {:.4}", result.efficiency));
```

```
        buffer.add_line("-".repeat(60));

        // Schedule Table
        buffer.add_line(format!(
        "{:<12} | {:<10} | {:<10} | {:<15}",
        "Processor", "Start", "End", "Operation"
        ));
        buffer.add_line("-".repeat(60));

        // Sort by start time for better readability
        let mut sorted_schedule = result.schedule.clone();
        sorted_schedule.sort_by_key(|t| t.start_time);

        for task in sorted_schedule {
          buffer.add_line(format!(
          "{:<12} | {:<10} | {:<10} | {:<15}",
          task.processor.to_string(),
          task.start_time,
          task.end_time,
          task.name
          ));
        }

        buffer.add_line("\n".to_string());
        buffer.add_line("Tick-by-Tick Execution Log:".to_string());
        buffer.add_line("-".repeat(60));

        for log in &result.tick_logs {
          buffer.add_line(format!("Tick {:02}:", log.tick));

          // Show Queue
          if log.ready_queue.is_empty() {
            buffer.add_line("  Queue: [Empty]".to_string());
          } else {
            buffer.add_line(format!("  Queue: {:?}", log.ready_queue));
          }

          // Show Processors
          // Sort keys for consistent output
          let mut keys: Vec<&String> = log.processor_states.keys().collect();
          keys.sort();

          for proc in keys {
            let state = match log.processor_states.get(proc) {
              Some(state) => state,
              None => unreachable!("Non-existent processor in log"),
            };
            buffer.add_line(format!("  {:<10}: {}", proc, state));
          }
          buffer.add_line("".to_string());
```

```
        }

        buffer.get()
      }
    }

```

pcs/research.rs

```
use crate::compiler::ast::tree::AbstractSyntaxTree;
use crate::compiler::pcs::SystemConfiguration;
use crate::compiler::pcs::vector::{SimulationResult, VectorSystemSimulator};
use crate::compiler::reports::Reporter;
use crate::utils::StringBuffer;

pub struct Researcher<'a> {
  forms: &'a Vec<AbstractSyntaxTree>,
  configuration: &'a SystemConfiguration,
}

pub struct OptimizationReport {
  pub index: usize,
  pub canonical_string: String,
  pub result: SimulationResult,
}

impl<'a> Researcher<'a> {
  pub fn new(
  equivalent_forms: &'a Vec<AbstractSyntaxTree>,
  system_configuration: &'a SystemConfiguration,
  ) -> Self {
    Self {
      forms: equivalent_forms,
      configuration: system_configuration,
    }
  }

  pub fn run(&self) -> Result<Vec<OptimizationReport>, String> {
    let mut results = Vec::new();

    for (index, form) in self.forms.iter().enumerate() {
      let simulator = VectorSystemSimulator::new(form, self.configuration);
      let result = simulator.simulate();
      results.push(OptimizationReport {
        index,
        canonical_string: form.to_canonical_string(),
        result,
      });
    }

    Ok(results)
```

```
    }
  }

  impl Reporter {
    pub fn generate_optimization_report(reports: &[OptimizationReport]) -> String {
      let mut buffer = StringBuffer::default();

      buffer.add_line("Optimization Research".to_string());
      buffer.add_line(
      "Goal: Find the optimal parallel form for the given architecture."
      .to_string(),
      );
      buffer.add_line("-".repeat(100));

      // First line contains system configuration
      match reports.first() {
        Some(first_report) => {
          let configuration = &first_report.result.configuration;
          let processors = &configuration.processors;
          let time = &configuration.time;

          buffer.add_line(format!(
          "System Config: Add({}), Sub({}), Mul({}), Div({}) | Costs: A={}, S={}, M={}, D={}",
          processors.add,
          processors.sub,
          processors.mul,
          processors.div,
          time.add,
          time.sub,
          time.mul,
          time.div,
          ));

          buffer.add_line("-".repeat(100));
        },
        None => {
          buffer.add_line("No optimization results available.".to_string());
          return buffer.get();
        },
      }

      // Table header
      buffer.add_line(format!(
      "{:<4} | {:<40} | {:<5} | {:<5} | {:<8} | {:<8}",
      "ID", "Form (Snippet)", "T1", "Tp", "Kp (Spd)", "Ep (Eff)"
      ));
      buffer.add_line("-".repeat(100));

      let mut best_tp = usize::MAX;
      let mut best_efficiency = f64::MAX;
```

```
        let mut best_index = 0;

        // Table rows
        for report in reports {
          let result = &report.result;
          let form_str = &report.canonical_string;

          // Shortening variable names for clarity
          let short_form = if form_str.len() > 37 {
            format!("{}...", &form_str[0..37])
          } else {
            form_str.clone()
          };
          buffer.add_line(format!(
          "{:<4} | {:<40} | {:<5} | {:<5} | {:<8.4} | {:<8.4}",
          report.index,
          short_form,
          result.t1,
          result.tp,
          result.speedup,
          result.efficiency
          ));

          // Searching for optimal form
          if result.tp < best_tp {
            best_tp = result.tp;
            best_index = report.index;
          } else if result.tp == best_tp && result.efficiency > best_efficiency {
            best_tp = result.tp;
            best_index = report.index;
            best_efficiency = result.efficiency;
          }
        }

        buffer.add_line("-".repeat(100));

        // Conclusion with the best form
        let best_report = &reports[best_index];
        let best_result = &best_report.result;
        buffer.add_line(format!("\nOptimal Form Found: ID #{}", best_index));
        buffer.add_line(format!("Expression: {}", best_report.canonical_string));
        buffer.add_line(format!(
        "Metrics: Tp = {} ticks, Speedup = {:.4}, Efficiency = {:.4}",
        best_result.tp, best_result.speedup, best_result.efficiency
        ));

        buffer.get()
      }
    }

```