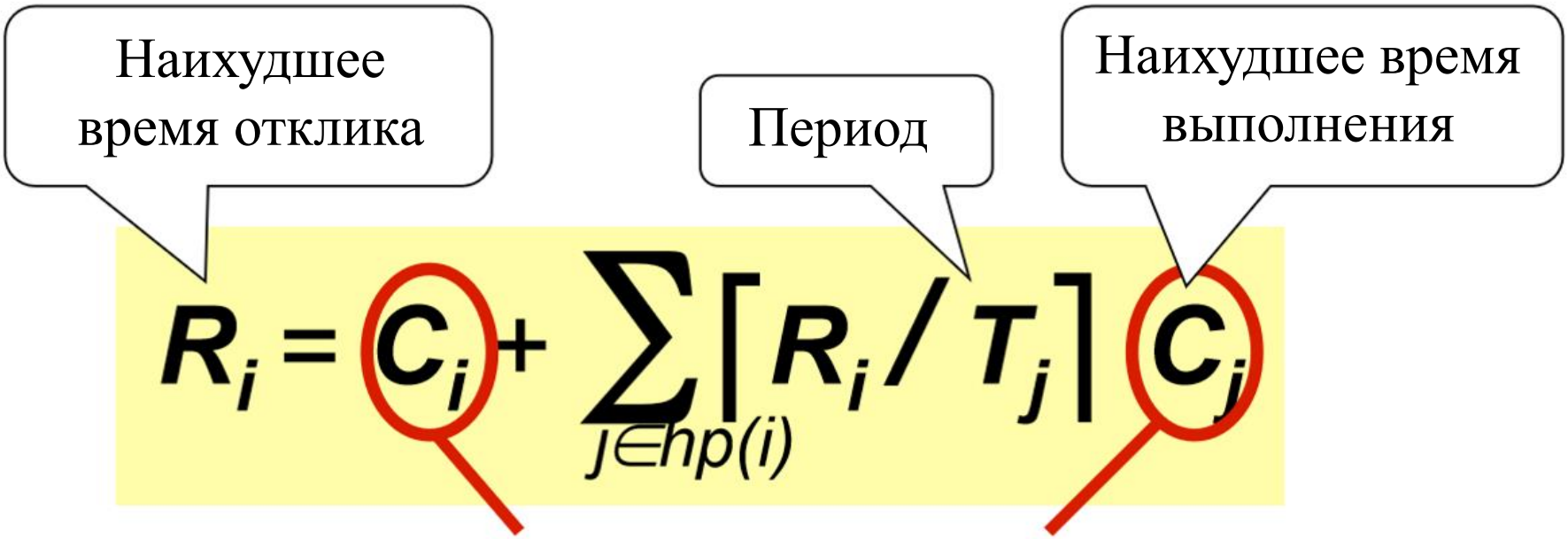
ВСТРОЕННЫЕ ИНФОРМАЦИОННО-УПРАВЛЯЮЩИЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Лекция 3:
Оценка наихудшего времени выполнения программ (WCET)

Кафедра АСВК,
Лаборатория Вычислительных Комплексов
Балашов В.В.

Почему важен WCET

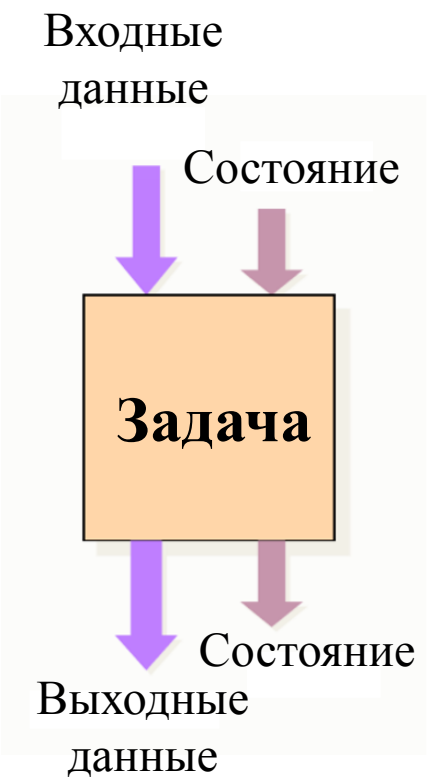
- *Время выполнения* программ играет ключевую роль при анализе систем реального времени
- Это время, например, встречается в формуле:



Откуда берутся значения C_i, C_j?

Простейшая вычислительная задача

- Входные данные доступны в момент старта
- Выходные данные готовы в момент завершения
- Нет блокировок в процессе выполнения
- Нет синхронизации или обмена данными в процессе выполнения
- Время выполнения зависит только от:
  - входных данных
  - состояния задачи в момент старта (внешние воздействия отсутствуют)

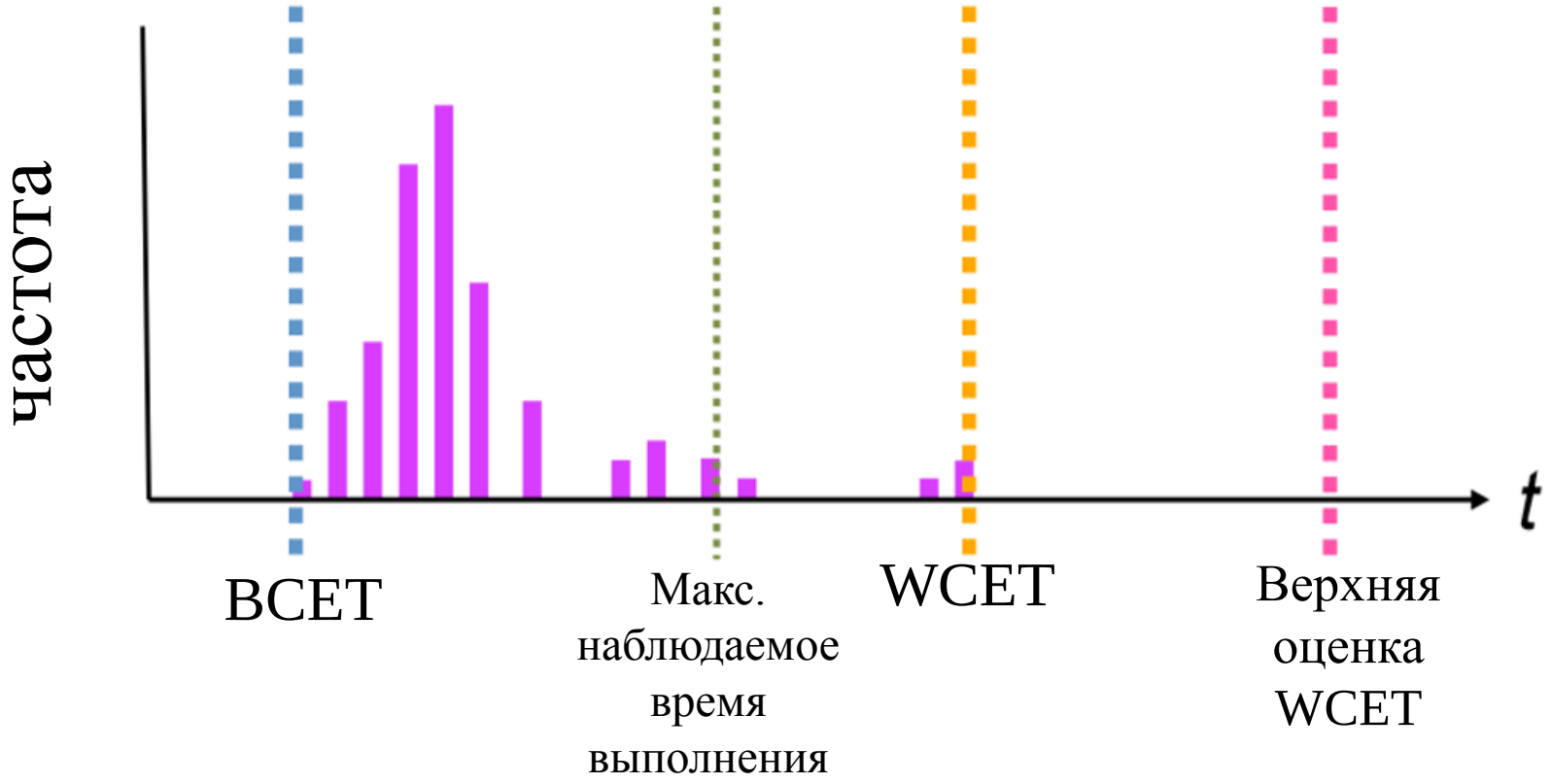


Наихудшее (максимальное) время выполнения

Наихудшее время выполнения программного кода (worst case execution time, WCET) – это максимальное время, которое требуется для выполнения

- данного фрагмента кода
- в данном контексте (входные данные, состояние)
- на заданном аппаратном вычислителе

Время выполнения: терминология



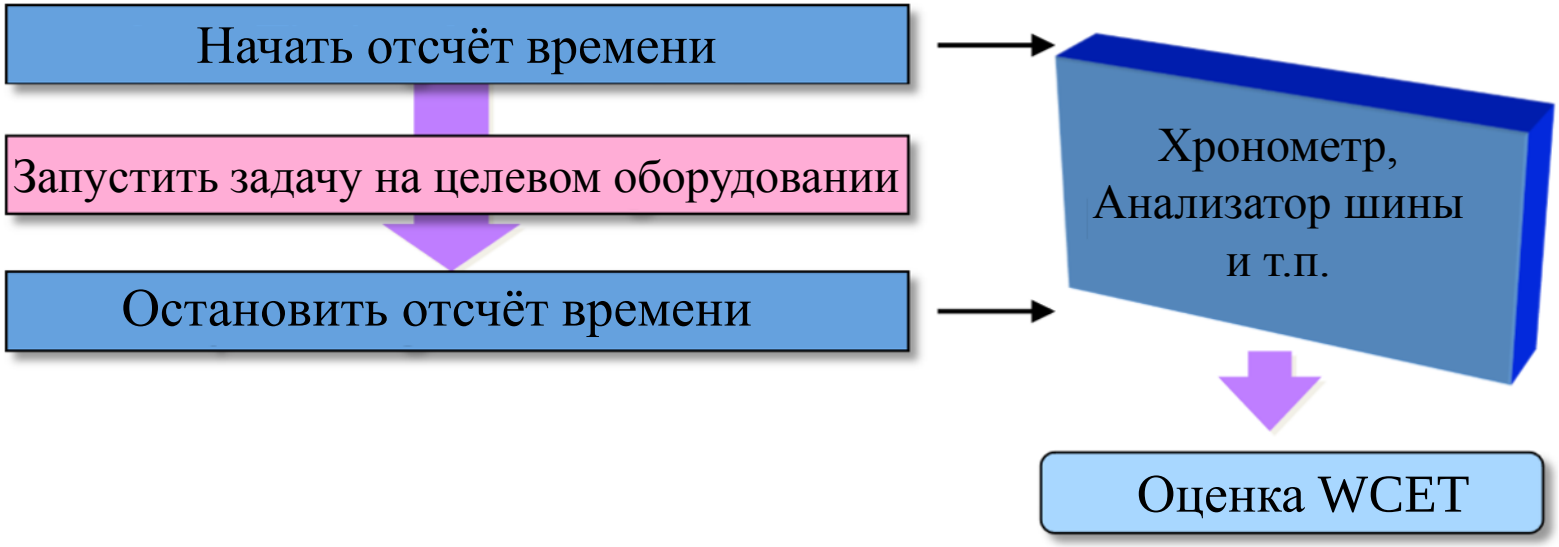
BCET – best-case execution time (минимальное время выполнения)
WCET – worst-case execution time (максимальное время выполнения)

Анализ WCET

Цель анализа WCET: оценить сверху время выполнения фрагмента кода

- оценка WCET должна быть безопасной (недопустимо ошибаться в меньшую сторону)
- оценка WCET должна быть достаточно точной (её завышенность приведет к излишнему резервированию ресурсов системы)
- затраты вычислительных ресурсов на анализ должны быть разумными

Измерение WCET



Почему нельзя просто измерить WCET?

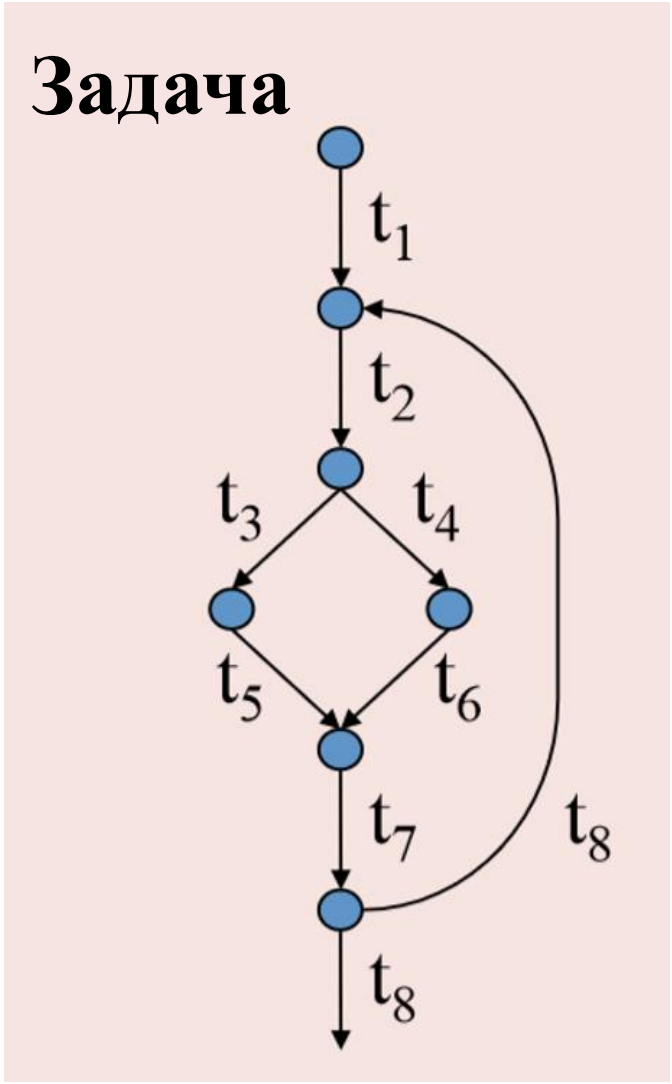
- Замер времени выполнения на всех путях выполнения реалистичной программы – на практике невозможен
- При определении тестовой выборки могут быть упущены редкие сценарии выполнения (обработка ошибочных ситуаций и т.п.)
- Выбранные тестовые данные могут не породить самую длинную (по времени) трассу выполнения
- Внутреннее состояние процессора на момент старта измерений может не быть наихудшим

Простые замеры могут послужить источником первоначальной (грубой) *нижней* оценки WCET

Статический анализ WCET

- При статическом анализе WCET вычисляется верхняя оценка времени выполнения фрагментов кода
- Моделируются аппаратные и программные средства, а также контекст выполнения
  - Программные средства: исходный код, двоичный код (с привязкой к физическим адресам)
  - Аппаратные средства: процессор (в т.ч. конвейер), память (в т.ч. кэш-память)
  - Контекст: начальное состояние аппаратных и программных средств

Чем определяется WCET

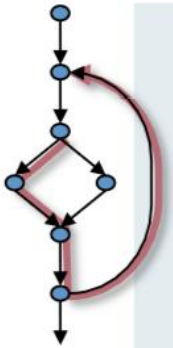


- Возможные последовательности действий задачи (пути выполнения)
- Длительность выполнения каждого действия на каждом допустимом (т.е. практически возможном) пути выполнения

Чем определяется WCET

Последовательности действия определяются:

- Семантикой программного кода (спецификой реализации, в т.ч. аппаратно-зависимой семантикой)
- Входными данными, возможными в данном контексте вызова программы



Длительность действий определяется:

- Аппаратной реализацией команд процессора
- Состоянием аппаратных средств, влияющих на тайминги (кэш-память, конвейер и т.п.)
  - Факторы, внутренние для задачи
  - Внешние факторы – состояние на момент старта; состояние после вытеснения задачи

Длительность выполнения путей – простой и сложный случаи

Длительность выполнения пути k : $xt(p_k)$

Простейшая архитектура:

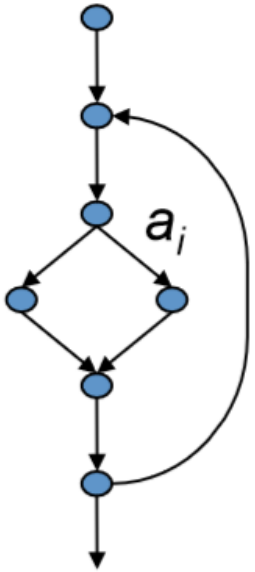
длительность каждого действия a_i – константа:

$$xt(p_k) = \sum n_{k,i} t(a_i)$$

Реалистичная архитектура:

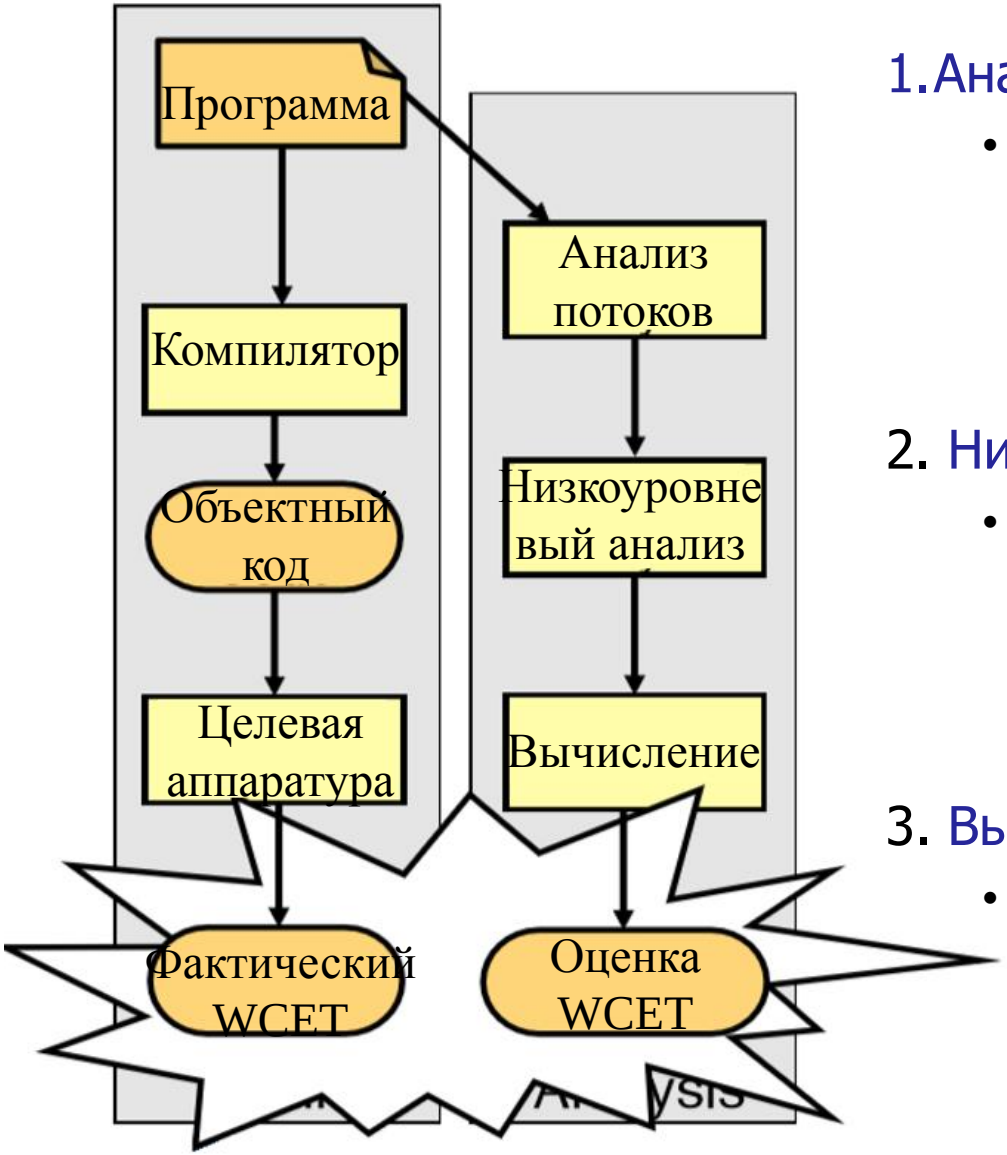
длительности действия варьируются:

$$xt(p_k) = \sum t(a_{i,j(k)})$$

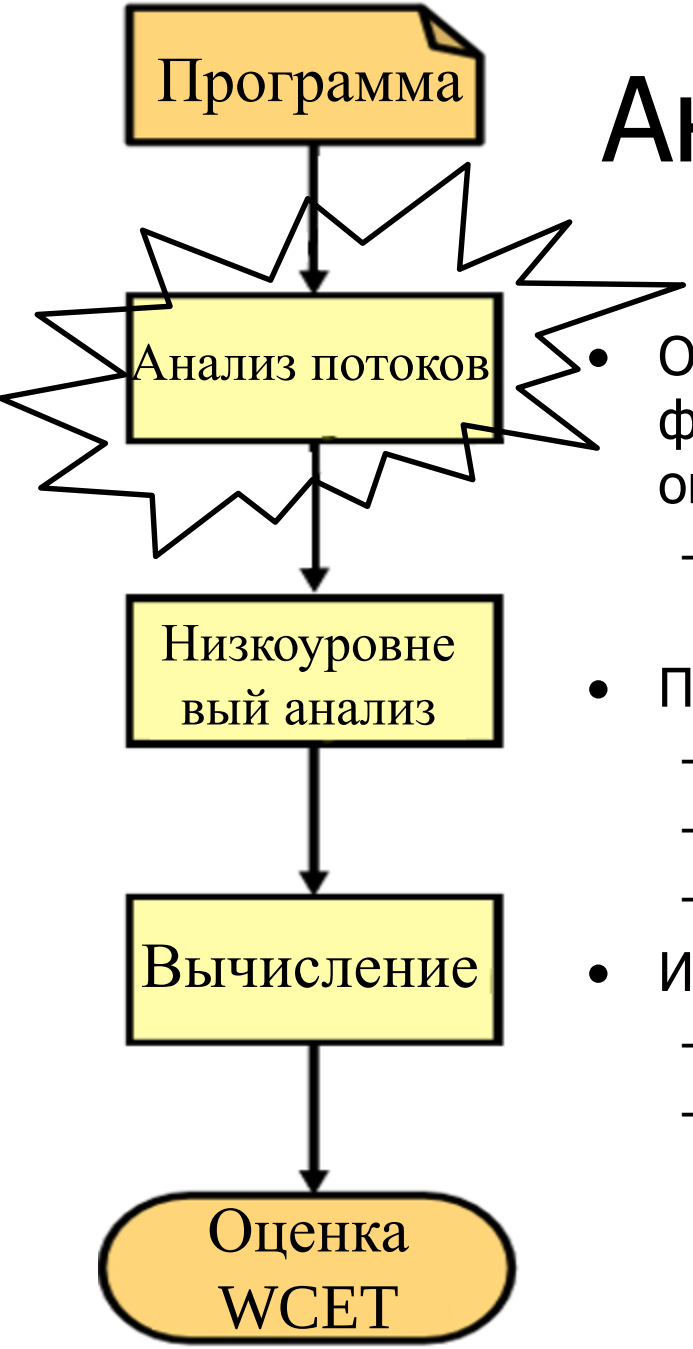


Причины: конвейер, кэш-память, параллелизм в процессоре, …

Фазы анализа WCET



1. Анализ потоков.
   - Ограничить (сверху) число выполнений различных фрагментов программы (в основном, анализ программной составляющей)
2. Низкоуровневый анализ
   - Ограничить (сверху) время выполнения различных фрагментов программы (сочетание анализа программной и аппаратной составляющих)
3. Вычисление
   - Объединить результаты анализа потоков и низкоуровневого анализ, чтобы получить верхнюю оценку WCET

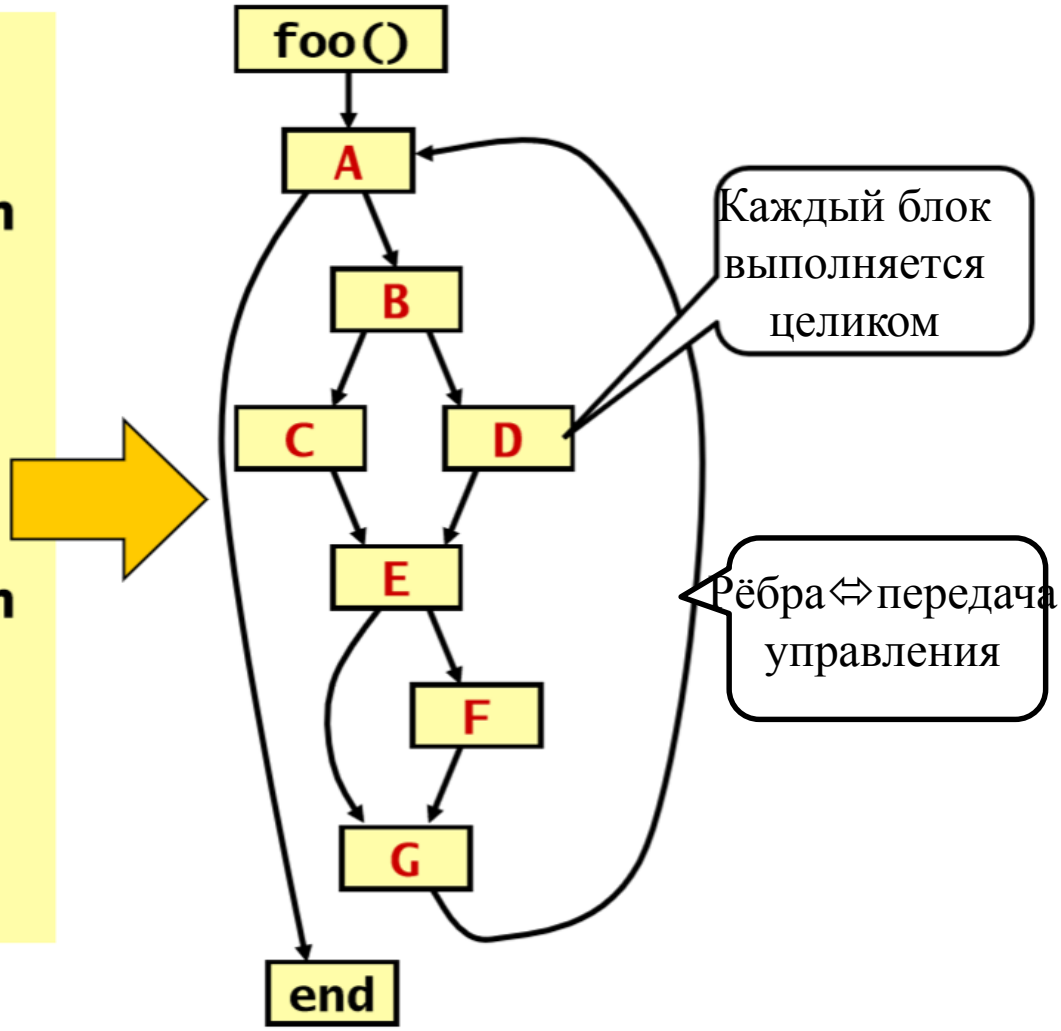


Анализ потоков

- Ограничивает число выполнений различных фрагментов программы (вычисляет верхнюю оценку)
  - оценка корректна для всех возможных трасс выполнения
- Примеры предоставляемой информации:
  - Ограничение на число итераций цикла
  - Ограничение на глубину рекурсии
  - Недопустимые пути выполнения
- Источники информации:
  - Статический анализ программы
  - Ручные аннотации кода

Граф потока управления

```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```



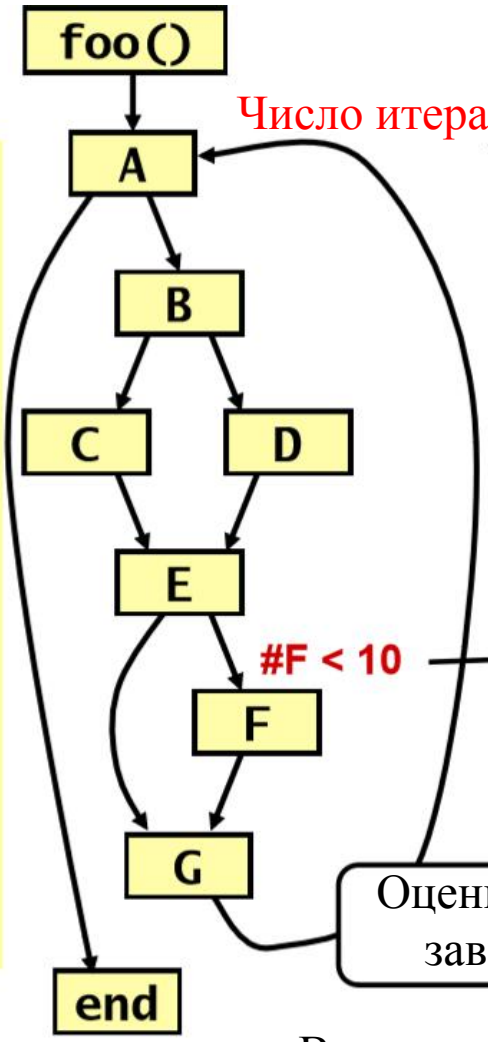
Потоковая информация и недопустимые пути

$1 \le i \le 100$

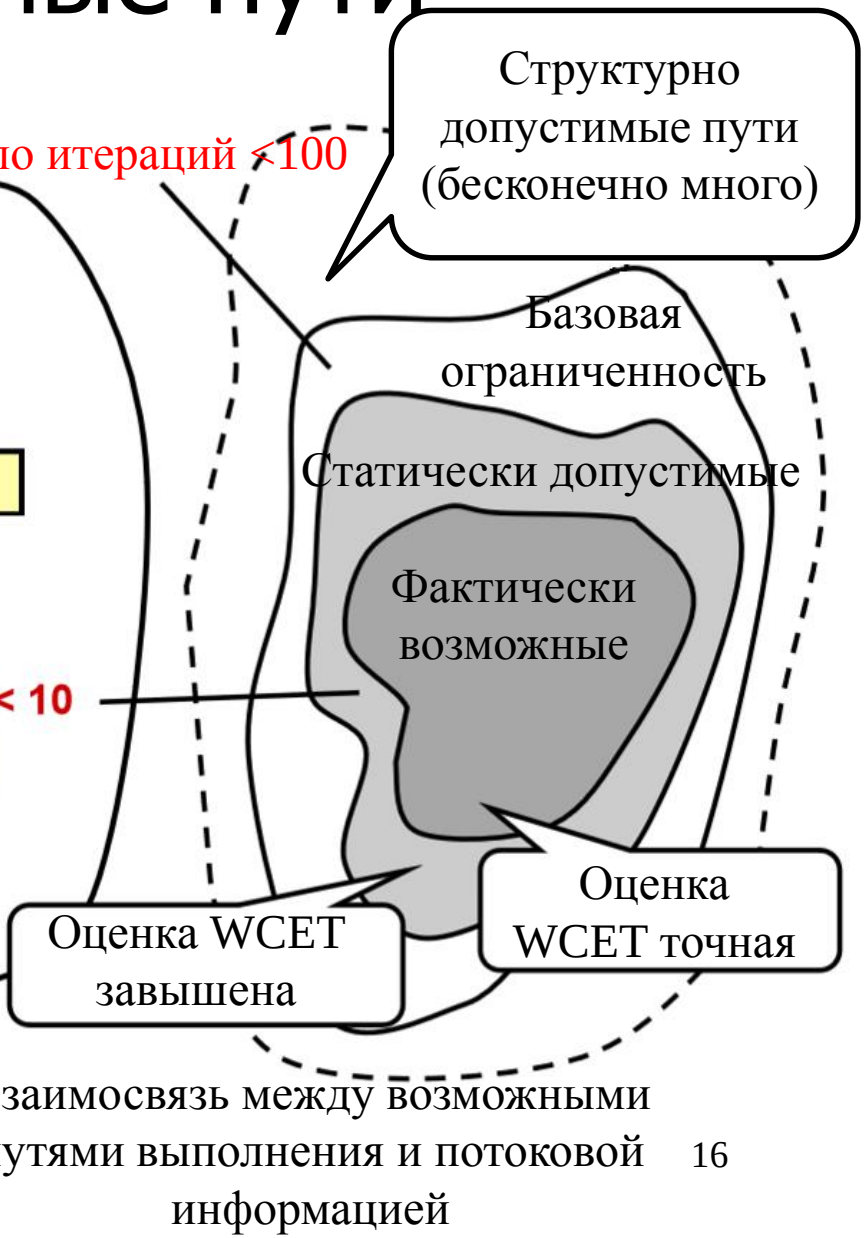
$-22 \le x \le 20$

```
foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
       else
D:       x = x+2;
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     i = i+1;
     end
```

Пример программы



Граф потока управления



Взаимосвязь между возможными путями выполнения и потоковой информацией

Пример: ограничение числа итераций

```
    foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
       else
D:       x = x+2;
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     i = i+1;
     end
```

Ограничение числа итераций:

- Зависит от возможных значений входной переменной I
  - например, если $1 \leq i \leq 100$, то число итераций не превышает 100
- В общем случае – очень сложная задача
- Имеет решение для многих частных случаев (типов циклов)

Требование базовой ограниченности:

- Для каждого цикла должно быть известно (вычислено или задано) ограничение на число итераций

Пример: недопустимый путь

```
    foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
       else
D:       x = x+2;
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     i=i+1;
     end
```

Недопустимый путь:

- Путь A-B-C-E-F-G не может быть выполнен, т.к. выполнение C исключает выполнение F
- Если (x > 5), то невозможно, чтобы (x * 2) < 0

Недопустимые пути исключаются из множества статически допустимых путей

- Это может уточнить оценку WCET

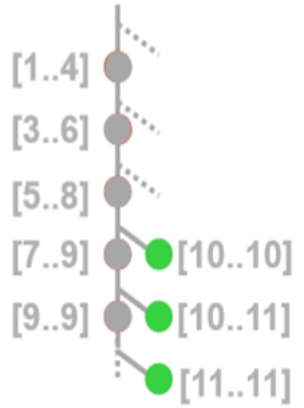
Абстрактная интерпретация

- Ограничивает число итераций циклов и выявляет недопустимые пути
  - Вычисляет безопасную (расширенную) оценку множества значений каждой переменной для различных точек выполнения программы
  - В ходе АИ, переменной сопоставляется не конкретное значение, а множество значений («абстрактное» значение)
- Программа «выполняется» с использованием абстрактных значений переменных
- Результат выполнения: безопасная (расширенная) оценка множества допустимых путей выполнения
  - Все фактически допустимые пути входят в полученное множество
  - Также в него могут входить некоторые фактически НЕдопустимые пути
  - Пути, не вошедшие в полученное множество, гарантированно не допустимы

Ограничение числа итераций цикла с помощью АИ

```
i = INPUT;
// i = [1..4]
while(i < 10) {
    //точка p
    ...
    i = i + 2;
}
//точка q
```

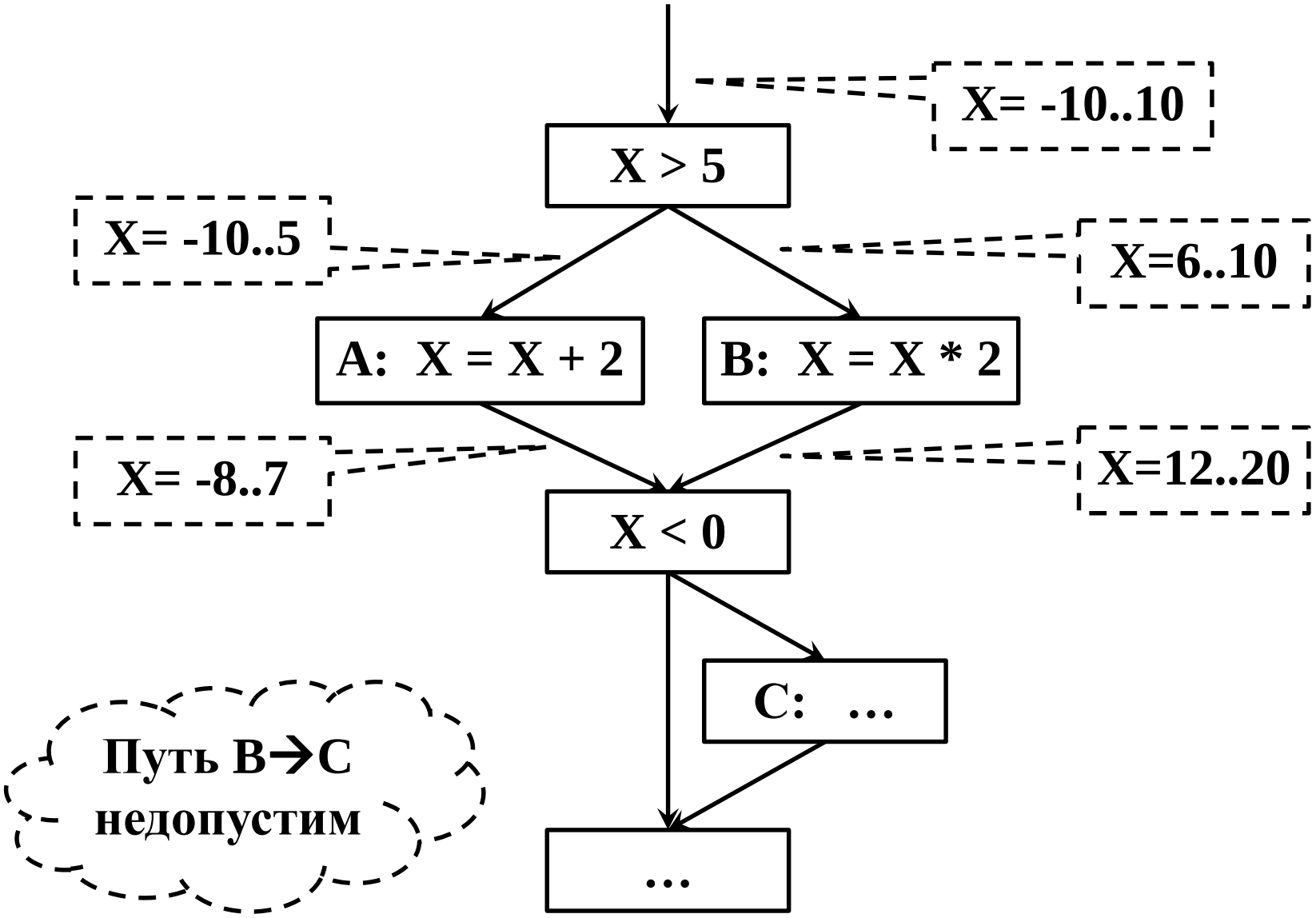
Итерация цикла	Абстрактное состояние в p	Абстрактное состояние в q
1	i = [1..4]	⊥
2	i = [3..6]	⊥
3	i = [5..8]	⊥
4	i = [7..9]	i = [10..10]
5	i = [9..9]	i = [10..11]
6	⊥	i = [11..11]



Результат:
Мин. число итераций: 3
Макс. число итераций: 5

- Анализируются все возможные пути выполнения цикла
- В точке *q* система может находиться в одном из трёх абстрактных состояний
  - Они могут быть объединены в одно абстрактное состояние

Выявление недопустимого пути с помощью АИ



Треугольный цикл

```
    triangle(a,b):
A:    loop(i=1..100)
B:     loop(j=i..100)
C:       a[i,j]=...
       end loop
     end loop
```

- Два вложенных цикла
  - Итераций в цикле A: не более 100
  - Итераций в цикле B: не более 100
- Число выполнений блока C:
  - Исходя из ограничений на число итераций циклов: 100 * 100 = 10 000
  - Фактически: 100 + … + 1 = 5 050

=> Аннотации кода

Виды аннотаций кода

Простейшая архитектура

- Сведения о частоте выполнения действий
  - Границы и соотношения для частот выполнения
  - Нотация: метки (marker), соотношения (relation), области (scope)

Сложная (реалистичная) архитектура

- Сведения о частоте выполнения последовательностей действий
  - Информация о (не)допустимых путях
  - Нотация: на основе регулярных выражений, например IDL (path Information Description Language)

Пример аннотации путей

```
for (i=0; i<N; i++)
{
    if (i % 3 == 0)
    {
        M1
    }
    if (i % 3 != 0)
    {
        M2
    }
}
```

N= 3*k

N= 3*k+1

N= 3*k+2

Выражение для пути
Сложная архитектура:

$(M1.M2.M2)^{\lfloor N/3 \rfloor} +$

$(M1.M2.M2)^{\lfloor N/3 \rfloor}.M1 +$

$(M1.M2.M2)^{\lfloor N/3 \rfloor}.M1.M2$

Простая архитектура:

$f(M1) = \lceil N/3 \rceil,$

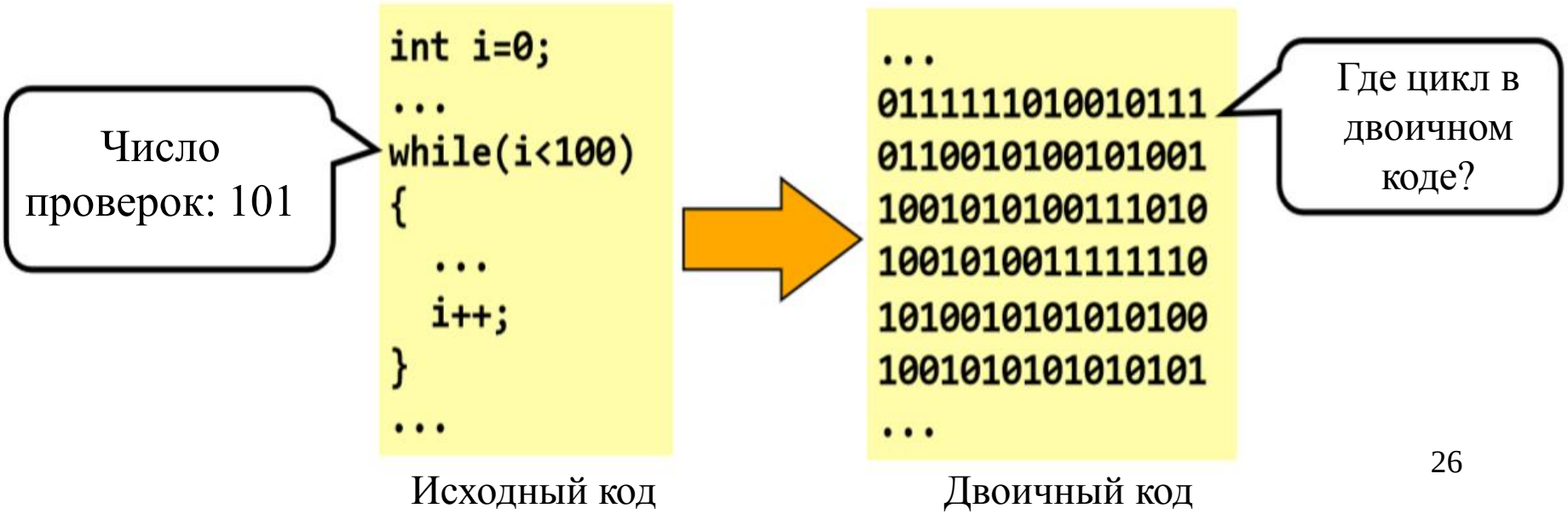
$f(M1) + f(M2) = N$

Маркеры, соотношения и области

```
SCOPE
{
    for (i=0; i<N; i++)
    {
        MAX_ITERATIONS(N);
        for (j=0; j<i; j++)
        {
            MAX_ITERATIONS(N);
            MARKER(M1);
                ...
        }
    }
    REL(FREQ(M1) == N * (N+1) / 2);
}
```

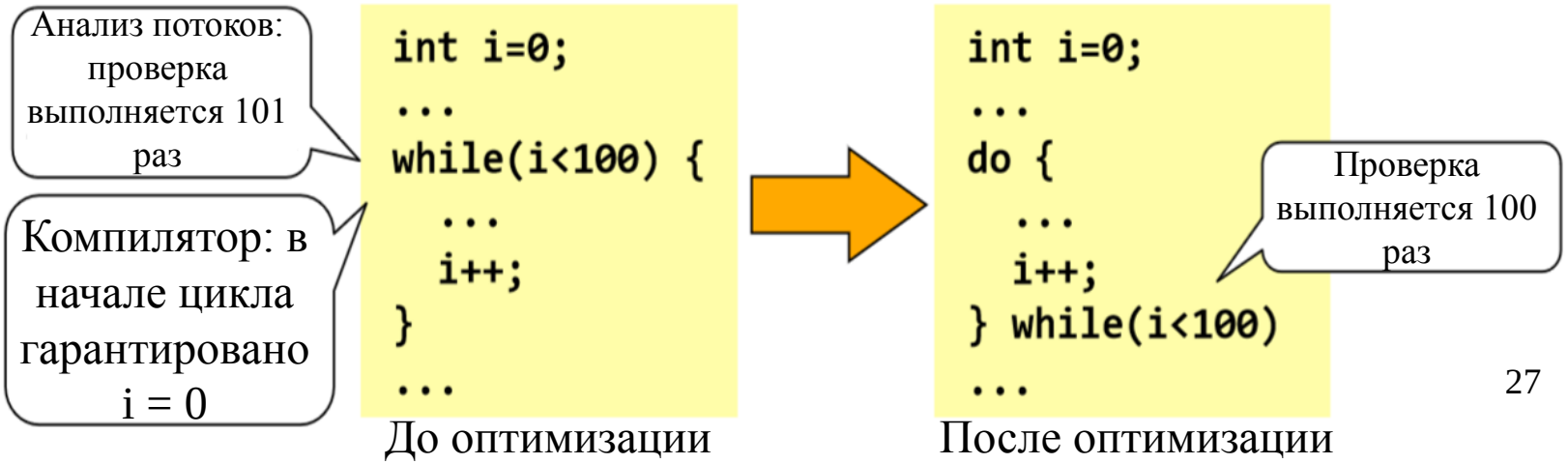
Проблема соответствия между исходным и двоичным кодом

- Анализ потоков проще проводить на уровне исходного кода
  - Более ясная семантика кода
  - Проще получить потоковую информацию (и для программиста, и для автоматических инструментов)
- Низкоуровневый анализ работает с двоичным кодом, фактически выполняемым на процессоре
- Как отобразить потоковую информацию с уровня исходного кода на двоичный код?

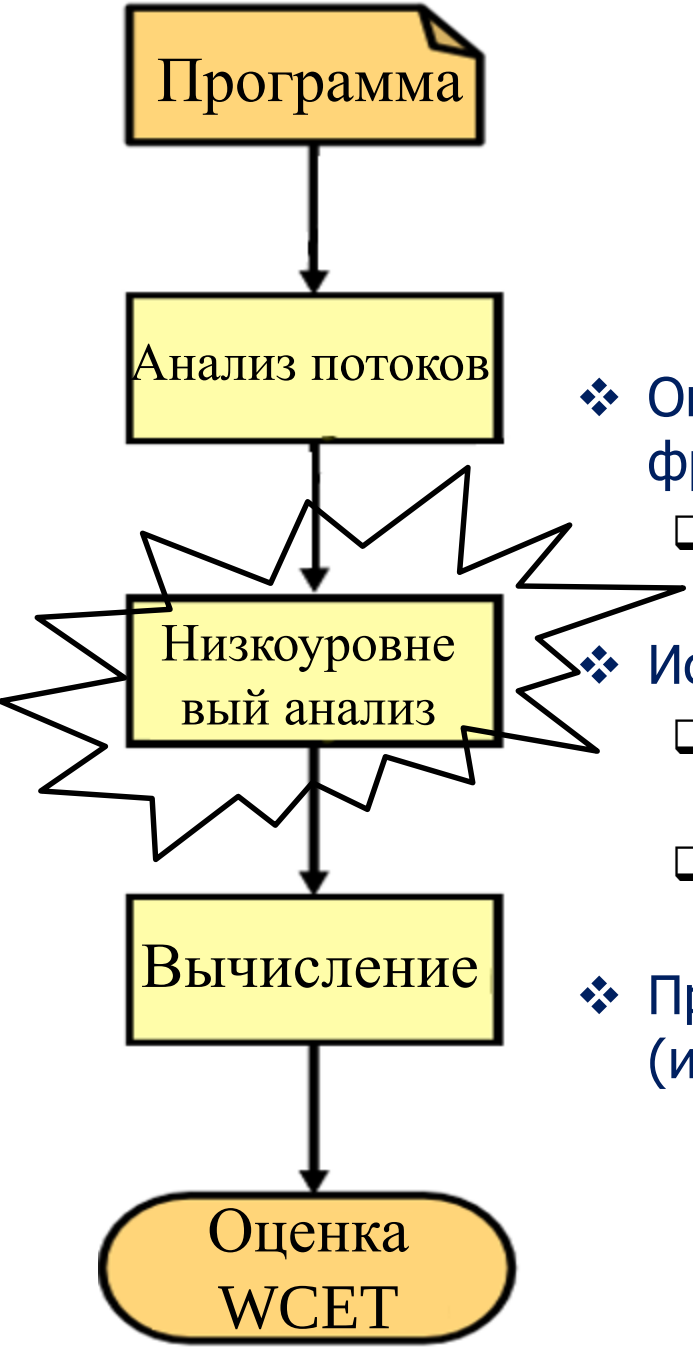


Проблема соответствия между исходным и двоичным кодом

- Компиляторы для встроенных систем реального времени выполняют глубокую оптимизацию кода
  - Нужно уложиться в ограничения по времени и объему памяти
- Оптимизации могут существенно изменить размещение кода и данных
  - В результате оптимизаций потоковая информация с уровня исходного кода становится неприменимой
- Решения:
  - Реализовать в компиляторе средства отображения потоковой информации (недостижимый идеал)
  - Использовать компиляцию с отладочной информацией (работает только при отсутствии или с минимумом оптимизаций)
  - Для систем с большим объемом памяти – не производить оптимизации кода с целью сокращения объема
  - Проводить потоковый анализ на двоичном коде (так чаще всего и делают)



Низкоуровневый анализ



- Ограничить время выполнения различных фрагментов программы
  - Основная задача большей части исследований по WCET
- Использовать модель целевой аппаратуры
  - Не требуется моделировать все подробности работы аппаратуры
  - При этом необходимо безопасно (сверху) оценить все задержки при работе аппаратуры
- Применяется к скомпонованному двоичному коду (исполняемой программе)

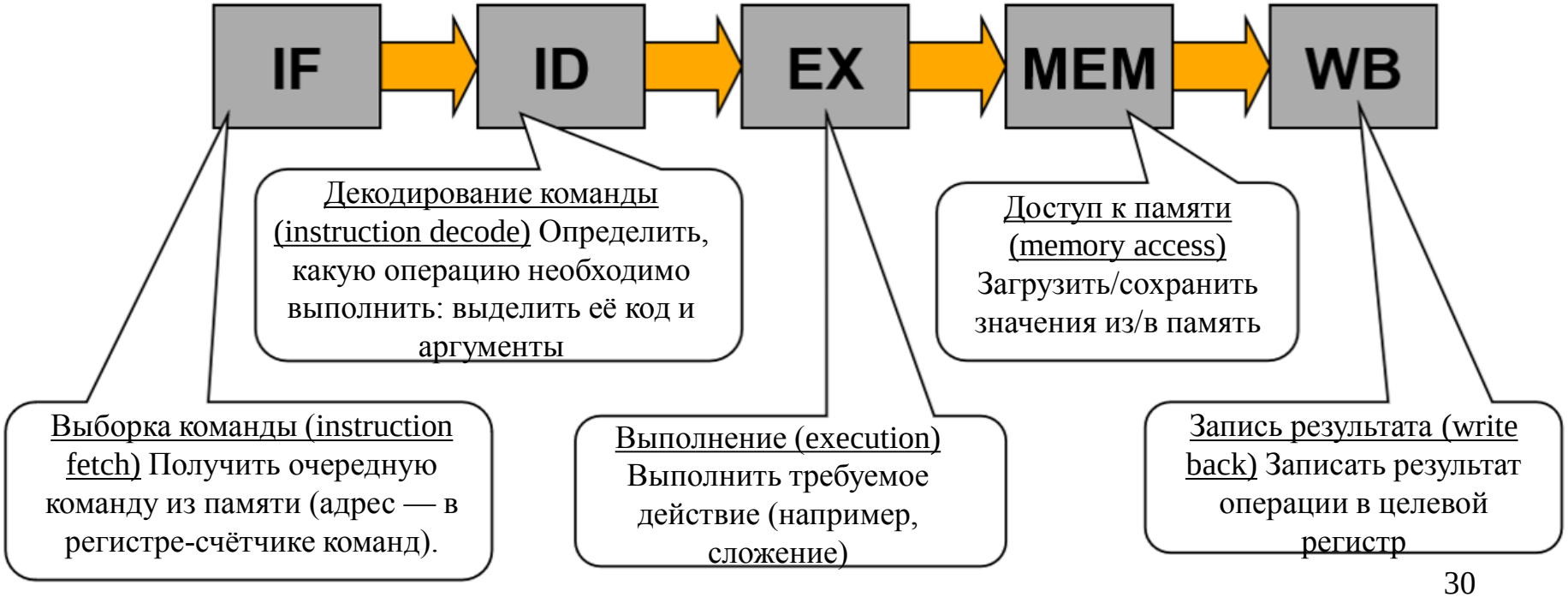
Проблемы моделирования аппаратуры

- Высокая сложность моделирования внутренней работы процессора
  - Конвейер, предсказание ветвлений, суперскалярность, внеочередное выполнение…
- Высокая сложность моделирования иерархической памяти
  - Необходимо детальное моделирование кэш-памяти
  - Другие виды памяти также являются источником задержек
- Многие аспекты функционирования сложных процессоров должны моделироваться совместно
  - Тайминги выполнения команд процессора зависят от предыстории
- Разработка безопасной временной модели функционирования процессора – сложная задача
  - Занимает месяцы и даже годы работы
  - Необходимо учесть все факторы, влияющие на время выполнения (как минимум, оценить сверху их влияние)

=> Аппаратура с предсказуемым временем работы важнее, чем очень быстрая аппаратура

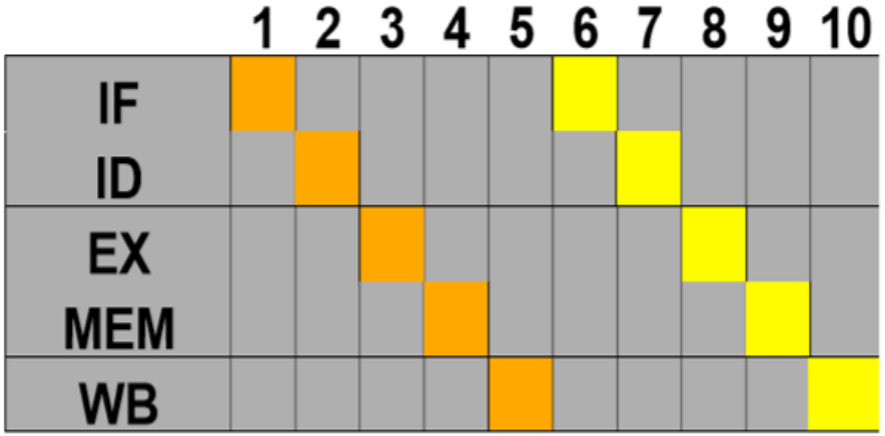
Простой конвейер

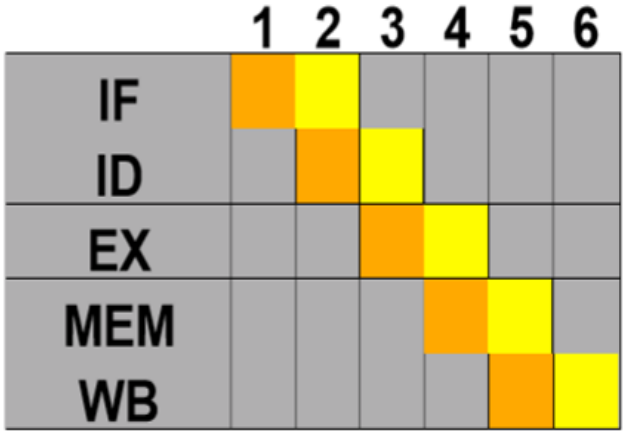
- Большинство команд проходят в процессоре через одни и те же этапы выполнения
- Пример: классический 5-ступенчатый конвейер RISC-процессора



Работа конвейера

- Параллельная работа различных стадий конвейера для различных команд (=> ускорение)
- Нет конвейера:
  - следующая команда не может начать выполнение, пока текущая команда не завершит последнюю стадию выполнения
- Конвейер:
  - В идеале: коэффициент ускорения равен числу ступеней конвейера
  - Фактически: между командами есть зависимости по данным; «слом» конвейера при неверном предсказании ветвления; ожидание данных из памяти





Непосредственная зависимость по данным

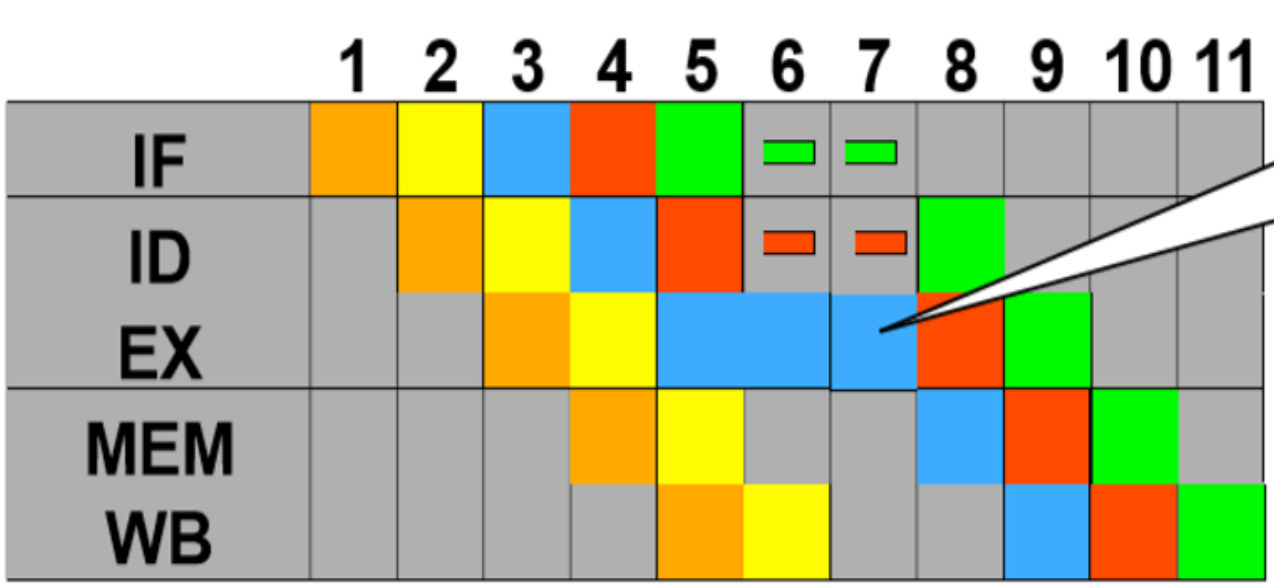
```
I1. add $r0, $r1, $r2
I2. sub $r3, $r0, $r4
```

I2 ждёт вычисления результата I1

Может возникнуть задержка в конвейере

Виды конвейеров

- Отсутствует: простейшие процессоры (68HC11, 8051, …)
- Скалярный: единственный конвейер (ARM7, ARM9, V850, …)
- VLIW: несколько конвейеров, статическое планирование их загрузки на уровне компилятора (DSPs, Itanium, Crusoe, …)
- Суперскалярный: несколько конвейеров, внеочередное выполнение команд (PowerPC 7xx, Pentium, UltraSPARC, …)



Голубая команда находится на этапе EX два дополнительных такта

Это приводит к задержке выполнения всех последующих команд

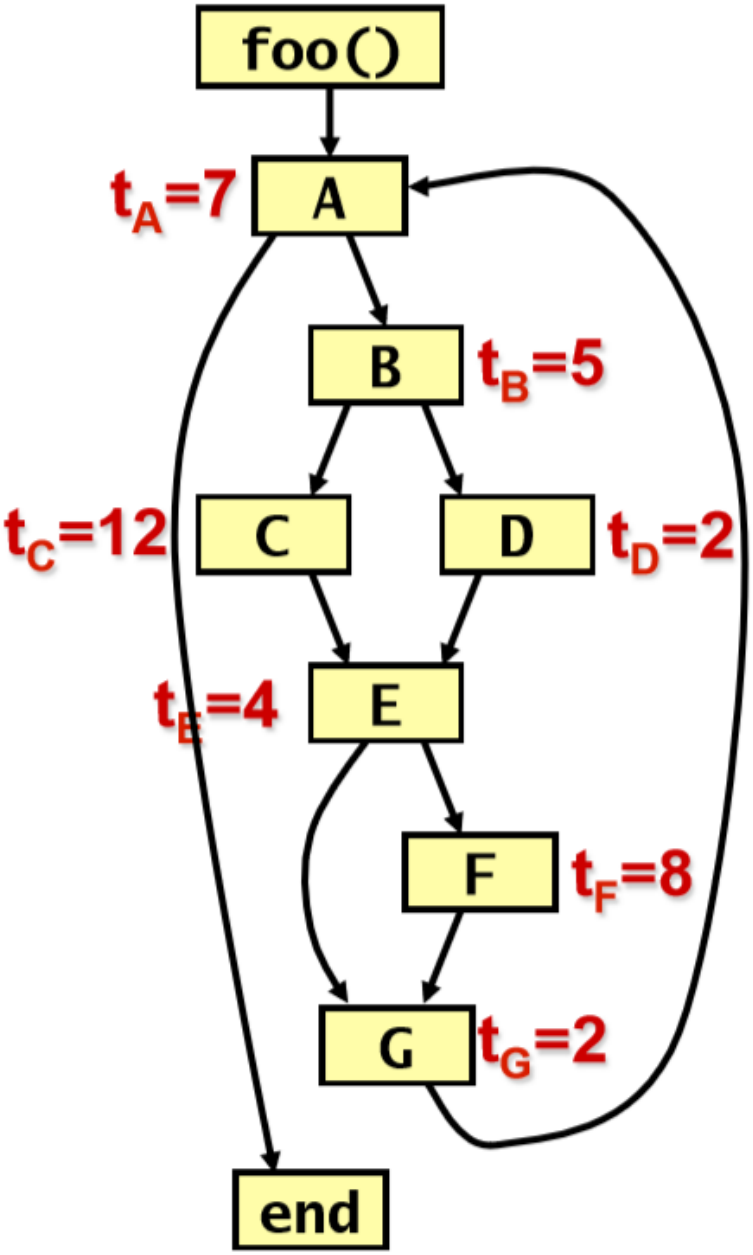
Задержки: нет конвейера

```
    foo(x,i):
A:   while(i < 100)      (7 cycles)
B:     if (x > 5) then   (5 c)
C:       x = x*2;        (12 c)
       else
D:       x = x+2;        (2 c)
       end
E:     if (x < 0) then   (4 c)
F:       b[i] = a[i];    (8 c)
       end
G:     i = i+1;          (2 c)
     end
```

- Фиксированное время для каждого фрагмента кода
- Двоичный код не показан

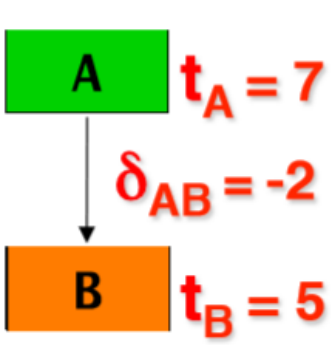
Задержки: нет конвейера

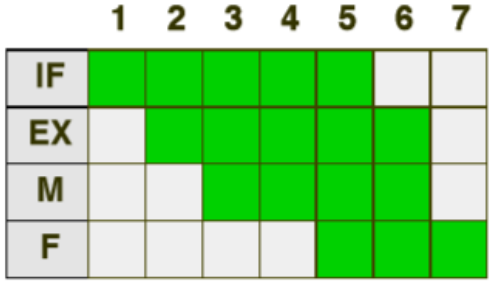
```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```

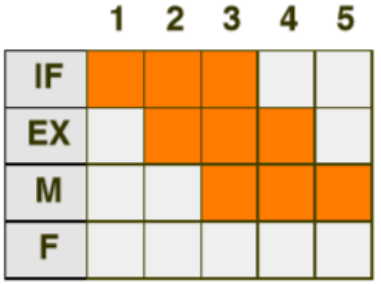


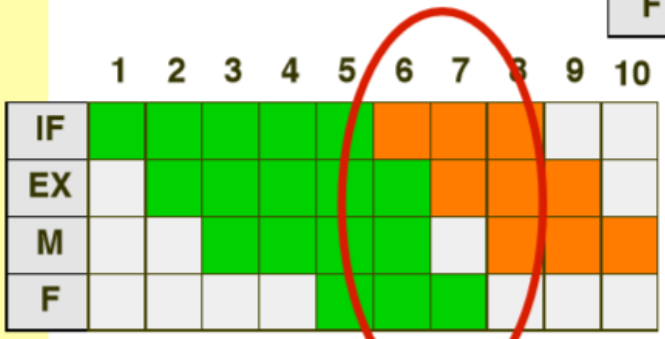
Задержки: простой конвейер

```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```







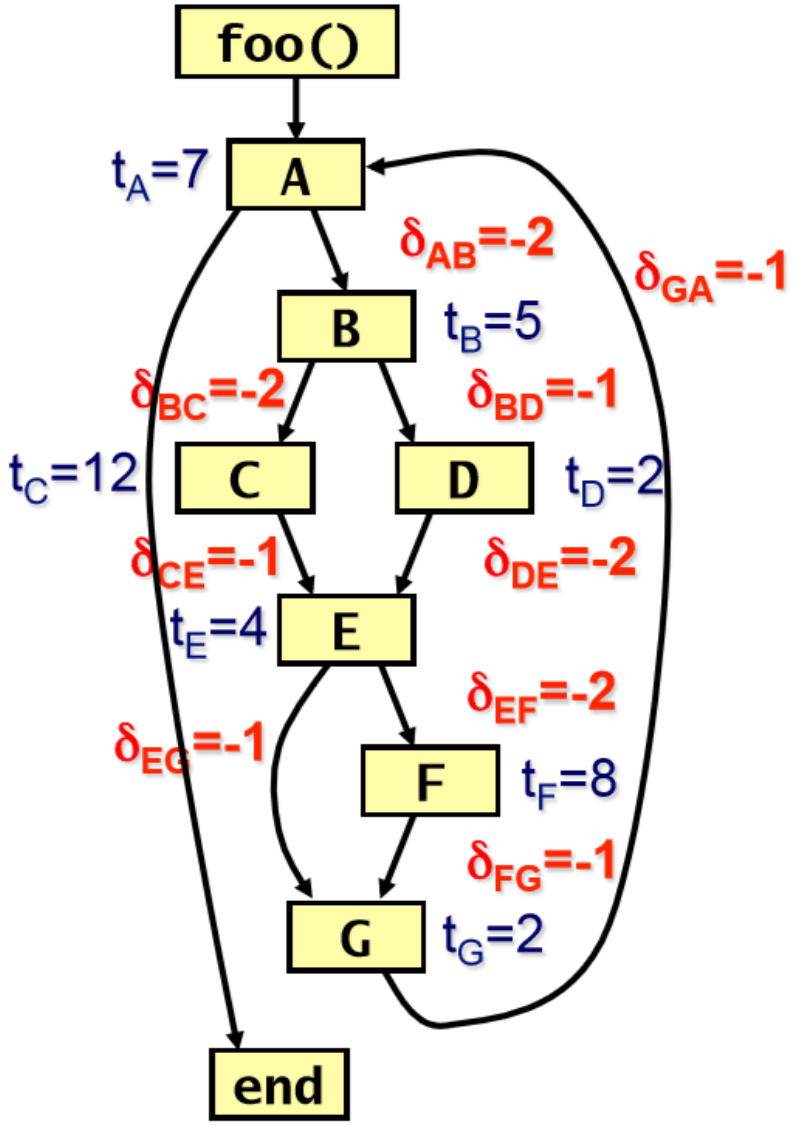


$t_{AB} = 10$

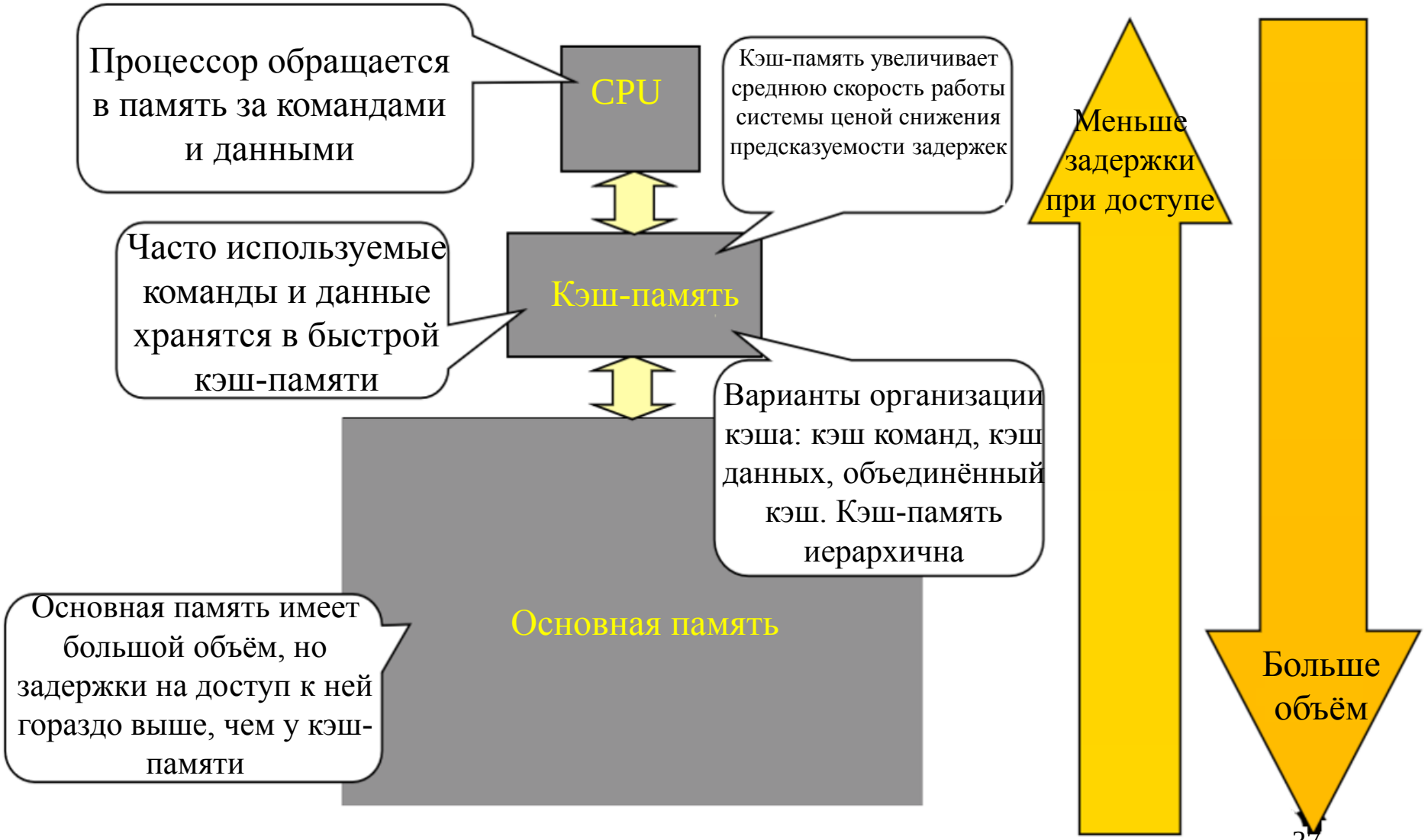
$\delta_{AB} = 10 - (7 + 5) = -2$

Задержки: простой конвейер

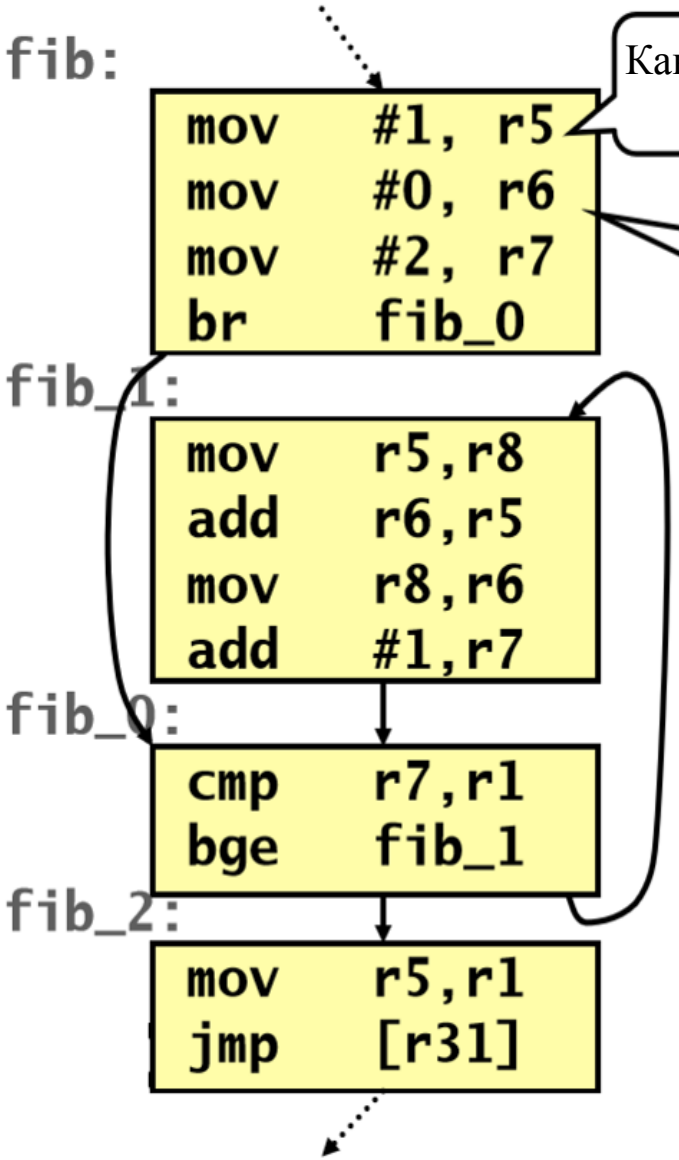
```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```



Иерархическая память

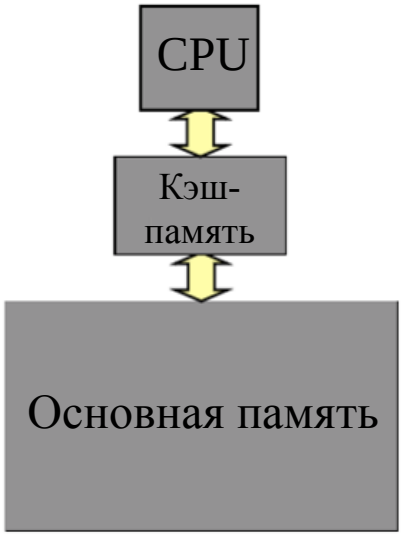


Анализ влияния кэш-памяти



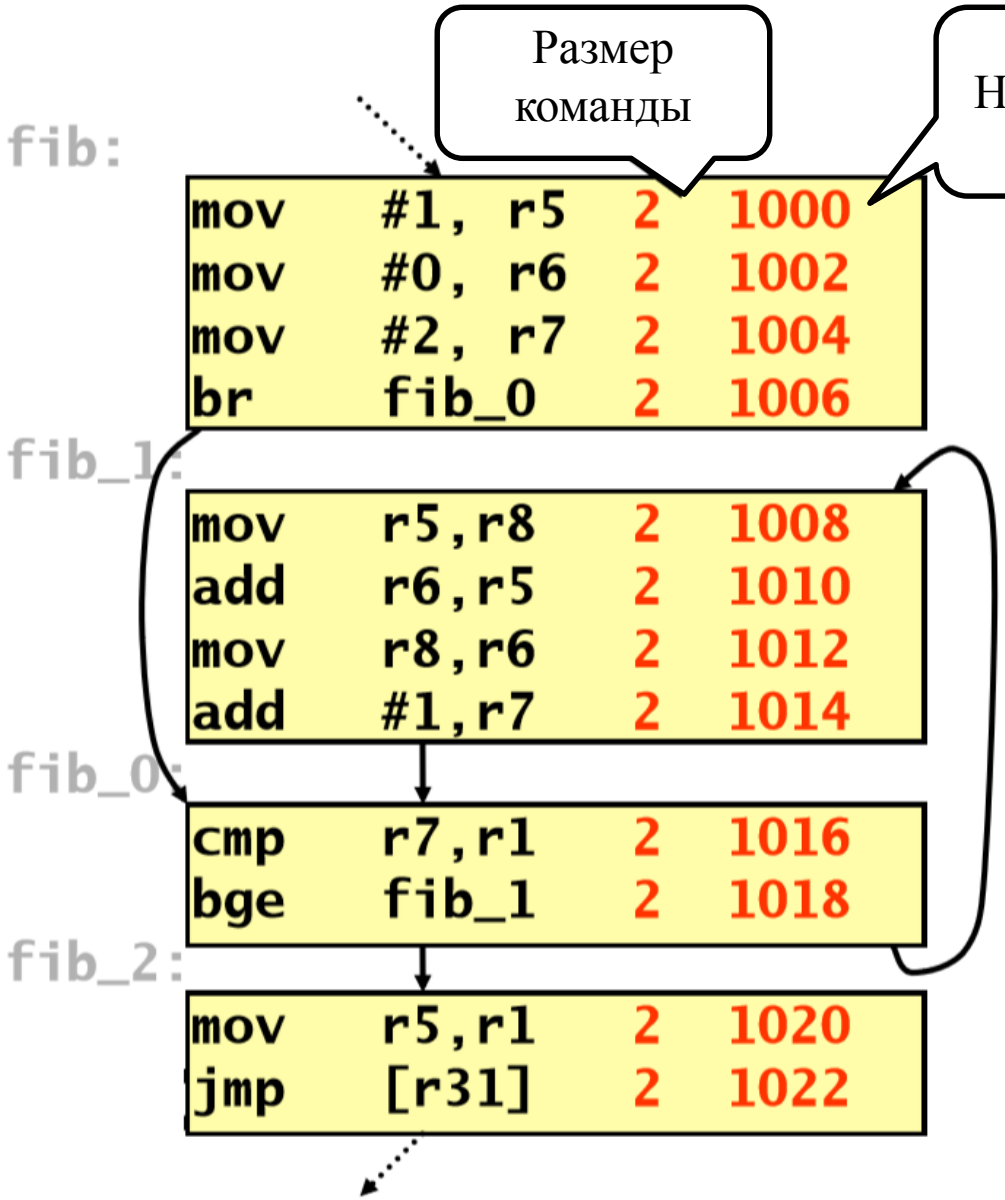
Какие команды приведут к промахам в кэш?

Промахи в кэш приводят к значительно большим задержкам, чем попадания



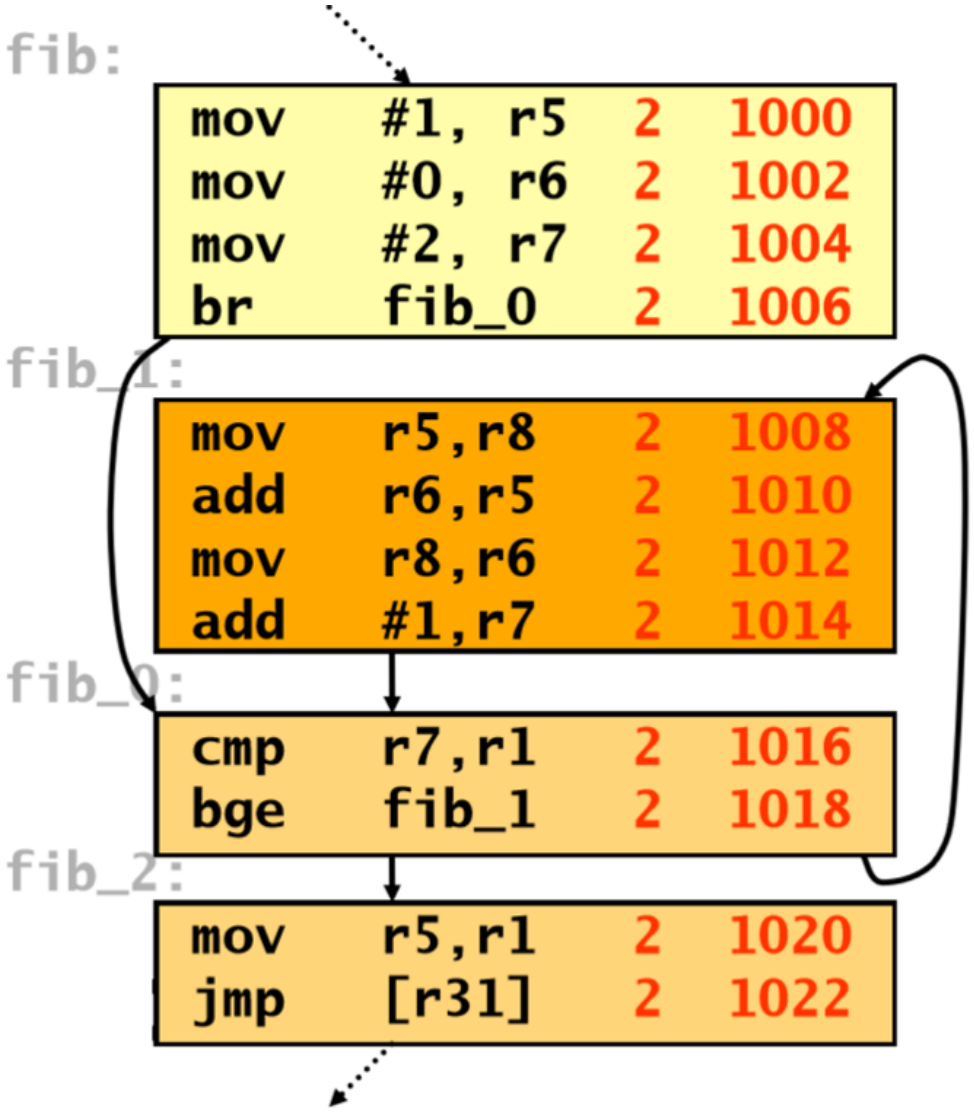
- Анализируется двоичный код
- В данном примере – только кэш с прямым отображением (каждому адресу в основной памяти соответствует единственный адрес в кэш-памяти)

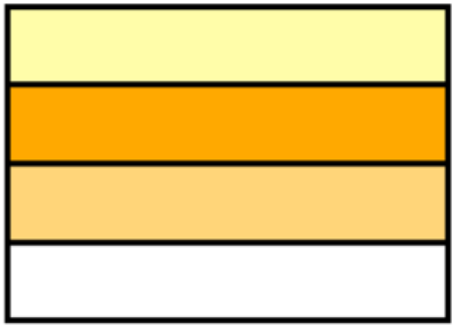
Анализ влияния кэш-памяти



- Информация для анализа влияния кэша команд

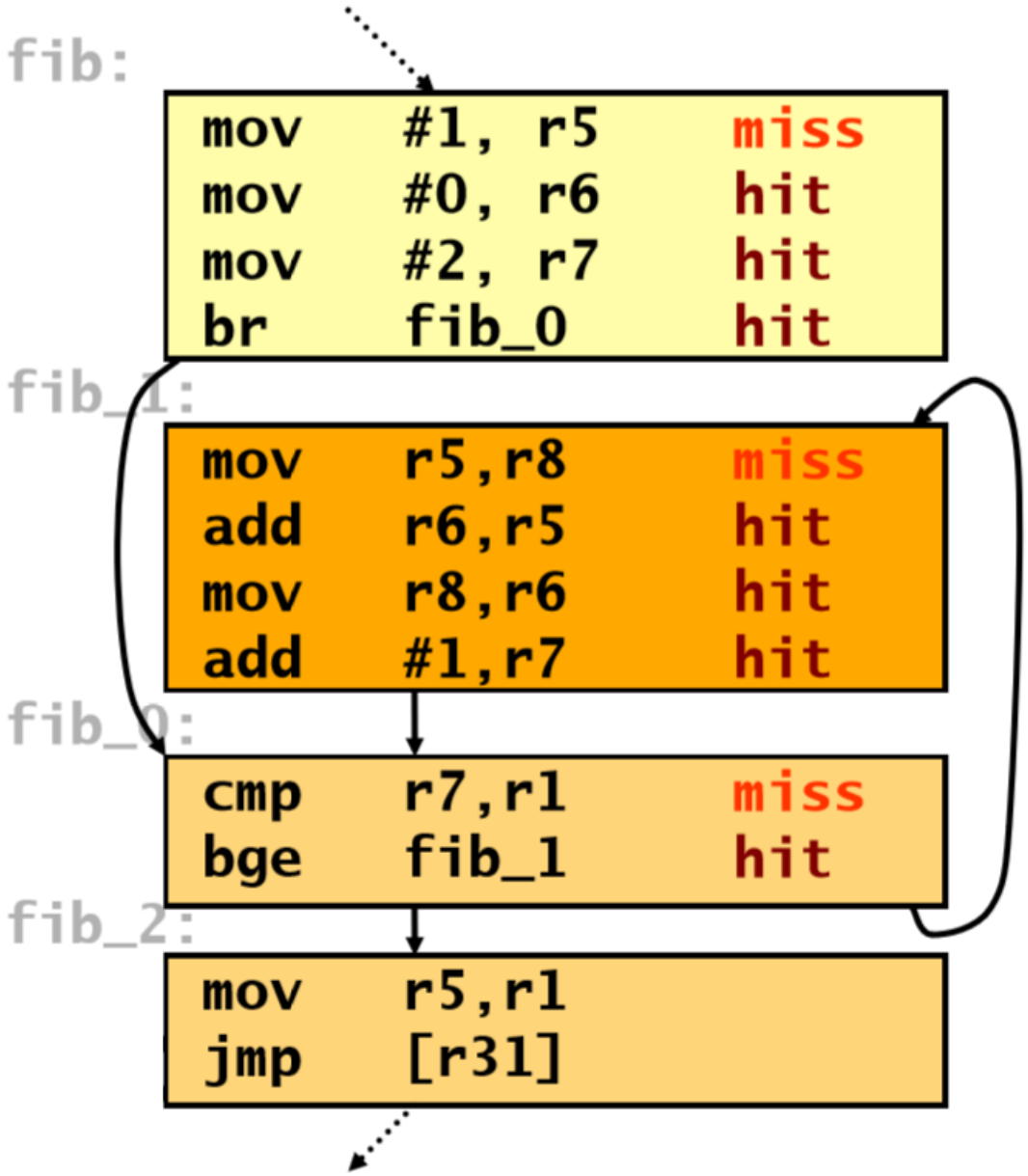
Анализ влияния кэш-памяти

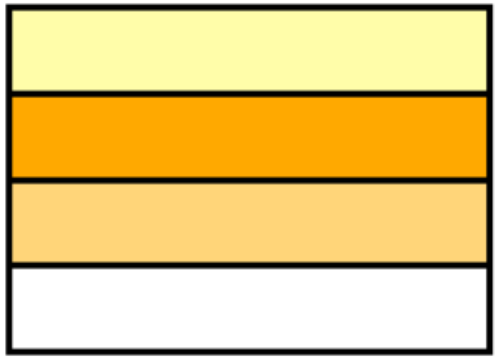




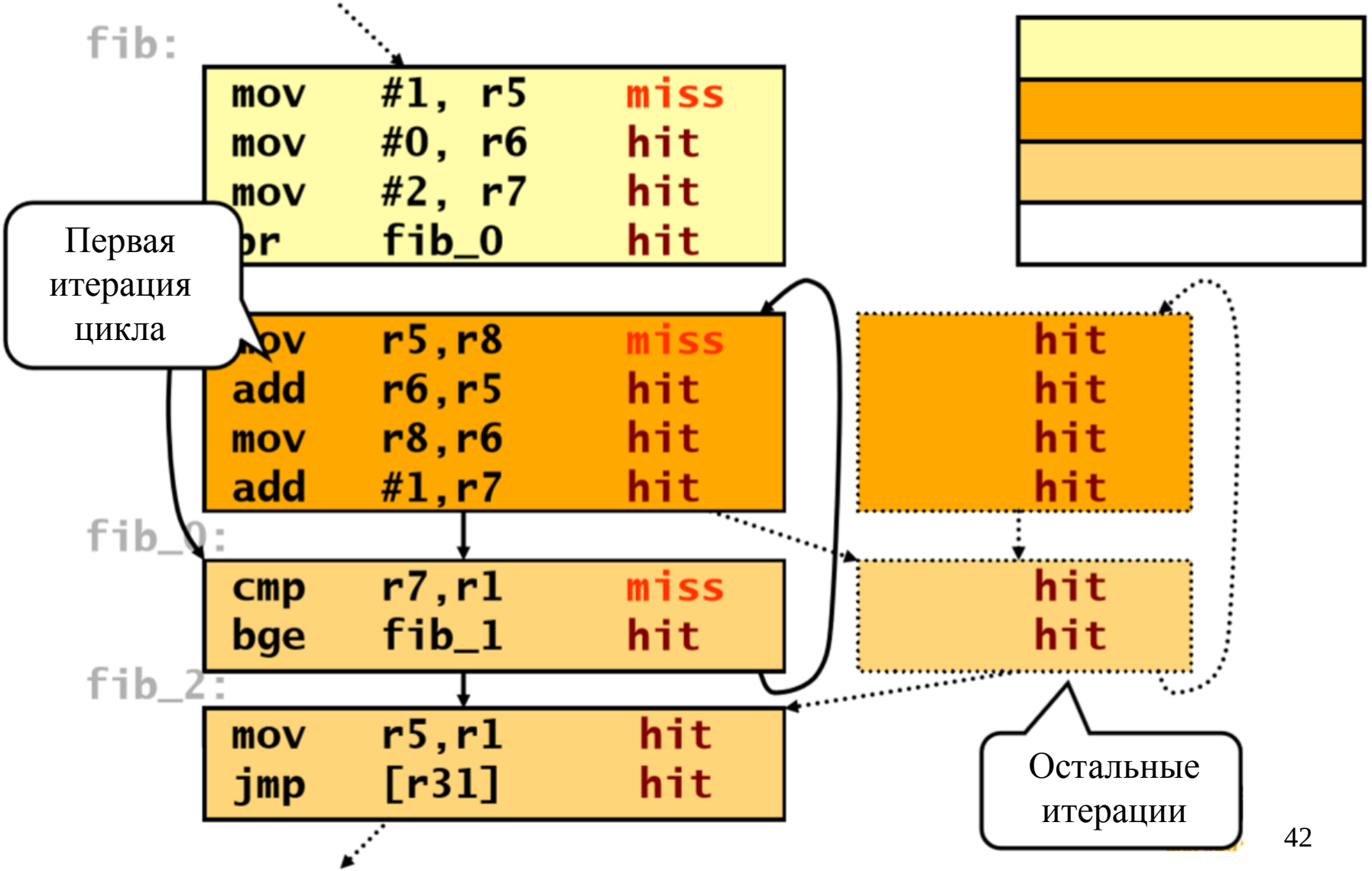
- Отображение в кэш команд

Анализ влияния кэш-памяти



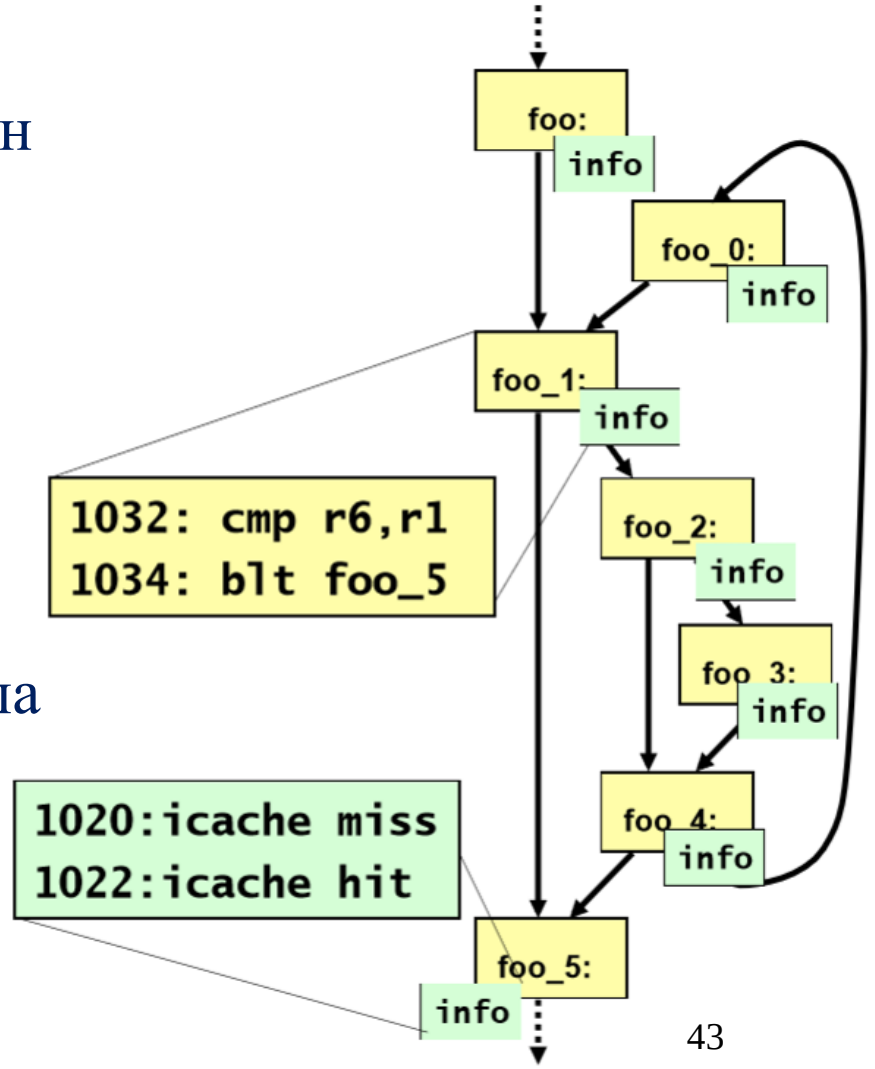


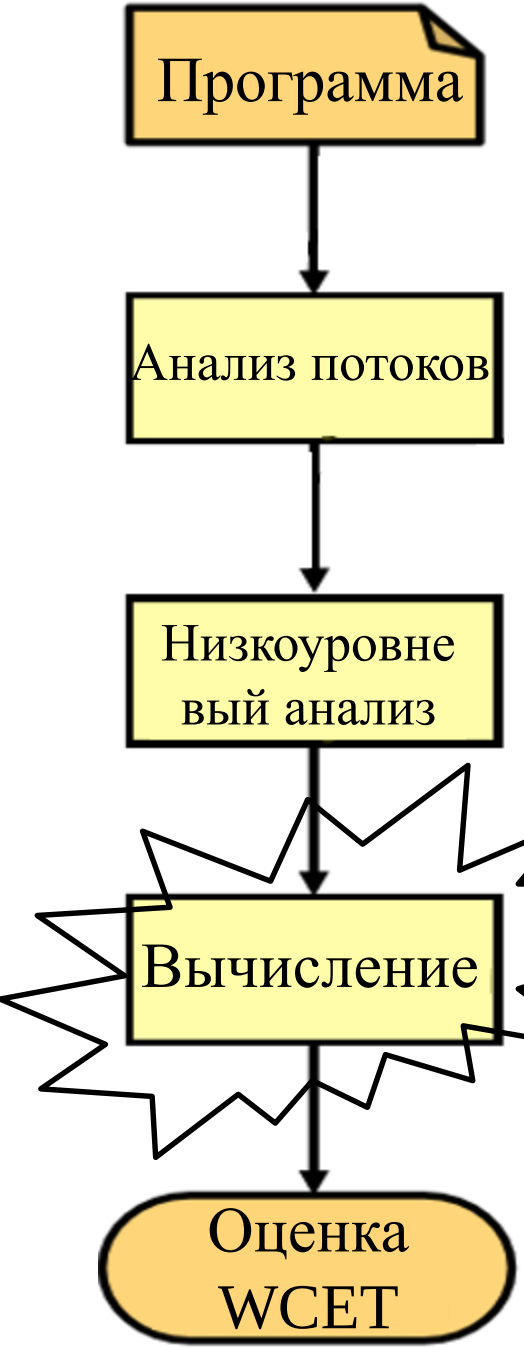
Анализ влияния кэш-памяти



Учет совместного влияния кэша и конвейера

- Анализ влияния конвейера должен брать на вход результаты анализа влияния кэш-памяти
  - Команды помечаются попаданием/промахом в кэш
  - Попадания/промахи влияют на задержки в конвейере
- Сложная аппаратура требует совместного анализа влияния кэша и конвейера





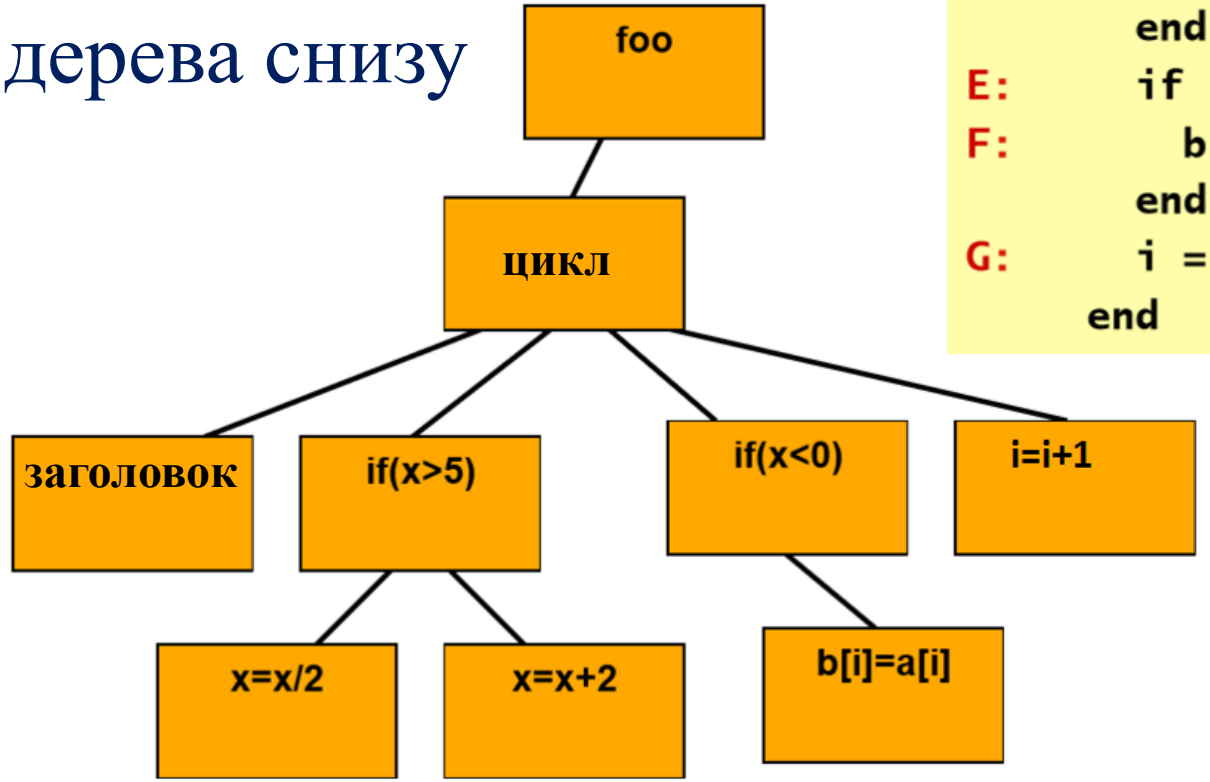
Вычисление WCET

- Вычислить верхнюю оценку WCET программы
  - Исходные данные: информация о задержках и потоковая информация
- Примеры подходов:
  - Расчёт по синтаксическому дереву
  - Расчёт по путям выполнения
  - Неявный перебор путей (IPET)

Расчёт WCET по дереву

- Анализируется синтаксическое дерево программы
- Обход дерева снизу вверх

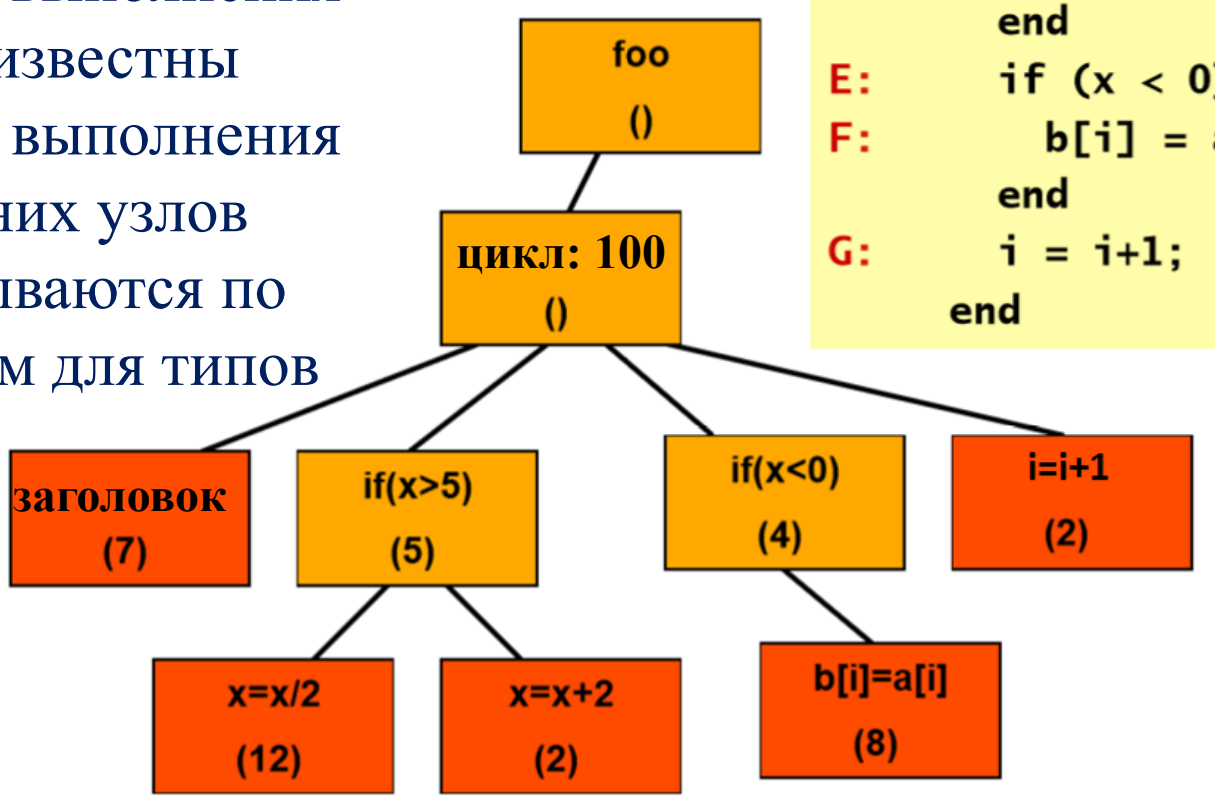
```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```



Расчёт WCET по дереву

- Фиксированные времена выполнения узлов
- Времена выполнения листьев известны
- Времена выполнения внутренних узлов рассчитываются по формулам для типов узлов

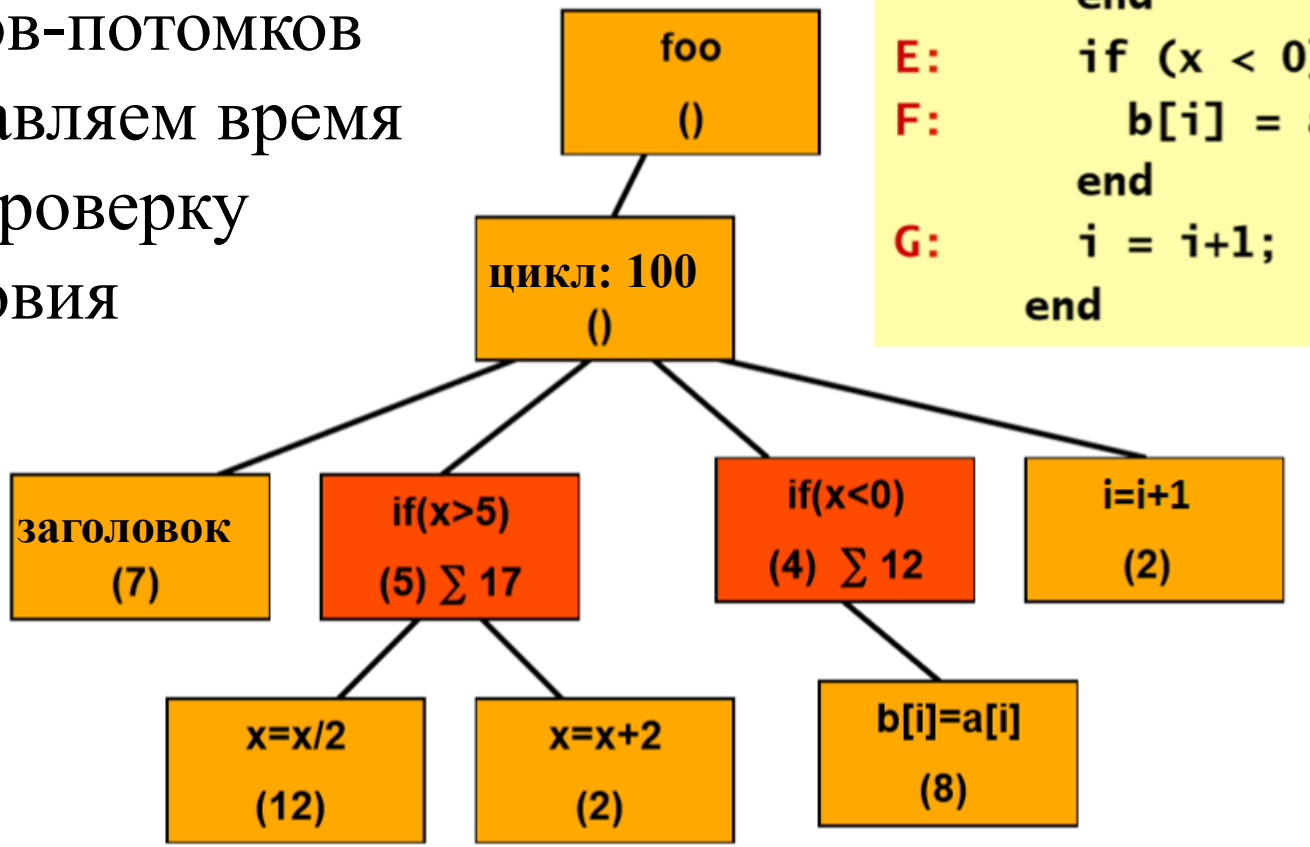
```
     foo(x,i):
A:    while(i < 100)        (7 c)
B:      if (x > 5) then     (5 c)
C:        x = x*2;          (12 c)
        else
D:        x = x+2;          (2 c)
        end
E:      if (x < 0) then     (4 c)
F:        b[i] = a[i];      (8 c)
        end
G:      i = i+1;            (2 c)
      end
```



Правило для оператора ветвления

- Ветвление:
  - берем максимум из значений для узлов-потомков
  - добавляем время на проверку условия

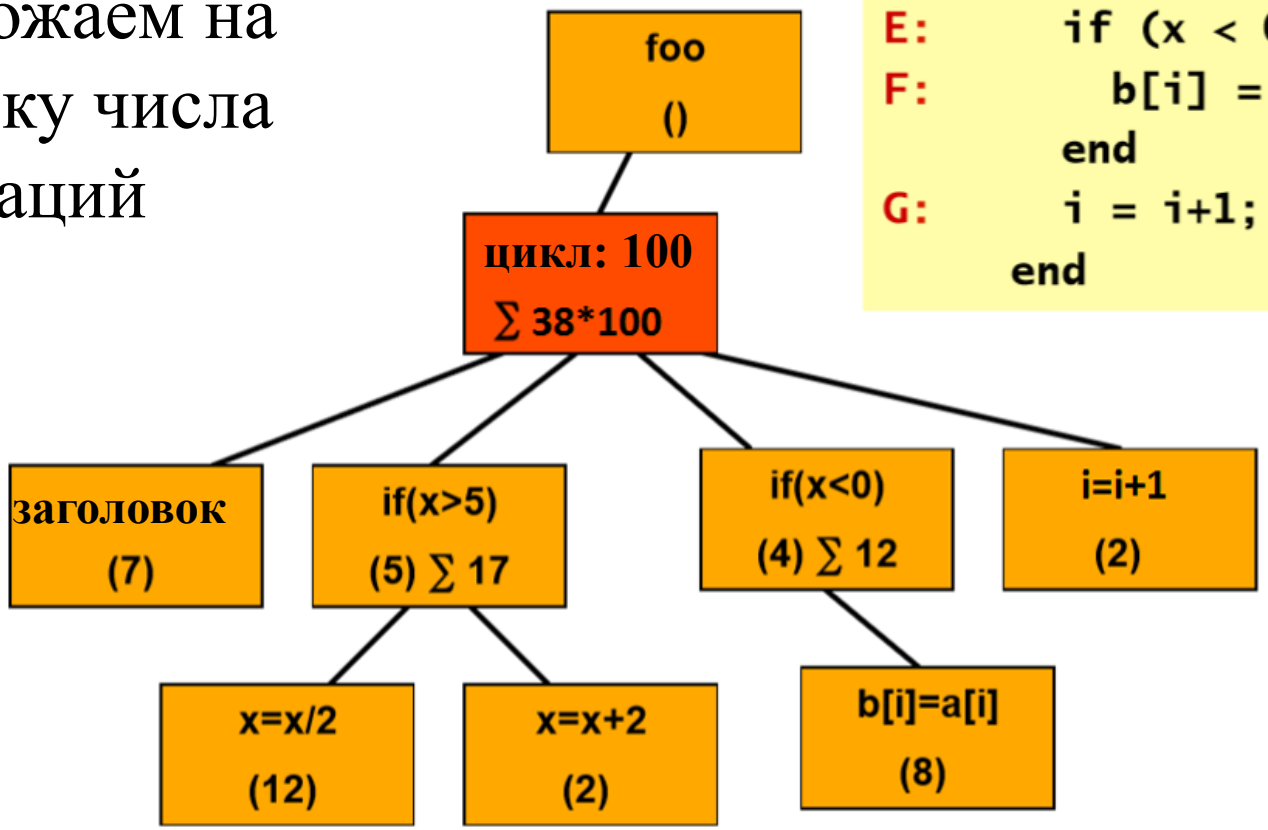
```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```



Правило для оператора цикла

- Цикл:
  1. Суммируем оценки для узлов-потомков
  2. Умножаем на оценку числа итераций

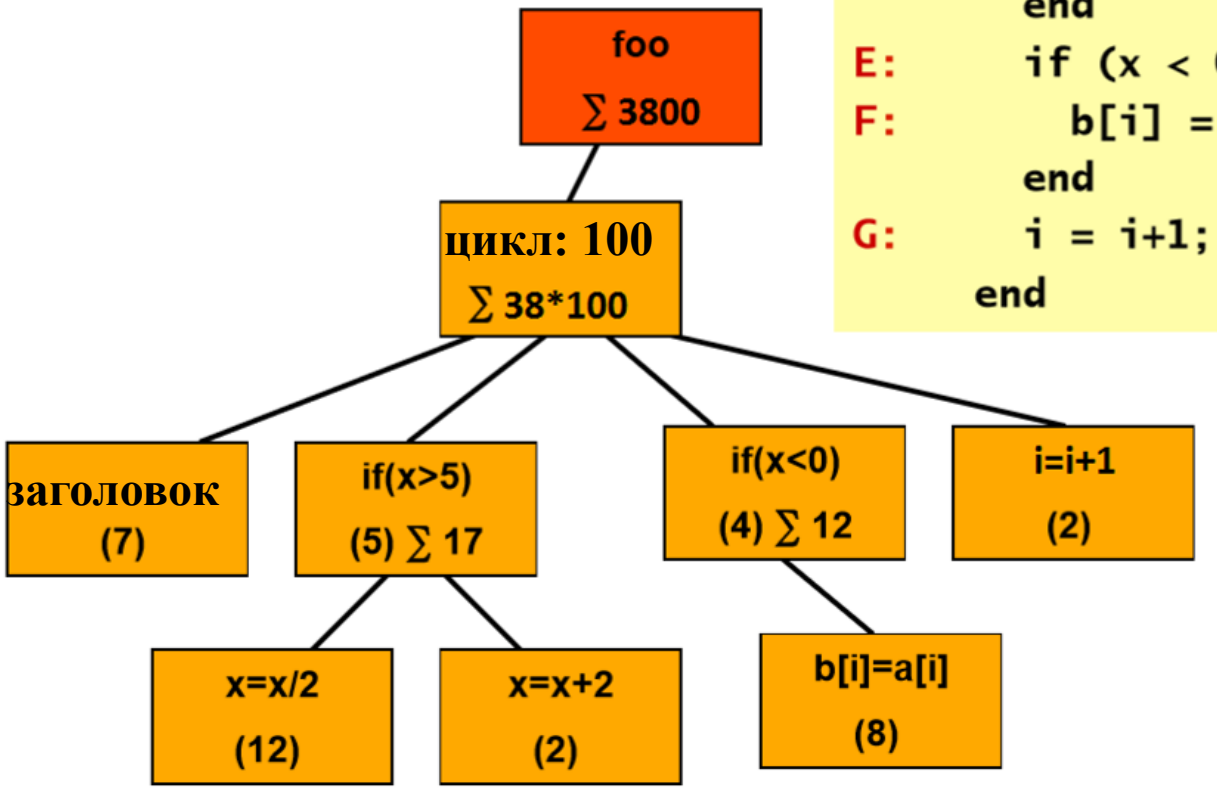
```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```



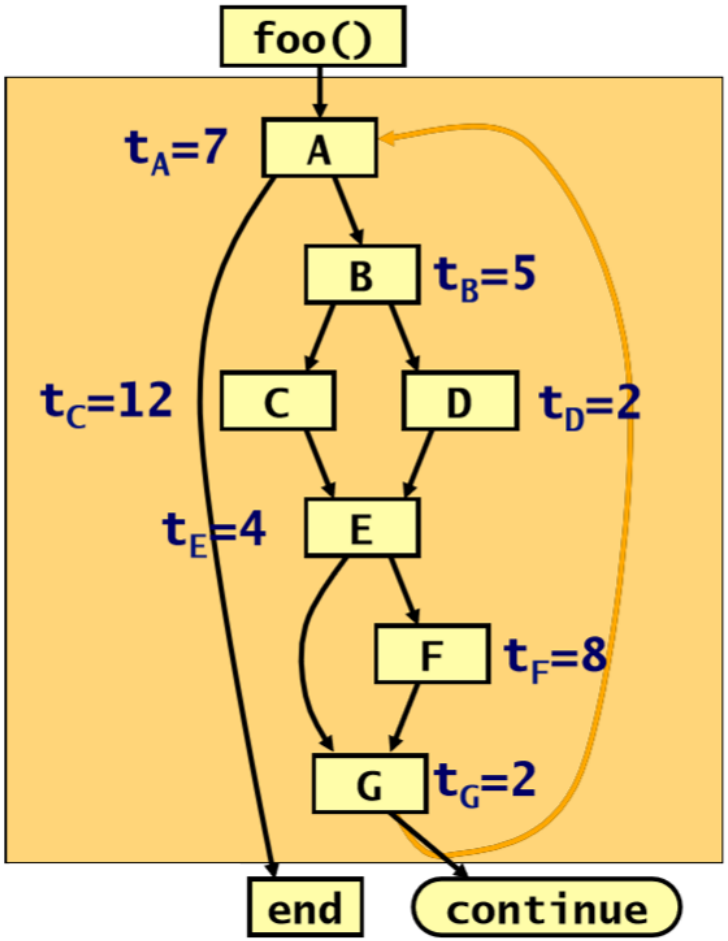
Результат расчёта по дереву

- WCET функции foo() равен 3800 тактам

```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```



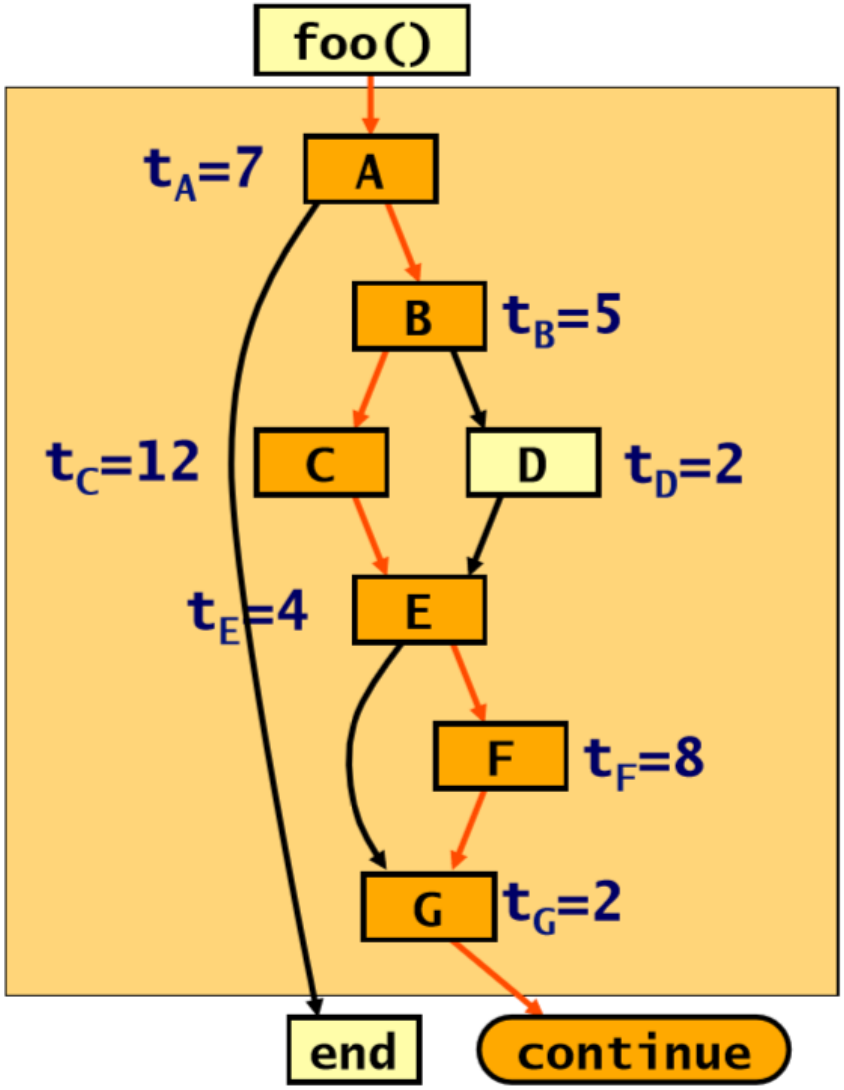
Расчёт WCET по путям




```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```

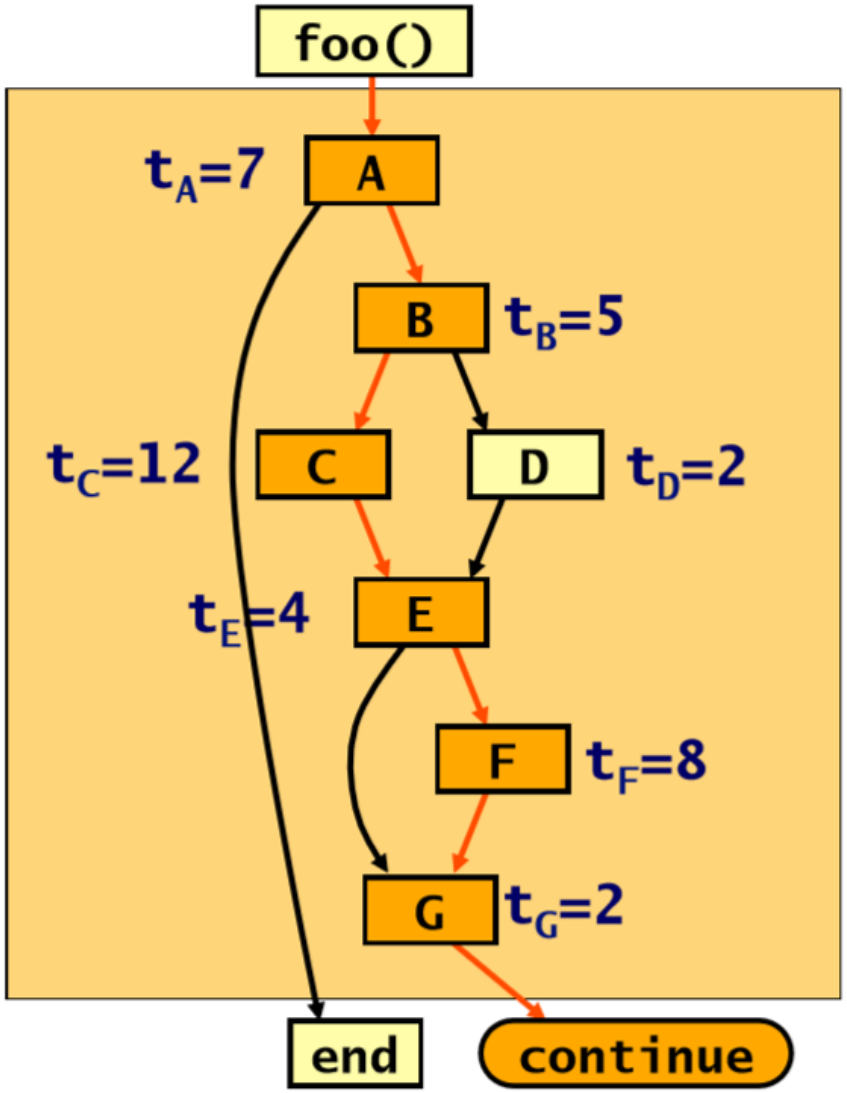
- Найти самый длинный путь
  - Рассматриваем итерации цикла по одной
- Подготовить цикл
  - Убрать обратные дуги
  - Перенаправить их на специальные узлы «continue»

Расчёт WCET по путям



- Самый длинный путь:
  - A-B-C-E-F-G
  - 7 + 5 + 12 + 4 + 8 + 2 = 38 тактов
- Суммарное время:
  - 100 итераций
  - 38 тактов на итерацию
  - Итого: 3800 тактов

Расчёт WCET по путям

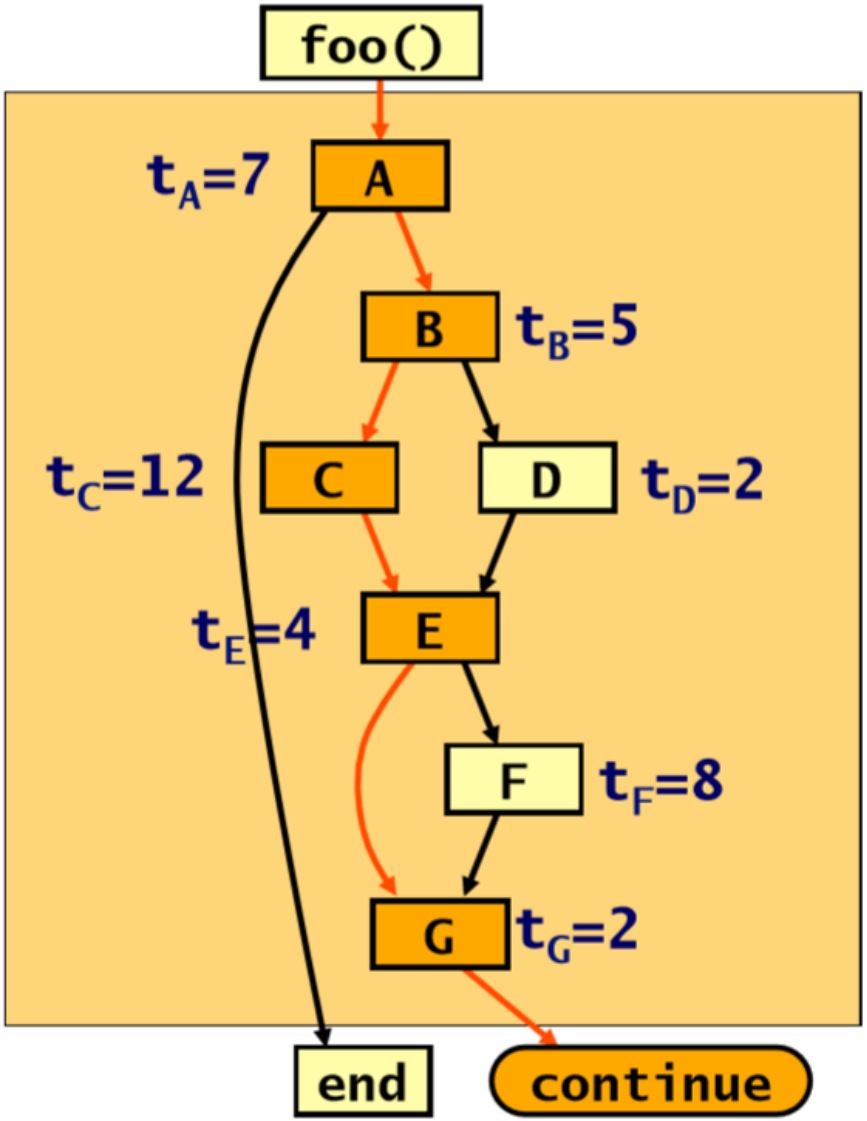



```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```

C и F никогда не выполняются совместно

- Недопустимый путь:
  - A-B-C-E-F-G
  - Отбрасываем, ищем следующий по длине

Расчёт WCET по путям



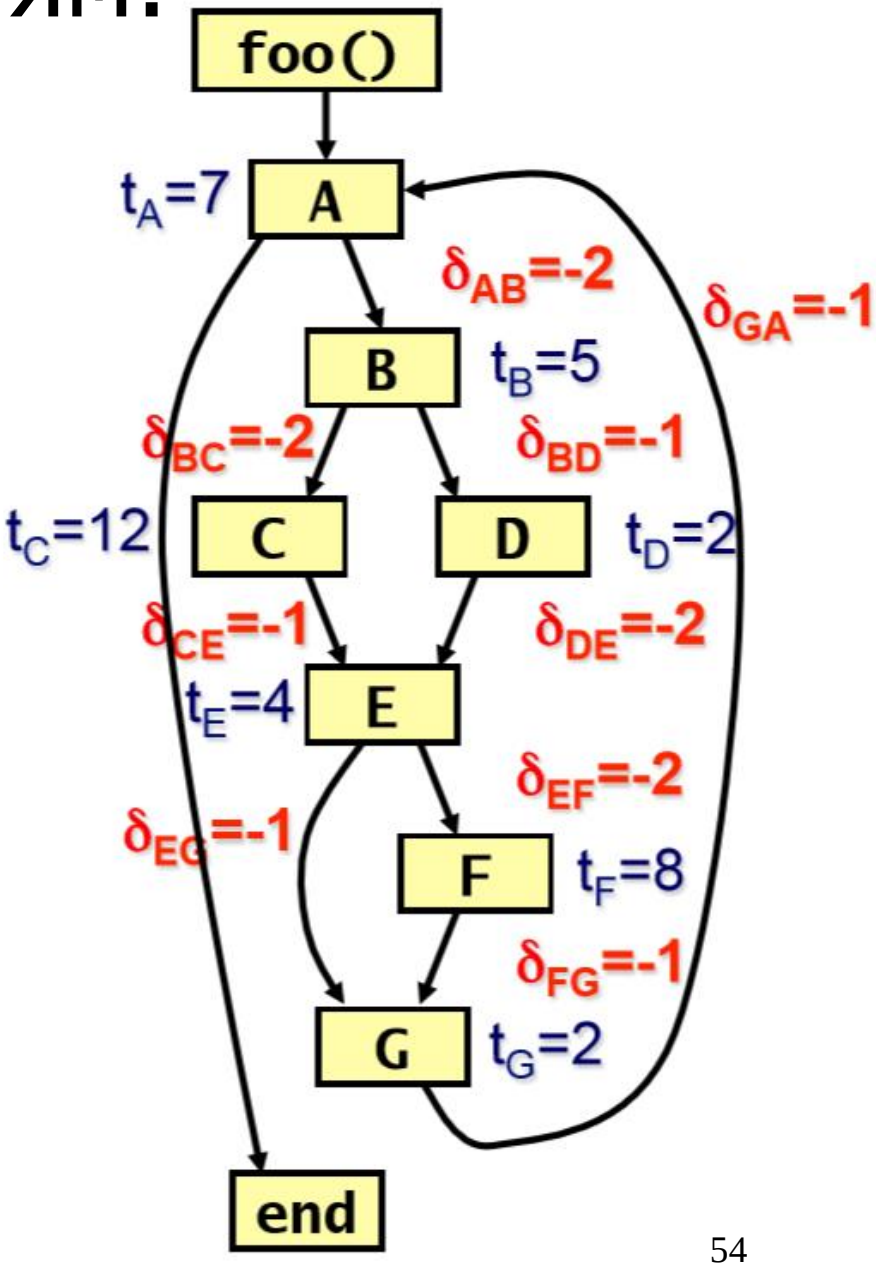
C и F никогда не выполняются совместно

```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```

- Недопустимый путь:
  - A-B-C-E-F-G
  - Отбрасываем, ищем следующий по длине
- Новый самый длинный путь:
  - A-B-C-E-G
  - 30 тактов
- Итого: 3000 тактов

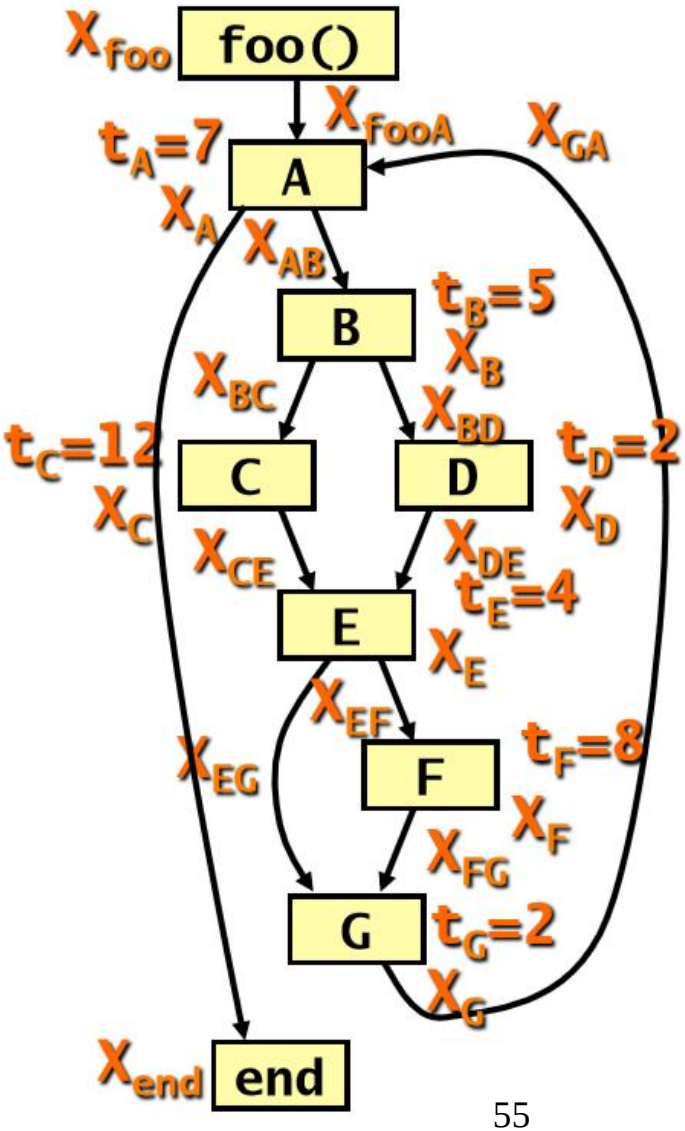
Расчёт WCET по путям: учёт конвейера

- Упорядочиваем допустимые пути по убыванию грубой оценки WCET (сумма оценок WCET для участков путей)
- для *x* из {допустимые_пути}
  - Вычислить $WCET_{PL}(x)$ с учётом «экономии» δ_{XY} от конвейерного выполнения последовательных участков
  - Если $WCET_{PL}(x)$ больше, чем наибольшая из грубых оценок WCET для оставшихся путей, или если других путей не осталось, то x – самый длинный (наихудший) путь; стоп
    иначе продолжить цикл



Неявный перебор путей

- Implicit path enumeration technique (IPET)
  - Пути выполнения не обрабатываются в явном виде
- Представление программы
  - Информация о задержках (t_{entity})
    - Значения в узлах: выполнение участков
    - Значение на дугах: экономия за счёт конвейера
  - Число выполнений (X_{entity})



Неявный перебор путей

$$*WCET = \max \sum(X_{entity} * t_{entity})$$

где совокупность X_{entity} удовлетворяет ограничениям:

- начальные и конечные условия
- структура программы
- ограничения на число итераций
- прочая потоковая информация

$X_{foo}=1$

$X_{end}=1$

$X_A=X_{fooA}+X_{GA}$

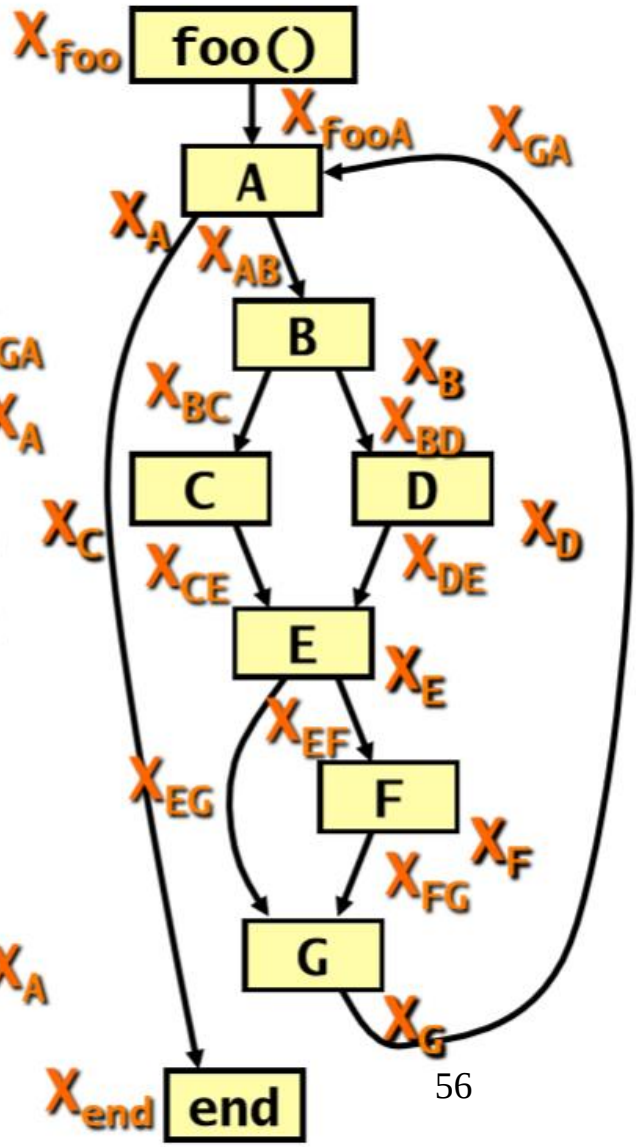
$X_{AB}+X_{Aend}=X_A$

$X_{BC}+X_{BD}=X_B$

$X_E=X_{CE}+X_{DE}$

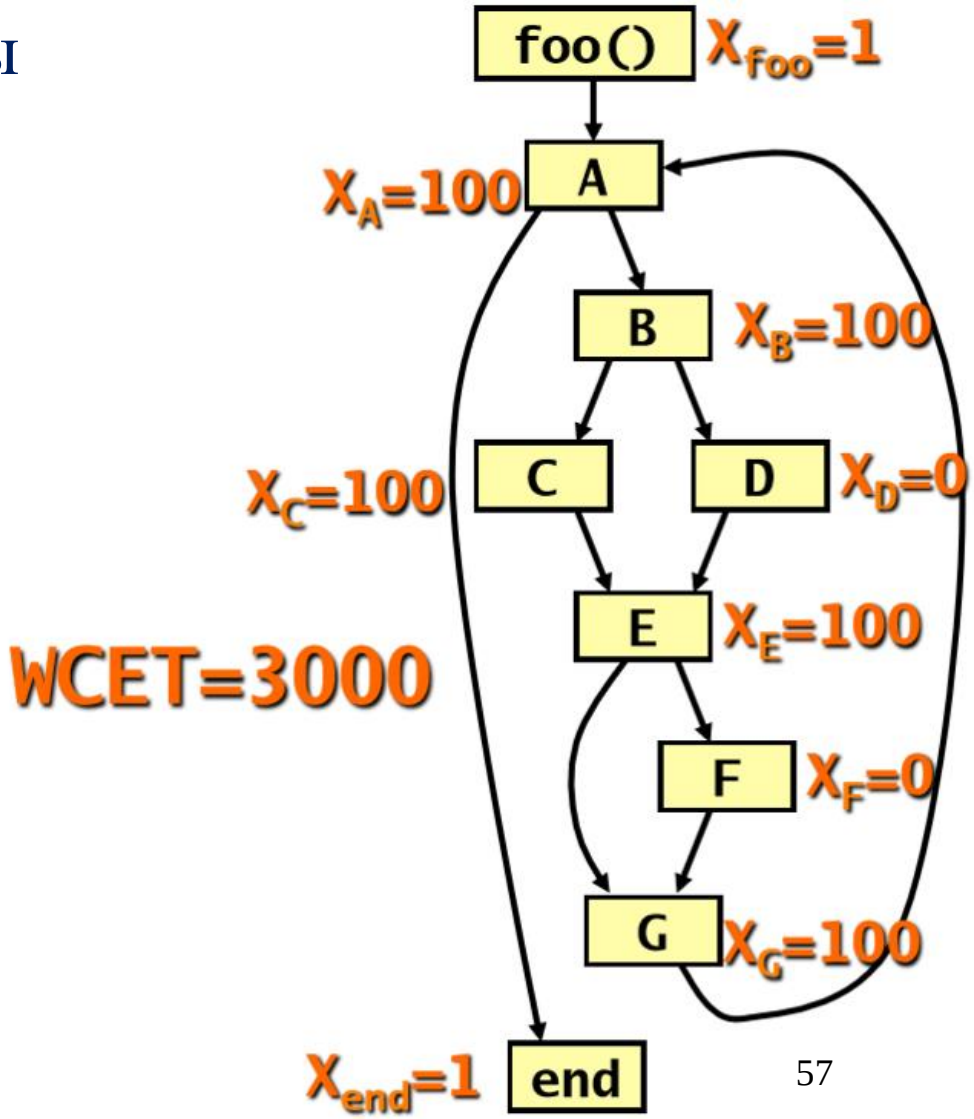
$X_A<=100$

$X_C+X_F<=X_A$



Неявный перебор путей

- Методы решения системы ограничений:
  - Целочисленное линейное программирование
  - Разрешение ограничений (constraint satisfaction)
- Результат
  - Число выполнений для узлов и дуг
  - Оценка WCET



Спасибо за внимание!