

# Архитектура компьютера. Конвейер

IPL/IPP Rubtsov Anton 2022

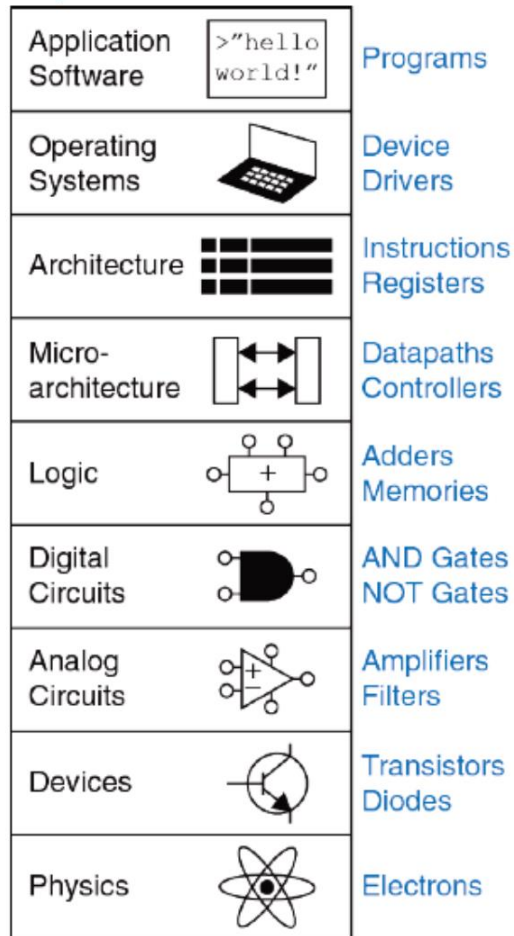


intel<sup>®</sup>

# План

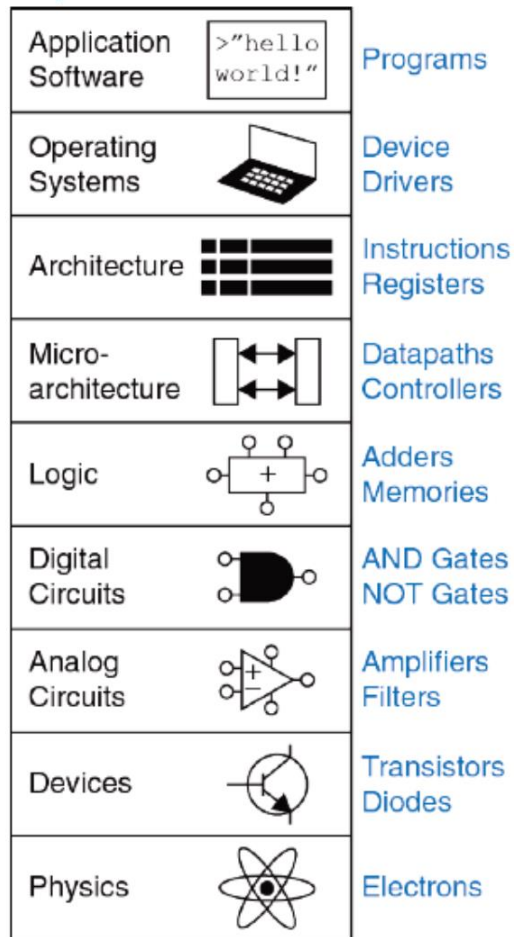
- ▶ Архитектура компьютера.
- ▶ Архитектура набора команд.
- ▶ Принцип процессорных вычислений.
- ▶ Современные процессоры.
- ▶ Конвейер.
- ▶ Параллелизм уровня инструкций. Суперскалярность.

# Уровни абстракции



- Физика (Physics) – физические явления, такие как поведение заряженных частиц, квантовые эффекты и электромагнетизм.
- Схемотехнические элементы (Devices) – преимущественно полупроводниковые элементы, такие как транзисторы.
- Аналоговые схемы (Analog Circuits) – это такие схемы, в которой сигналы могут существовать в непрерывном диапазоне величин и каждая из них одинаково значима (усилители, генераторы, преобразователи сигналов и фильтры)
- Цифровые схемы (Digital Circuits) – это схемы, предназначенные для преобразования и обработки сигналов, изменяющихся по закону дискретной функции. Здесь – уровень логических вентилей.
- Логический уровень (Logic) – комбинированная логика, объединяющая набор логических вентилей в функциональные логические устройства, такие как суматоры и т.п.

# Уровни абстракции



- Микроархитектура (Micro-Architecture) – объединение функциональных логических устройств в вычислительный тракт, выполняющий определённые команды.
- Архитектура набора команд (Architecture) – описание вычислительного устройства с точки зрения программиста, как некоторого набора ресурсов и команд.
- Операционная система (Operating Systems) – управление операциями нижнего уровня, такие как доступ к памяти и периферии и тд.
- Прикладное программное обеспечение (Application Software) – решение конкретных прикладных задач

\*Иллюстрация: Дэвид М. Харрис и Сара Л. Харрис. Цифровая схемотехника и архитектура компьютера

# Архитектура набора команд

На уровне архитектуры набора команд определяются:

- Архитектура памяти
- Взаимодействие с устройствами ввода вывода
- Режимы адресации
- Регистры
- Машинные команды
- Типы внутренних данных
- Обработчики прерываний и исключительных состояний

На уровне инструкции предоставляемые архитектурой набора команд можно разделить на:

- Системные
- Управления
- Передачи данных
- Обработки данных

C++ source #1 X

C++ ▾

```

1
2 unsigned int scalar_multiplication(unsigned int a[16], unsigned int b[16])
3 {
4     unsigned int result = 0.0;
5     for(unsigned int i = 0; i < 16; i++){
6         result = result + a[i] * b[i];
7     }
8
9     return result;
10 }
11

```


Код хранится в оперативной памяти\*





Инструкций обслуживающих цикл (серый цвет) : 4

Инструкций в теле цикла (розовый цвет): 12

Итого инструкций для вычислений:  $(4 + 12) * 16 = 256$

x86-64 gcc 9.2 (C++, Editor #1, Compiler #1) X





x86-64 gcc 9.2 ▾  -O0 ▾

A ▾  Output... ▾  Filter... ▾  Libraries + Add new... ▾  Add tool... ▾

```

1 scalar_multiplication(unsigned int*, unsigned int*):
2     push rbp
3     mov rbp, rsp
4     mov QWORD PTR [rbp-24], rdi
5     mov QWORD PTR [rbp-32], rsi
6     mov DWORD PTR [rbp-4], 0
7     mov DWORD PTR [rbp-8], 0
8     .L3:
9     cmp DWORD PTR [rbp-8], 15
10    ja .L2
11    mov eax, DWORD PTR [rbp-8]
12    lea rdx, [0+rax*4]
13    mov rax, QWORD PTR [rbp-24]
14    add rax, rdx
15    mov edx, DWORD PTR [rax]
16    mov eax, DWORD PTR [rbp-8]
17    lea rcx, [0+rax*4]
18    mov rax, QWORD PTR [rbp-32]
19    add rax, rcx
20    mov eax, DWORD PTR [rax]
21    imul eax, edx
22    add DWORD PTR [rbp-4], eax
23    add DWORD PTR [rbp-8], 1
24    jmp .L3
25    .L2:
26    mov eax, DWORD PTR [rbp-4]
27    pop rbp
28    ret

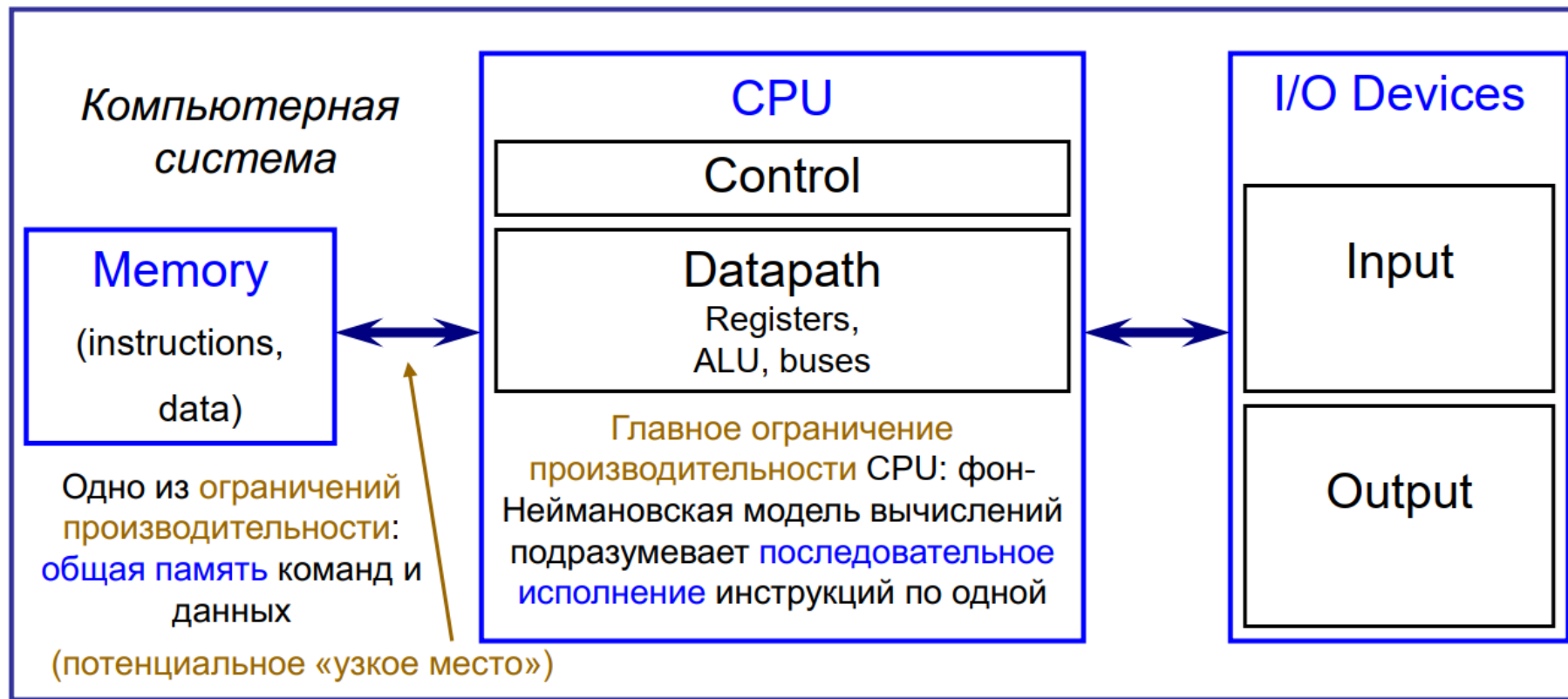
```


 Output (0/0)
 x86-64 gcc 9.2
  - 1270ms (4703B) ~269 lines filtered
 

# Топ 10 инструкций x86

Ранг	инструкция	процент от всех исполняющихся в среднем
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	<b>Всего</b>	<hr/> 96%

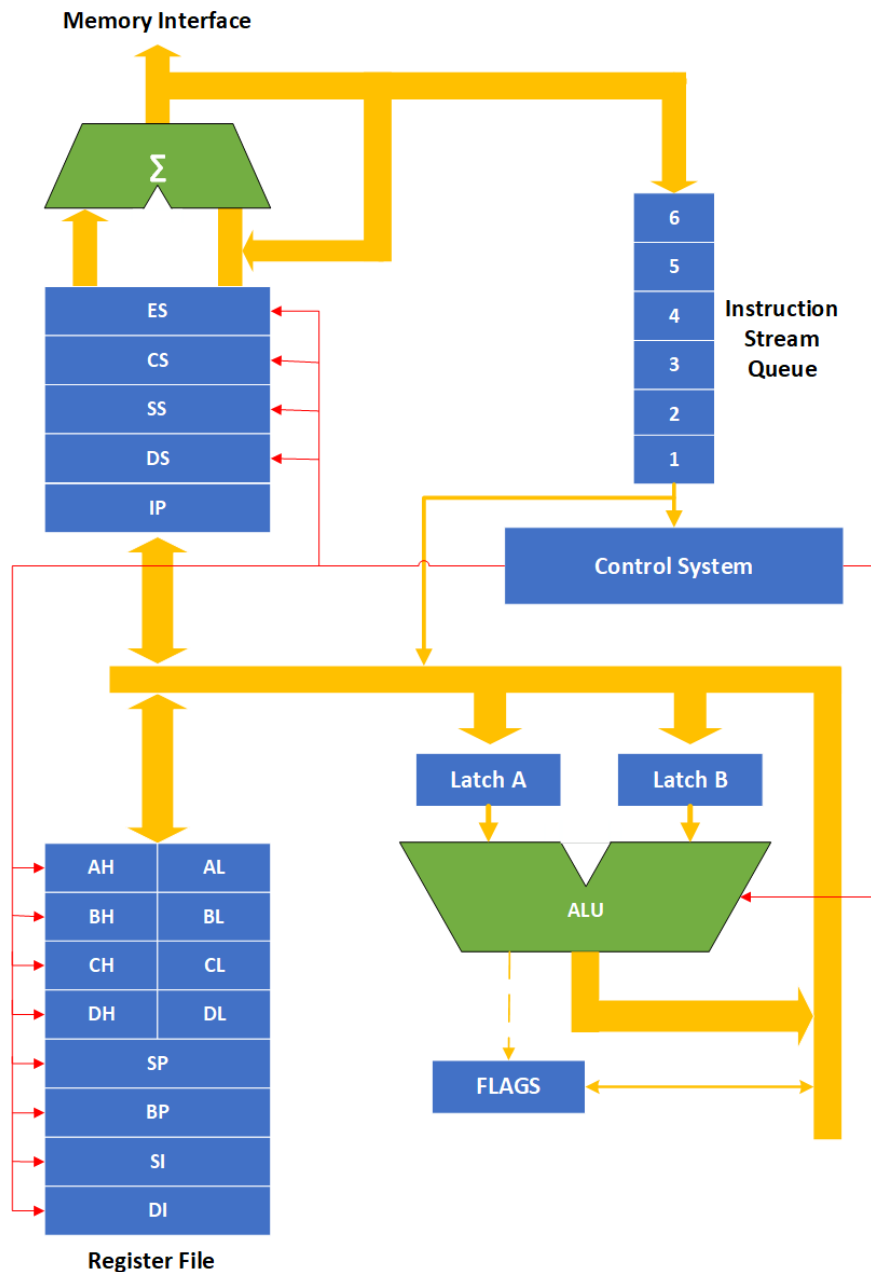
# Фон Неймановская модель компьютера



**Процессор** – программно управляемое устройство, предназначенное для обработки цифровой информации и управления процессом этой обработки.



# Архитектура Intel 8086



....
0x43
0xff
0x32
0x45
0x7e
0xe7
0x90
0x90
0x90
0x90
0x90
0x90
0x23
0xda
0xef
0x12
...

Intel 8086 можно разделить на:

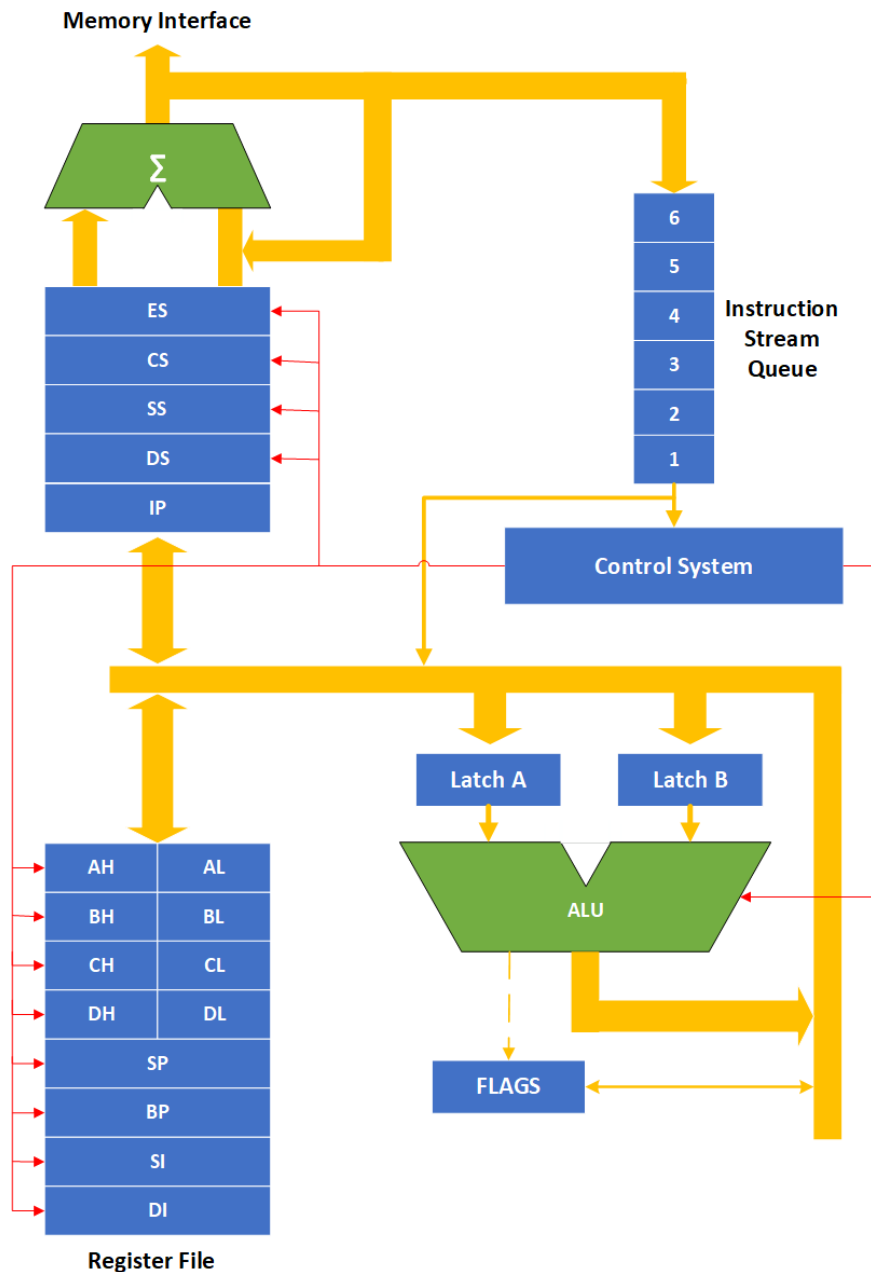
1. Устройство Работы с Шинной (BIU) (Верхняя половина)
2. Исполняющее устройство (EU) (Нижняя половина)

# Шаги выполнения инструкции CPU -



1. **Выборка Инструкции:** Загрузка инструкции расположенной по адресу записаном в Счетчике Инструкций (PC)
2. **Декодирование Инструкции**
3. **Выборка операндов**
4. **Исполнение**
5. **Запись результатов вычислений в память.**
  1. Запись в регистры
  2. Запись в оперативную память
6. **Обновление счетчика инструкций.**

# Архитектура Intel 8086

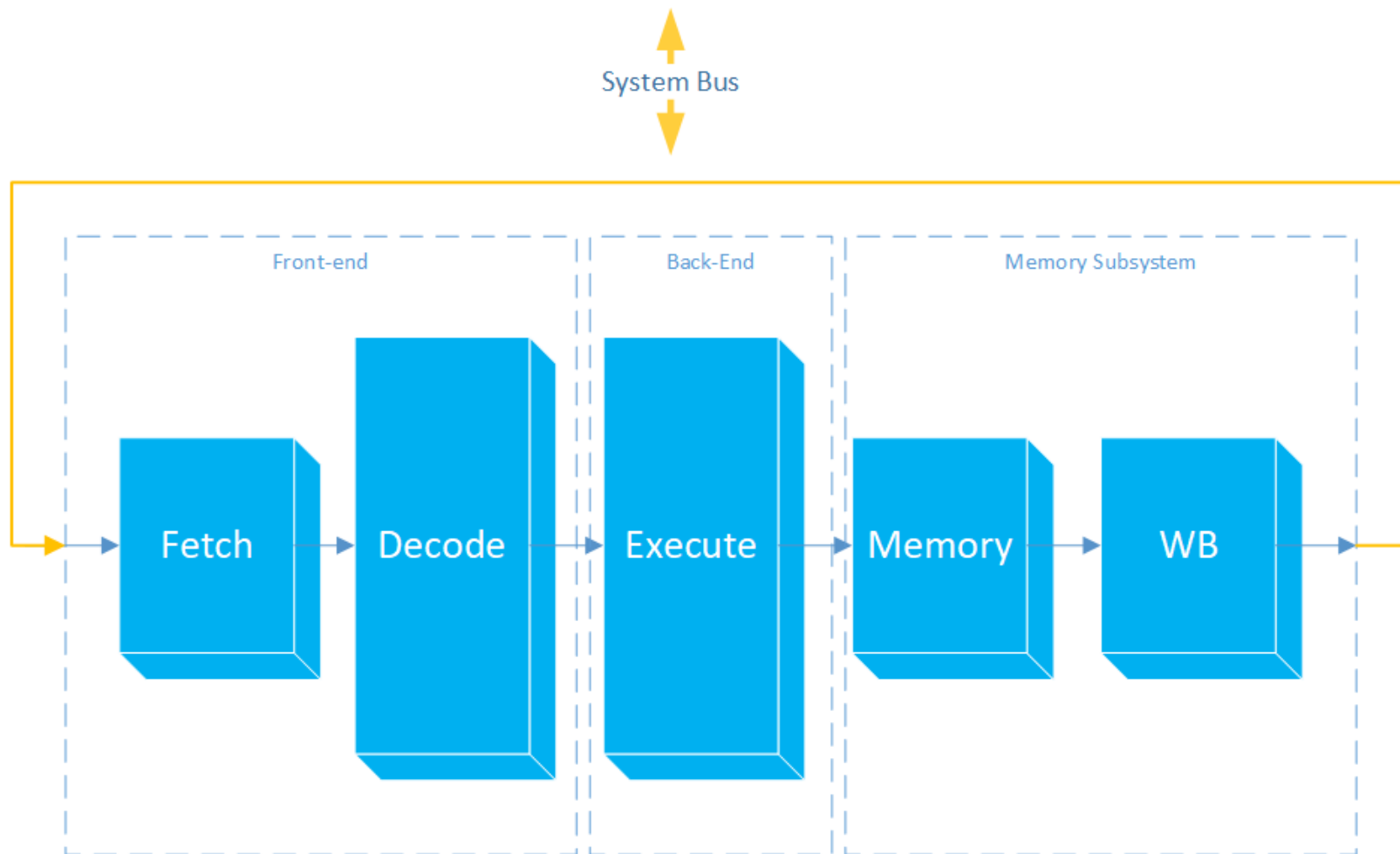


As 8086 does 2-stage pipelining (overlapping fetching and execution), its architecture is divided into two units:

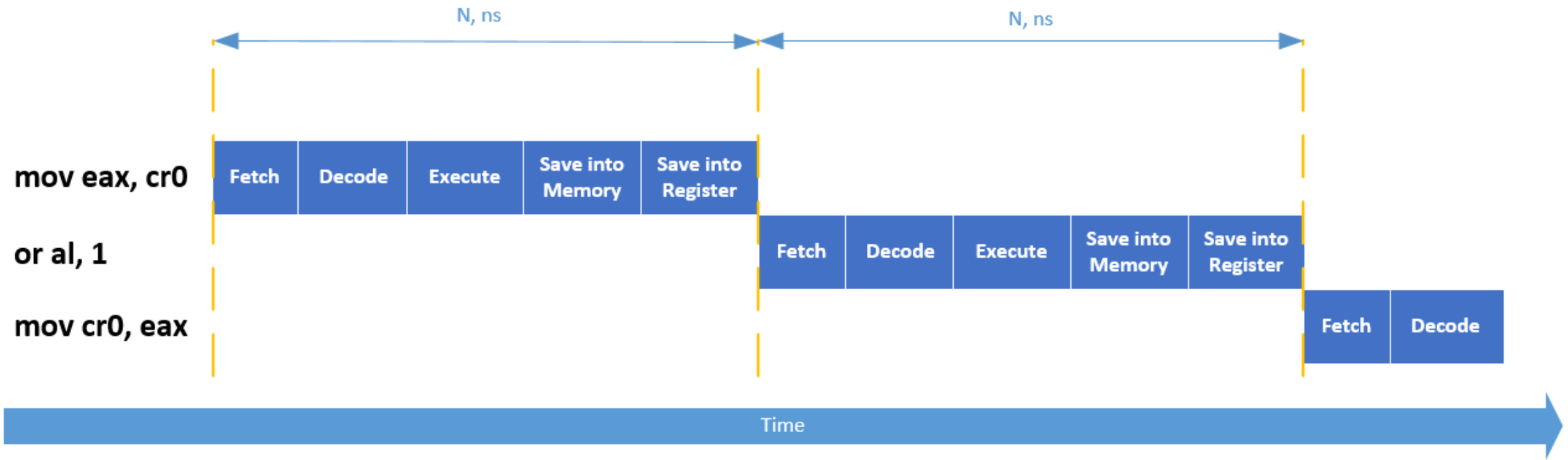
1. Bus Interfacing Unit (BIU)
2. Execution Unit (EU)

....
0x43
0xff
0x32
0x45
0x7e
0xe7
0x90
0x90
0x90
0x90
0x90
0x23
0xda
0xef
0x12
...

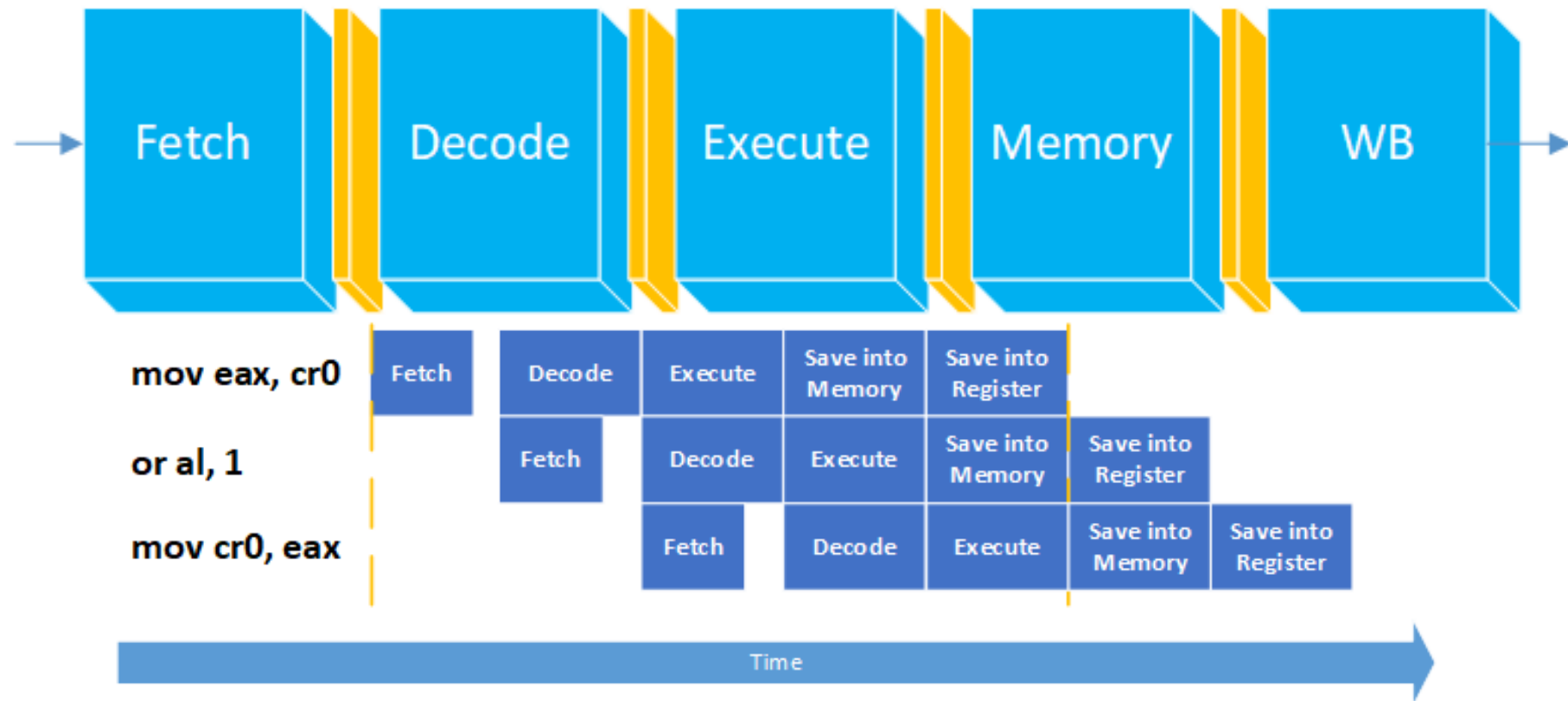
# Устройство ядра



# Стадии исполнения инструкций CPU



# Конвейер CPU



# Риски управления

Проблемы управления возникают в случае исполнения инструкций перехода. Когда во время исполнения CPU обнаруживает переход и вынужден изменить адрес следующей инструкции в РС. При этом весь конвейер должен быть очищен.

# Риски управления: Спекулятивное исполнение

В случае последовательной программы:

$$\text{CPI} = \text{CPI}_{\text{ideal}} \qquad \text{CPI}_{\text{ideal}} = 1.$$

В случае если попадают инструкции перехода:

$$\text{CPI} = \text{CPI}_{\text{ideal}} + \text{penalty} * \text{branch\_frequency}$$

Для:

$$\text{penalty} = 20 \text{ cycles}$$

$$\text{branch\_frequency} = 20\%$$

Тогда:

$$\text{CPI} = 1 + 20 * 0.2 = 5$$



# Риски управления: предсказатель переходов.

Если добавить предсказатель переходов с ошибкой предсказания (misprediction rate) 10%, тогда:

- $$\text{CPI} = \text{CPI}_{\text{ideal}} + \text{penalty} * \text{branch\_frequency} * \text{miss\_rate}$$

Для: penalty = 20 cycles, branch\_frequency = 20%

Без предсказателя	С предсказателем
$\text{CPI} = 1 + 20 * 0.2 = 5$	$\text{CPI} = 1 + 20 * 0.2 * 0.1 = 1.4$



**Увеличение производительности 357.1%**

# Предсказатель переходов

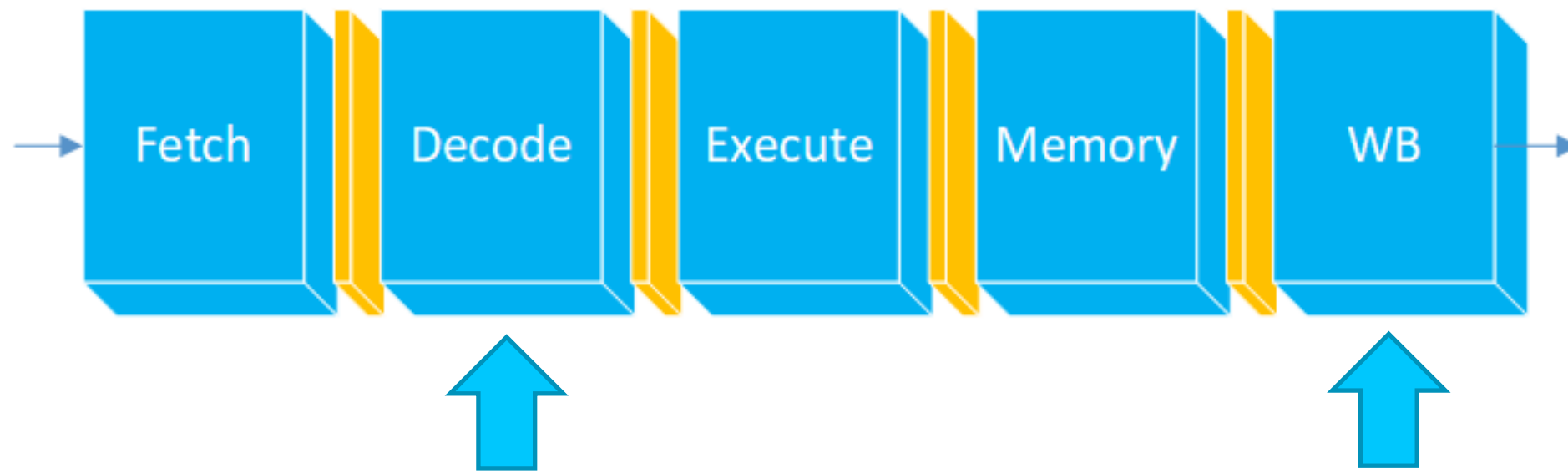
Реализован в виде наборно-ассоциативного кэша.

Хранит в себе адреса наиболее часто исполняемых инструкций управления

Строка кэша содержит:

1. Адрес куда выполнить переход;
2. Адрес инструкции управления для которой надо выполнить предсказание;
3. Тип перехода:
  - Условный
  - Безусловный направленный;
  - Безусловный ненаправленный;
  - Цикл;
  - Вызов/Возврат из функции.

# Структурные риски: Конфликт доступа к регистрам



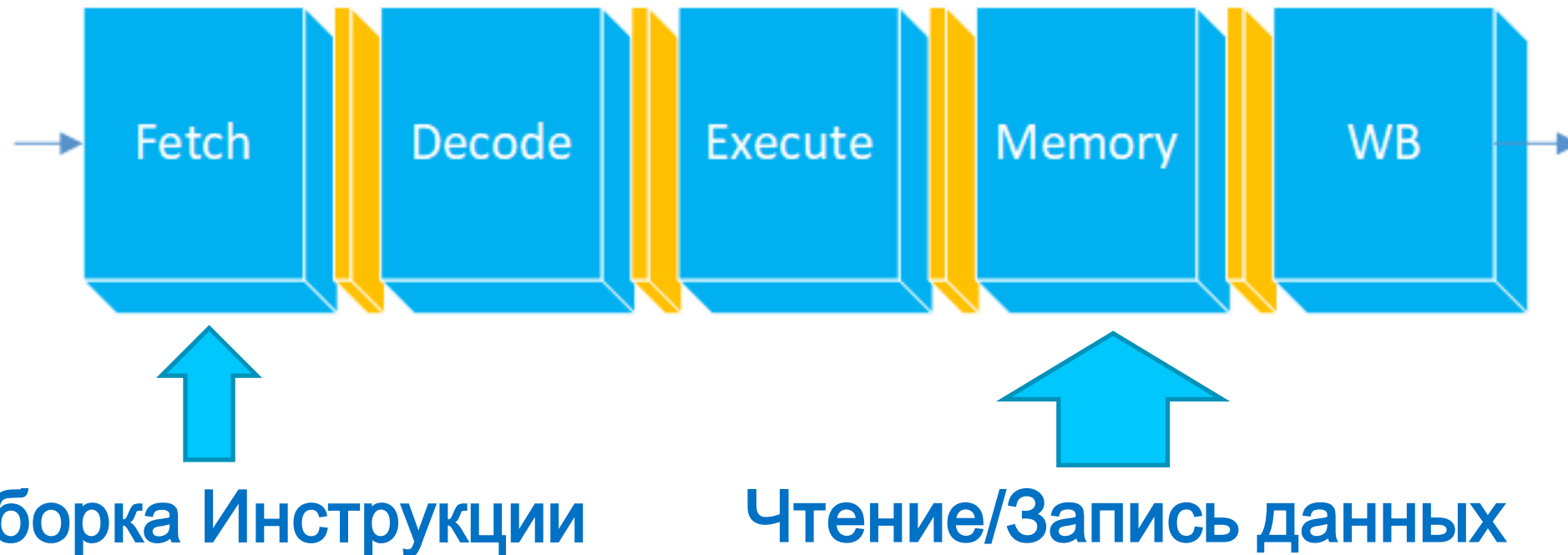
Подготовка операндов

Запись результата



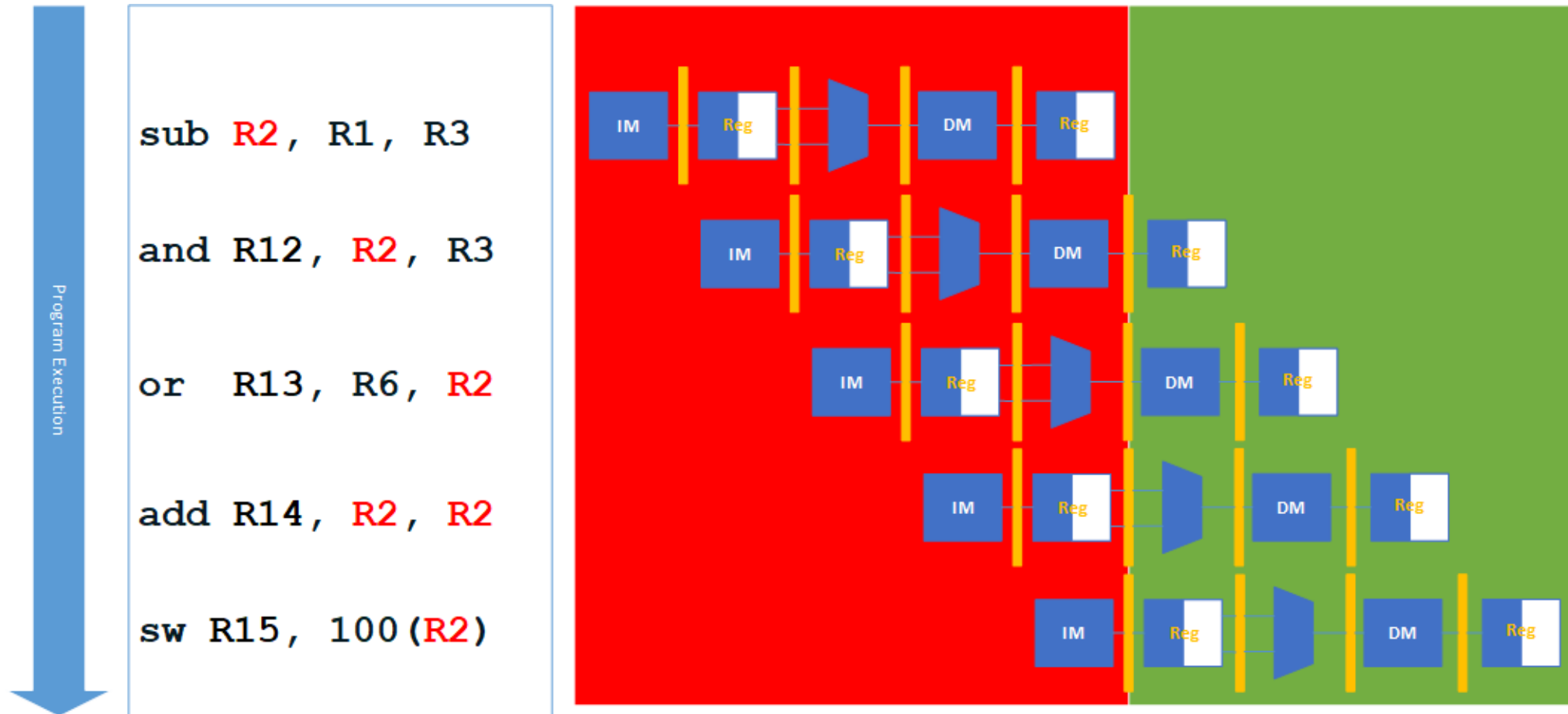
Решени: 2 порта чтения, 1 порт записи

# Структурные риски : Конфликт доступа к памяти



Решение: разделить кэш первого уровня на кэш данных и кэш инструкций

# Риск получить невалидные данные для следующей инструкции



## Решение 1: вставлять “пузыри” – приостанавливать конвейер для следующей инструкции

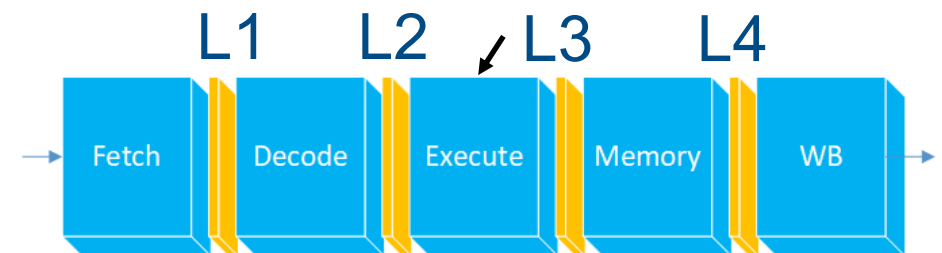


# Решение 2: Не дожидаться полного окончания записи регистра/памяти

## Прямая пересылка после стадии исполнения

if (L3.RegWrite and (L3.dst == L2.src1))  
ALUSelA = 1

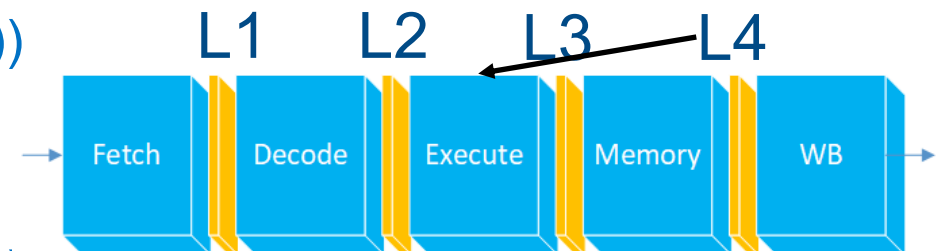
if (L3.RegWrite and (L3.dst == L2.src2))  
ALUSelB = 1



## Прямая пересылка после стадии записи в память:

if (L4.RegWrite and ((not L3.RegWrite)  
or (L3.dst == L2.src1)) and (L4.dst = L2.src1))  
ALUSelA = 2

if (L4.RegWrite and ((not L3.RegWrite)  
or (L3.dst == L2.src2)) and (L4.dst = L2.src2))  
ALUSelB = 2



# Риск исполнения инструкции для неактуальных данных сохраняется в случае если инструкция зависит от данных, которые читаются из памяти.

Пример ( все переменные находятся в памяти): :

```
a = b + c;  
d = e - f;
```

## Проблема

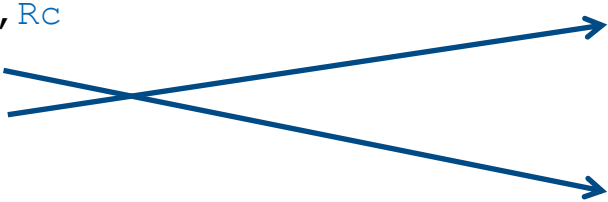
Stall

Stall

LW	Rb, b
LW	Rc, c
ADD	Ra, Rb, Rc
SW	a, Ra
LW	Re, e
LW	Rf, f
SUB	Rd, Re, Rf
SW	d, Rd

## Решение

LW	Rb, b
LW	Rc, c
LW	Re, e
ADD	Ra, Rb, Rc
LW	Rf, f
SW	a, Ra
SUB	Rd, Re, Rf
SW	d, Rd



Пока сохраняется алгоритмический порядок мы получим верный результат!



# Параллелизм уровня инструкций



Цель: Уменьшить время исполнения программы CPU

$$\text{CPU Time} = \text{duration of clock cycle} \times \text{CPI} \times \text{IC}$$



Варианты:

- Уменьшить продолжительность цикла
- Уменьшить CPI (Циклы на инструкцию)
- Уменьшить IC (Количество инструкций)



Увеличить количество этапов конвейера?

- НЕТ. Затраты на защиту от рисков описанных ранее станут слишком большими

Что же делать?



- Уменьшать CPI увеличивая внутренний параллелизм (ILP – instruction level parallelism).
  - Дублировать модули
  - Или?

# Внеочередное исполнение команд: Идея

Example:

```
(1) r1 ← r4 / r7  
(2) r8 ← r1 + r2  
(3) r5 ← r5 + 1  
(4) r6 ← r6 - r3  
(5) r4 ← r5 + r6  
(6) r7 ← r8 * r4
```

! Деление и умножение  
занимают много тактов

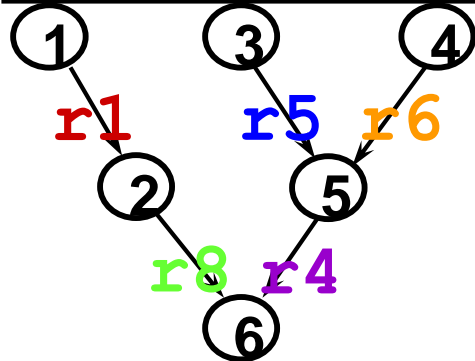
**In-order execution**

1	2	3	4	5	6
---	---	---	---	---	---

**Out-of-order execution**

1			
3	5	2	6
4			

Data Flow Graph



# Внеочередное исполнение команд: Имплементация

1. Заполнить окно инструкций (Специальная очередь инструкций после декодера)
2. Запомнить порядок
3. Распаралелить по исполнительным устройствам
4. Исполнить
5. Восстановить порядок
6. Продвинуть очередь

# Зависимости между инструкциями

# Чтение после чтения (Read-After-Read)

```
(1) r6 ← r1 / r7
(2) r8 ← r1 + r2
(3) r1 ← r5 + 1
(4) r6 ← r6 - r3
(5) r4 ← r5 + r6
(6) r7 ← r8 * r4
(7) r4 ← r5 + r6
(8) r7 ← r8 * r4
(9) r5 ← r6 * r4
```

...

```
(9) r5 ← r6 * r4
```

```
(8) r7 ← r8 * r4
```

...

Инструкция (9) исполнена до (8), r5  
содержит **верное** значение

# Чтение после записи (Read-After-Write)

(1)  $r1 \leftarrow r4 / r7$   
(2)  $r8 \leftarrow r1 + r2$   
(3)  $r1 \leftarrow r5 + 1$   
(4)  $r6 \leftarrow r6 - r3$   
(5)  $r4 \leftarrow r5 + r6$   
(6)  $r7 \leftarrow r8 * r4$   
(7)  $r4 \leftarrow r5 + r6$   
(8)  $r7 \leftarrow r8 * r4$

...

(2)  $r8 \leftarrow r1 + r2$   
(1)  $r1 \leftarrow r4 / r7$

...

Инструкция (2) исполнена до (1), r8 содержит **неправильное** значение

# Запись после чтения (Write-After-Read)

(1)  $r1 \leftarrow r4 / r7$   
(2)  $r8 \leftarrow r1 + r2$   
(3)  $r1 \leftarrow r5 + 1$   
(4)  $r6 \leftarrow r6 - r3$   
(5)  $r4 \leftarrow r5 + r6$   
(6)  $r7 \leftarrow r8 * r4$   
(7)  $r4 \leftarrow r5 + r6$   
(8)  $r7 \leftarrow r8 * r4$

...

(7)  $r4 \leftarrow r5 + r6$

(6)  $r7 \leftarrow r8 * r4$

...



Инструкция (7) исполнена до (6),  $r7$  содержит **неправильное** значение

# Запись после записи (Write-After-Write)

(1)  $r1 \leftarrow r4 / r7$   
(2)  $r8 \leftarrow r1 + r2$   
(3)  $r1 \leftarrow r5 + 1$   
(4)  $r6 \leftarrow r6 - r3$   
(5)  $r4 \leftarrow r5 + r6$   
(6)  $r7 \leftarrow r8 * r4$   
(7)  $r4 \leftarrow r5 + r6$   
(8)  $r7 \leftarrow r8 * r4$

...

(3)  $r1 \leftarrow r5 + 1$

(1)  $r1 \leftarrow r4 / r7$

...

Инструкция (3) исполнена до (1), r1  
содержит **неправильное** значение



# Спекулятивное исполнение

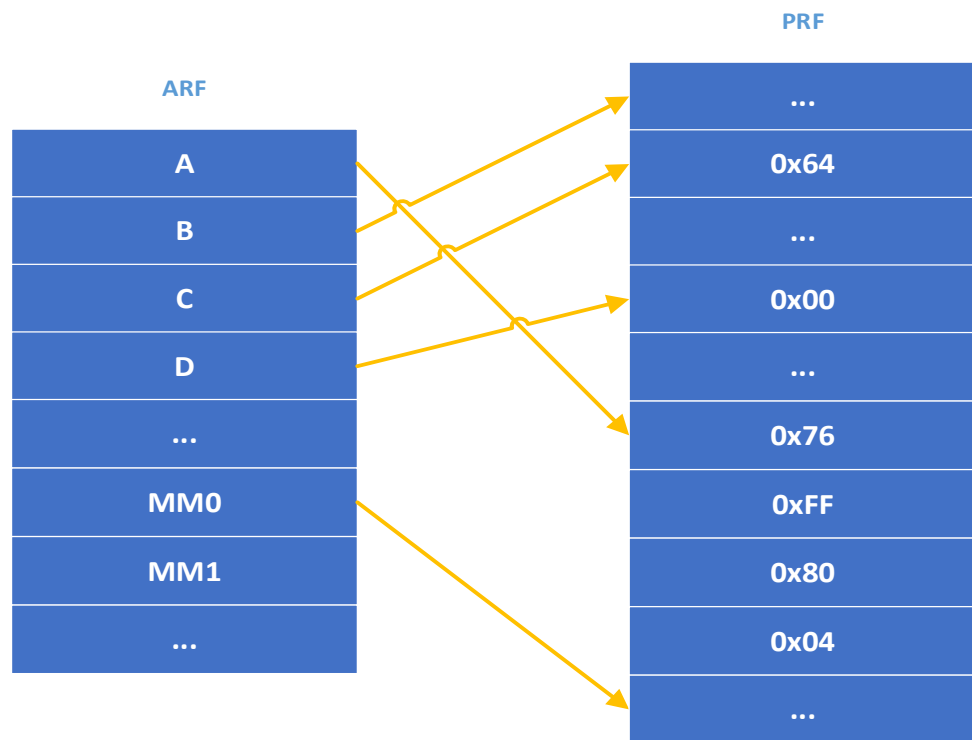
```
(1)      r1 ← r4 / r7
(2)      r8 ← r1 + r2
(3)      r1 ← r5 + 1
(4)      r6 ← r6 - r3
(5)      jcc L2
(6) L2 r4 ← r5 + r6
(7)      r7 ← r8 * r4
(8)      r4 ← r5 + r6
(9)      r7 ← r8 * r4
```

# Переименование регистров



Процессор должен:

- Иметь пул физических регистров;
- Поддерживать отображение архитектурных регистров на физические;



# Как оно работает?

Для каждой инструкции:

1. Выделить свободный физический регистр из пула физических регистров.
2. Обновить отображение архитектурных регистров на физические
3. Подменить архитектурные регистры используемые инструкцией физическими

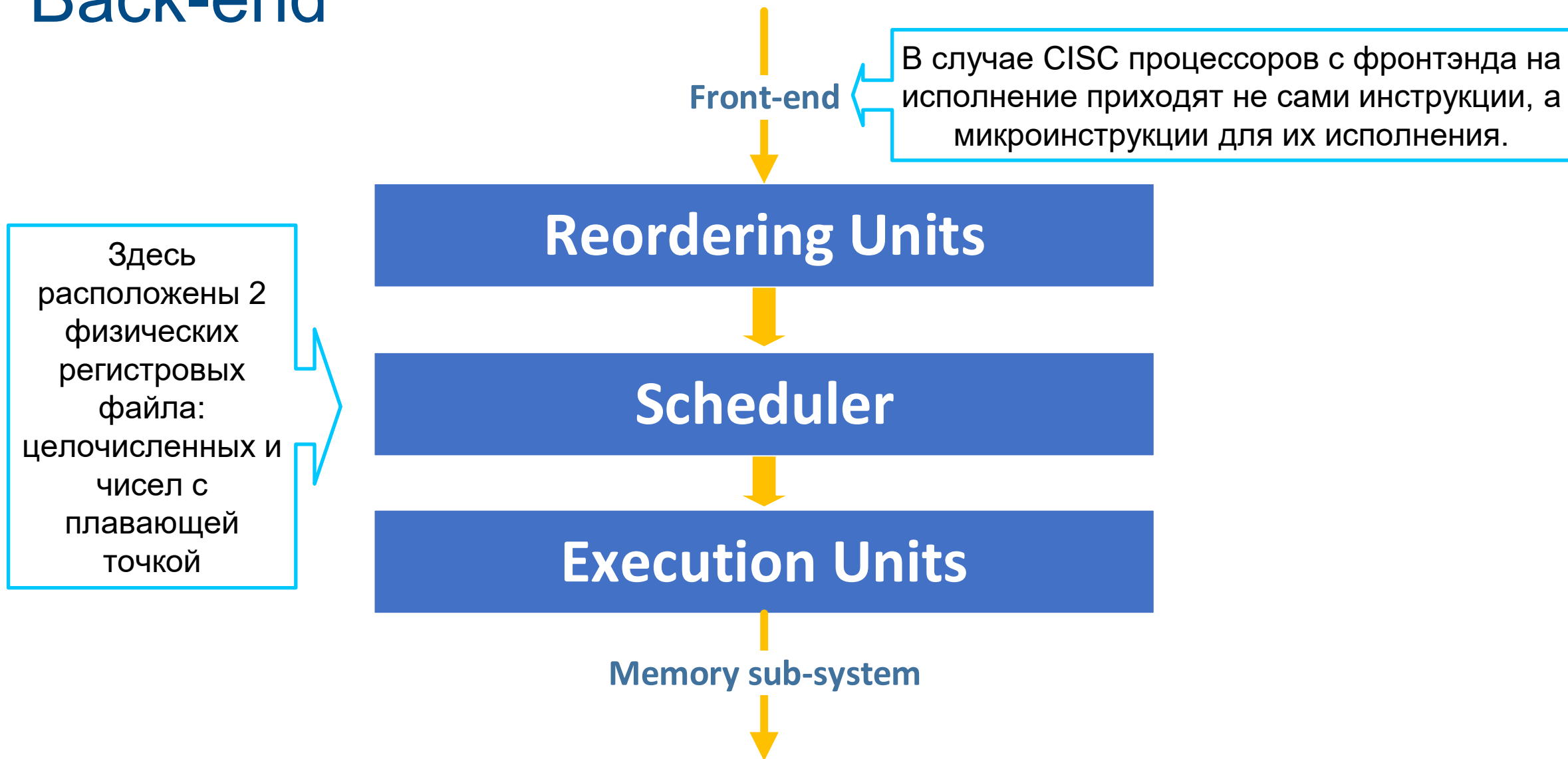
# Пример:

```
(1) r1 ← r4 / r7
(2) r8 ← r1 + r2
(3) r1 ← r5 + 1
(4) r6 ← r6 - r3
(5) r4 ← r5 + r6
(6) r7 ← r8 * r4
(7) r4 ← r5 + r6
(8) r7 ← r8 * r4
```

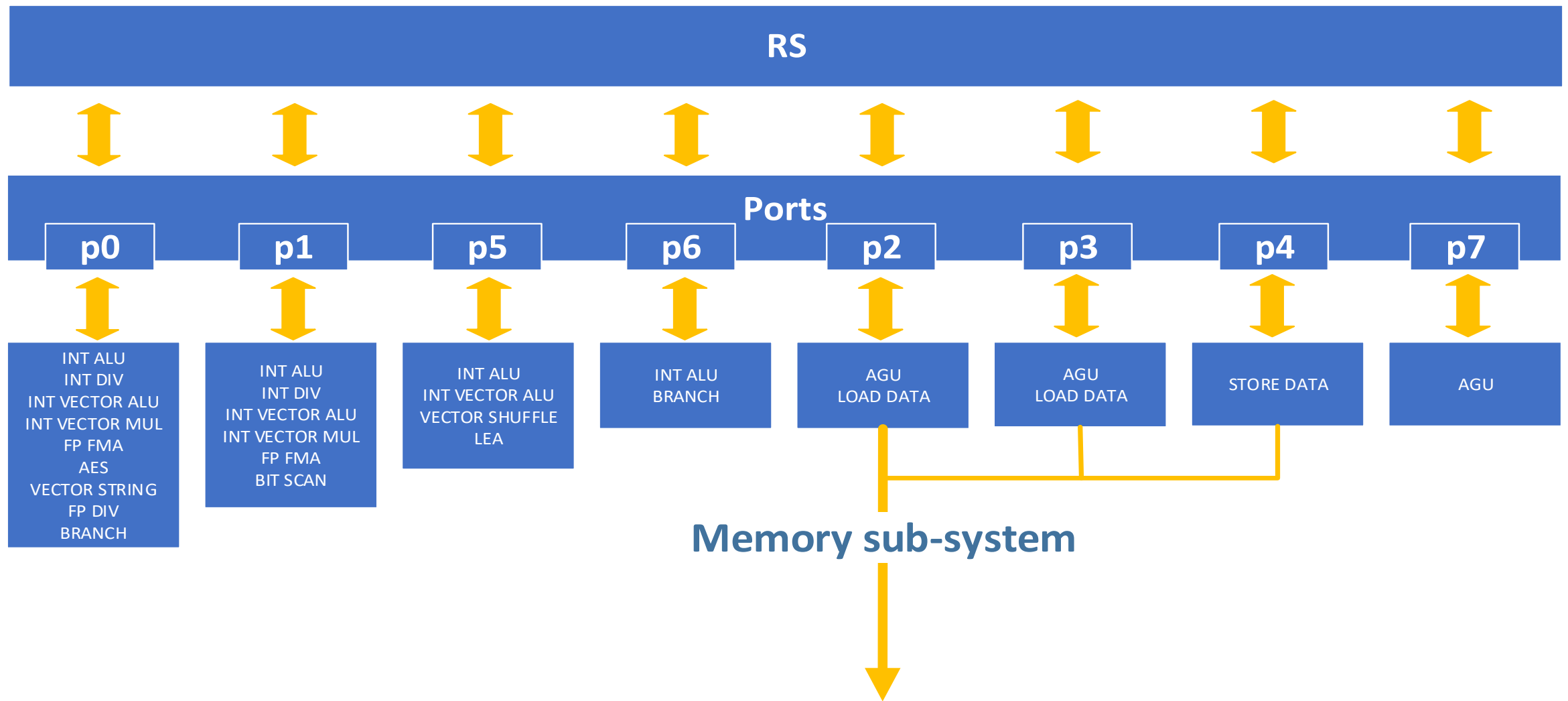
```
(1) r1 = &pr1
(2) r8 = &pr2
(3) r1 = &pr3
(4) r6 = &pr4
(5) r4 = &pr5
(6) r7 = &pr6
(7) r4 = &pr7
(8) r7 = &pr8
```

```
(1) pr1 ← r4 / r7
(2) pr2 ← pr1 + r2
(3) pr3 ← r5 + 1
(4) pr4 ← r6 - r3
(5) pr5 ← r5 + pr4
(6) pr6 ← pr2 * pr5
(7) pr7 ← r5 + pr4
(8) pr8 ← pr2 * pr7
```

# Back-end



# Исполнительные устройства (Skylake)



# Throughput & Latency



**Latency (задержка)** - количество тактов процессора через которые результат вычисления инструкции будет доступен для использования другой инструкцией.



**Throughput (пропускная способность)** - количество тактов процессора необходимое на исполнение инструкции. Инструкция с пропускной способностью в 2 такта может занять исполнительный блок на такое количество циклов, что заблокирует выполнение “следующей” инструкции, требующей этого исполнительного блока. Только после того, как инструкция выполнена исполнительным блоком, может поступить следующая инструкция.

**Pipeline Width (Ширина конвейера)** – максимальное количество операций, которые могут покинуть конвейер за 1 такт

# Нотация инструкции

## Skylake

Inst	Latency	Throughput	Uops	Ports
<u><a href="#">ADC (AL, I8)</a></u>	[1;2]	0.50 / <u><a href="#">1.00</a></u>	<u><a href="#">2</a></u>	<u><a href="#">1*p0156+1*p06</a></u>
<u><a href="#">SHR (R16, 1)</a></u>	<u><a href="#">1</a></u>	0.50 / <u><a href="#">0.50</a></u>	<u><a href="#">1</a></u>	<u><a href="#">1*p06</a></u>

<https://uops.info/table.html>

1\*p0156+1\*p06:

- Колличество микроопераций: 2
- Первая микрооперация может исполниться на портах: 0, 1, 5, 6
- Вторая микрооперация может исполниться на портах : 0, 6



# Параметры трактов внеочередного исполнения. Что учитывать?

Feature	Sandy Bridge	Haswell	SkyLake	IceLake
Out-of-order Window	168	192	224	352
Scheduler Entries	54	60	97	?
Integer Register File	160	168	180	?
FP Register File	144	168	168	?
Allocation Queue	28	56	64	?

# Resources

- Agner Fog: “The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers”
- <https://en.wikichip.org/>
- Etc. 😊