



УНИВЕРСИТЕТ  
ЛОБАЧЕВСКОГО



# Конвейерное исполнение команд

Сергей Шишлов  
3 февраля 2016 г.

# План лекции

- Опишем простую систему команд и алгоритмы исполнения каждой команды
- Построим однотактную неконвейерную схему процессора для данной системы команд
- Конвейеризуем данную схему
- Введем блокировки и байпасы для разрешения конфликтов по данным и по управлению
- Оценим производительность

# Упрощенная система команд

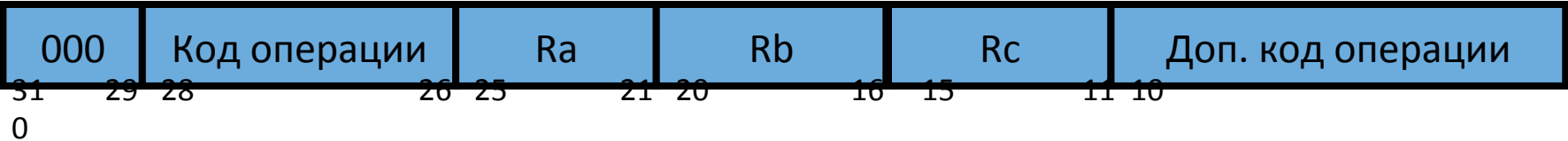
Регистровый контекст:

*PC* – счетчик команд

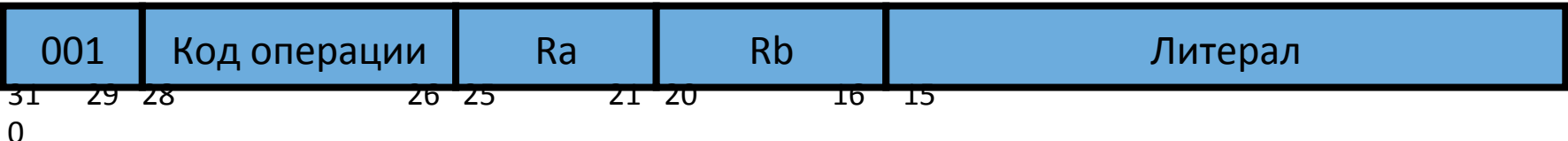
*R0* = 0

*R1..R31* – регистры общего назначения

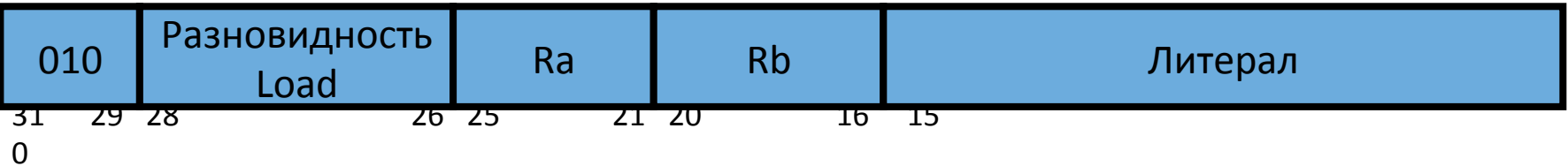
Арифметика  
(регистровые операнды):  
Op Ra <= Rb, Rc  
Op = ADD, SUB, AND, OR,  
XOR, SHR, SHL, CMP...



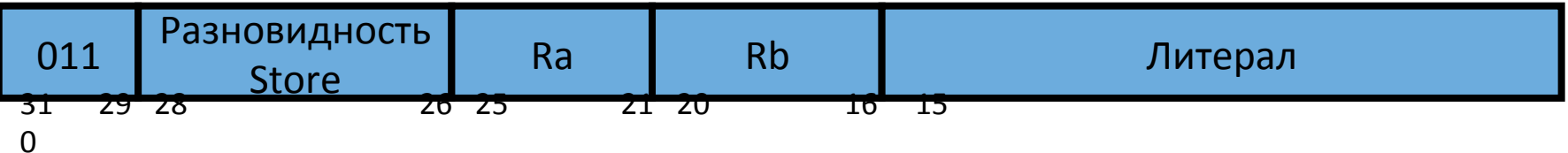
Арифметика  
(непосредственный операнд):  
Op Ra <= Rb, Imm



Чтение из памяти:  
LD Ra <= [Rb + Imm]



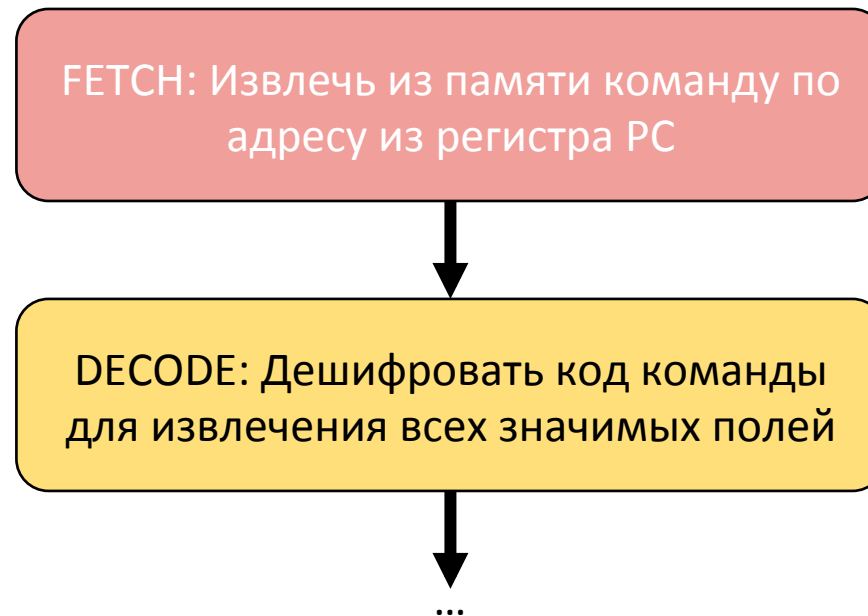
Запись в память:  
ST Ra => [Rb + Imm]



Условный переход:  
Jcc Ra, [PC + Imm] => PC  
cc = Equal, Zero, More, Less

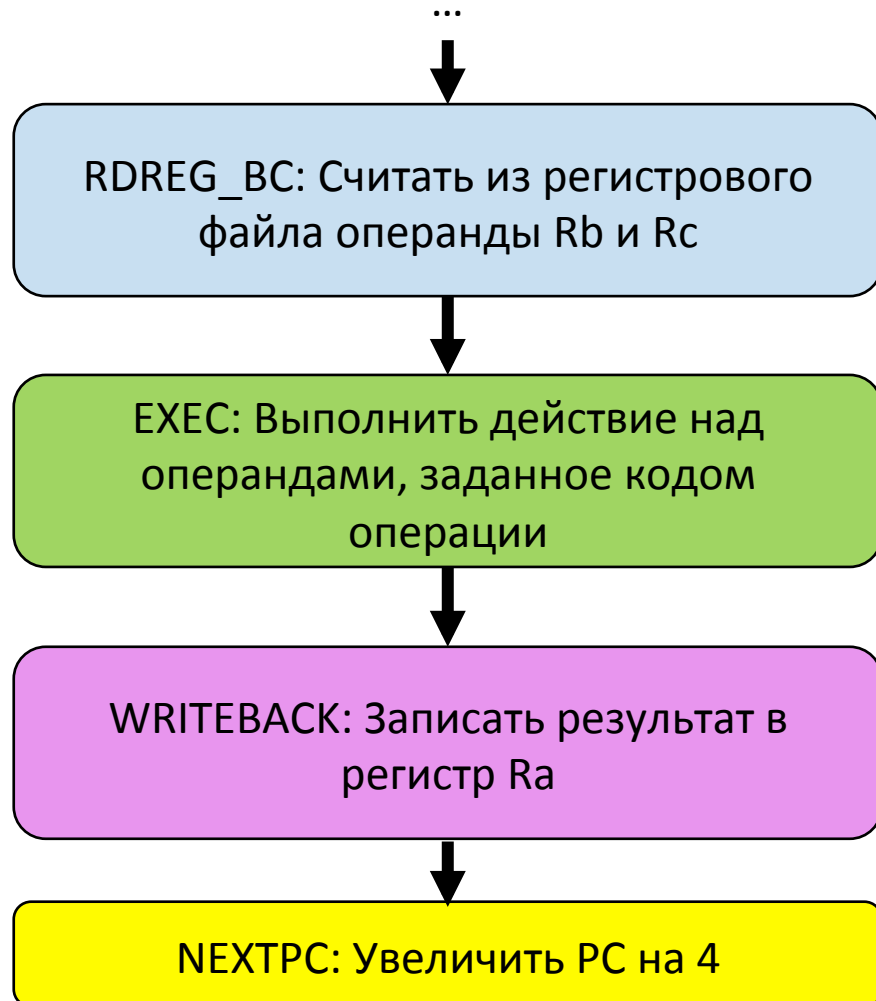


# Алгоритм исполнения команды (общая часть для любых команд)

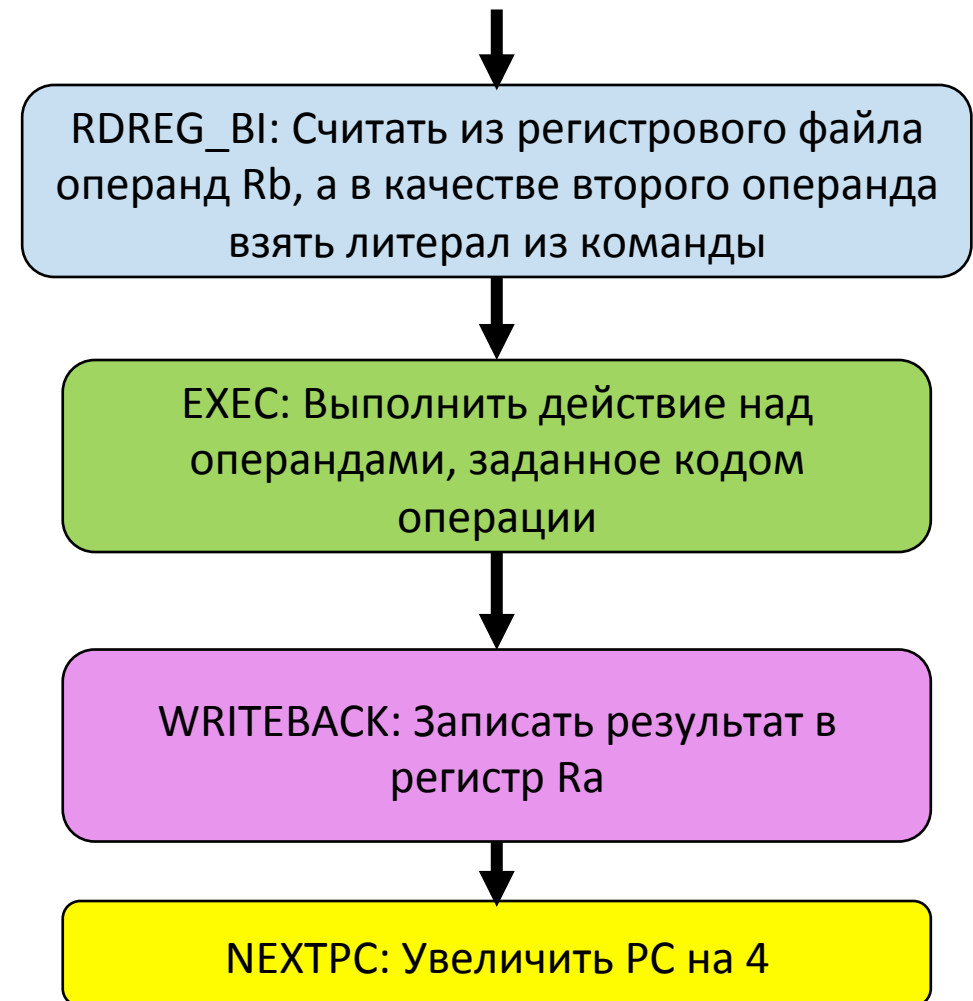


# Алгоритм исполнения арифметической команды

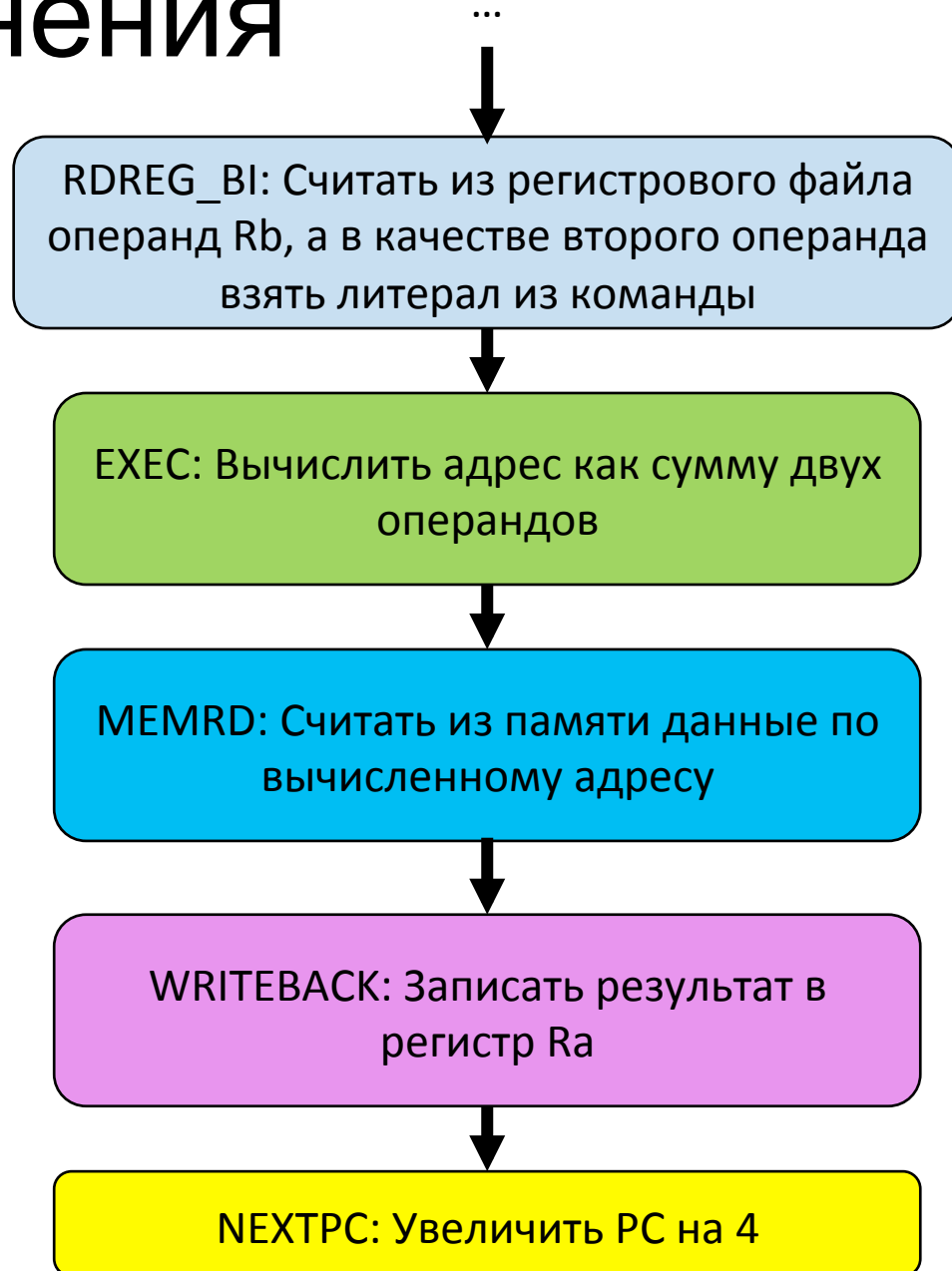
С регистровыми операндами:



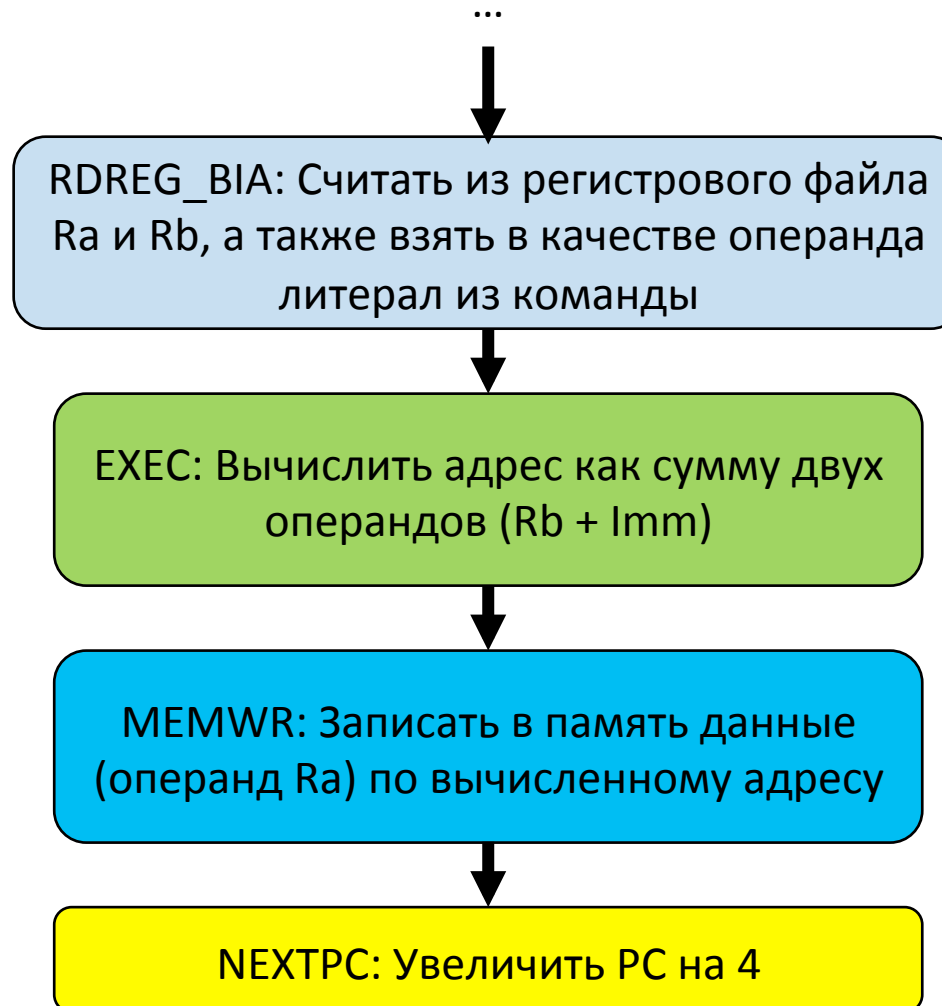
С непосредственным операндом:



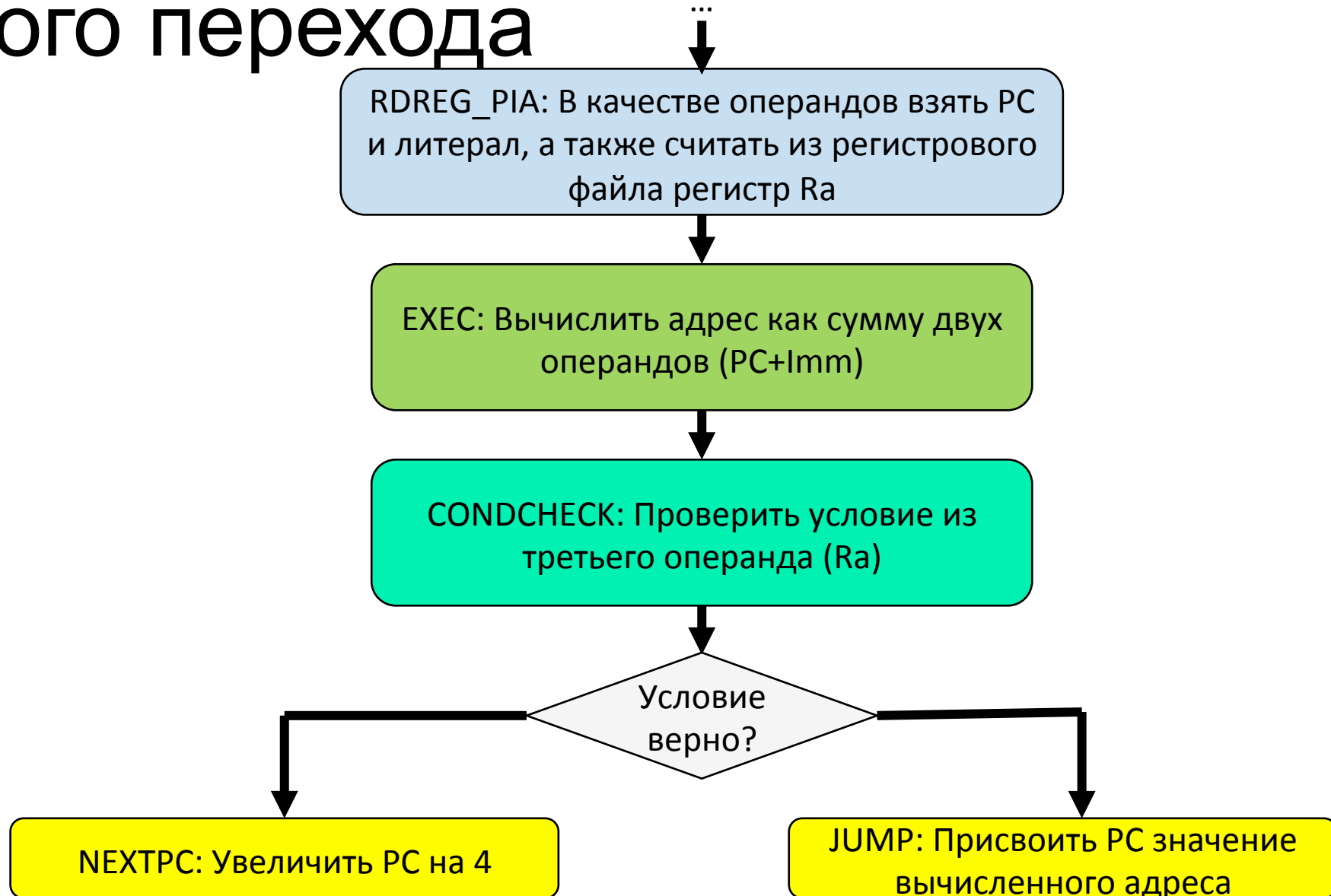
# Алгоритм исполнения команды Load



# Алгоритм исполнения команды Store

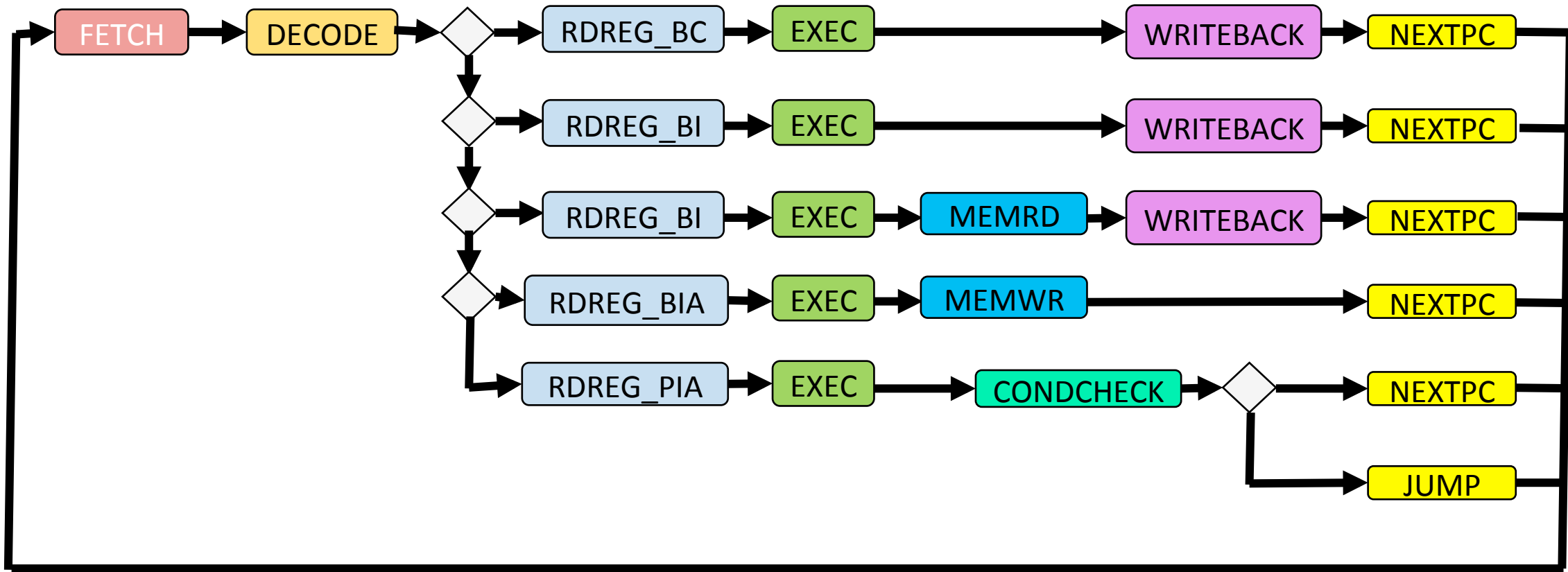


# Алгоритм исполнения команды условного перехода

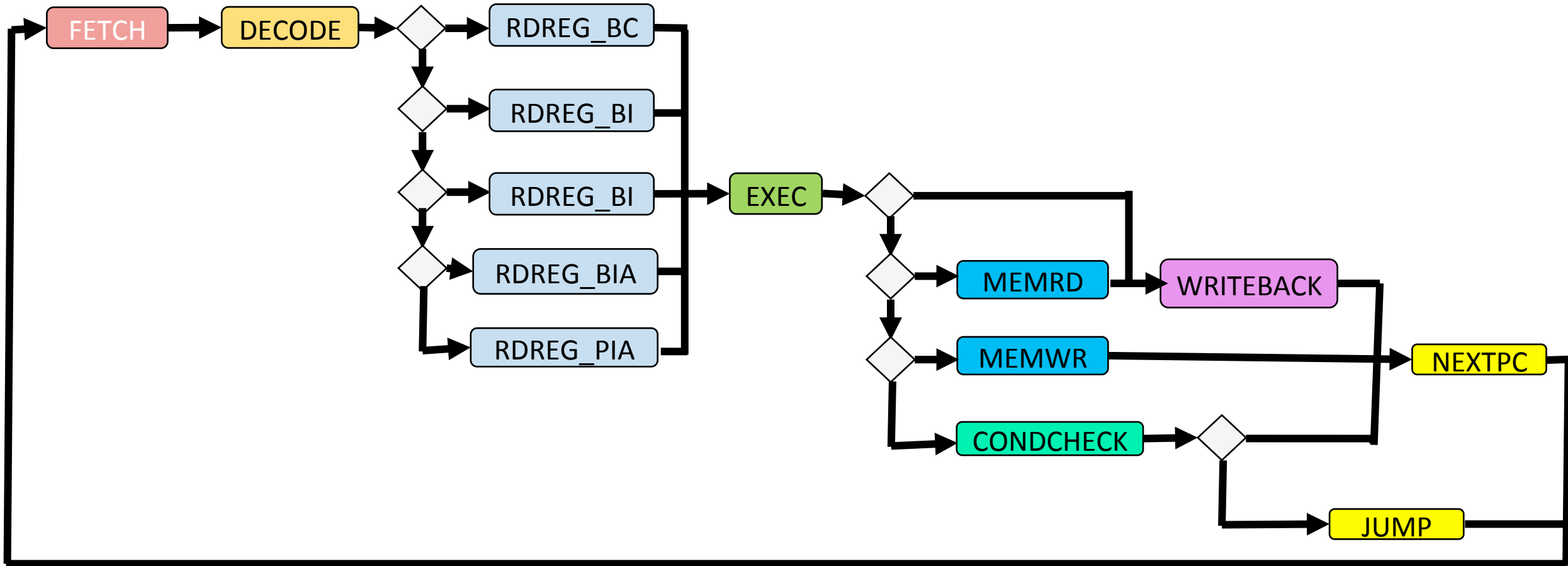




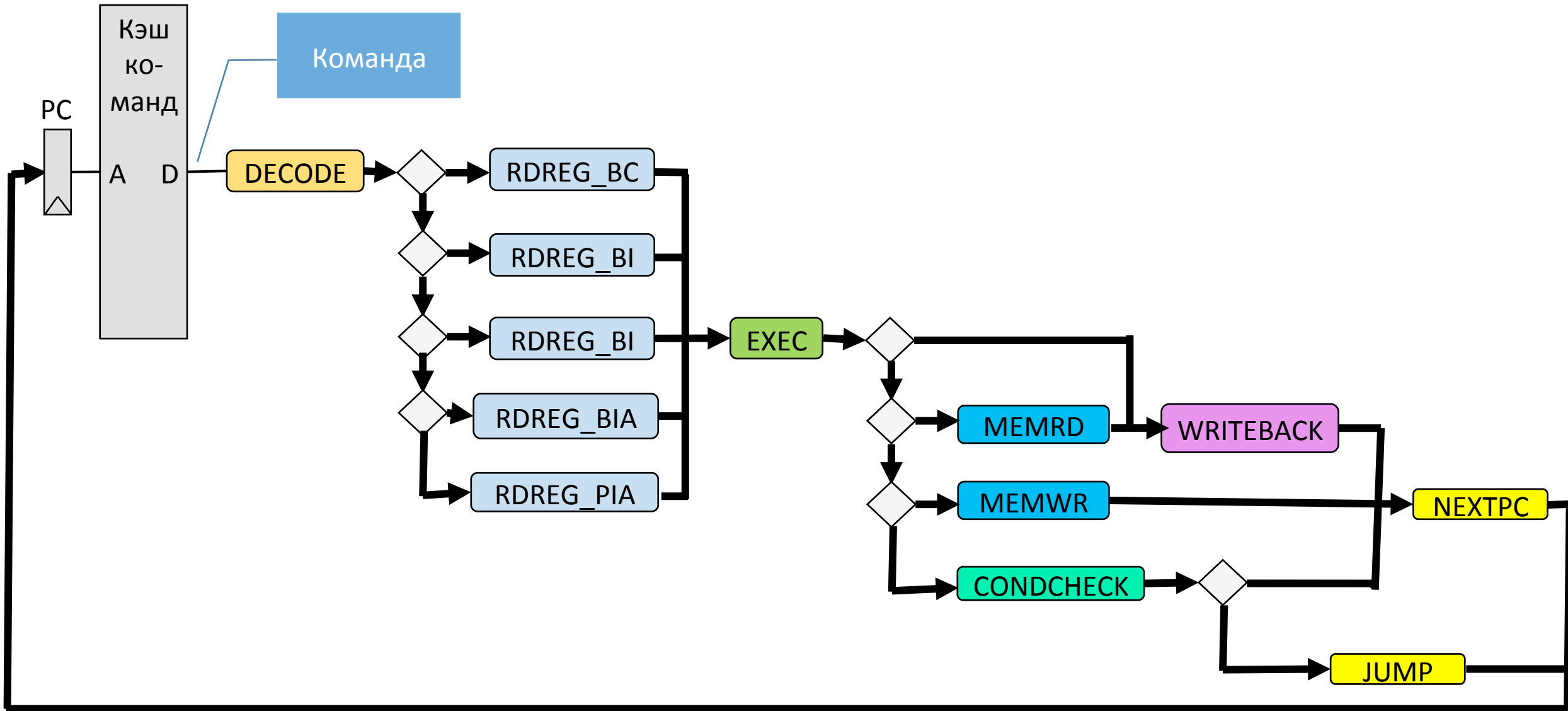
# Объединенный алгоритм исполнения команды



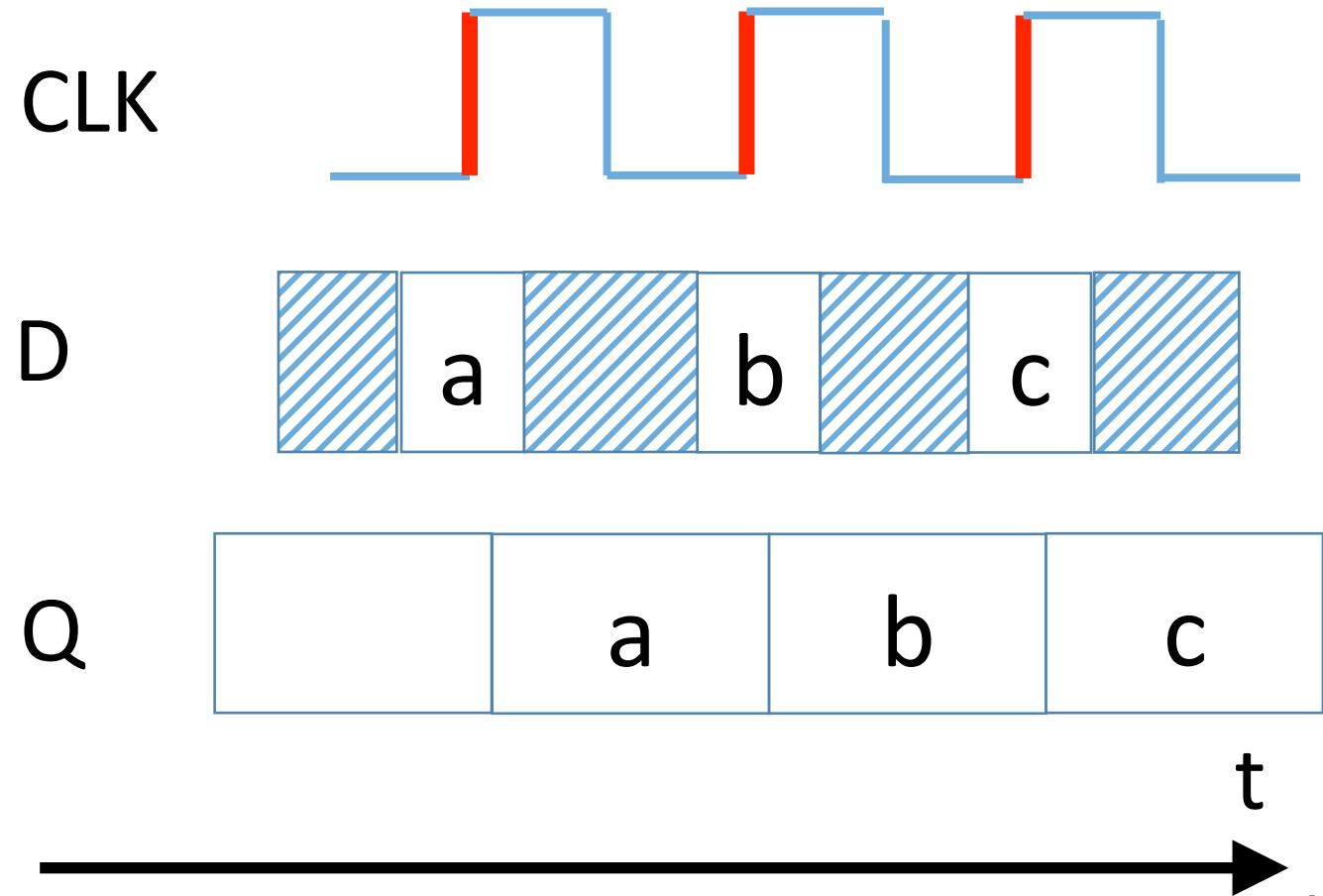
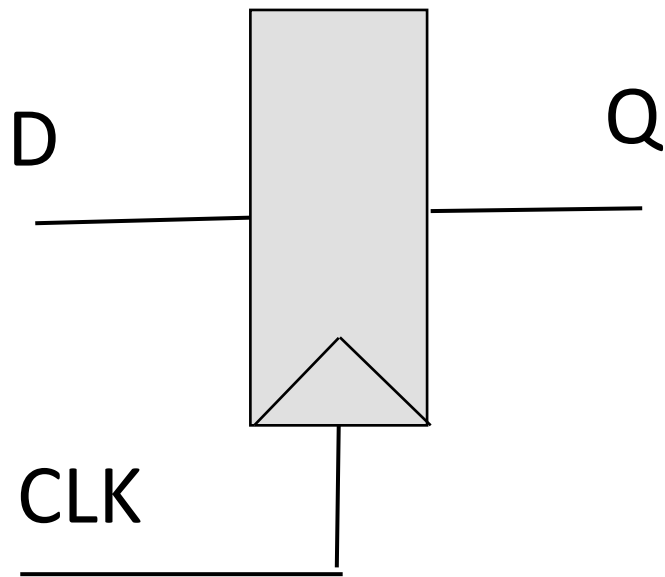
# Объединенный алгоритм исполнения команды



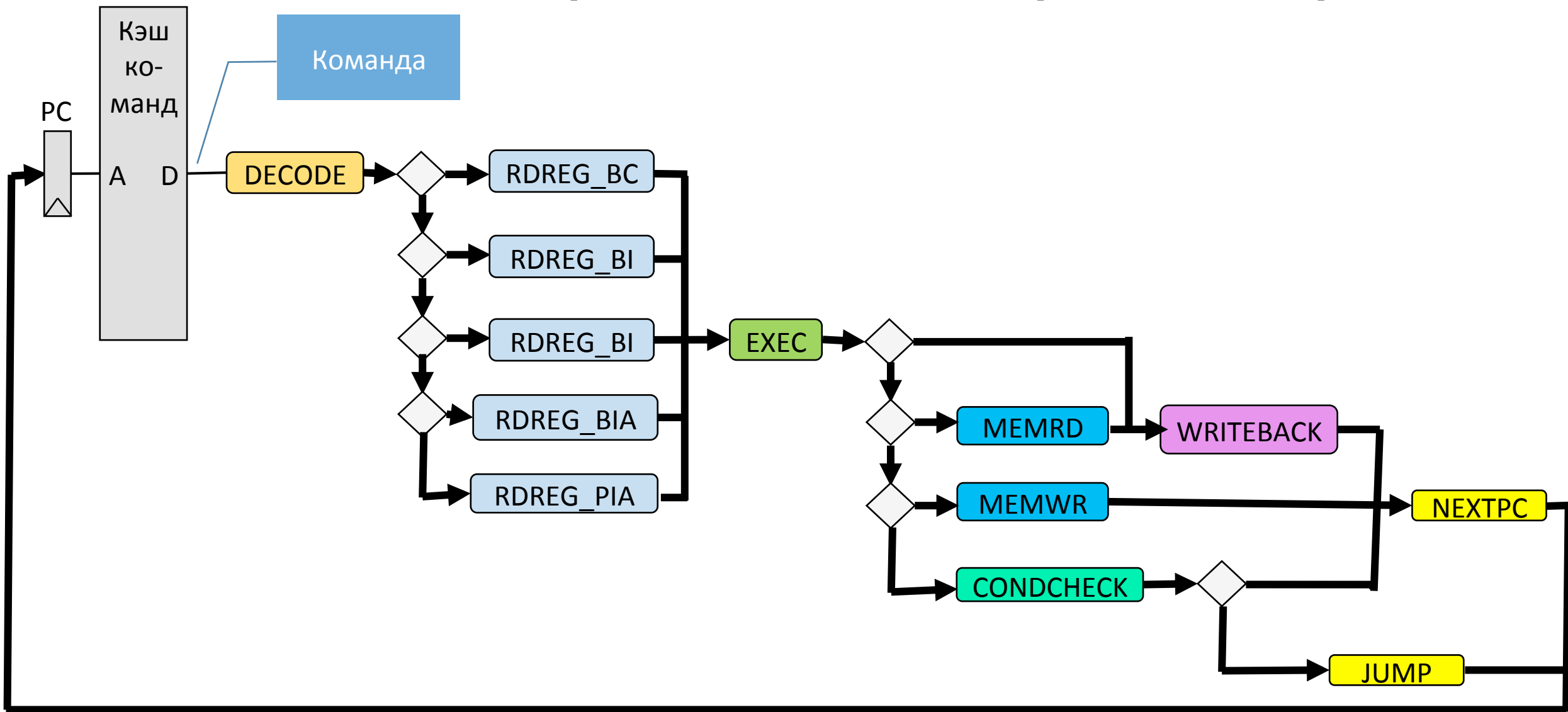
# Однотактная реализация процессора



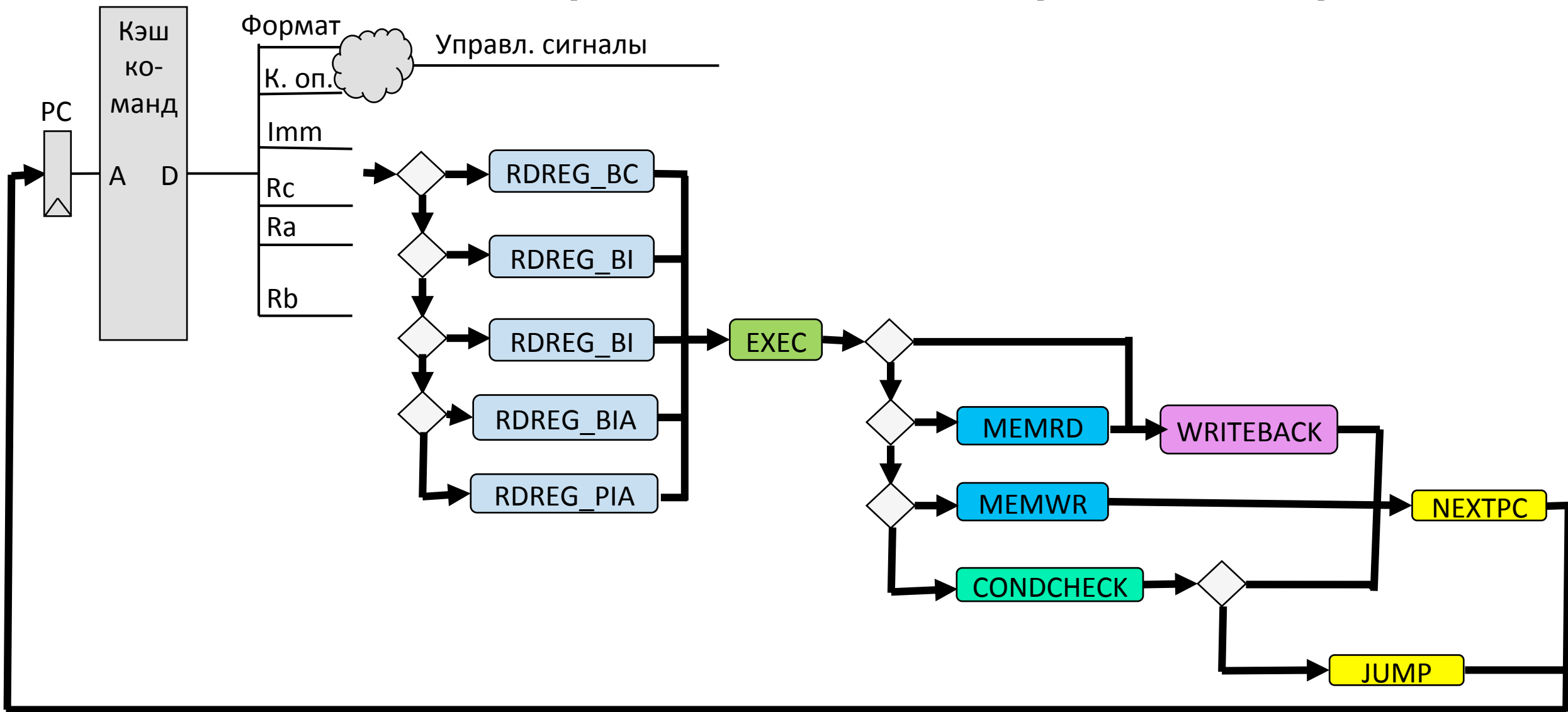
# Регистр на D-триггерах (flip-flop)



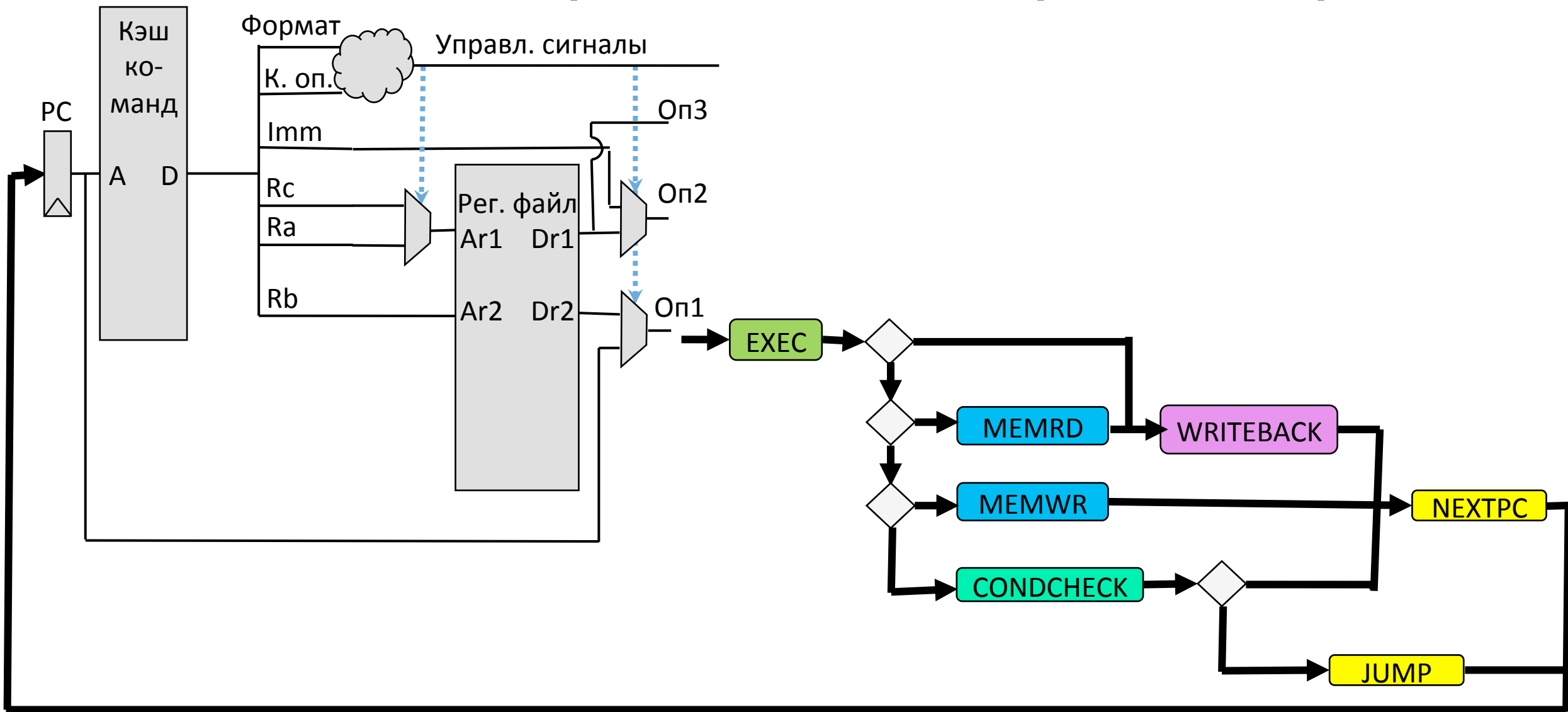
# Однотактная реализация процессора



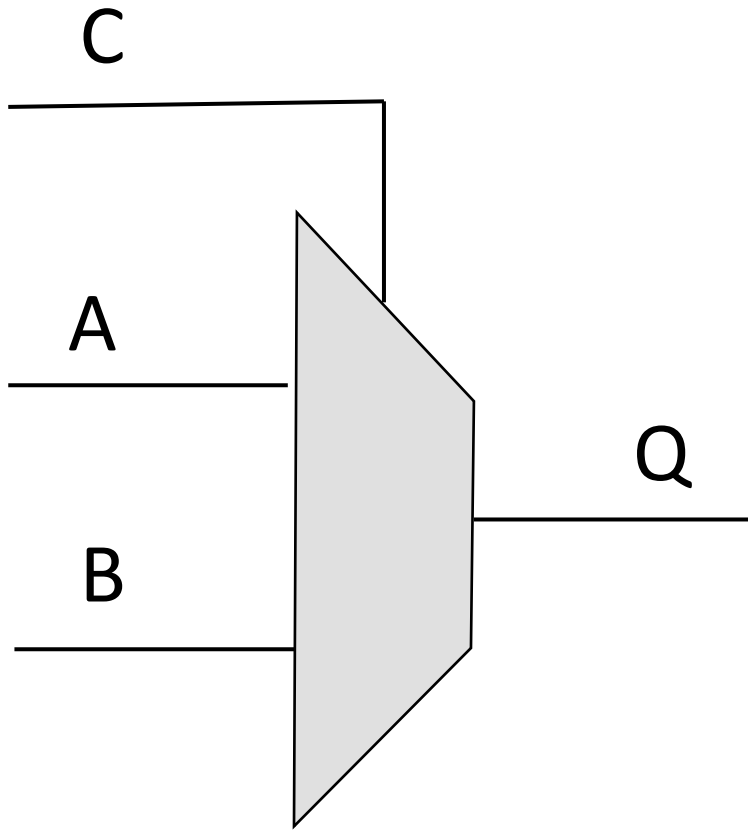
# Однотактная реализация процессора



# Однотактная реализация процессора



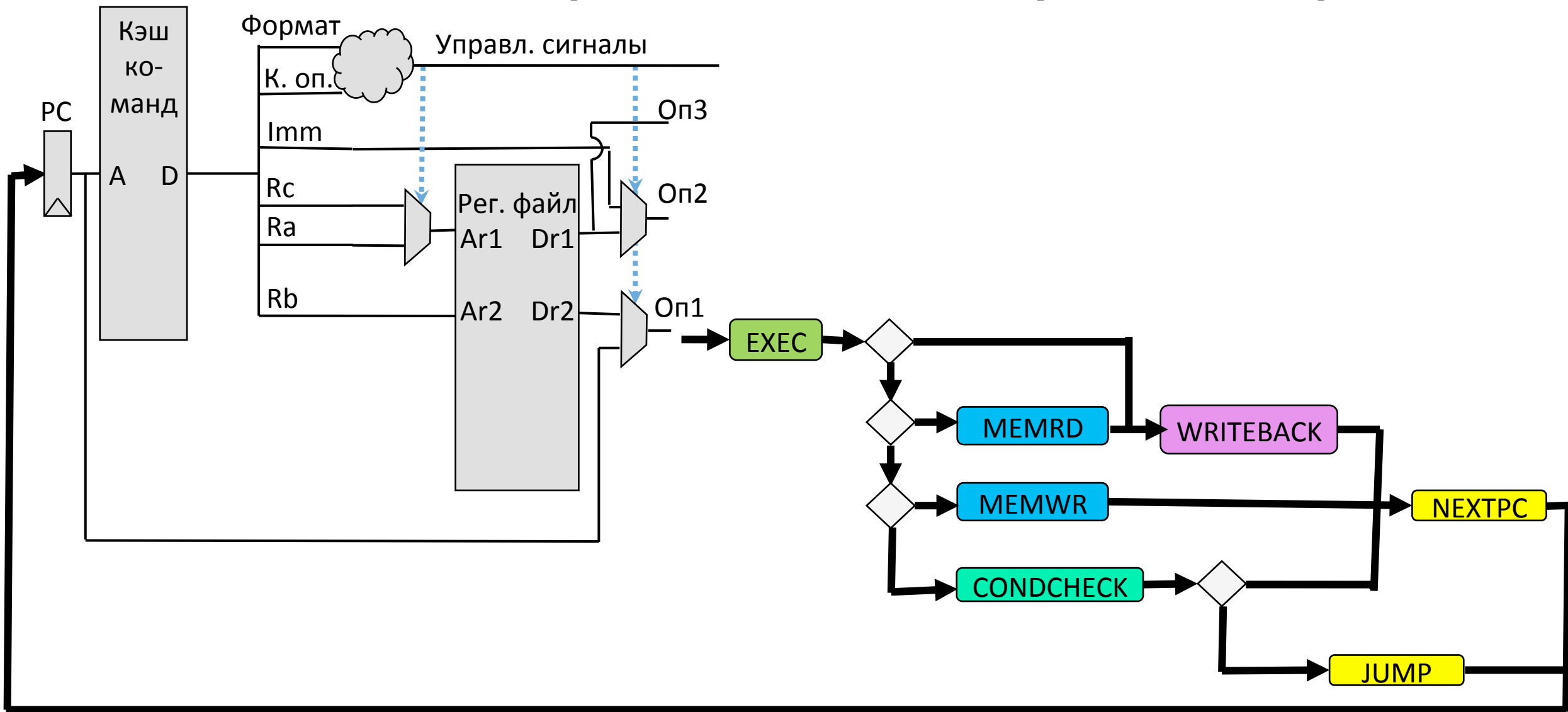
# Мультиплексор



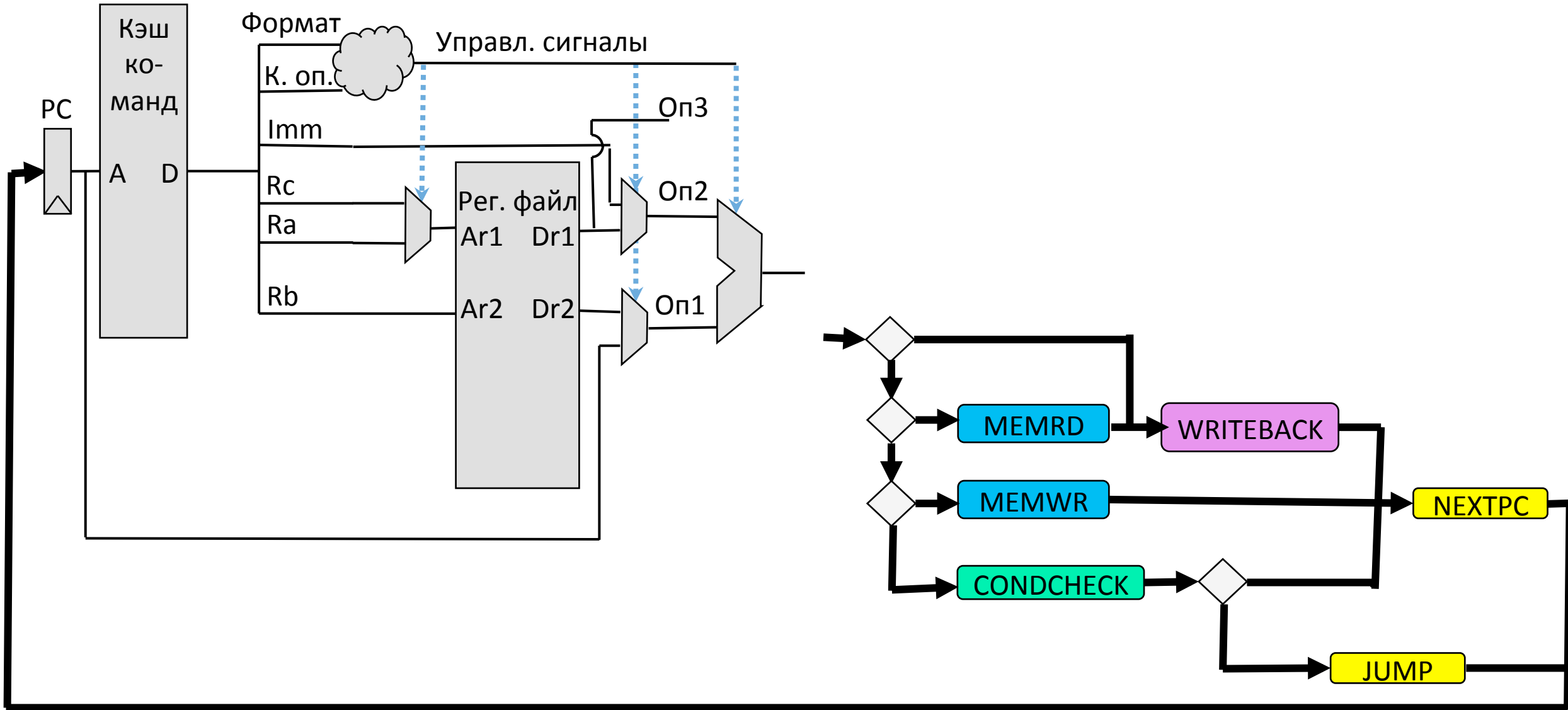
$$Q = \begin{cases} A, & \text{если } C = 1 \\ B, & \text{если } C = 0 \end{cases}$$



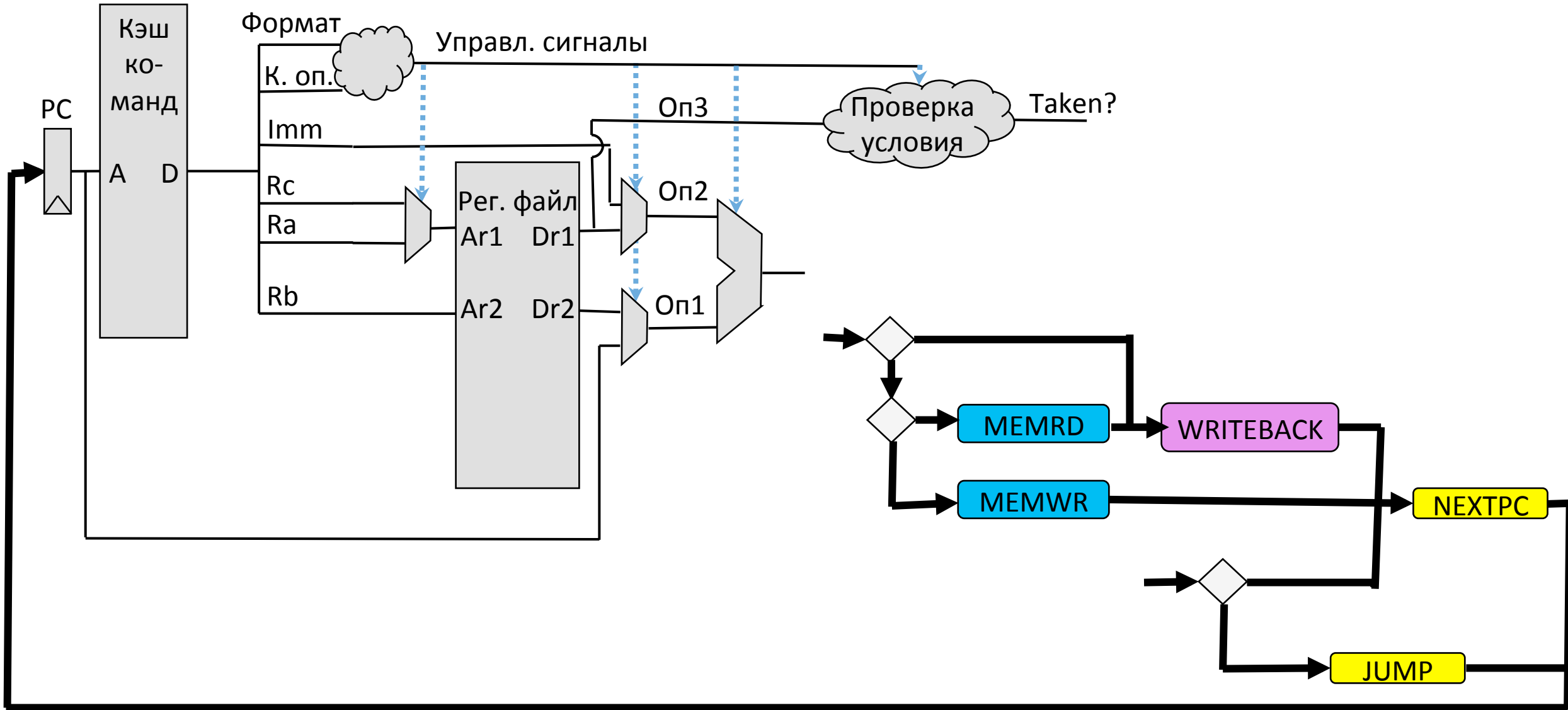
# Однотактная реализация процессора



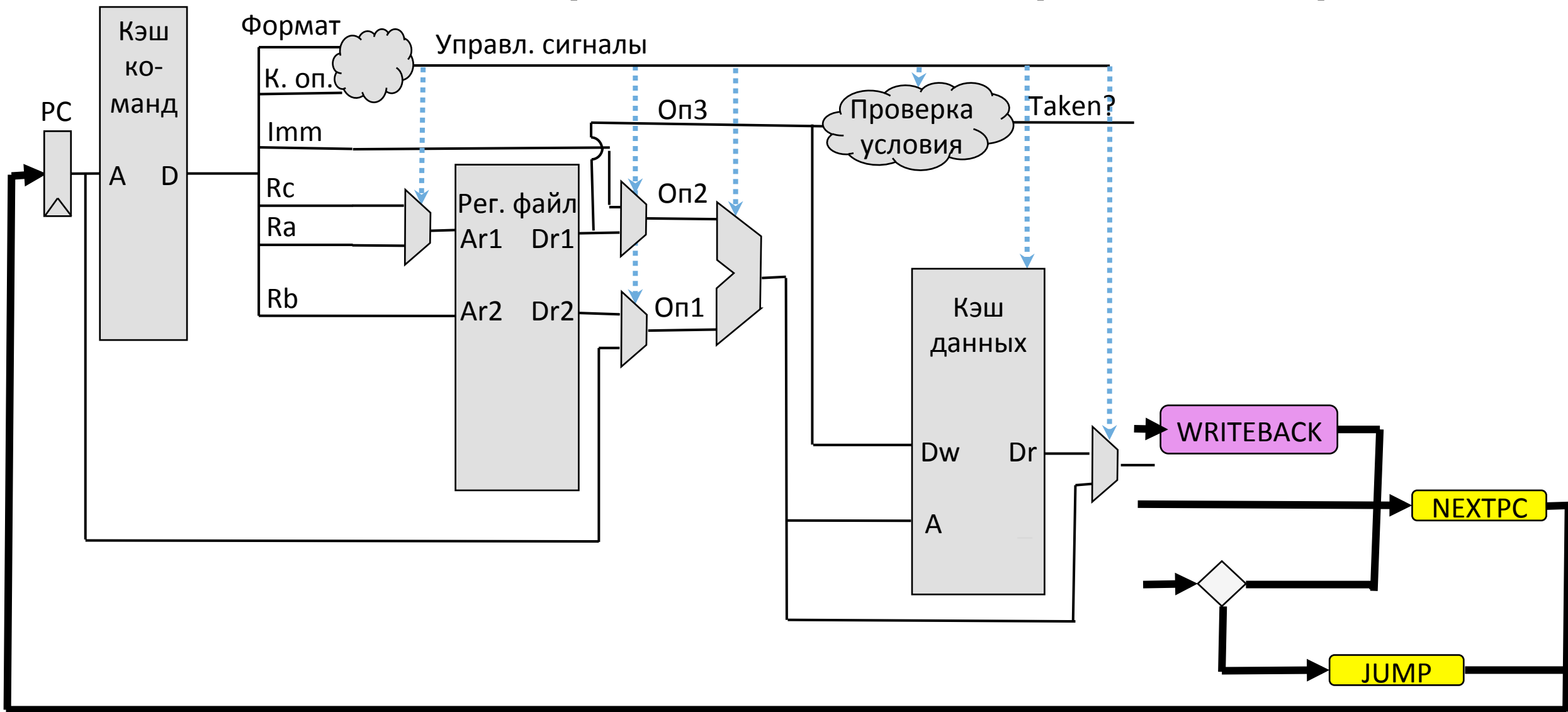
# Однотактная реализация процессора



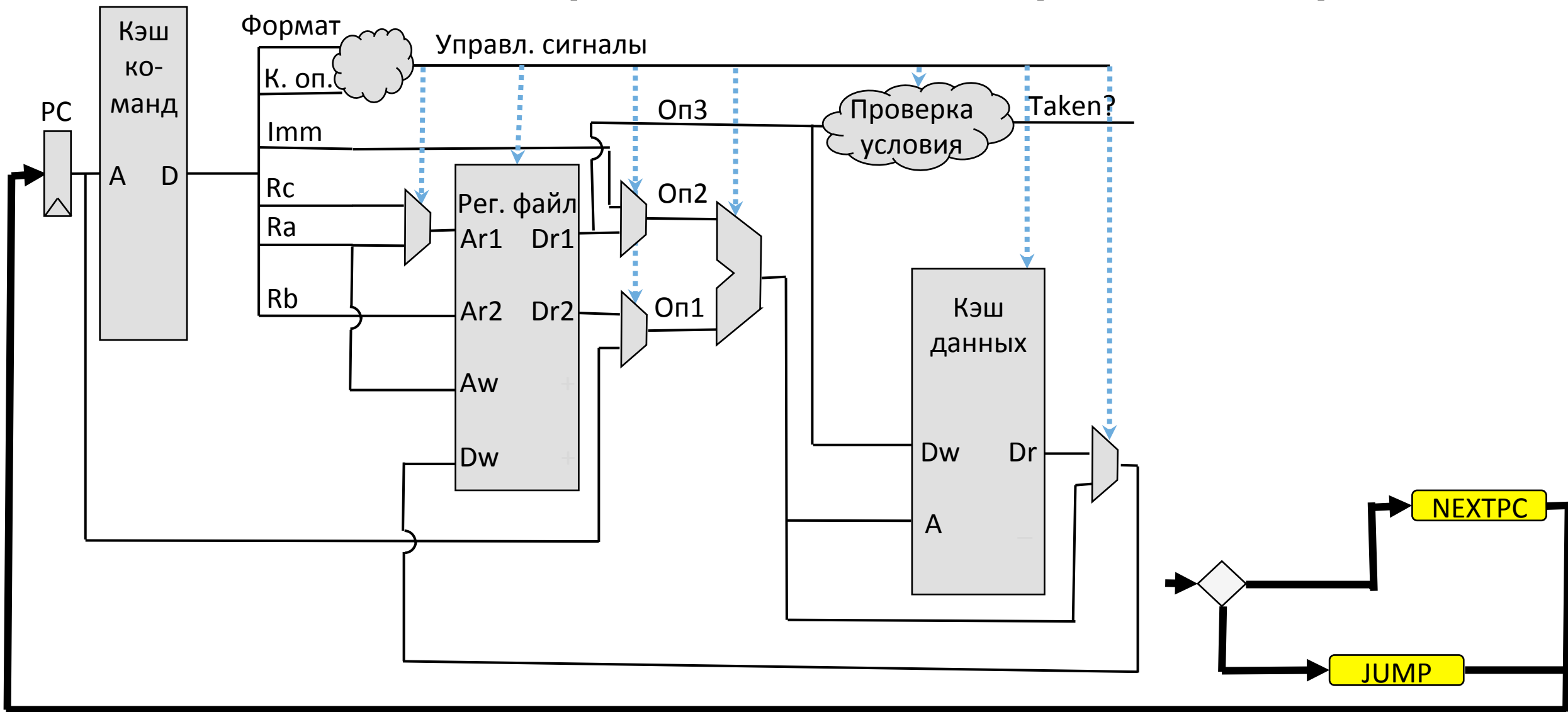
# Однотактная реализация процессора



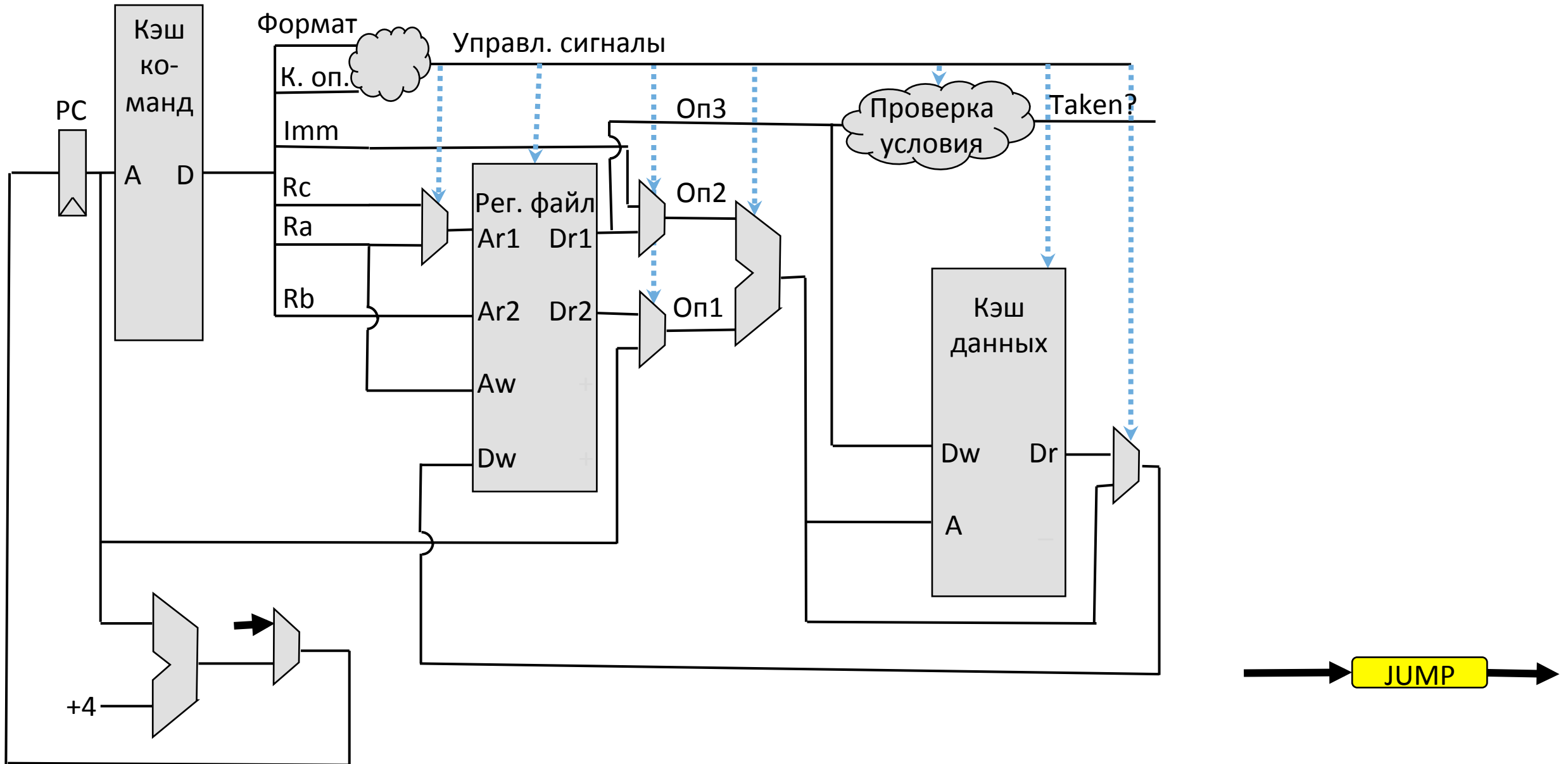
# Однотактная реализация процессора



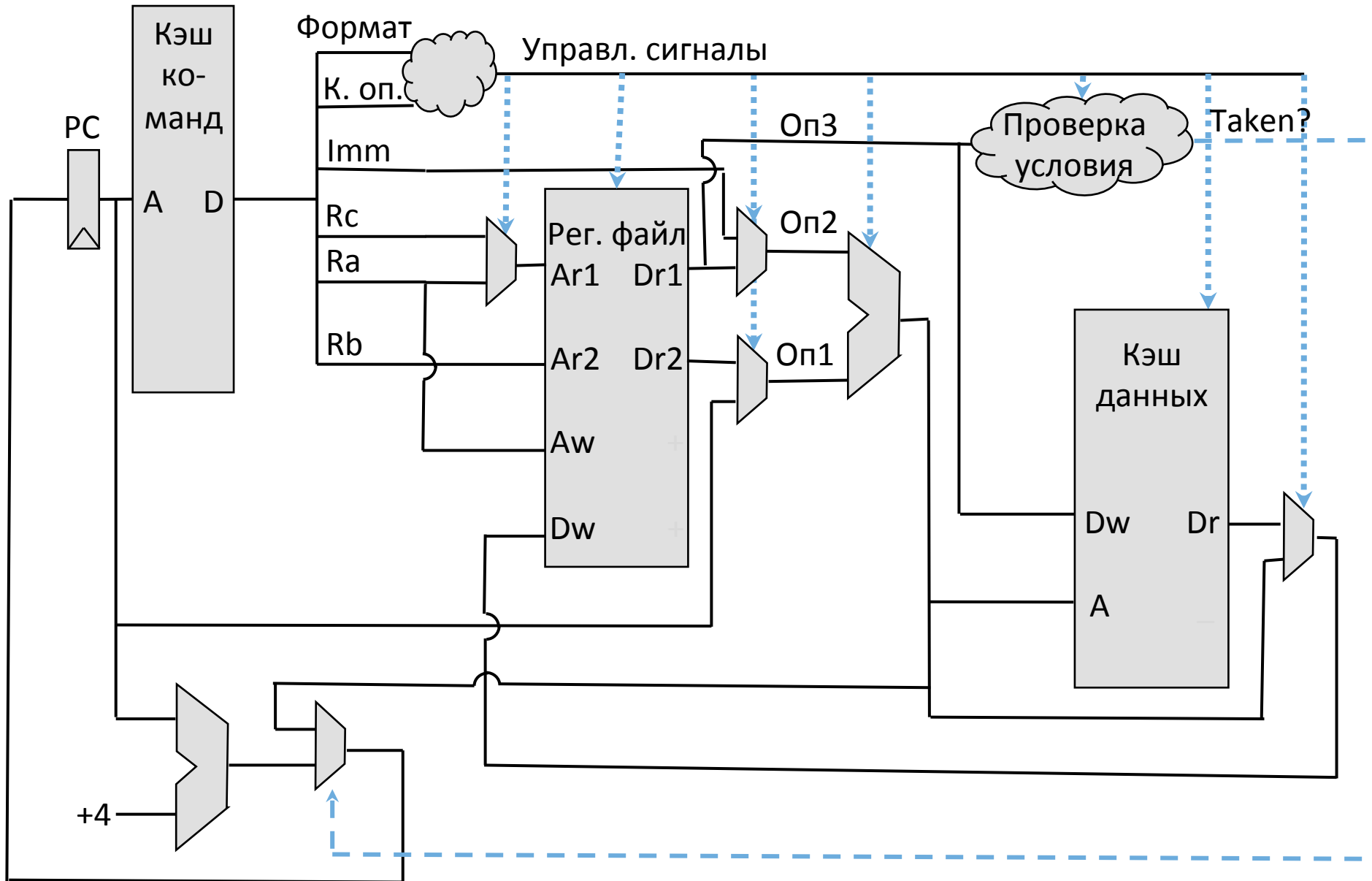
# Однотактная реализация процессора



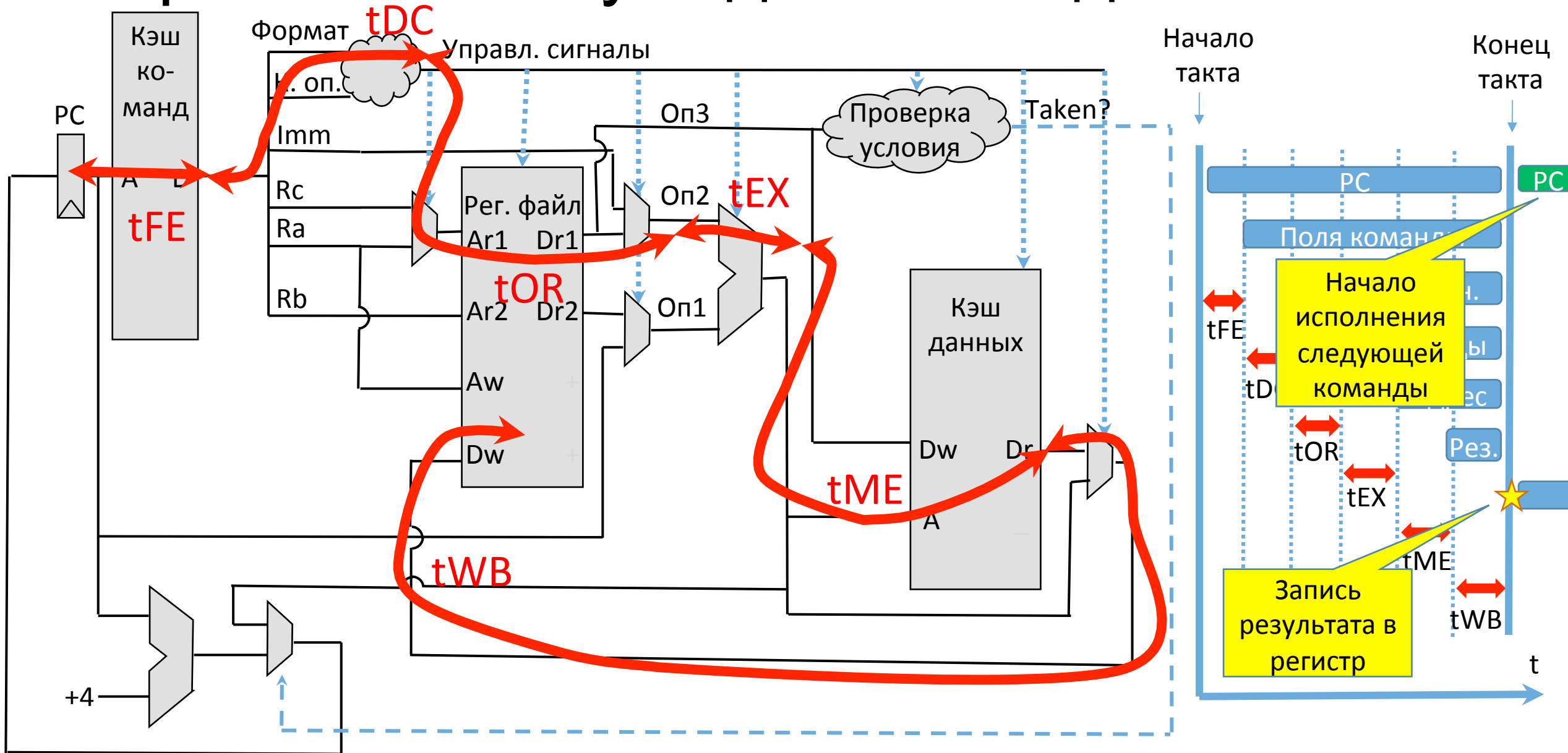
# Однотактная реализация процессора



# Однотактная реализация процессора



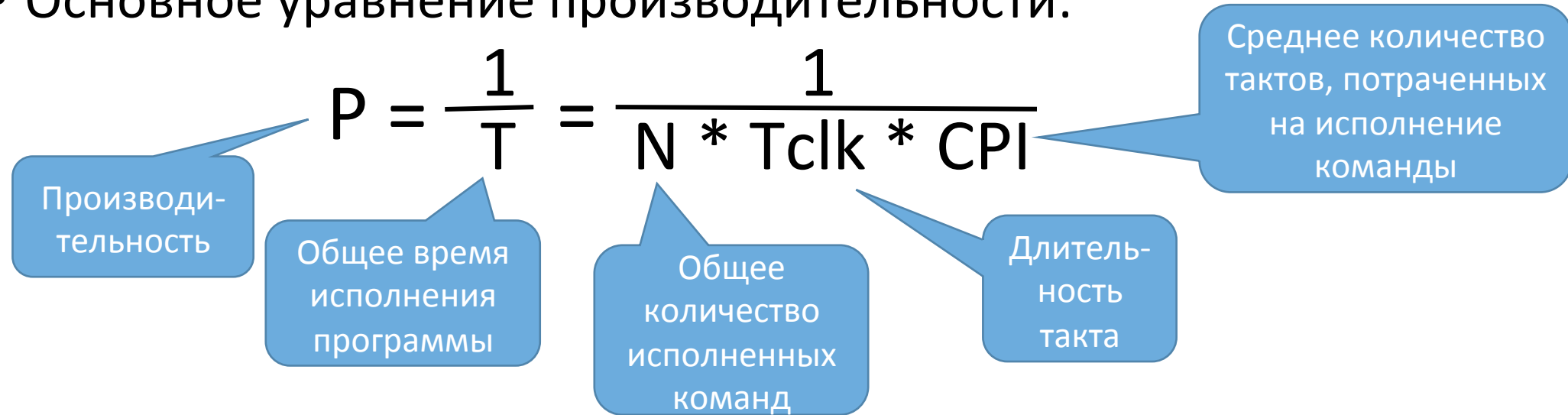
# Критический путь для команды Load





# Производительность одноктактной реализации

- Основное уравнение производительности:

$$P = \frac{1}{T} = \frac{1}{N * T_{clk} * CPI}$$


Производительность

Общее время исполнения программы

Общее количество исполненных команд

Длительность такта

Среднее количество тактов, потраченных на исполнение команды

- Для одноктактной реализации:  $CPI = 1$ ;  $T = t_{FE} + t_{DC} + t_{OR} + t_{EX} + t_{ME} + t_{WB}$

$$P = \frac{1}{N * (t_{FE} + t_{DC} + t_{OR} + t_{EX} + t_{ME} + t_{WB})}$$

# Идея конвейеризации

- Как повысить производительность для последовательности Load'ов?

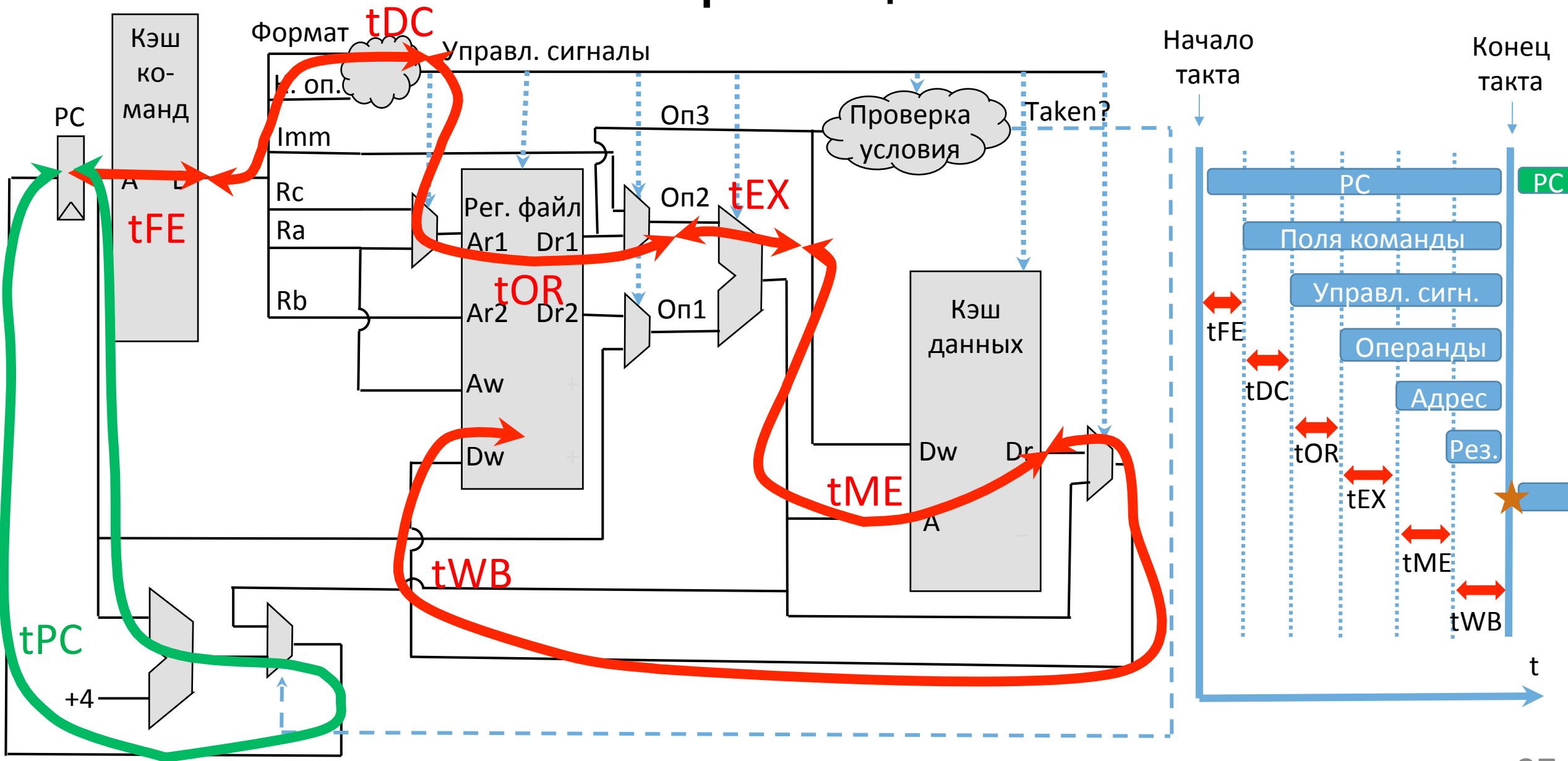
Load R1  $\leq$  [R7 + 10]

Load R2  $\leq$  [R8 + 14]

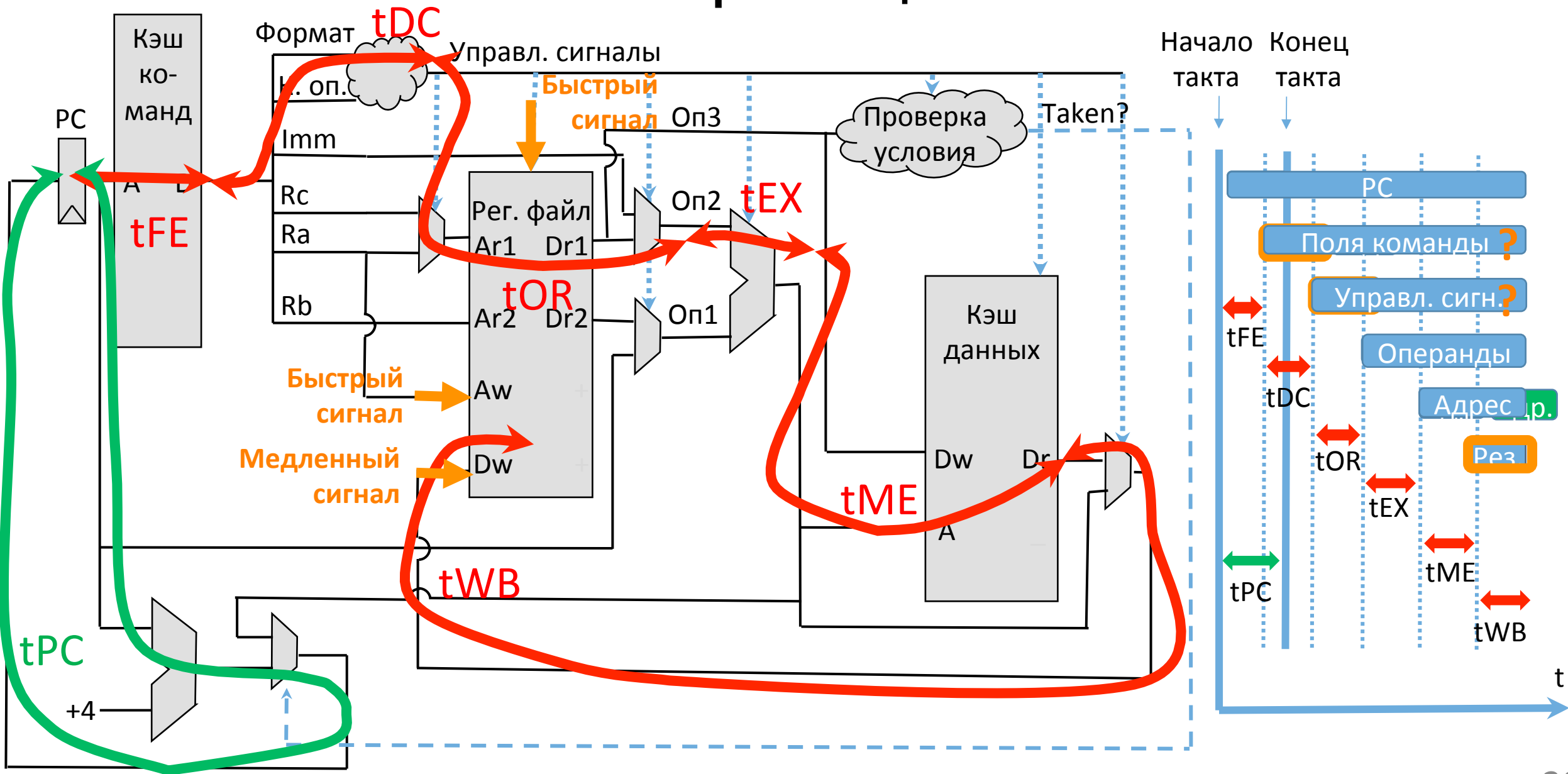
Load R3  $\leq$  [R9 + 18]

- Можем ли мы начинать исполнять каждый следующий Load раньше, чем завершится исполнение предыдущего?
- Тогда общая пропускная способность повысится, хотя время исполнения каждого отдельного Load'а не изменится

# Попытка конвейеризации



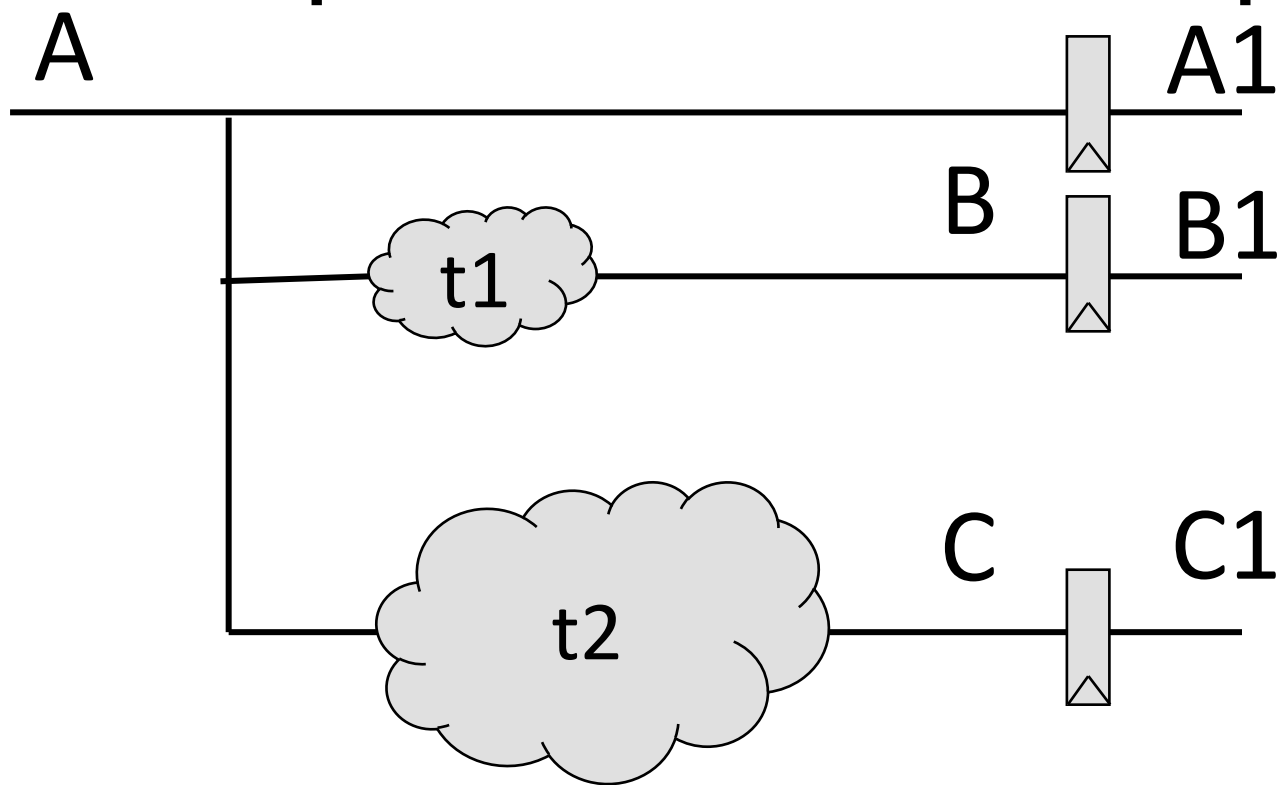
# Попытка конвейеризации



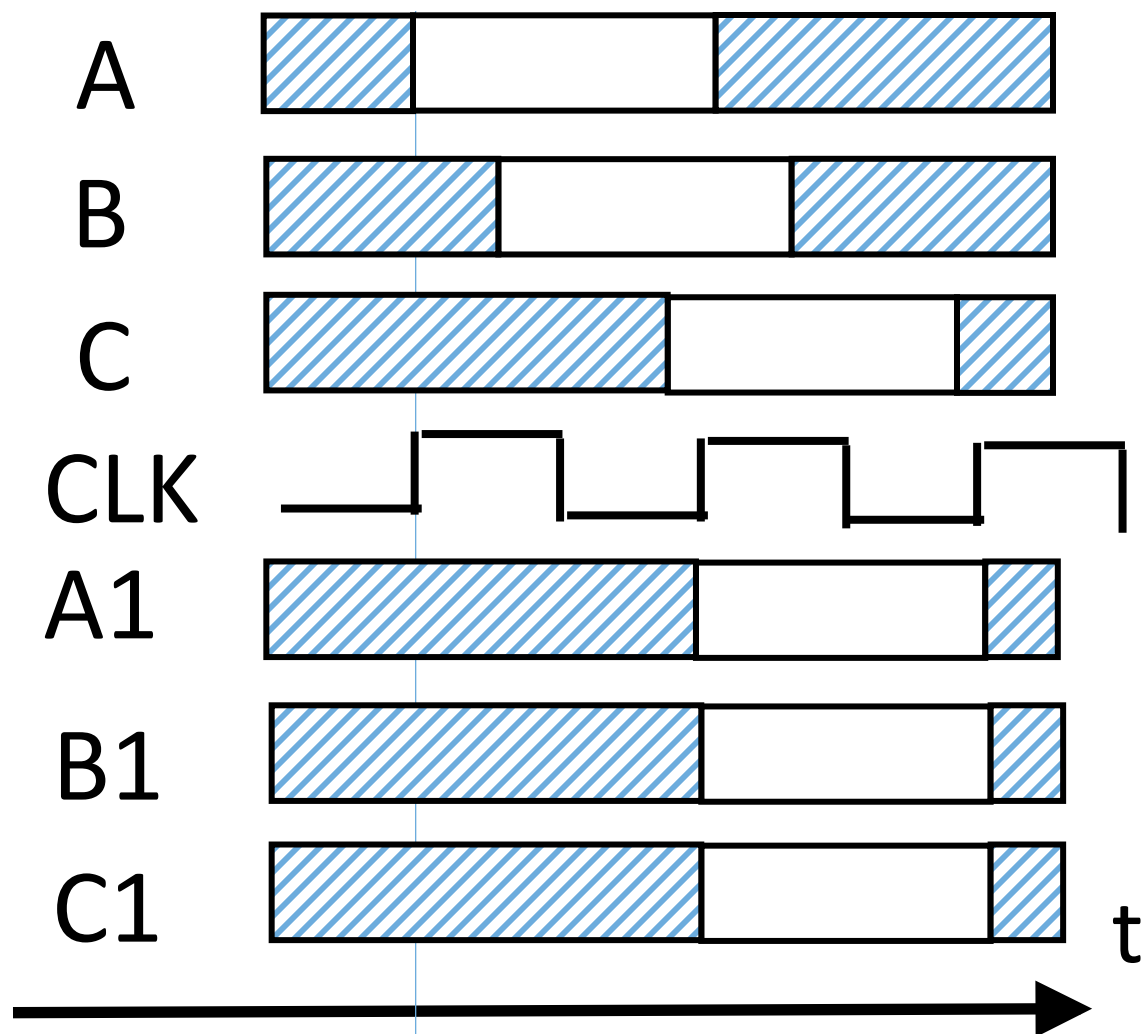
# Введение регистров в конвейер

- Проблема: адрес и управляющий сигнал для записи в регистровый файл сбрасываются слишком рано - еще до того, как будут готовы записываемые данные
- Решение: дополнительно задерживать «быстрые» сигналы для выравнивания времени задержек взаимодействующих между собой сигналов
- Универсальное средство для выравнивания задержек – промежуточные регистры

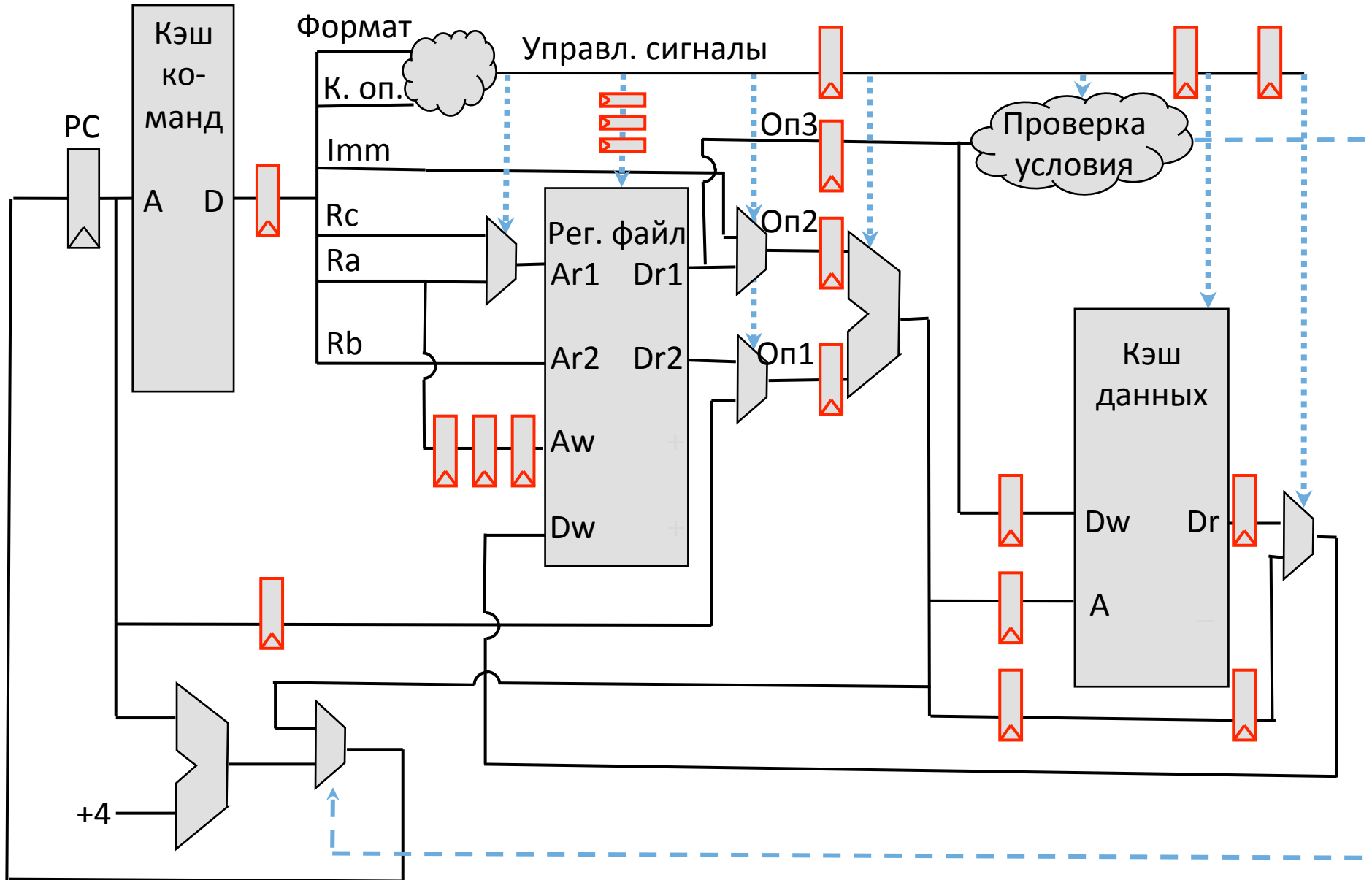
# Использование регистров для выравнивания задержек



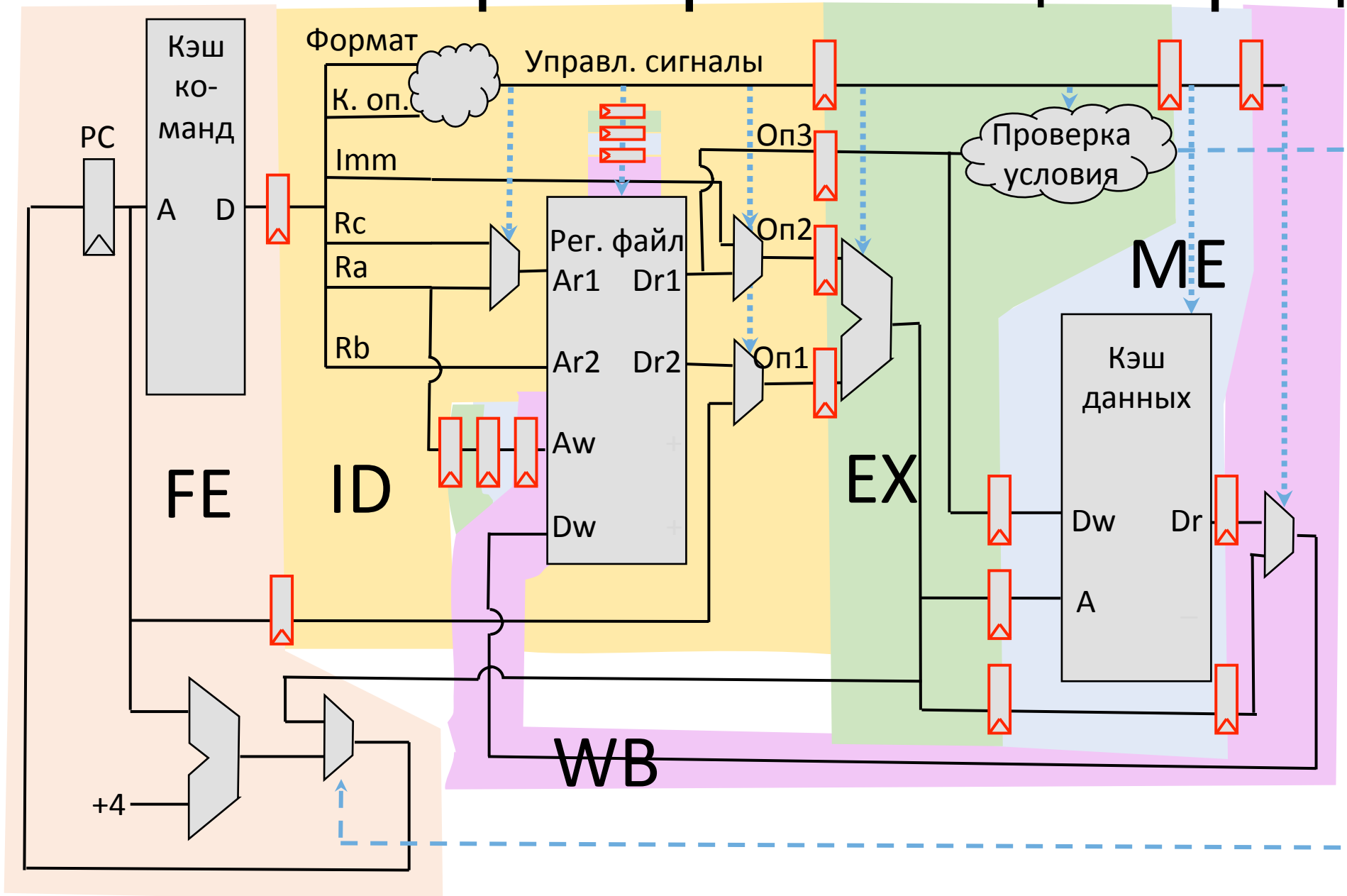
$$T_{clk} > t_{max}$$



# Конвейерная реализация процессора



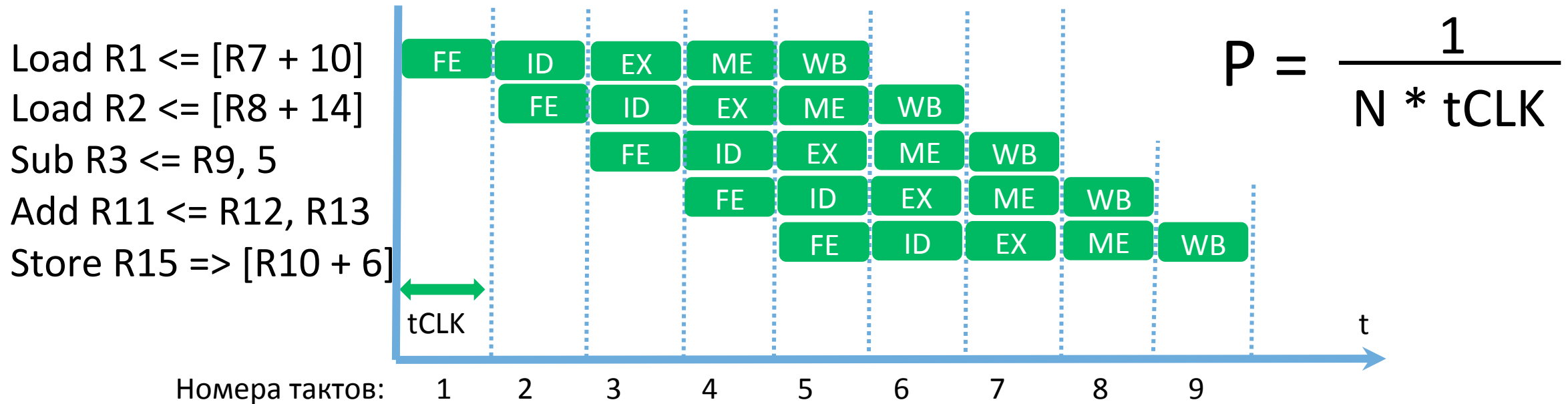
# Конвейерная реализация процессора





# Производительность конвейерной реализации

Период тактовой частоты:  $t_{CLK} = \max(t_{PC}, t_{FE}, t_{DC} + t_{OR}, t_{EX}, t_{ME}, t_{WB})$

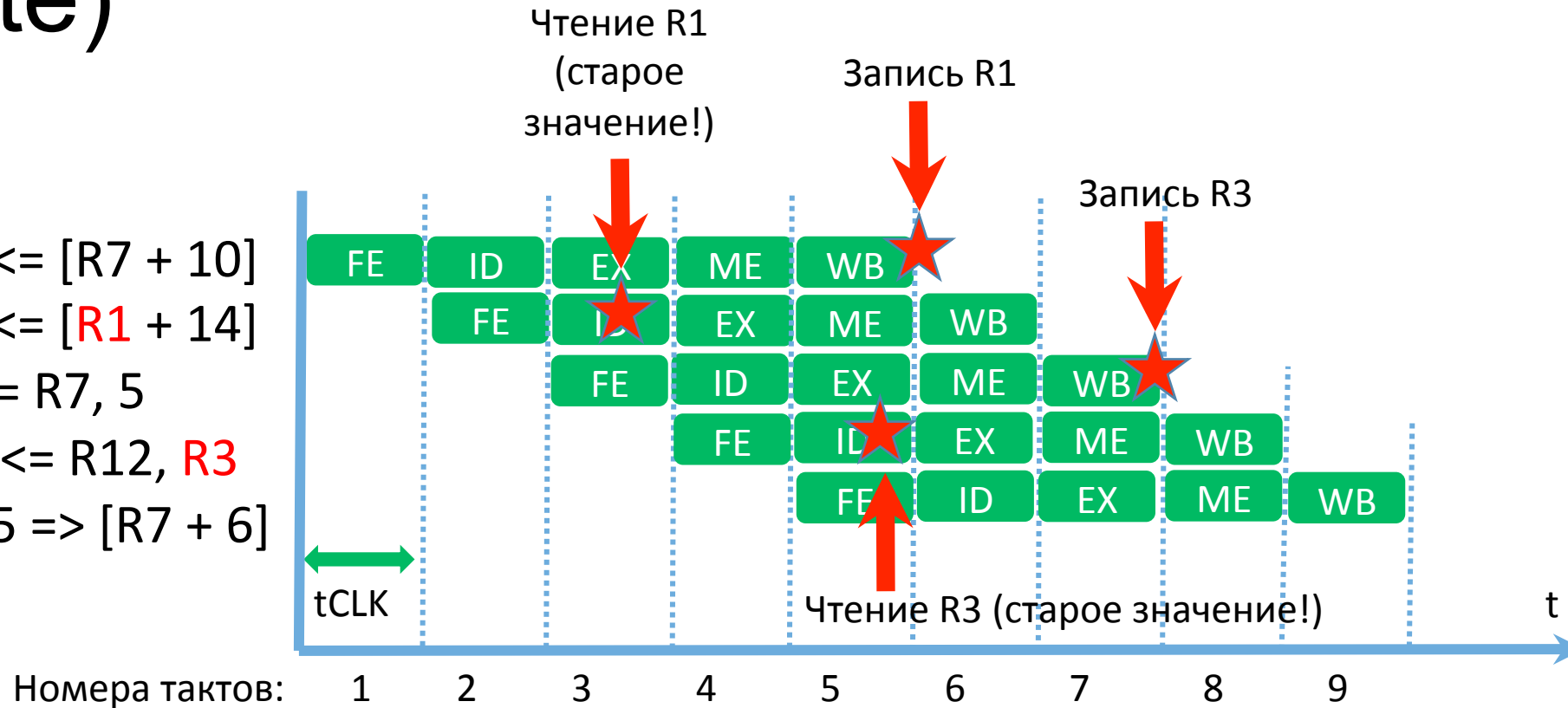


Ускорение по сравнению с однотактной реализацией:

$$\text{Ускорение} = \frac{t_{FE} + t_{DC} + t_{OR} + t_{EX} + t_{ME} + t_{WB}}{\max(t_{PC}, t_{FE}, t_{DC} + t_{OR}, t_{EX}, t_{ME}, t_{WB})}$$

# Конфликты по данным (Read-After-Write)

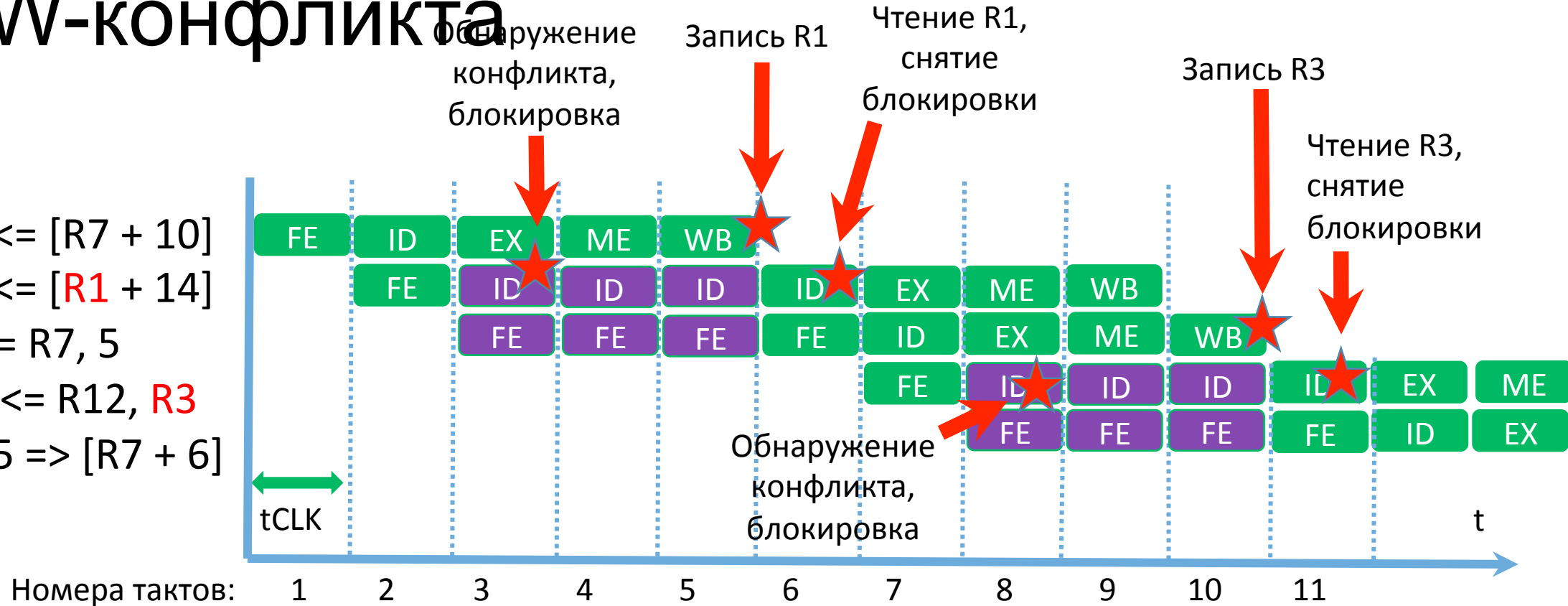
Load **R1**  $\leftarrow [R7 + 10]$   
Load  $R2 \leftarrow [R1 + 14]$   
Sub **R3**  $\leftarrow R7, 5$   
Add  $R11 \leftarrow R12, R3$   
Store  $R15 \Rightarrow [R7 + 6]$



- RAW-конфликт возникает, если из-за конвейера команда может прочитать устаревшее значение операнда

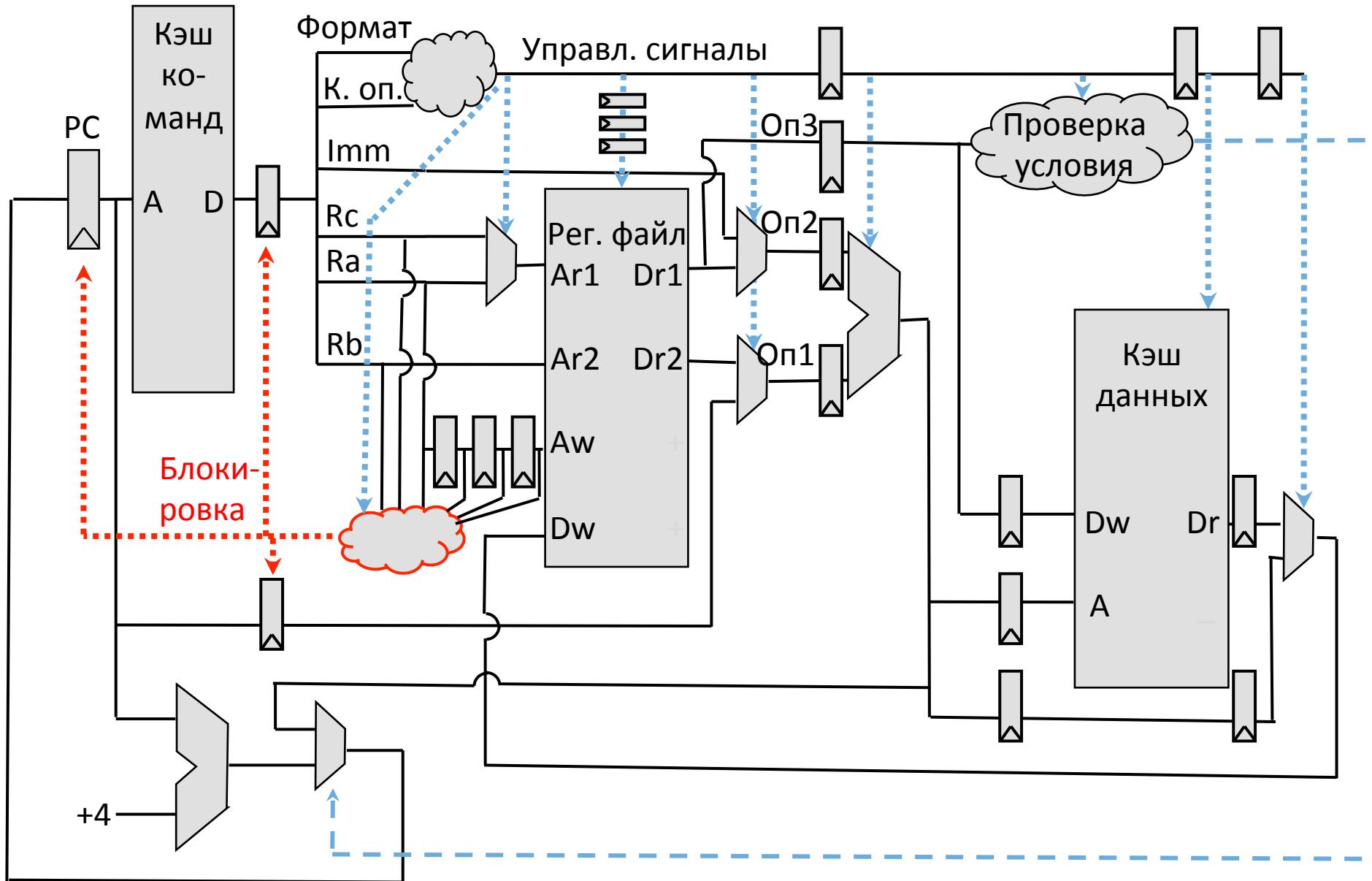
# Блокировка (stall) для разрешения RAW-конфликта

Load **R1**  $\leq [R7 + 10]$   
Load R2  $\leq [R1 + 14]$   
Sub **R3**  $\leq R7, 5$   
Add R11  $\leq R12, R3$   
Store R15  $\Rightarrow [R7 + 6]$



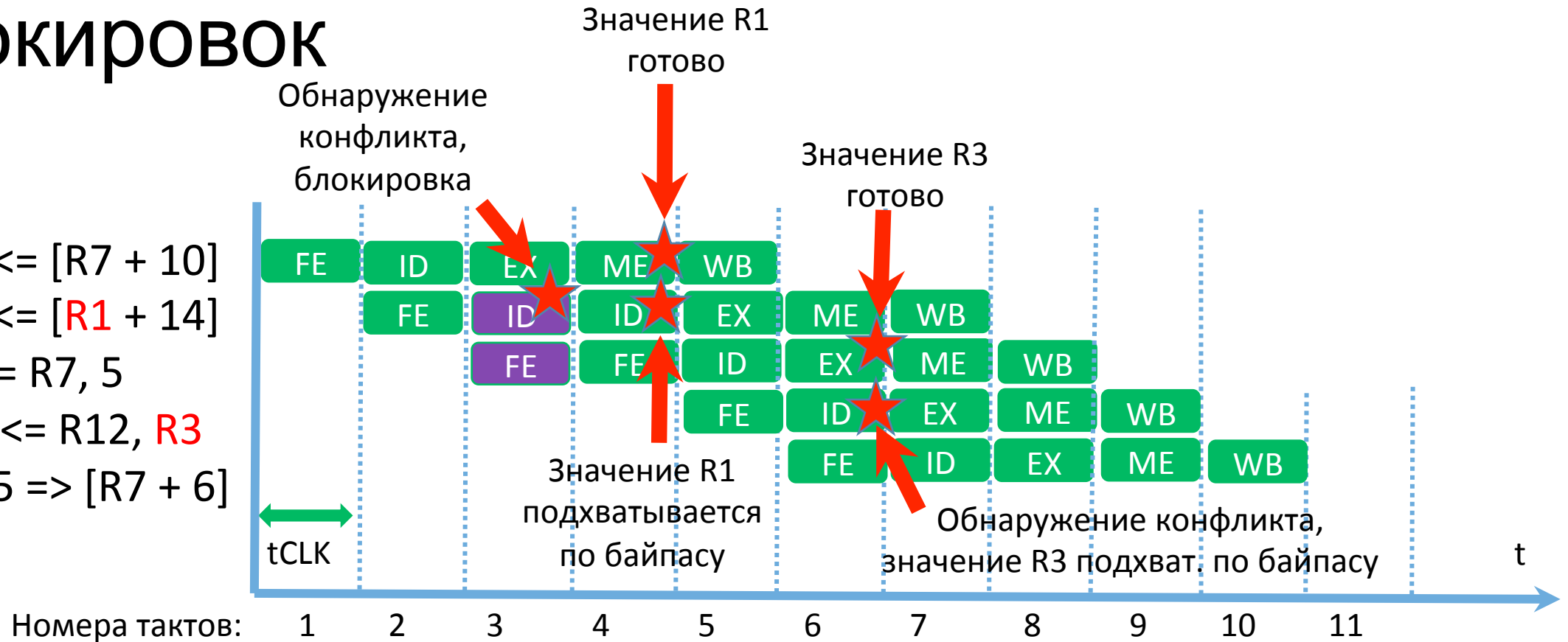
- Блокируется команда на стадии ID и следующая за ней на стадии FE
- Блокировка снижает производительность за счет уменьшения IPC

# Реализация блокировки конвейера



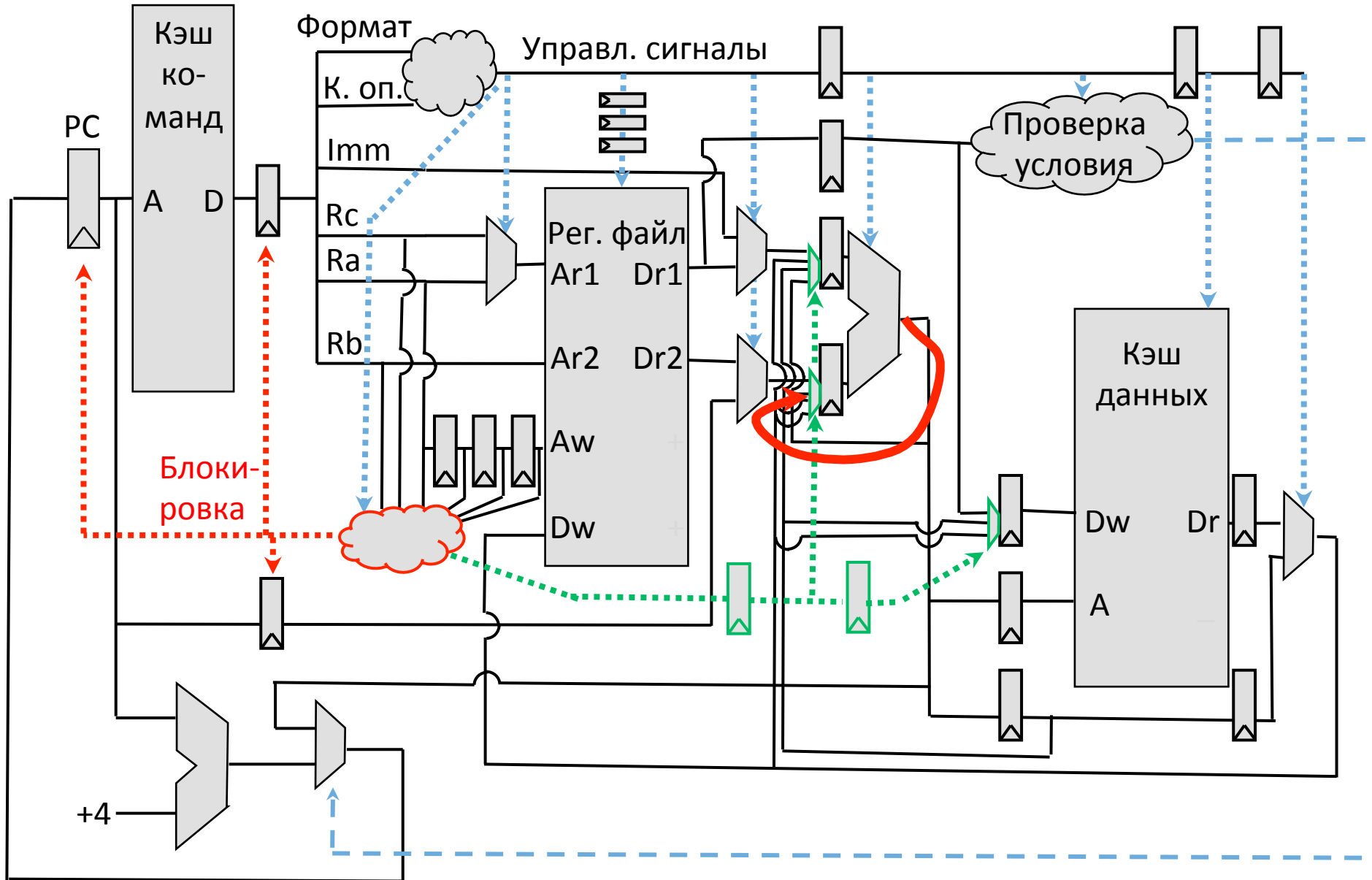
# Байпасы для уменьшения количества блокировок

Load **R1** <= [R7 + 10]  
Load R2 <= [**R1** + 14]  
Sub **R3** <= R7, 5  
Add R11 <= R12, **R3**  
Store R15 => [R7 + 6]

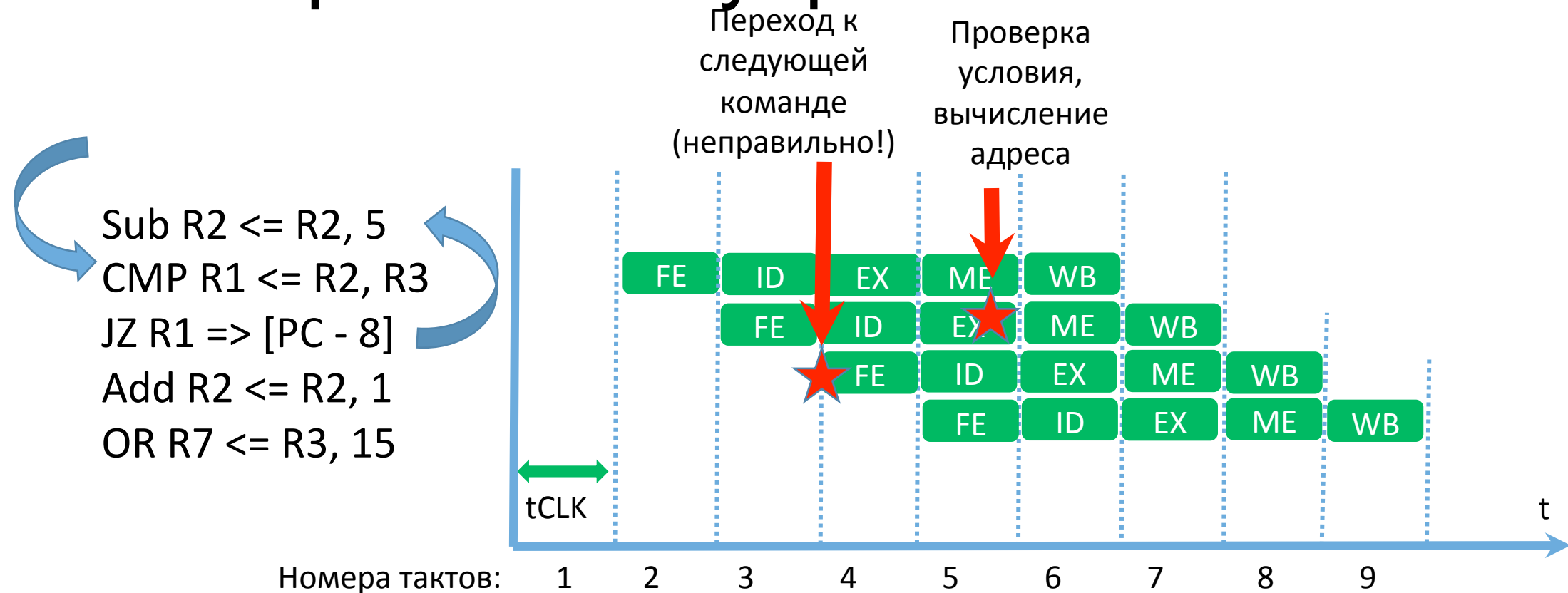


- Вместо чтения из регистрового файла данные передаются потребителю непосредственно с последующих стадий конвейера
- Повышается IPC по сравнению с полной блокировкой

# Реализация байпасов



# Конфликты по управлению



- Конфликт по управлению возникает, когда условие или адрес перехода еще не известны к моменту начала считывания кода

# Блокировка (stall) для разрешения конфликта по управлению

Проведена  
условия,

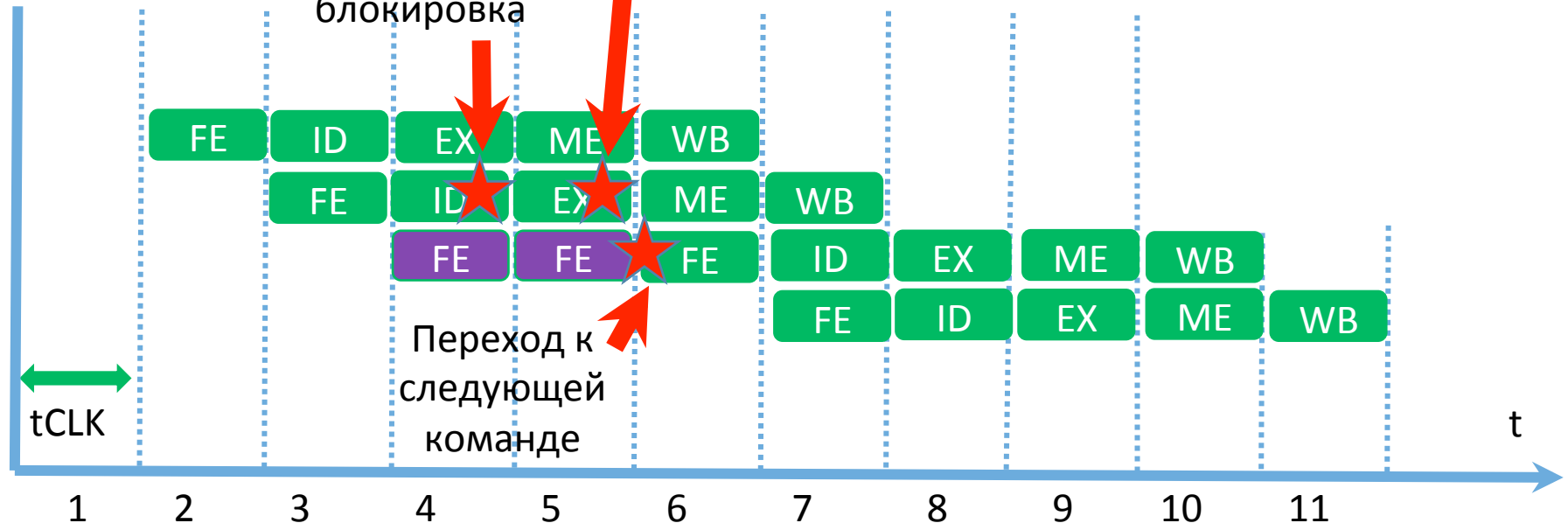
Если условие ложно  
(Not taken)

Обнаружение  
конфликта,  
блокировка

вычисление  
адреса

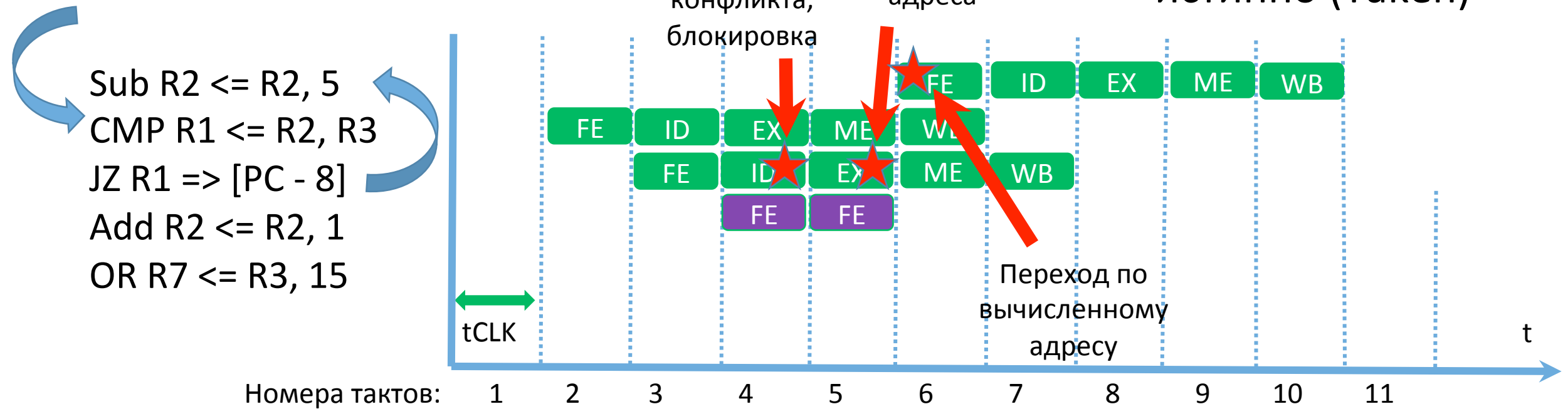
Переход к  
следующей  
команде

Sub R2 <= R2, 5  
CMP R1 <= R2, R3  
JZ R1 => [PC - 8]  
Add R2 <= R2, 1  
OR R7 <= R3, 15





# Блокировка (stall) для разрешения конфликта по управлению



- Вне зависимости от направления, при полной блокировке каждая выполненная команда условного перехода приводит к потере 2 тактов.

# Конфликты по управлению: ОПТИМИЗАЦИЯ

- Перенос логики проверки условия и вычисления адреса со стадии EX на стадию ID
  - Остается только один потерянный такт вместо двух
- Предсказание перехода: блокировки нет, но в случае истинного условия (Not Taken) отправленная в конвейер команда отменяется
  - Потеря такта имеет место только в случае истинного условия (Not Taken)
- Отложенное ветвление: изменение семантики системы команд, чтобы эффект от команды ветвления начинался не сразу со следующей команды, а только через одну
  - Потери IPC полностью устраняются, но компилятору труднее эффективно составить такую программу

# Какие еще могут быть конфликты?

- Write-After-Write (WAW):

Add R1 <= R2, R3

Sub R1 <= R4, R5

CMP R8 <= R1, 10

- Недопустимо, чтобы команда CMP использовала результат команды Add
- Актуально для конвейеров, имеющих переменную длину

- Write-After-Read (WAR):

Add R1 <= R2, R3

Sub R2 <= R4, R5

- Недопустимо, чтобы команда Add прочитала результат команды Sub.
- Актуально для конвейеров, допускающих изменение порядка команд

- Структурные

- Если бы мы использовали один и тот же кэш для команд и данных, то когда Load или Store находится на стадии ME, другая команда не может находиться на стадии FE

# Подходы к устранению конфликтов

- Блокировки
  - Приводят к потерям IPC, ухудшают задержки в схемах
- Сокращение разрыва между потребителем и производителем
  - Нужно стремиться использовать данные от других команд как можно позже в конвейере, а вырабатывать как можно раньше
- Изоляция стадий конвейера, выравнивание длин конвейеров
  - Разумный компромисс между экономией аппаратных затрат (за счет совместного использования оборудования разными стадиями) и отсутствием структурных конфликтов
- Предсказание / спекулятивное выполнение
  - Команда может запускаться без достоверного знания всей информации, а затем при необходимости отменяться или перезапускаться
- Переименование
  - Если ресурс переиспользуется, можно его продублировать и использовать несколько «инкарнаций» одновременно

**Спасибо за внимание!**

# Backup