2.3. Конвейерная архитектура. MIPS.

*MIPS = Microprocessor without Interlocked Pipeline Stages (микропроцессор без задержки конвейера)

1. Конвейерная архитектура

- один из способов увеличения производительности

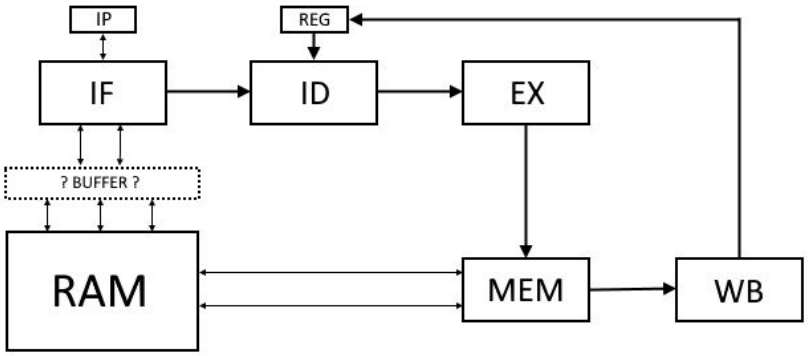
Увеличения производительности можно было добиться двумя способами:

- увеличение скорости работы компонент
  - становилось все более и более сложной задачей со временем
- параллельное выполнение
  - не так сложно в реализации
    - реализация на уровне команд
    - реализация на уровне процессоров

Реализация на уровне команд - одновременная обработка нескольких команд за счет разбиения каждой команды на стадии обработки, которые происходят по очереди в определенном порядке.
За счет такого разбиения одновременно можно обрабатывать по одной команде на каждую стадию.

Реализация на уровне процессоров - разбиение обработки команд на части работы, выполняемые одновременно несколькими процессорами

Первый подход реализуется с помощью конвейера. По сути он представляет собой выстроенные подряд стадии обработки команды.
(Стоит также отметить, что прямое обращение к командам в памяти требует очень много времени ~100-200 тактов на один запрос, поэтому перед выполнением команды выгружаются в буфер (кэш) и дальше берутся конвейером из него).



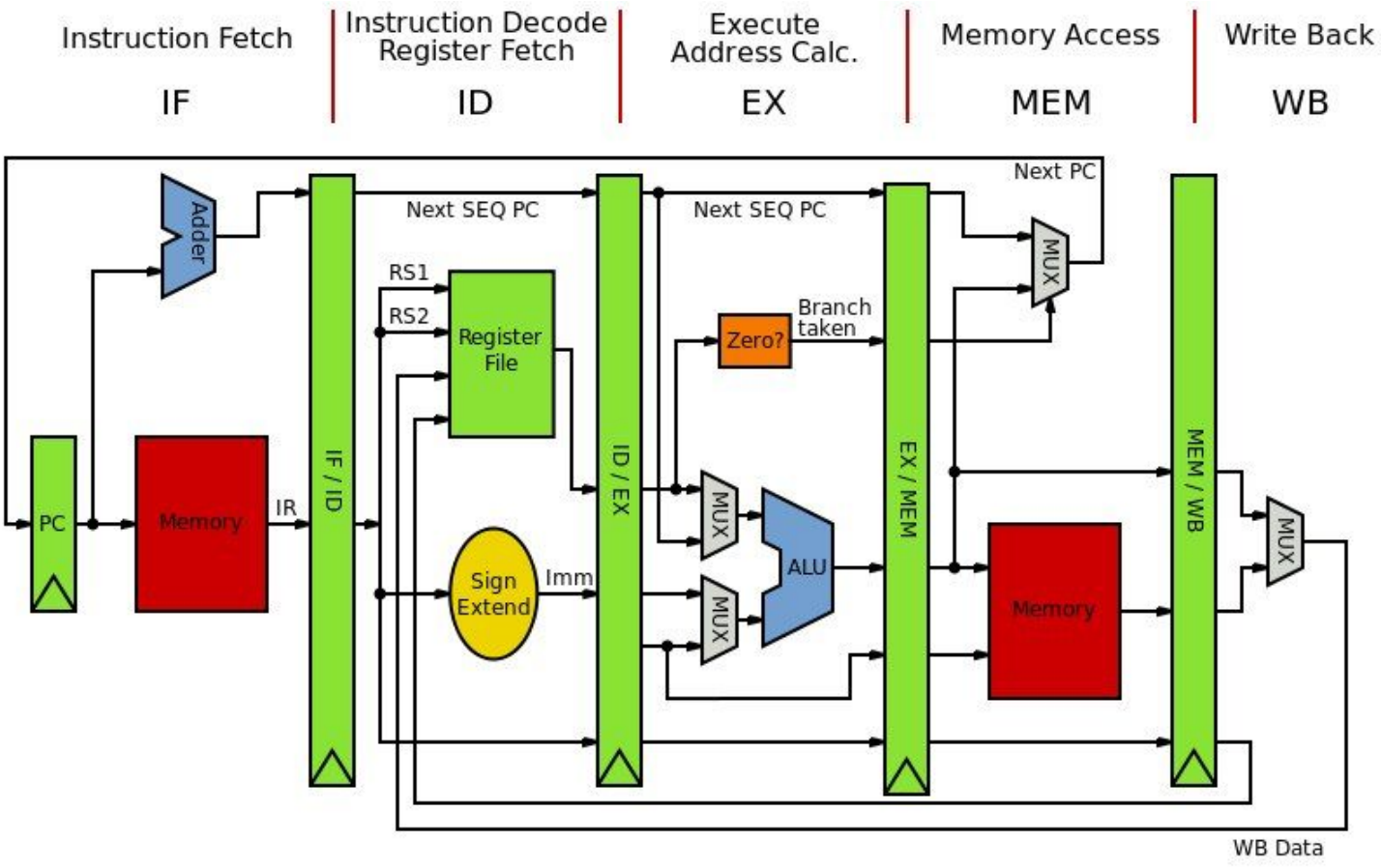
Его основу можно описать как рисунок слева. Всего (в самом простом варианте, то есть **MIPS**, 5 стадий конвейера), и это стадии считывания, декодирования, выполнения, обращения к памяти и записи в регистры.
По мере выполнения одной команды, другие могут обрабатываться на других этапах.

Порядок выполнения:

1. **IF** (instruction fetch)
   a. происходит обращение к **IP** (instruction pointer), который указывает, по какому адресу находится следующая команда (**IP** - специальный регистр)
   b. после чего **IP** увеличивается на размер команды и сама команда передается дальше
2. **ID** (instruction decode)
   a. по полученному коду инструкции **ID** находит, какое действие надо выполнить, после чего считывает те регистры и константы, которые участвуют в этой инструкции
   b. декодированная информация передается в **EX**

3. **EX** (execution)
    a. выполняет операцию
4. **MEM** (memory)
    a. выполняет операцию обращения к памяти, например
        - ld R1 R2 - загрузить в регистр R1 ячейку памяти с адресом R2
        - st R1 R2 - загрузить в память по адресу R2 значение регистра R1
5. **WB** (writeback)
    a. записывает в нужный регистр результат вычислений

Более полная схема работы конвейера выглядит так:



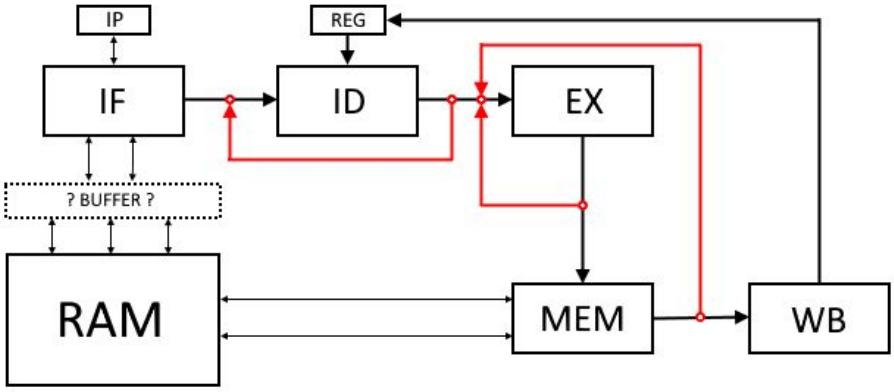
Скорость выполнения - количество тактов, необходимое на обработку одной команды целиком.
Пропускная способность - количество тактов, необходимое перед запуском второй команды после запуска первой - максимальное число тактов на прохождение одной стадии (конвейер синхронизирован так, чтобы на каждую операцию тратилось одинаковое количество тактов, иначе новые команды поступали бы раньше конца обработки старых).
+ из-за простого устройства этапов конвейера, такт системы длиннее такта конвейера

Современные конвейеры имеют в несколько раз больше блоков, что делает схемы в разы сложнее, но позволяет сильно ускорить выполнение команд.

2. FAQ

- Нарисовать схему MIPS, рассказать что такое MIPS

MIPS - простейшая реализация конвейерной архитектуры в 5 стадий. В ней все hazardы решаются без использования **NOP**, Data - с помощью форвардинга, остальные - с помощью документации.



- Если разбивать стадии на меньшие, до какого момента мы сможем получать выигрыш по времени?

До тех пор, пока минусы не станут слишком заметны.
Чем больше стадий, тем:

- Проще стадии, тем больше тактовая частота, до которой их можно разогнать
- Больше энергопотребление (см п.1)
- Требуется решать больше конфликтов
- Больше параллельность, так как большее количество стадий может выполнять большее количество потоков

В энергопотребление упираются намного раньше, чем в конфликты, поэтому количество стадий ограничивается в основном электропотреблением.
Ныне используются порядка 15-30 стадий. В пентиуме 4 юзалось около 30 стадий, из-за чего он очень сильно нагревался.

- Откуда попадают команды на конвейер?

CPU делает обращение к RAM напрямую, так как логически кэш для него не существует. Однако команды будут кэшироваться, поэтому к большей части из них доступ будет быстрым
Хотя, наверное, может быть отдельный буфер, хз.

- Почему MEM идет после EX, а не наоборот?

В **reg-reg** операция загрузки данных - отдельная операция. На самом деле понятно, что после работы с памятью **EX** не имеет смысла. При этом **MEM** после **EX** имеет смысл, чтобы уметь выполнять команду “загрузить данные по адресу <R+CONST>”

- Когда конвейер не дает выигрыша или даже ухудшает производительность?

спросить

2.4. Проблемы конвейера и их решения

1. Data hazards

1.1. RaW (read after write)

Это проблема, возникающая при попытке обращения к регистру, данные в которые должны быть записаны из предыдущей команды, которая не дошла до стадии **writeback**.

К примеру, код
ADD R1 R2 R3
SUB R4 R1 R2

	IF	**ID**	**EX**	**MEM**	**WB**
T = 1	ADD	-	-	-	-
T = 2	SUB	ADD	-	-	-
T = 3	-	SUB	ADD	-	-
T = 4	-	-	SUB	ADD	-

При передаче **SUB** на стадию выполнения, из регистра R1 взято значение. При этом предыдущая команды **ADD**, которая должна была выполниться раньше, еще не дошла до записи нового значения R1 в регистры, поэтому **SUB** будет вычислен с помощью старого значения R1.

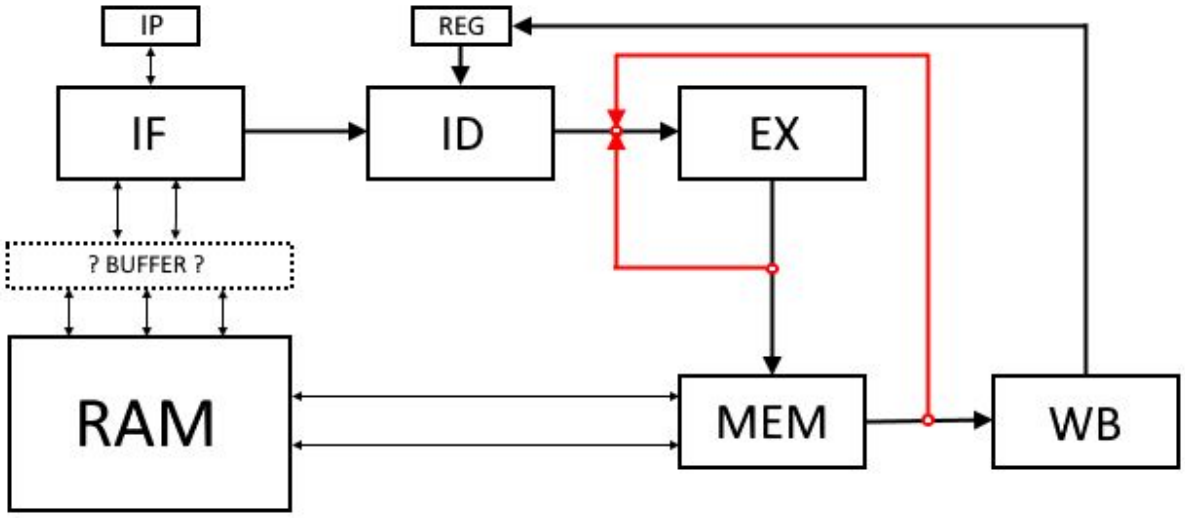
Способы решения:

1. Документация - определение такого поведения корректным, и в этом случае избежать неверной последовательности выполнения - задача разработчика
2. **NOP** - команда “ничего не делать”. У регистров хранится флаг - изменяется ли его значение еще не дошедшими до **WB** командами. И если новая команда пытается к нему обратиться,
   - **ID** вправляет **IP** на одну команду назад (чтобы остаться на том же **SUB**)
   - в **EX** в **T = 3** передается **NOP**
   - в момент, когда **ADD** дошел до **WB**, флаг с регистра R1 снимается и **ID** передает дальше **SUB**, а уже не **NOP**

T = 3	SUB	SUB	ADD	-	-
T = 4	SUB	SUB	**NOP**	ADD	-
T = 5	SUB	SUB	**NOP**	**NOP**	ADD
T = 6	SUB	SUB	**NOP**	**NOP**	**NOP**
T = 7	-	-	SUB	**NOP**	**NOP**

- Однако, минус этого способа в том, что это значительно замедляет время работы. Если каждая команда обращается к результату предыдущего, это увеличит время в 4 раза

3. **Forwarding** - способ, при котором результат выполнения предыдущей операции сохраняется и передается в предыдущие стадии конвейера, чтобы его можно было использовать. Поскольку команда на 3 раньше текущей уже на **WB**, когда текущая на **ID**, то при подаче ее на **EX**, **WB** уже завершится. Следовательно, надо знать результаты вычисления предыдущих двух.
    - Красным на схеме обозначены добавленные цепи forwarding. Если **ID** просит значение у регистра, который еще не посчитан, **EX** считывает его из них, а не из регистров.



1.2. WaR (write after read)

Это проблема, возникающая, если команда, которая должна была использовать старое значение регистра, получает на вход его измененное значение.

ADD R1 R2 R3
ADD R2 R3 R4
С условием, что первая команда почему-то выполнилась после второй. Это может случаться при параллельном вычислении, однако на MIPS такое невозможно.

1.3. WaW (write after write)

Это проблема, при которой две команды записывают значение в один и тот же регистр, и это происходит не в том порядке.

ADD R1 R2 R3
ADD R1 R4 R5
С условием, что первая команда выполнилась после второй. Опять же, на MIPS такое невозможно, так как MIPS - довольно просто устроенный конвейер.

2. Structure hazards

Возьмем для примера код
LD
ADD
SUB
XOR

	IF	**ID**	**EX**	**MEM**	**WB**
T = 1	LD	-	-	-	-
T = 2	ADD	LD	-	-	-
T = 3	SUB	ADD	LD	-	-
T = 4	XOR	SUB	ADD	LD	-

На четвертом такте происходят:

- обращение к памяти для считывания команды **XOR**
- обращение к памяти для получения значения командой **LD**

Но на одном такте не может произойти более одного обращения к памяти.
Эта проблема обходится с помощью пропуска одного **IF** с помощью **NOP**. Если проходит команда, требующая обращения к памяти, то через 2 такта **ID** возвращает **IP** на один назад, а **IF** получает инструкцию передать **NOP** вместо считывания следующей команды

T = 3	SUB	ADD	LD	-	-
T = 4	**NOP**	SUB	ADD	LD	-
T = 5	XOR	**NOP**	SUB	ADD	LD
T = 6	-	XOR	**NOP**	SUB	ADD

Вторым способом так же является объявление этого корректным поведением, чтобы разработчик учитывал это при написании программы. Любое добавление **NOP** увеличивает время работы.

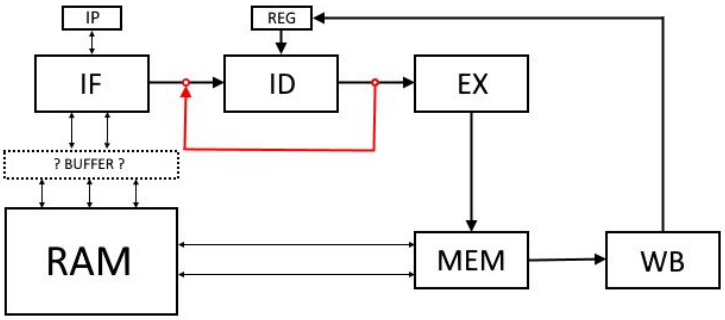
при использовании **Гарвардской архитектуры**, structure hazard не возникает, потому что инструкции и данные хранятся на разных физических устройствах и доступ осуществляется по разным каналам.

3. Control hazards

Достигается, например, кодом
ADD
JMP L1
SUB
...
L1: **XOR**

	IF	ID	EX	MEM	WB
T = 1	ADD	-	-	-	-
T = 2	JMP	ADD	-	-	-
T = 3	SUB	JMP	ADD	-	-
T = 4	XOR	SUB	JMP	ADD	-
T = 5	-	XOR	SUB	JMP	ADD

Проблема в том, что при поступлении команды **JMP** до того, когда она дойдет до **ID**, будет считана следующая после нее команда. При этом по после **JMP** должна быть выполнена команда после соответствующей метки.



ID при получении **JMP** меняет **IP** на соответствующий. Давайте сделаем **Forwarding** из **ID** в **ID**. При получении **JMP**, **ID** будет передавать, что следующая считанная команда недействительна, и при получении такого сигнала, **ID** будет передавать дальше **NOP** вместо декодированной команды.

T = 3	SUB	JMP	ADD	-	-
T = 4	XOR	**NOP**	JMP	ADD	-
T = 5	-	XOR	**NOP**	JMP	ADD

Таким образом, следующая после **JMP** команда, которую не нужно выполнять, игнорируется.

в MIPS эта проблема решена документацией, поэтому чтобы приведенный код работал правильно, необходимо поставить **JMP** на одну операцию выше, чем нужно, то есть
JMP L1
ADD
SUB
…
L1: **XOR**

4. FAQ

- Написать код, вызывающий каждый из hazard’ов

- Объяснить, почему WaR и WaW на MIPS не случаются