

Branch prediction:

или откуда процессор берет
производительность (Часть 1)

Что такое Branch

Branch (Ветвление или Команда перехода) — команда процессора, которая нарушает непрерывную последовательность исполнения команд, вынуждая выбирать и исполнять последующие команды с произвольно заданного адреса.

Инструкции перехода - **15-25%**

https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_%D0%BF%D0%B5%D1%80%D0%B5%D1%85%D0%BE%D0%B4%D0%B0

Branch Predictor Unit (BPU)

Branch Predictor Unit (BPU) - блок процессора, который говорит процессору при выборке инструкций, какие инструкции выбирать дальше, если встретилась инструкция ветвления.

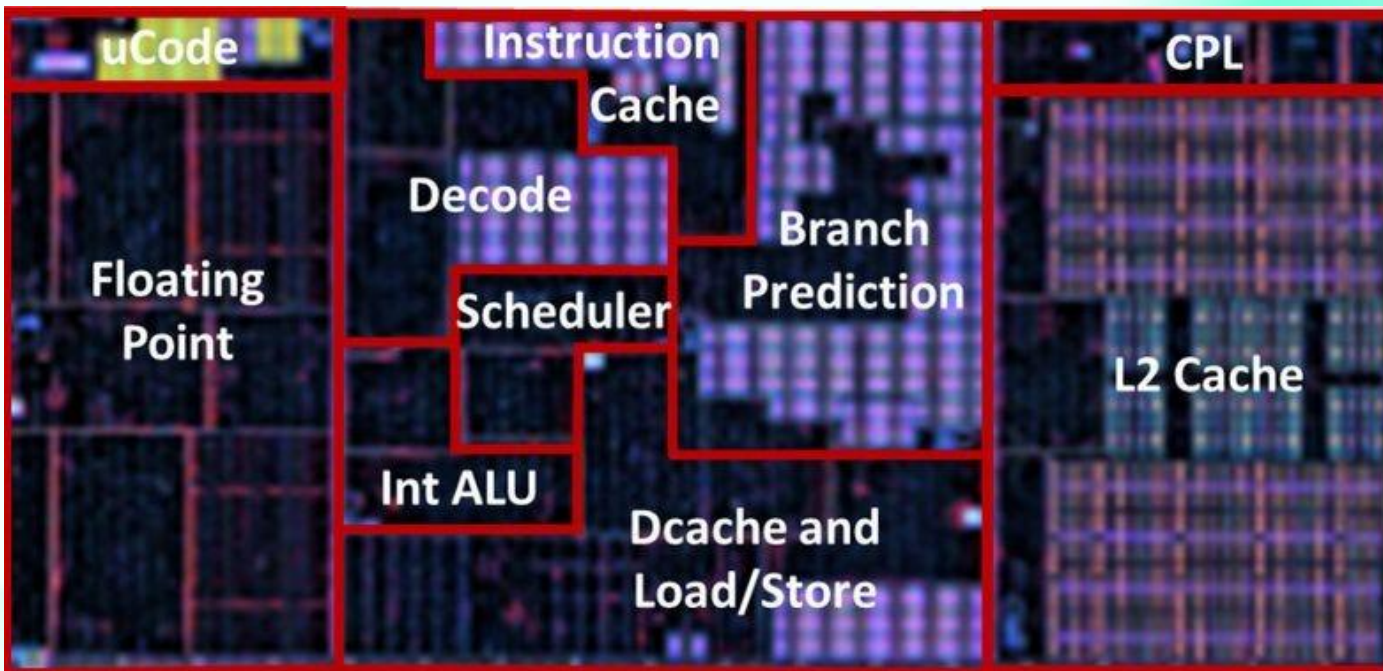
А зачем нам все это?

Внутрянка процессора напрямую определяет производительность нашего кода.

Понимание его механизмов дает знание, что в нашем коде может работать эффективно а что нет.

Branch Predictor один из важнейших узлов процессора. И понимание принципов его работы дает мощный инструмент для оптимизации производительности кода.

Размер бранч предиктора на кристале



AMD Zen4

Ограничимся RISC

Будем рассматривать как пример **RISC** архитектуры, т.к. CISC привносят много лишних проблем.

Как устроена программа

Виды бранчей

Безусловные - инструкции выполняющие переход к выполнению другого кода без условий.

Можно разделить на:

- **Относительные** - по смещению от текущей позиции
- **По абсолютному адресу** - адрес загружается из памяти или берется из регистра.

Условные - если условие выполняется, то делаем переход в другую часть кода, если НЕТ - то продолжаем исполнение кода дальше.

Безусловные и Условные бранчи

Конструкции языка

- **goto**
- **while / for** - прыжок после тела на условие
- **break / continue**
- **if** - прыжок с после тела истинного условия, на конец выражения
- **switch** - с jump таблицей
- **Вызов функций**
- **Возврат из функции**

Инструкции процессора

- **x86** - **jmp, call, ret**
- **aarch64** - **b, br, bl, ret**
- **risc-v** - **j, jalr, jal**

Конструкции языка

- **while / for** - прыжок на проверке условия
- **if** - прыжок на else или конец выражения
- **switch** - без jump таблицы

Инструкции процессора

- **x86** - **je, jne, ...**
- **aarch64** - **b.ne, cbz, tbz, ...**
- **risc-v** - **beq, bne, ...**

Структура программы

```

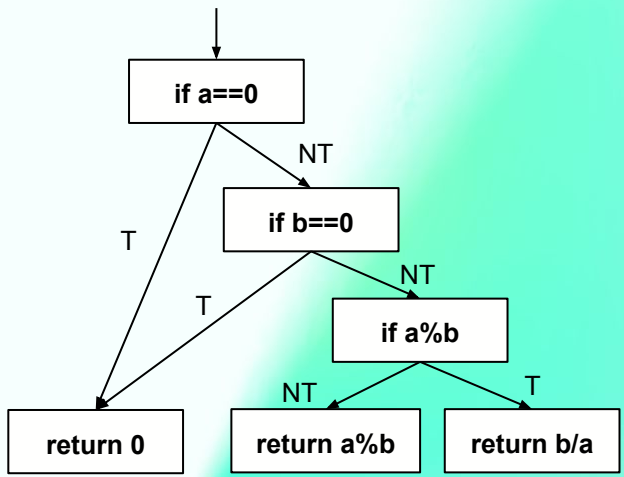
int fn1(int a, int b)
{
  int c;
  if (a == 0 || b == 0)
    return 0;
  if (a % b)
    c = a % b;
  else
    c = b / a;
  return c;
}

```

```

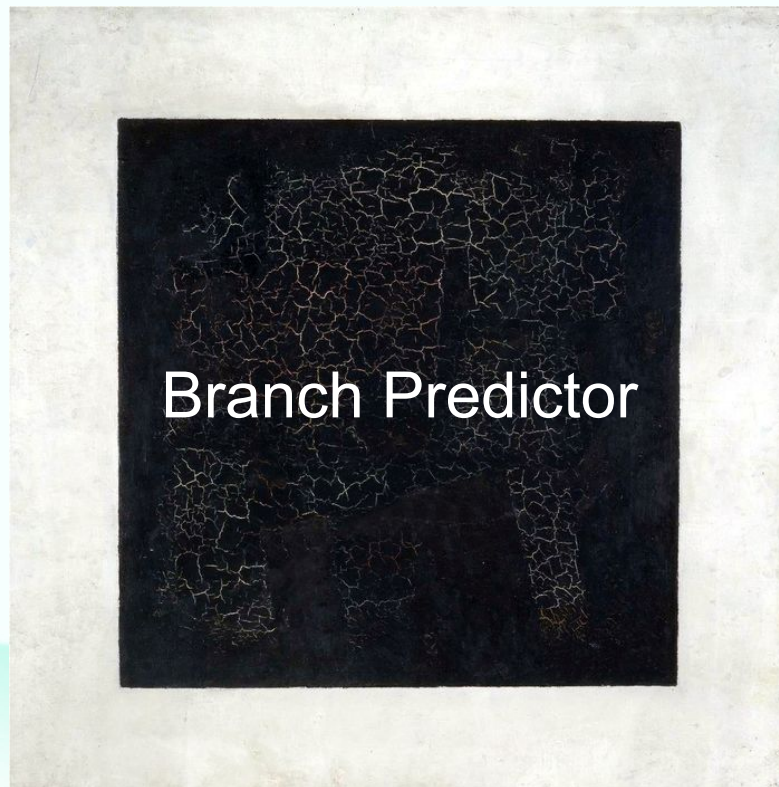
fn1(int, int):
  test edi, edi
  je .L3
  test esi, esi
  je .L3
  mov eax, edi
  cdq
  idiv esi
  mov ecx, edx
  test edx, edx
  je .L5
  mov eax, ecx
  ret
.L5:
  mov eax, esi
  cdq
  idiv edi
  mov ecx, eax
  mov eax, ecx
  ret
.L3:
  xor ecx, ecx
  mov eax, ecx
  ret

```



Как устроен процессор

Пока считаем что логика Branch-predictor - черный ящик



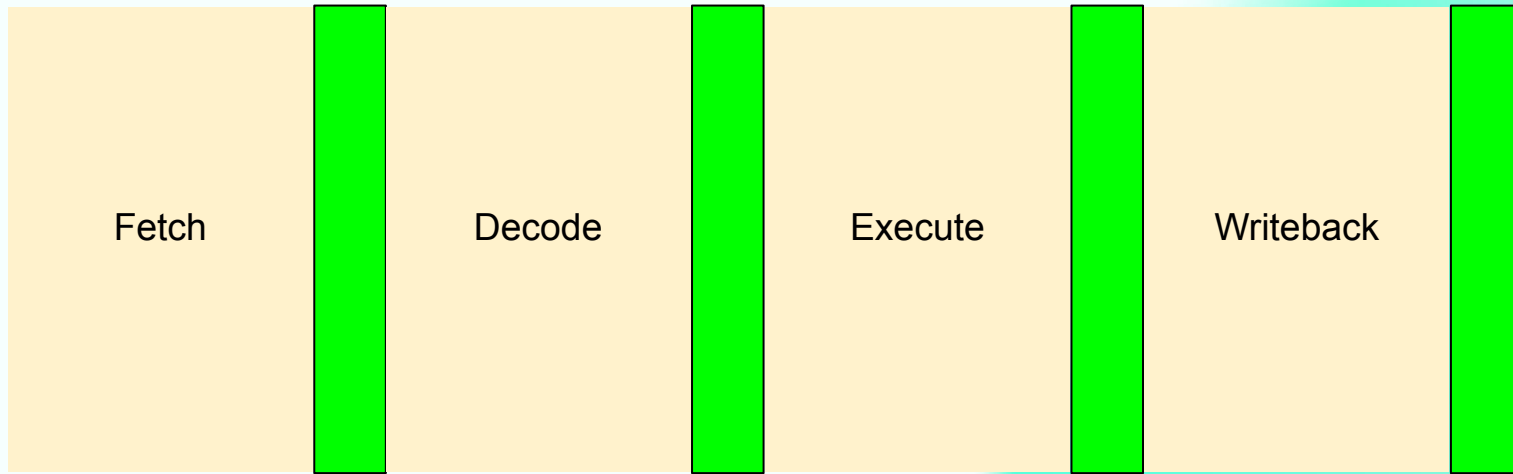
Branch-predictor of Intel



Пока считаем что логика Branch-predictor - черный ящик

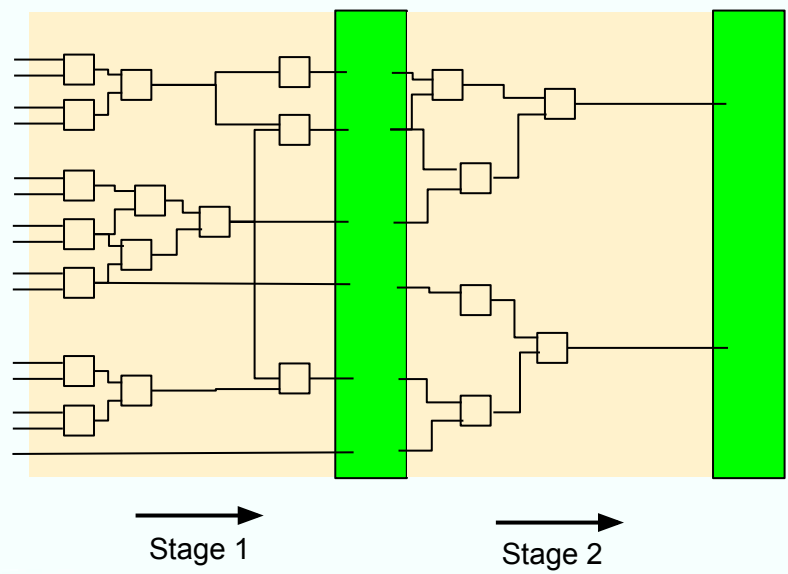


Конвейер (pipeline)

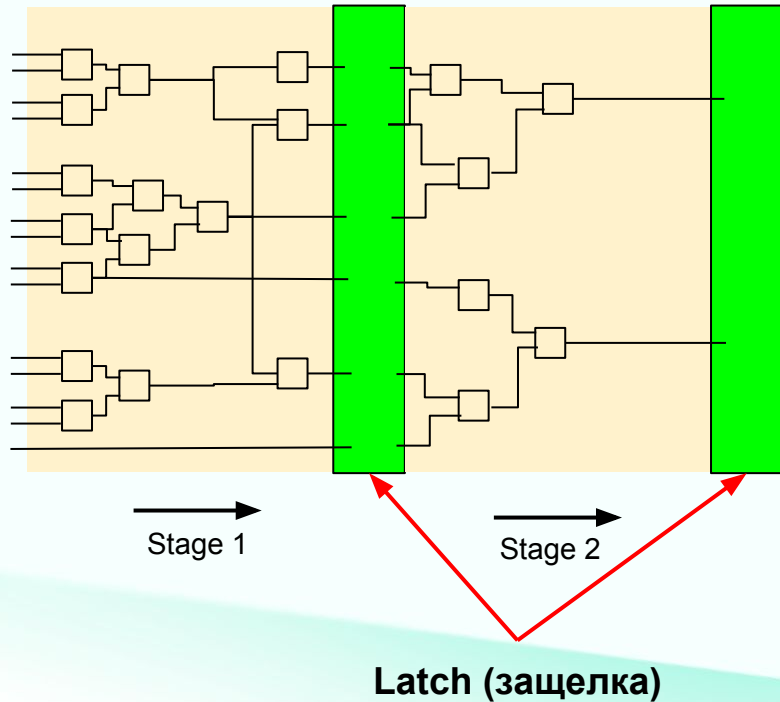


Вся логика вычислений в конвейере разделена на стадии одинаковой продолжительности.

Как на самом деле работает цифровая электроника

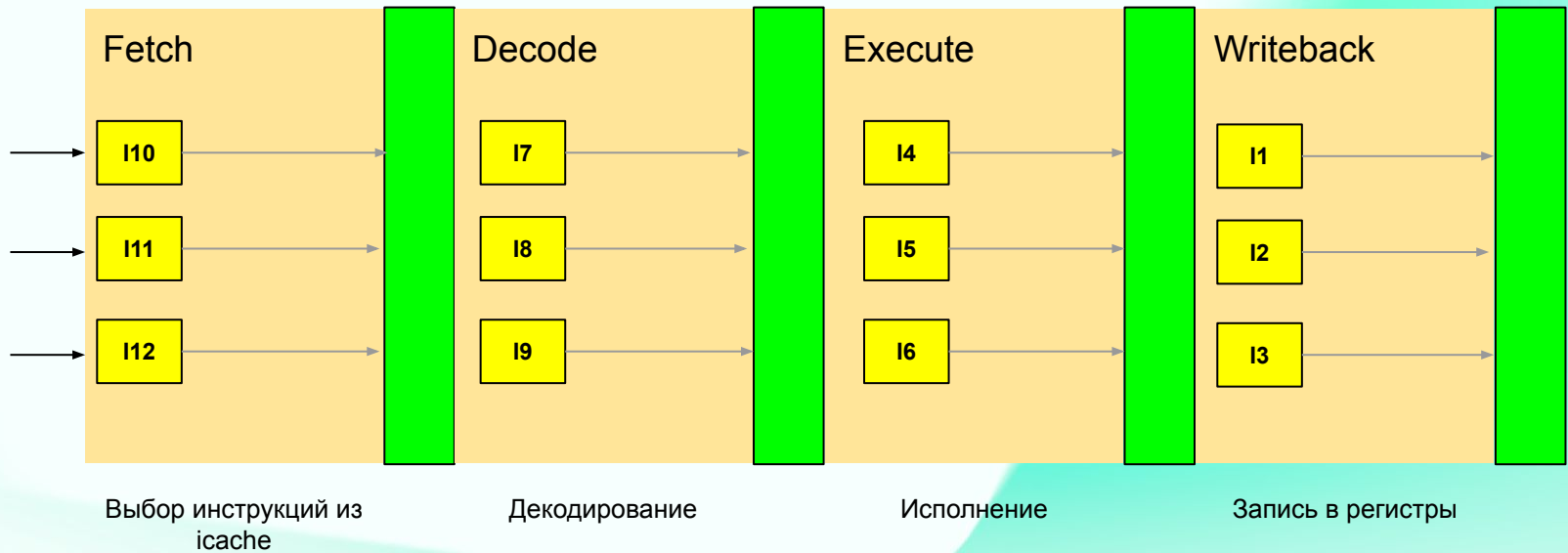


Как на самом деле работает цифровая электроника



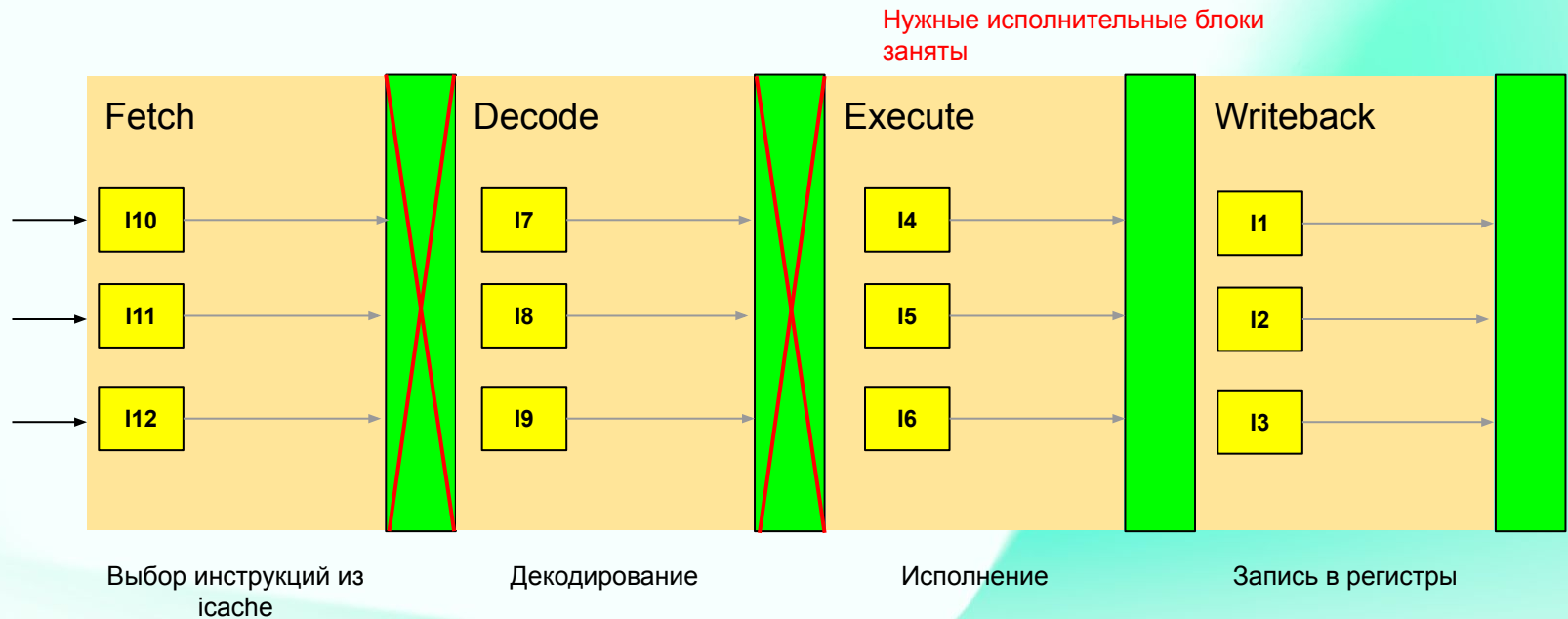
- Транзисторы работают в **режиме ключа**, и имеют время на **переходные процессы**
- На разных уровнях транзисторы включаются в работу в разное время
- **Такт** - это время которое достаточно для прохождения всех переходных процессов в глубь схемы, и является сигналом для **защелок (latch)**, что данные готовы.

Что такое на самом деле конвейер (pipeline)



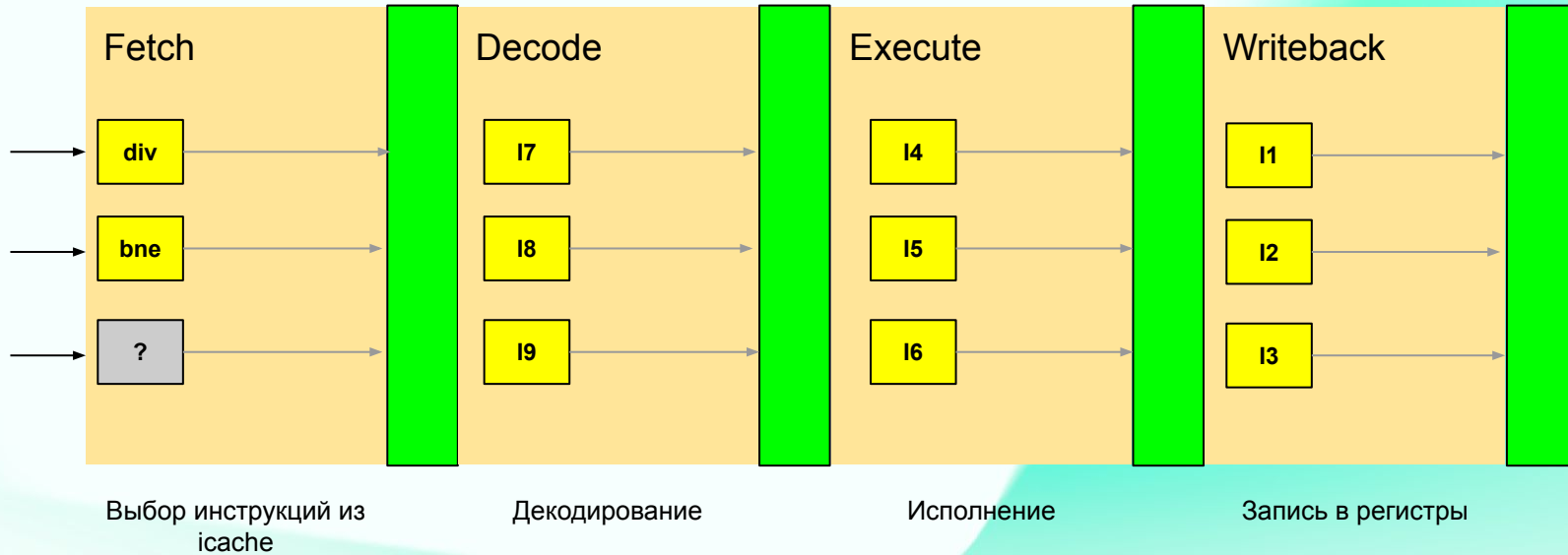
Superscalar процессор - выполнение нескольких инструкций сразу.

Buble (Пузырь) в конвейере



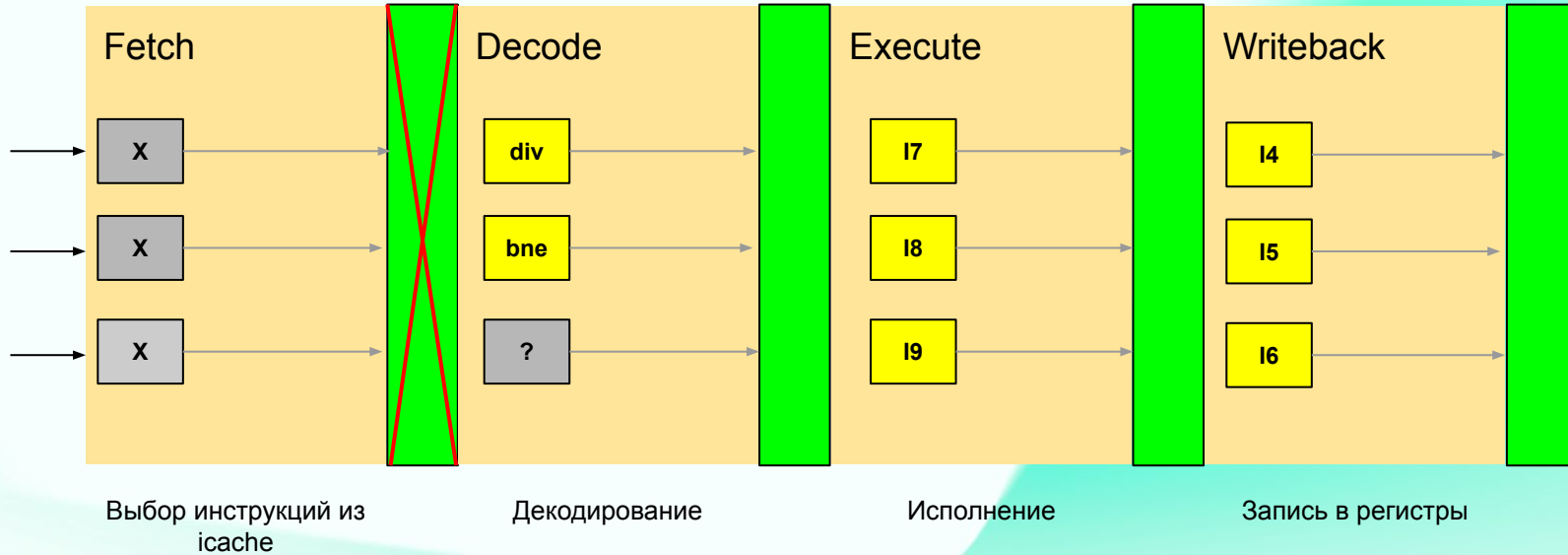
Если где-то в конвейере возникает затор (недоступны какие-то ресурсы), защелки (latch) перестают пропускать новые инструкции. Возникает так называемый **Buble** (пузырь).

Bubble (Пузырь) в pipeline из за бранча



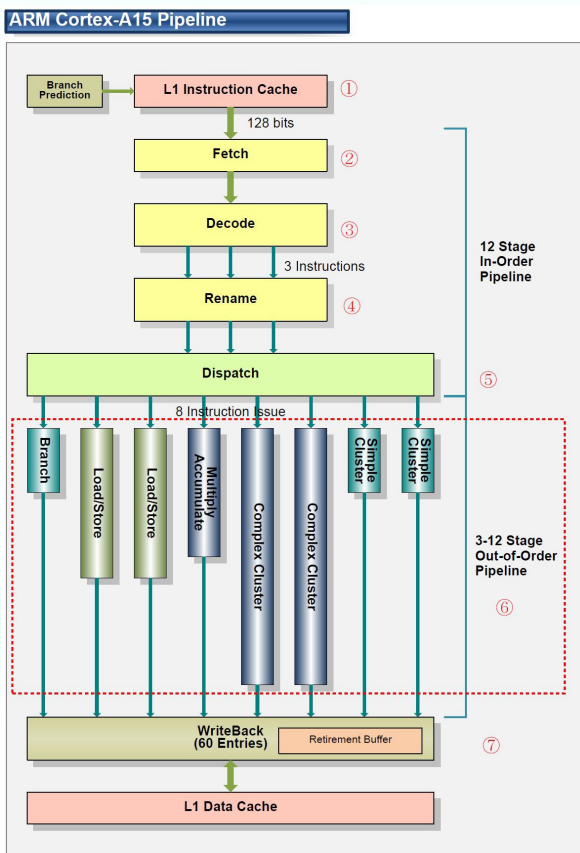
Если одна инструкция на входе **branch**, какая должна быть следующая?
Следующая или по адресу перехода?

Bubble (Пузырь) в pipeline из за бранча



Если одна инструкция на входе **branch**, какая должна быть следующая?
Следующая или по адресу перехода?

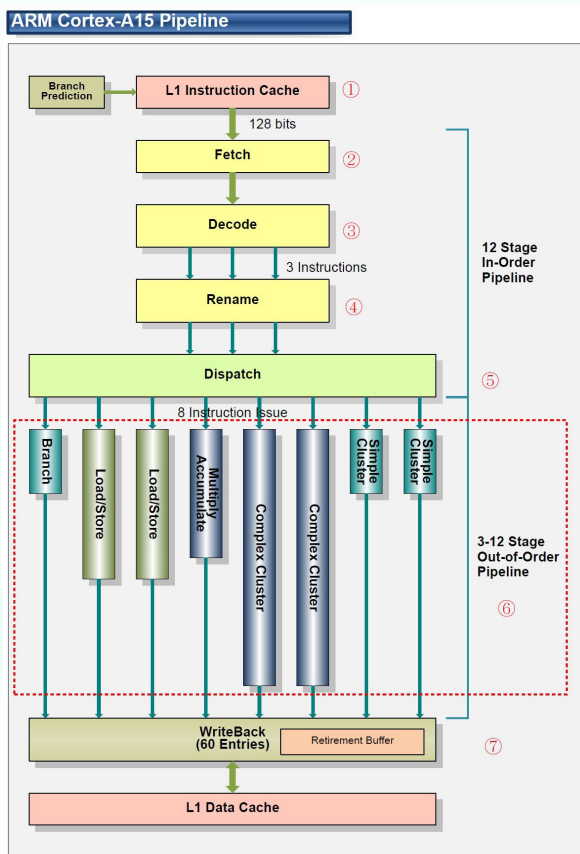
Конвейер современного процессора



Весь вычислительный процесс разбит на стадии:

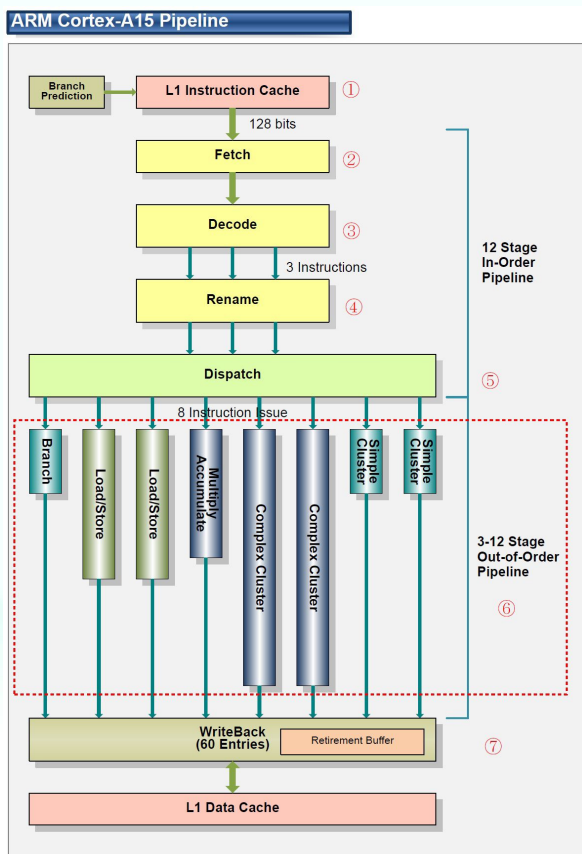
1. Доступ к кешу инструкций (взаимодействует с бранч предиктором)
2. **Fetch** - осуществляет выборку инструкций.
3. **Decode** - декодирование инструкций.
4. **Rename** - переименование регистров.
5. **Dispatch (Reservation station)** - накопление готовых и неготовых к выполнению инструкций (**OoO**).
6. **Execution** - блоки исполнения команд (**OoO**).
7. **Reorder buffer** - восстанавливаем последовательную логику инструкций (**In order**).

L1 Instruction cache + Fetch



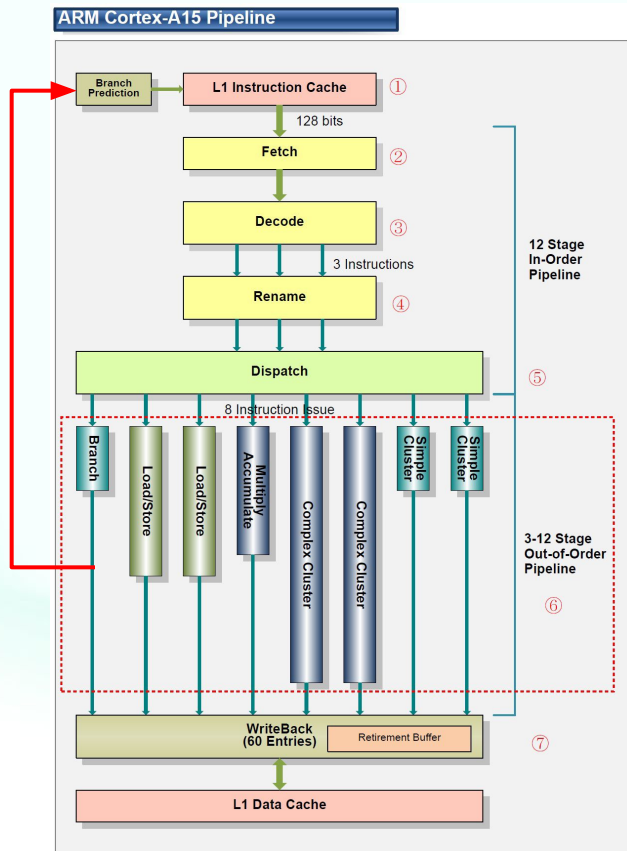
1. Из кеша вытаскиваются несколько инструкций (**super scalar**) - в современных **4/6/8**.
2. Кол-во выбираемых инструкций за такт - “**Ширина fetch-группы**”.
3. **Branch predictor (BP)** непосредственно вмешивается в выборку, и может менять ее в середине **fetch-группы**.
4. Адреса выборки **PC (Program Counter)/IP (Instruction Pointer)**.

Decode



1. Инструкции приходят разных форматов: `<dst_reg, reg, reg>`, `<reg, reg, imm>`, `<imm>`...
2. Инструкции с **целочисленные, float** ...
3. На следующую стадию отдается больше бит информации (декодированной).
4. Фактически определяется для последующих стадий - куда девать инструкцию.

Execute

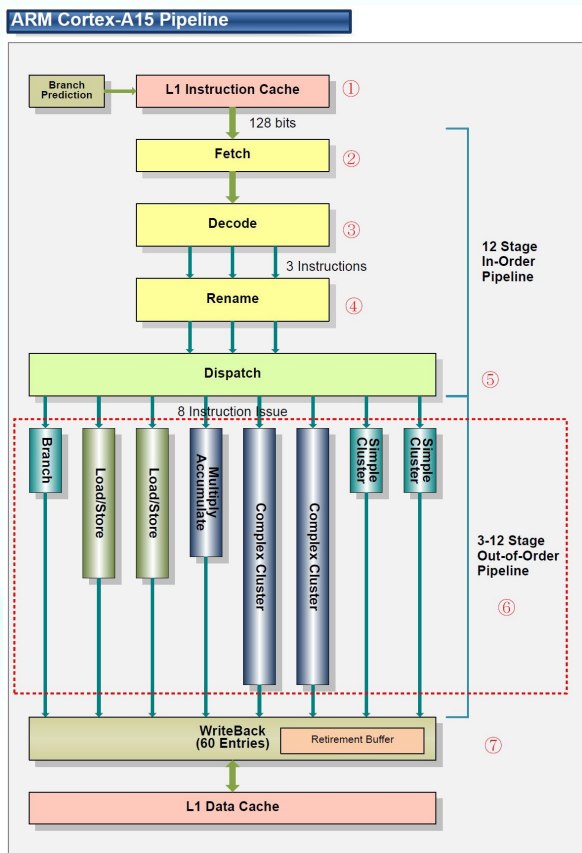


1. Нет смысла все параллельные исполнительные блоки делать одинаковыми по функционалу.
2. От 1 до 2 блоков выделены под обработку бранчей - обычно совмещены с АЛУ.
3. Разные цепочки исполнения имеют разную длину исполнения.

Out of Order (OoO) исполнение

Механизм OoO исполнения строится на основе алгоритма **Томасуло**.

Rename



1. В программе инструкции идут последовательно
2. НО они могут выполняться параллельно!
3. Между инструкция могут быть 3 типа зависимостей (**Hazards**): **RAW, WAW, WAR**

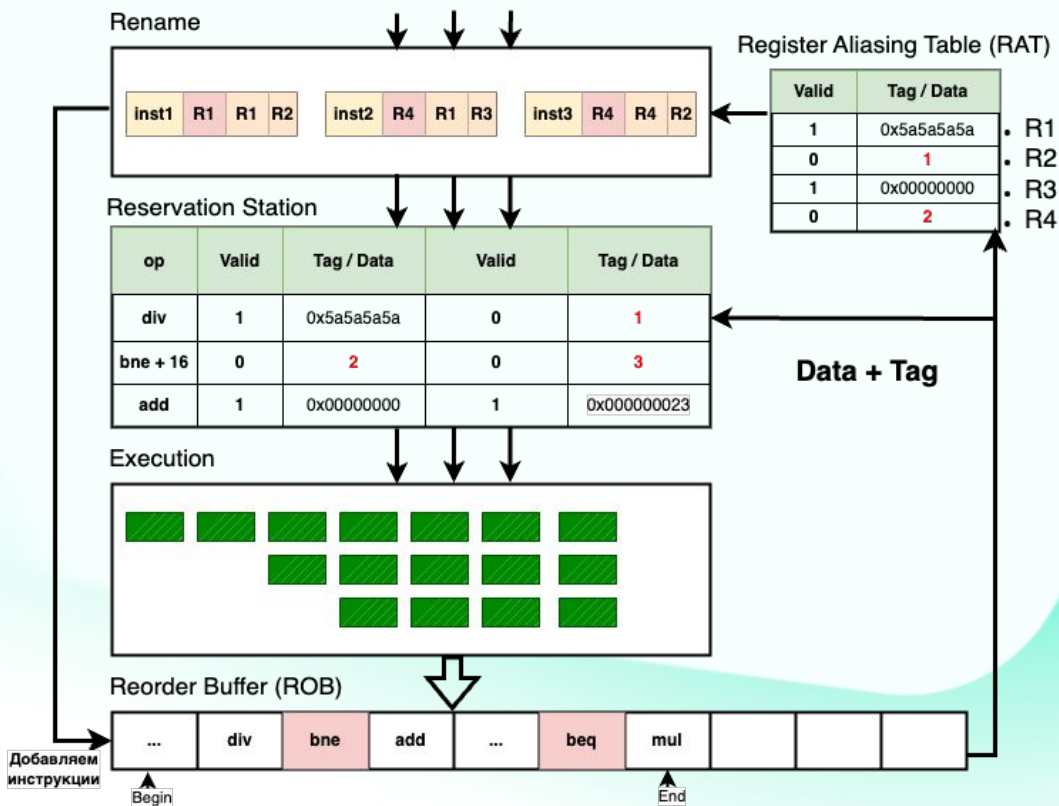
RAW (Read After Write) - инструкция требует результат записи в регистр от предыдущей (**True dependency**).

WAW (Write After Write) - инструкция пишет результат в тот же регистр.

WAR (Write After Read) - инструкция пишет в регистр, из которого читает предшествующая инструкция.

Идея переименования - давайте превратим поток инструкций, в подобие **Single Static Assignment (SSA)**. Каждое новое значение в свободное место.

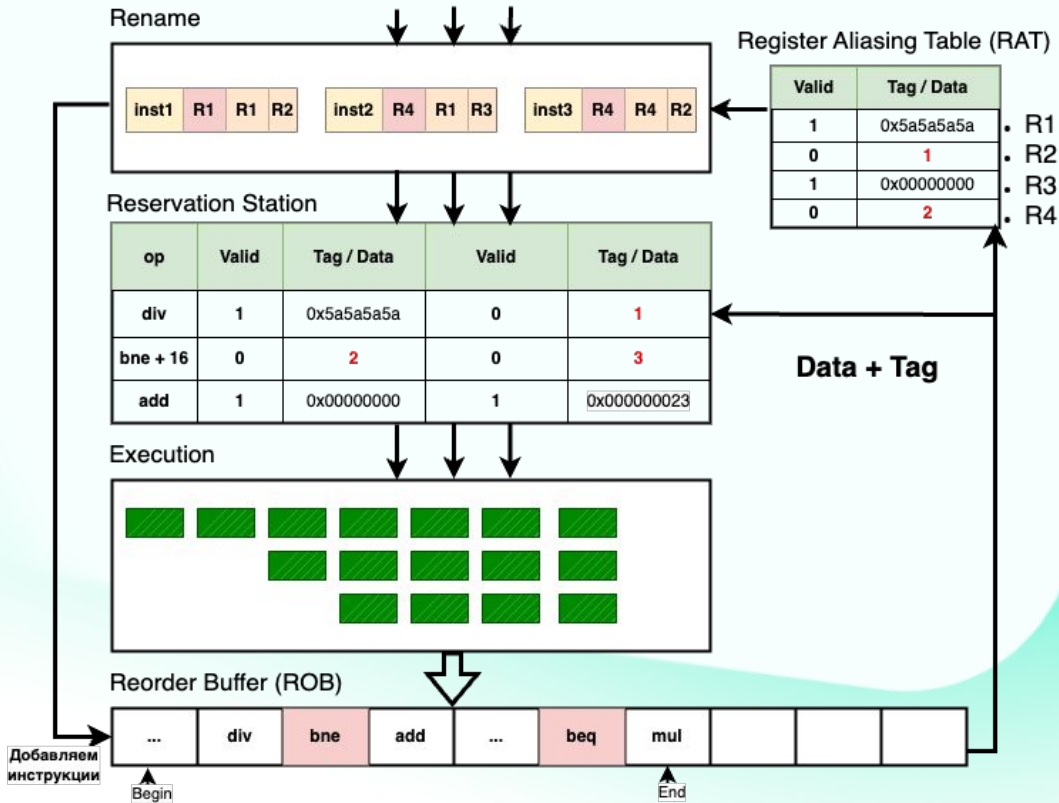
Томасуло алгоритм (Rename)



- Приходящие инструкции регистрируются в **Reorder Buffer (ROB)**.
- Переименование происходит в **RAT (Register Aliasing Table)**.
- Содержит или значение или **Tag** - индекс в **ROB**.
- Внутри инструкции может быть **WAR** зависимость.
- Между инструкциями могут быть все типы зависимостей.
- На следующую стадию уходят либо значения регистров, либо **Таги**.

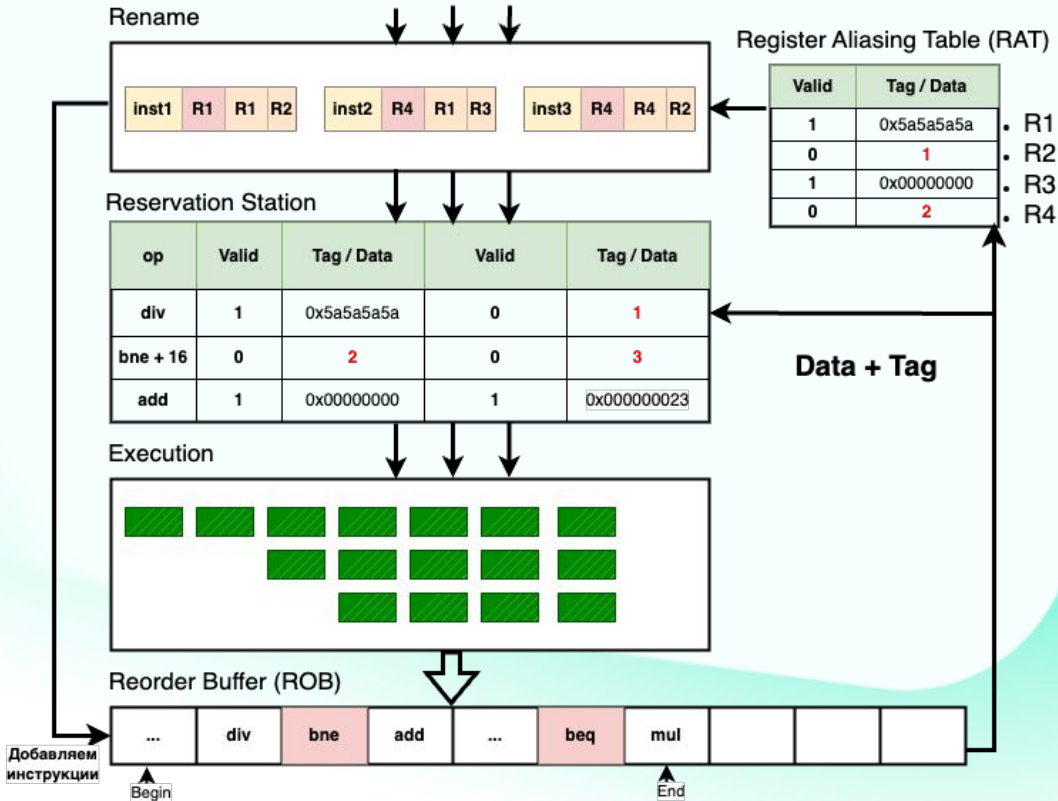
Tag: future + promise

Томасуло алгоритм (Reservation Station)



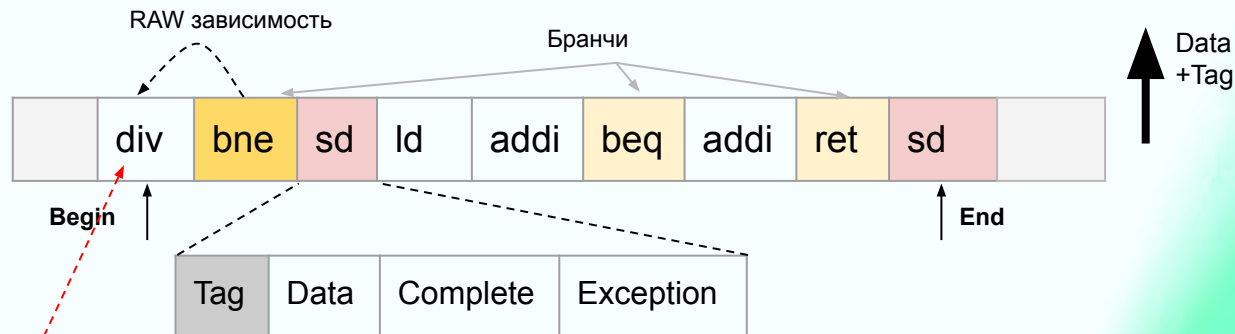
- **Reservation Station (RS)** - накапливает инструкции готовые и неготовые к исполнению.
- Когда все **данные готовы** и есть **свободные исполнители** - инструкция отправляется на обработку (**Issue**).
- Каждая занятая инструкцией запись в RS ожидает поступления пары **Tag+Data**, если таг совпал, то значение запоминается.

Томасуло алгоритм (Reorder Buffer)



- **Re-order Buffer (ROB)** - возвращает порядок следования инструкций после OoO исполнения.
- Инструкции в нем в порядке следования в программе.
- Представляет из себя кольцевой буфер.
- В нем находятся инструкции в состоянии **incomplete**, **complete** но не **retired**.
- Программный аналог - **git**, у нас есть комиты но они не запущены.

Re-order buffer (ROB)

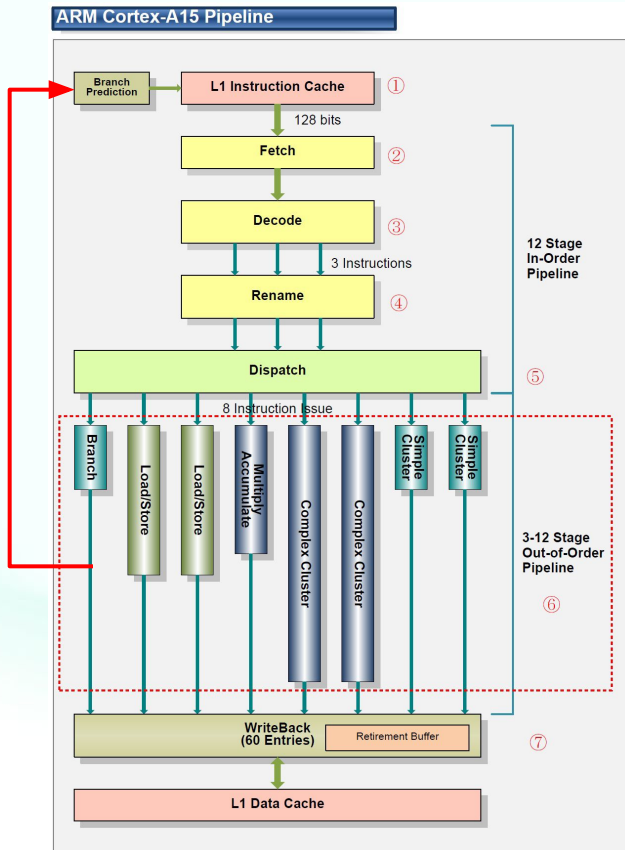


AMD Zen4 ~10-18 циклов

Кольцевой буфер, инструкции в порядке следования программы.

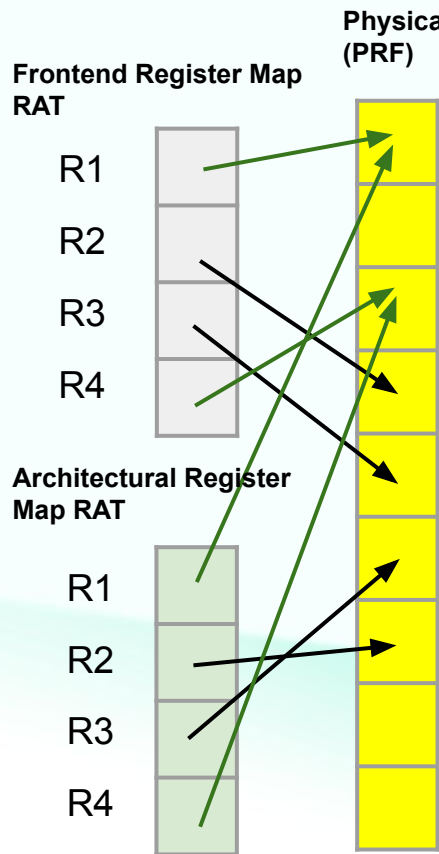
- В нем могут быть инструкции в состояниях **complete/incomplete/exception**.
- **Операции записи** могут произвести реальную запись в кеш, только когда бранч подтвержден.
- В проспекулированных инструкциях может произойти **исключение** (деление на 0, обращение за границы памяти), но пока бранч не подтвержден мы не знаем точно.
- **Complete** инструкции от **Begin** до первой не **incomplete** переходят в состояние **retired**. Но в количестве выходной ширины шины (в оригинале **Common Data Bus (CDB)**).

Если бранч не подтвержден



1. Все что после этой инструкции бранча - **сбрасываем!**
2. Выставляем новый PC.
3. Следующий Fetch идет с нового PC.

Новая структура регистрового файла

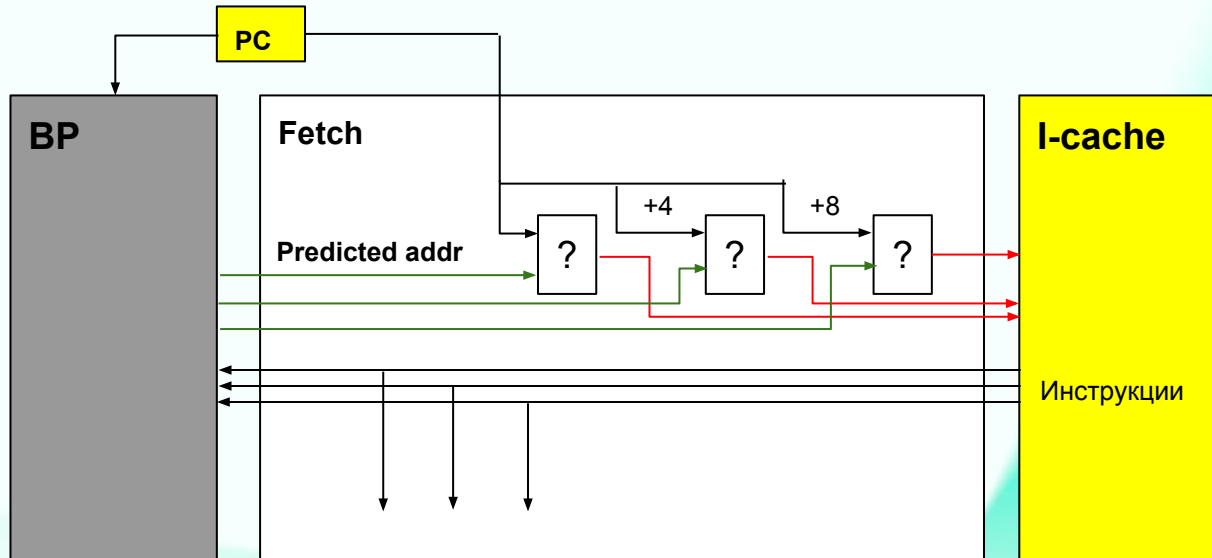


- Все значения вынесены в выделенный регистровый файл **PRF**.
- **RAT** только ссылается индексом на действительное значение в **PRF**.
- **Frontend Register Map** - содержит текущее спекулятивное состояние регистров.
- **Architectural Register Map** - содержит подтвержденное состояние регистров.
- Когда бранч **подтверждается**, мы обновляем **Архитектурную карту** из бранча.
- Если бранч не подтверждается (**misprediction**) - копируем архитектурный RAT в спекулятивный.

Архитектурная - означает адресуется из программы (на Intel RAX, RBX ...)

Branch Predictor -> Fetch (Очень грубая схема)

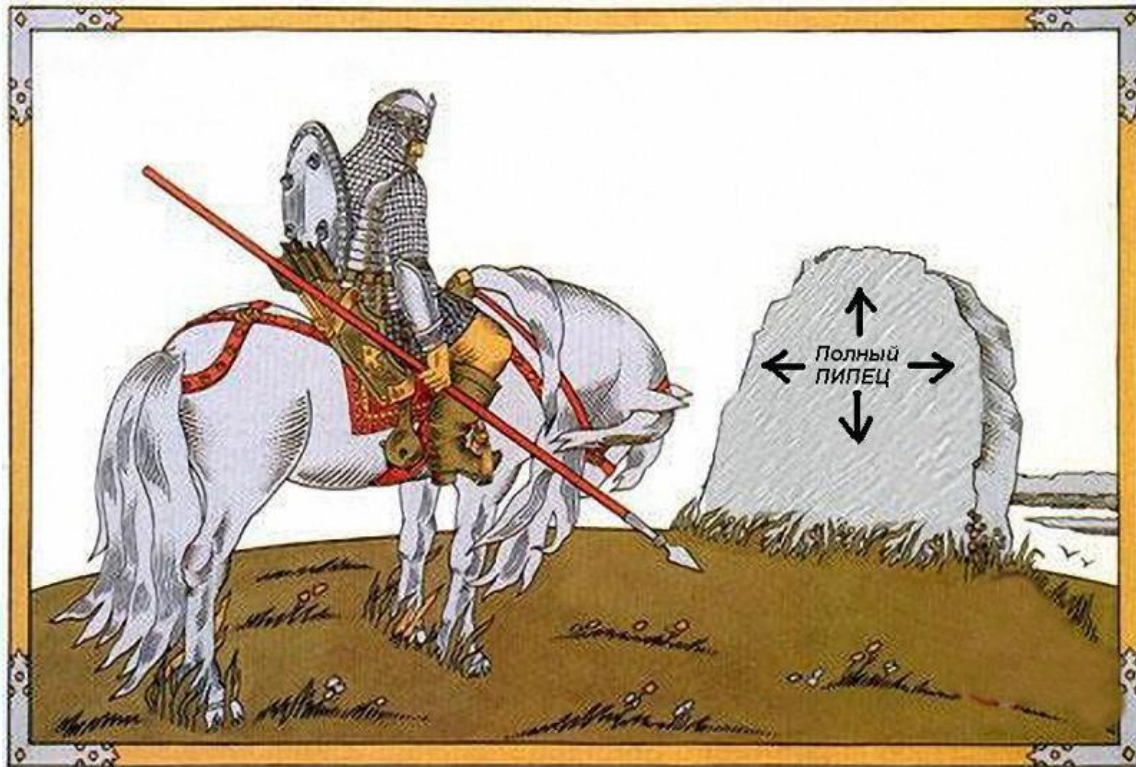
Выбор инструкции относительно **Program Counter (PC)** / **IP (Instruction Pointer)**.



- PC на следующий такт известен - BP может заранее выдать предсказания.
- Бранч может быть на **любом месте** fetch-группы.
- **Коды команд** из I-cache тоже источник информации для BP.

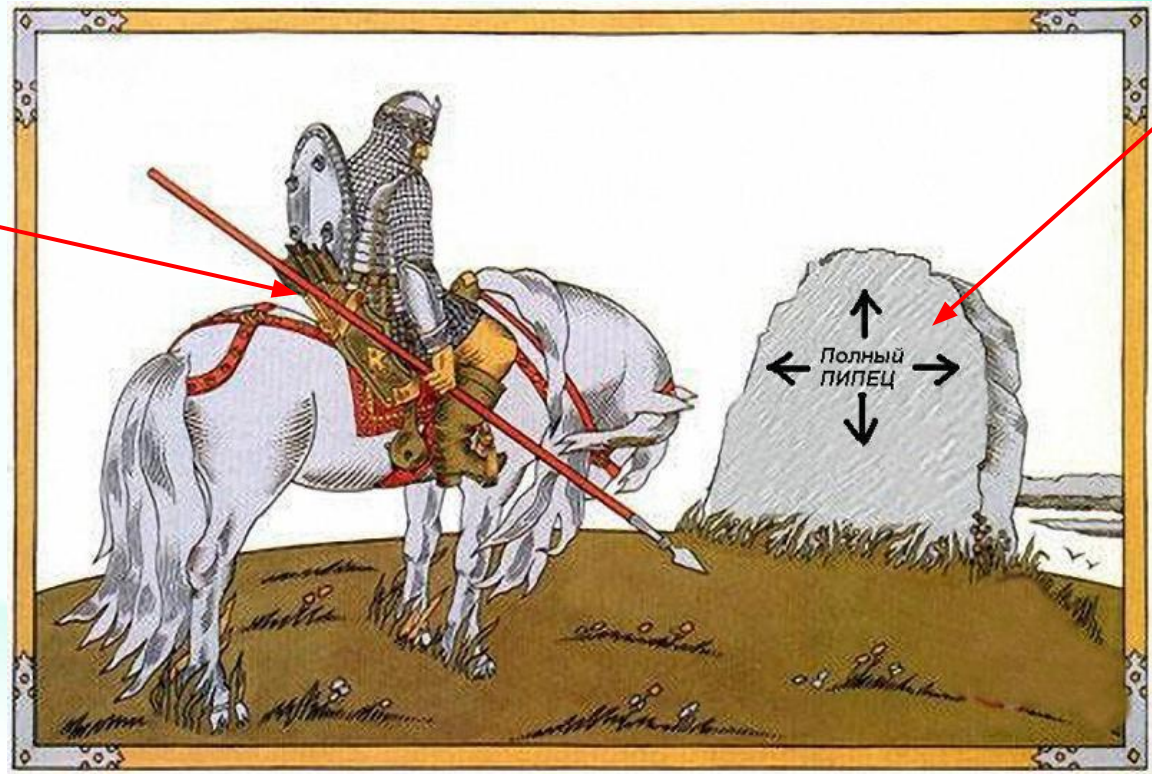
Что такое branch predictor

Branch(Ветвь) - куда пойдём?



Branch(Ветвь) - куда пойдём?

CPU



Я программист - я так вижу!

Немного цифр

Для конвейера **20 стадий**, ширина **фетч группы 5**. Сколько займет исполнение на **500 инструкций**. **1 из 5** инструкций **бранч**.

Точность:

- **100%** - **100** cycles
- **99%** 100 (correct path) + 20 (wrong path) = **120** cycles
- **98%** accuracy: 100 (correct path) + $20 * 2$ (wrong path) = **140** cycles
- **95%** 100 (correct path) + $20 * 5$ (wrong path) = **200** cycles

Статические и динамические предикторы

Нужны для условных ветвей.

Статические - используют минимальную информацию для предсказания. Используются когда процессор впервые видит этот ветвь.

Динамические - используют информацию о предыдущих проходах по ветви.

Статические предикторы

- Нужны когда мы в первый раз попадаем в бранч.
- Мы можем использовать минимум информации.
- Какая информация у нас есть:
 - адрес инструкции
 - код операции
 - направление перехода

Например в RISC-V инструкции бранчей:

branch rs1, rs2, imm

branch - опкод, например **beq**, **blt**, **bgt**, ...

rs1 и **rs2** - сравниваемые операнды

imm - 12 бит, один из которых знак, вшитое в инструкцию смещение.

Always taken/not taken

- **Always not-taken** - по статистике 30-40% срабатываний
- **Always taken** - по статистике 60-70% срабатываний

BTFN (Backward Taken Forward Not Taken)

Используем направление перехода (обычно кодируется в команде как immediate поле).

- Если переход идет **отрицательный** (смещение со знаком -), то скорее всего это цикл, а циклы чаще берутся чем нет.
- Если переход положительный, то не берем.

Показывает неплохую эффективность, и широко распространен.

Op-code based

Используется статистика собранная на типовых задачах - какие оп-коды
бранчей тяготеют братьяся или не-братьсяя.

Хочешь узнать будущее - учи историю!

Динамические предикторы - собирают информацию о предыдущих проходах по коду. И накапливается в виде исторической информации.

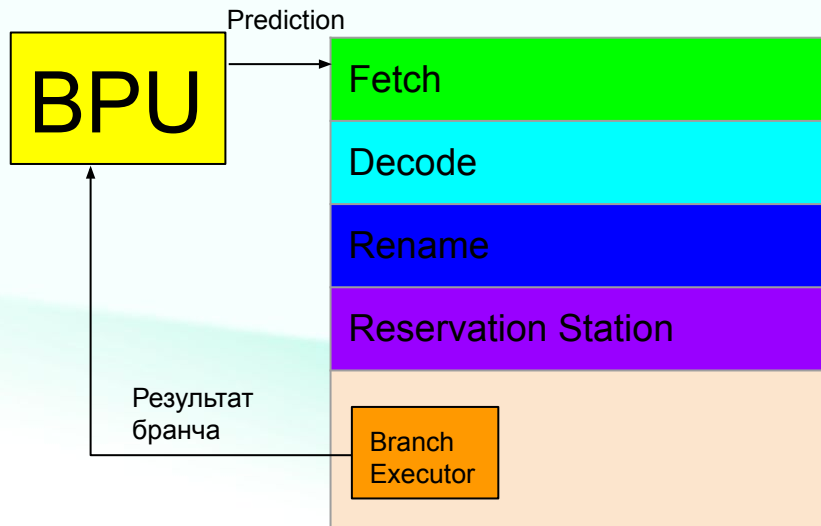
Используются для более сложных по поведению бранчей.

Должен отвечать на вопросы:

- Должен ли произойти переход.
- Куда идет переход.

Общие принципы динамического предсказания

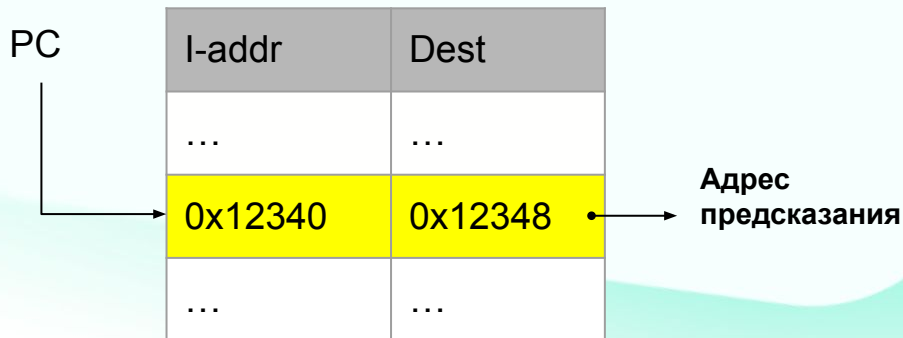
- Динамический предиктор **не используется** если справляется **статический**.
- Динамический предиктор начинает свое **обучение/прогрев (warm-up)** по первому **branch misprediction**.
- Каждый следующий проход используется для аккумуляции знания о поведении бранча.



Branch Target Buffer (BTB)

За хранение направления перехода отвечает **Branch Target Buffer (BTB)**. Он хранит пары **<адрес инструкции перехода, адрес перехода>**.

- Может использовать не все биты адреса.
- **Может** использоваться и со **статическим предиктором** для ускорения получения результата.
- Адрес назначения тоже может меняться!



- Оптимизирован под локальность кода.
- Правая часть может хранить не полный адрес, а **разницу** с текущим!
- Как правило **маленький**, или их 2 маленький и большой.

1BP (Last Time Predictor)

Хорошо подходит когда бранч берет одно из направлений надолго (**TTTTNNNNTTTTTT**).

Но паттерн: **TNTNTNT...** даст 100% misprediction.

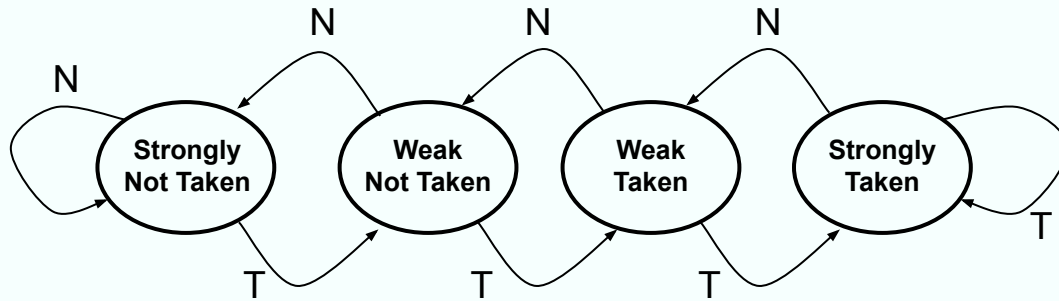
0
1

- бранч не берется (**NT**)

- бранч берется (**T**)

Приблизительная точность: ~85%

2BC(2BP, Bimodal)



0	0	Strongly Not Taken
0	1	Weak Not Taken
1	0	Weak Taken
1	1	Strongly Taken

Histerezis bit
 Prediction bit

2 битный счетчик с насыщением - показал себя как наиболее статистически удачное решение.

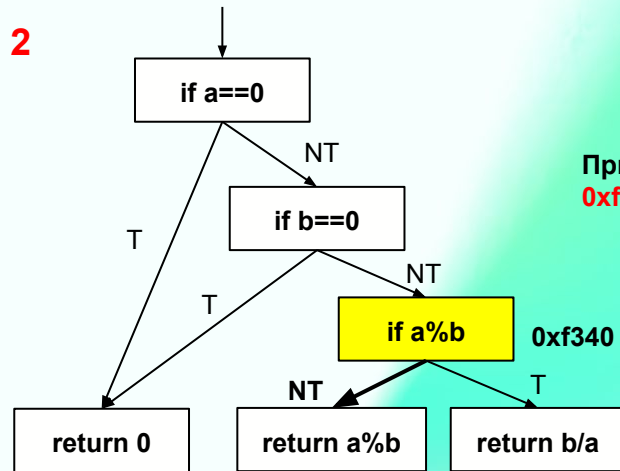
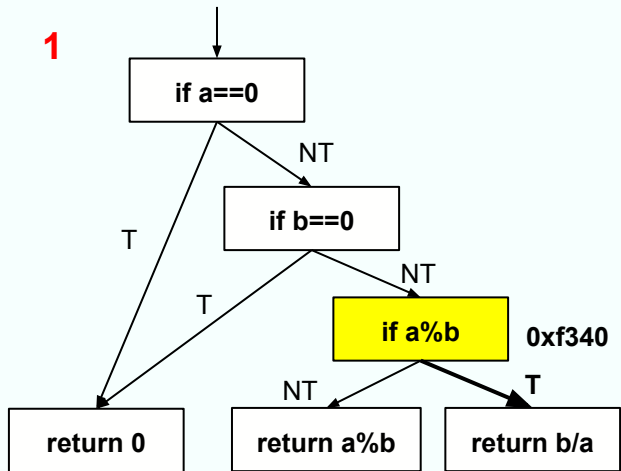
Хорошо подходит для ветвей тяготеющих больше выдавать Т либо не NT.

Например ветвь цикла.

Начальное состояние может быть как **weak** так и **strong**.

Private history

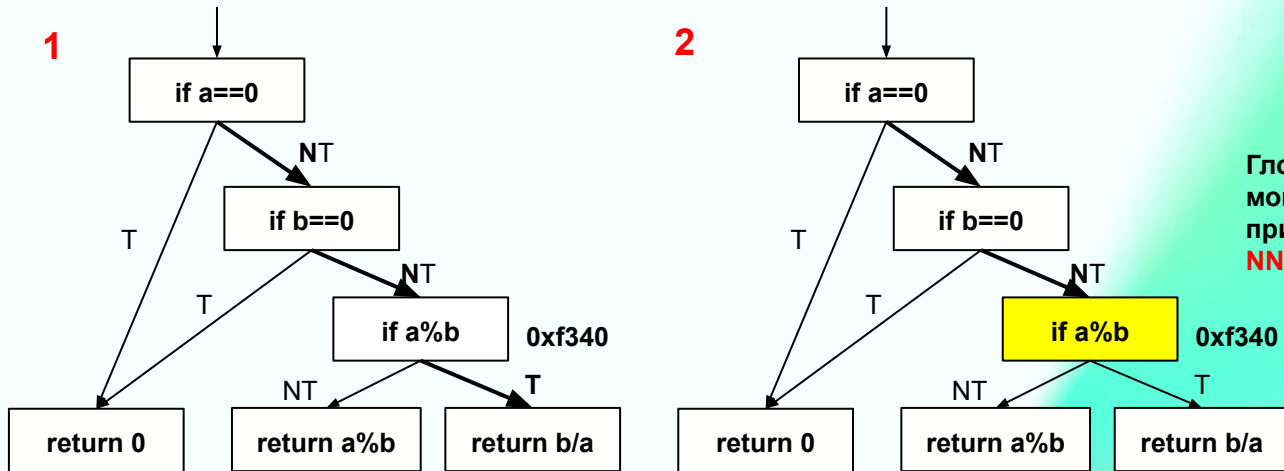
История привязанная к конкретному ветви, по его адресу.



Приватная история
0xf340: ...TN

Global history

История как мы идем по control flow, т.е. как мы пришли в точку. Пусть у нас 2 вызова подряд:



Сдвиговый регистр: на каждом бранче - **сдвиг влево** на 1, выставить правый бит **0=N, 1=T**.

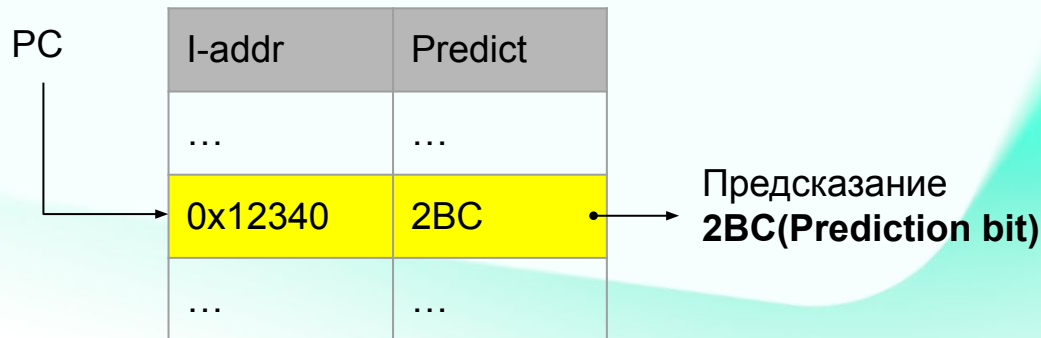
Branch History Table (BHT)

Самый базовый алгоритм предсказания - для пройденных бранчей.

I-addr как правило содержит **только часть адреса**. Т.е. **BHT** фактически является **хеш таблицей**.

Branch executor - возвращает результат, который обновляет 2BC.

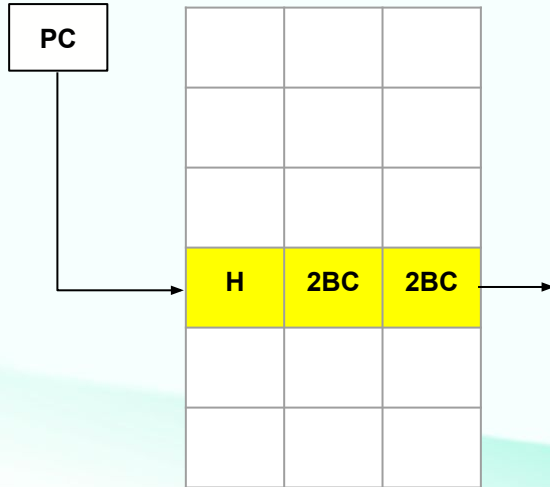
Хорошо работает на циклах: NNNNNTNNNNNT... (На цикл 1 misprediction*)



1 History 2 2BC

На самом деле мы можем распознать паттерны. Для этого нам понадобится **бит истории** (1 если последний раз было предсказано T, 0 если NT), он индексирует какой из 2BC выдает предсказание.

Позволяет распознать паттерн: **TNTNTNTNTN**



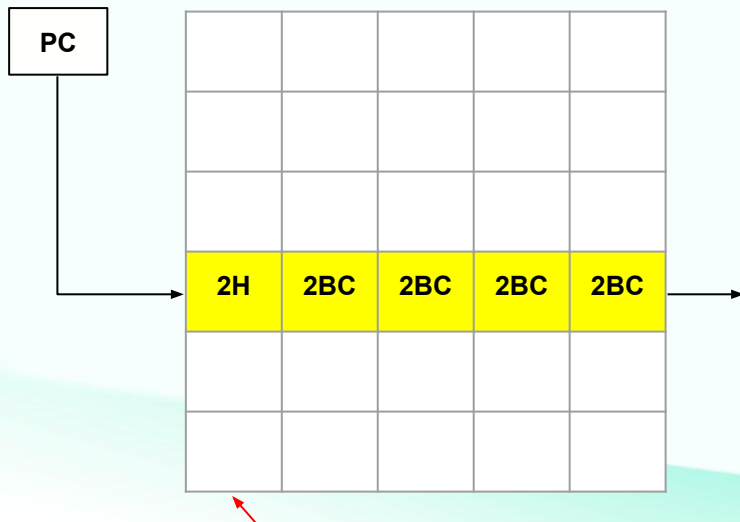
Состояние	Предсказание	Выход
(0,SN,SN)	N	T
(1,WN,SN)	N	N
(0,WN,SN)	N	T
(1,WT,SN)	N	N
(0,WT,SN)	T	T
(1,ST,SN)	N	N
(0,ST,SN)	T	T

2 History 4 2BC

H (history) поле может быть 2 и больше (n) бит, оно потребует **2 pow n** бит.

Позволяет обрабатывать более сложные паттерны, но как правило часть 2BC остаются не используемыми.

Позволяет распознать паттерн: **TTNTTN, NNTNNT, NNTTNNTT ...**



Когда бранч вычислен,
последовательность действий:

- Скорректировать **2BC**
- Сдвинуть на 1 бит BHSR
- Правый бит BHSR установить в результат.

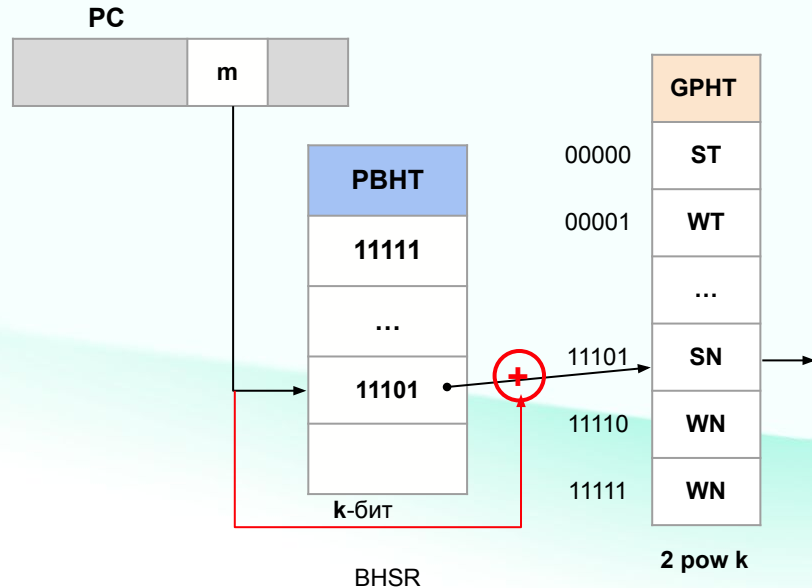
BHSR (Branch History Shift Register)

Сдвиг в лево и установка правого бита в значение вычисленного бранча.

Two level adaptive branch predictor

Двухуровневые адаптивные таблицы - это целый класс бранч предикторов, со своими подклассами.

В данном подвиде мы используем часть битов адреса инструкции, чтобы индексировать в таблице **PBHT (Per-address BHT)**. Каждое поле содержит локальную историю этого бранча. Оно индексирует в **GPHT(Global Pattern History Table)**.



Реальные имплементации, делают дополнительные действия для декоррелирования ячеек PHT. Самое простое это **История XOR Адрес**.

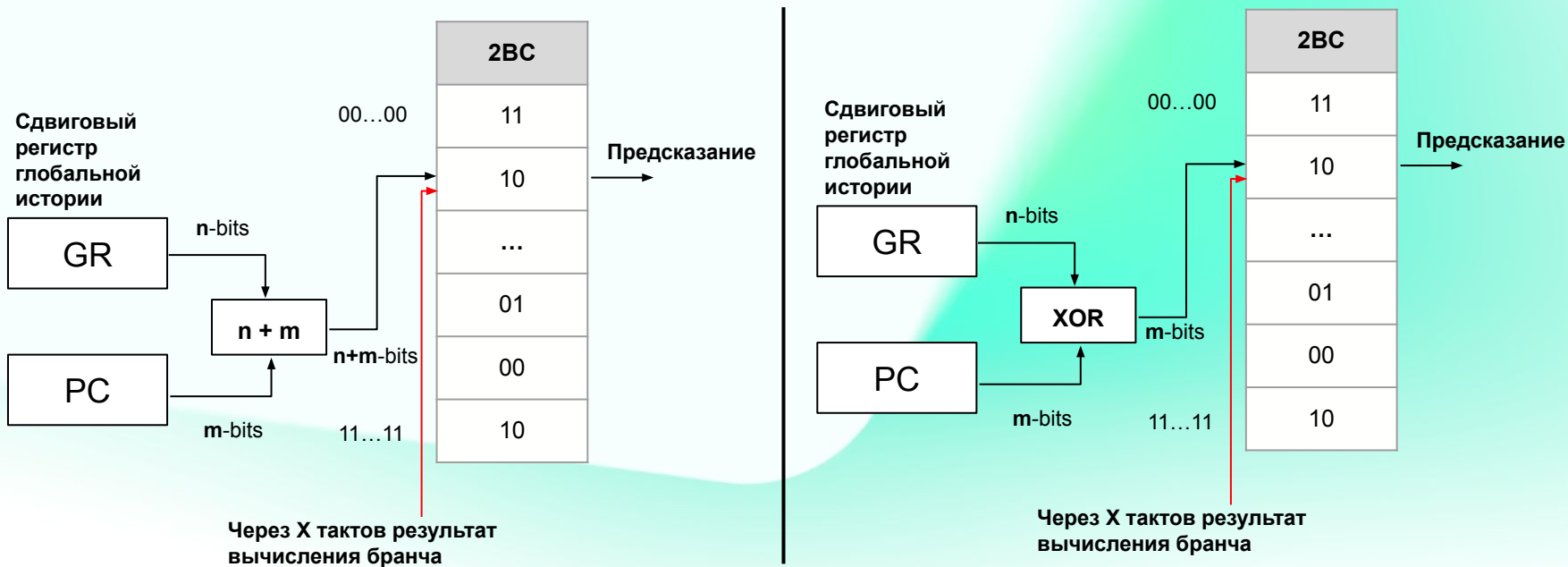
НО, **коллизии** так же реальны.

Забегая вперед, здесь представлен **PAg** предиктор по классификации **Tse-Yu Yeh**'а.

Полностью разберем во 2й части доклада.

G-select / G-share

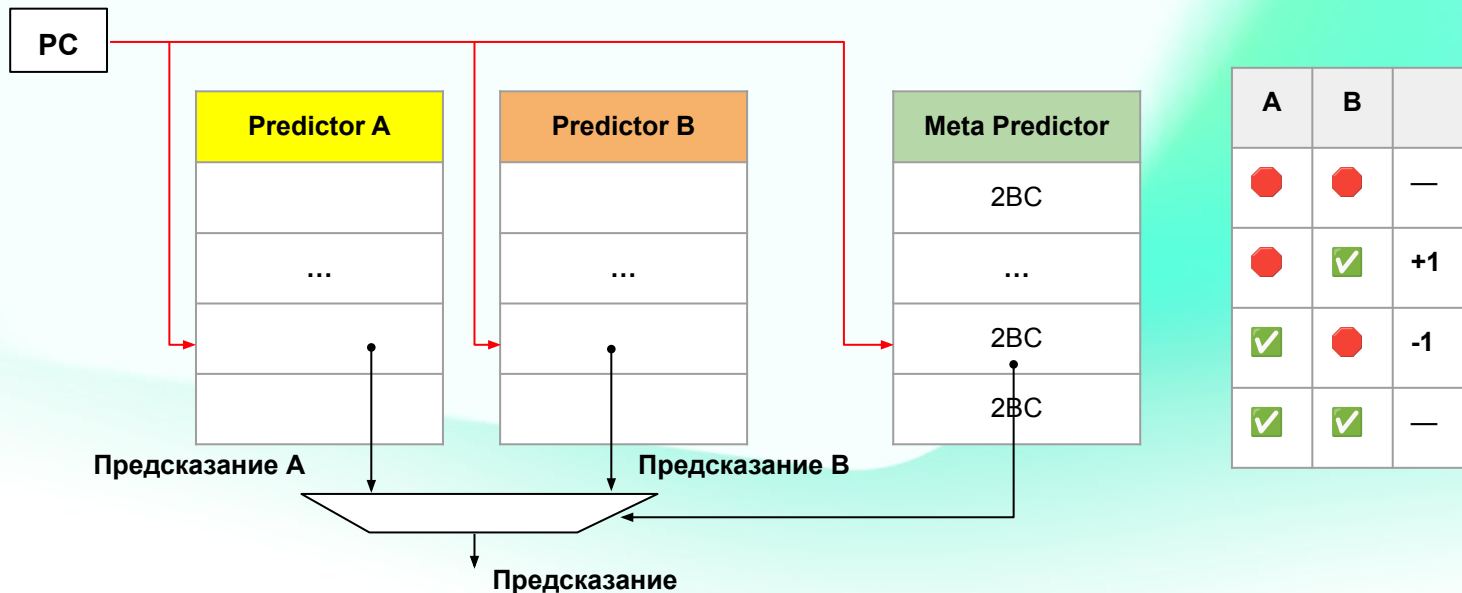
Один простых, но при этом весьма эффективный предикторов. Использует глобальную историю. Чаще используется вариант G-share.



Tournament predictor

Бранчи могут быть зависимыми как от предыдущих бранчей (**global history**), так и зависеть от своей истории (**private history**).

Тогда чтобы правильно предсказывать мы можем использовать сразу 2, и в зависимости от того кто угадывает чаще, выбирать более удачливый.



Hierarchical predictor

Идея такая же как и у Tournament.

Но мы используем по умолчанию **дешевый предиктор**, если он не справляется - **подключается более дорогой** (возможно несколько).

Таким образом мы минимизируем интерференцию в более дорогих предикторах!

RAS (Return Address Stack)

Достаточно большое количество бранчей дают вызовы функций.
У нас есть **инструкция вызова** (иногда несколько) и **инструкция возврата**.

В чем идея: инструкция возврата практически всегда переходит по адресу из стека вызовов, мы можем сделать небольшой быстрый стек в ВР.

Но все равно - **это спекулятивное выполнение**.



Чтобы эффективно отлавливать такие инструкции уже на стадии Fetch, при загрузке **I-cache** делается небольшой **пре-декодирование**.

Как бороться с бранчами

Предикация

Некоторые архитектуры поддерживают инструкции которые в зависимости от результата сравнения перед ней выполняются либо нет.

Мы преобразуем проблему: **control flow** -> **data flow**.

Intel:

```
cmp ax, dx    =>  if (ax < dx)
cmovl ax, dx      ax = dx;
```

Aarch64:

```
cinc X2, X0, X1, EQ => if (X0 == X1)
                        X2 = X1 + 1
                        else
                        X2 = X1
```

И еще куча таких инструкций!

Предикация в действии

```
int fn1(int a, int b)
{
    int c;
    if (a == 0 || b == 0)
        return 0;
    if (a % b)
        c = a % b;
    else
        c = b / a;
    return c;
}
```

GCC

```
fn1(int, int):
    test edi, edi
    je .L3
    test esi, esi
    je .L3
    mov eax, edi
    cdq
    idiv esi
    mov ecx, edx
    test edx, edx
    je .L5
    mov eax, ecx
    ret
.L5:
    mov eax, esi
    cdq
    idiv edi
    mov ecx, eax
    mov eax, ecx
    ret
.L3:
    xor ecx, ecx
    mov eax, ecx
    ret
```

```
fn1(int, int):
    test edi, edi
    sete cl
    test esi, esi
    sete dl
    xor eax, eax
    or dl, cl
    jne .LBB0_3
    mov eax, edi
    cdq
    idiv esi
    mov eax, edx
    test edx, edx
    je .LBB0_2
.LBB0_3:
    ret
.LBB0_2:
    mov eax, esi
    cdq
    idiv edi
    ret
```

Clang заменил 2
бранча - одним!

clang

Branchless programming

По теме **branchless programming** достаточно много информации. Здесь мы на ней не будем останавливаться.

Идея сводится к тому чтобы: **заменить вычисление требующее бранча, на эквивалентное вычисление без.**

Но результат не всегда будет лучше!

```
res = a > b ? c : d;
```

```
a > b: 1 0
```

```
res = (a > b) * c + (a <= b) * d;
```

```
a <= b: 0 1
```

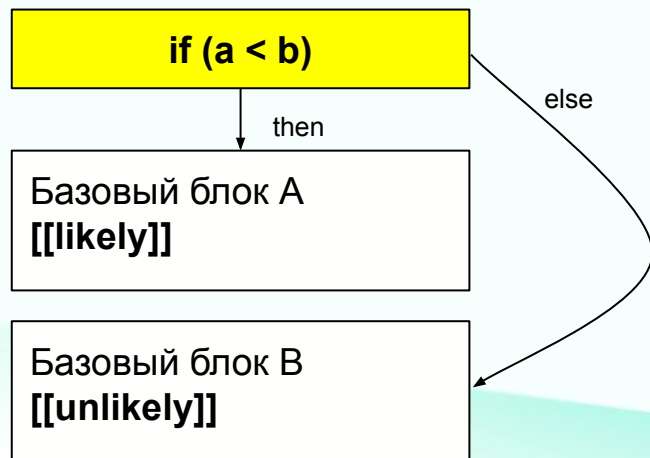
- В цикле предиктор учится (если данные предсказуемы) - и бранч стоит минимально.
- Умножение как правило дорогая операция.

На самом деле код примера компилятор и так хорошо оптимизирует!

likely / unlikely

[[likely]] / **[[unlikely]]** - спецификаторы которые могут помочь статическому бранч предиктору. Переставляя базовые блоки так, чтобы статический бранч предиктор мог их угадывать. Простое преобразование: инвертируем условие, и меняем блоки местами.

Например **BTFN**



Исключения

Исключения помогают решать проблему с бранчами сразу 2 способами:

- Нет проверок ошибок в глубине дерева вызовов
- Ветка на которой есть исключение для компилятора, очевидный **unlikely**.

PGO

Сбор статистики исполнения **Profile Guided Optimization (PGO)**:

- Собираем профиль.
- По частоте использования базовых блоков, строится весовая функция.
- По ней компилятор делает перестановки блоков так, чтобы меньше нагружался динамический предиктор.

Как ловить проблемы с branch misprediction?

- Используем утилиты сбора счетчиков процессора: **perf** ...
Счетчики архитектурно зависимы. Ищем **branches**, **branch-misses**.
- Ищем кривые **likely**, **unlikely**.
- **Causal profiling** - Emery D. Berger.
- В ручную можно попробовать добавить смещение подозрительному бранчу (**на горячих путях**), вставив перед ним пачку **nop** - развести коллизию хеш-таблиц.

Выводы!

- **Циклы ваши друзья!** - на линейном коде предиктор не обучится, и будет много промахов.
- Процессор достаточно “умен” чтобы подстраиваться под разные ситуации, но разные модели даже в одной архитектуре имеют разные возможности.
- Если вы знаете что ваши данные достаточно рандомны, можно обратиться к **branchless programming** (но если вы уже убедились, что компилятор за вас это не сделал).
- Старайтесь **не делать для условия бранча дорогих операций** - это может приводить к деградации предсказаний.

Заключение

Часть 1

Мы поговорили о

- Устройство процессора
- Общие проблемы предсказания ветвления
- Базовые алгоритмы предсказаний
- Статические и динамические предсказания

Часть 2

Мы поговорим о

- Углубимся дальше в теорию и практику алгоритмов предсказания
- Разберем более продвинутые методы предсказания



Слайды

Спасибо!

Евгений Ерохин

Telegram: [@the_gabber](https://t.me/the_gabber)

kaspersky