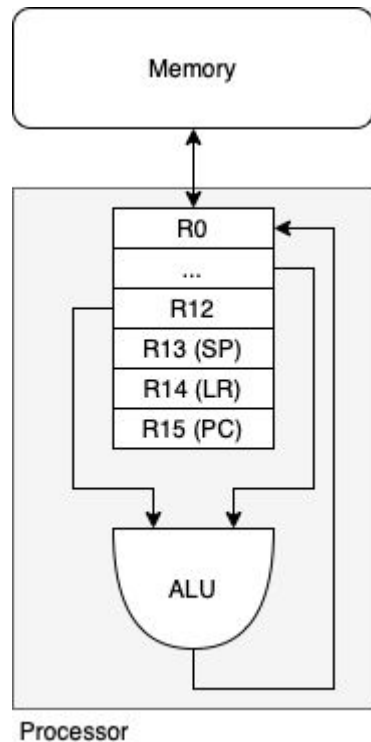
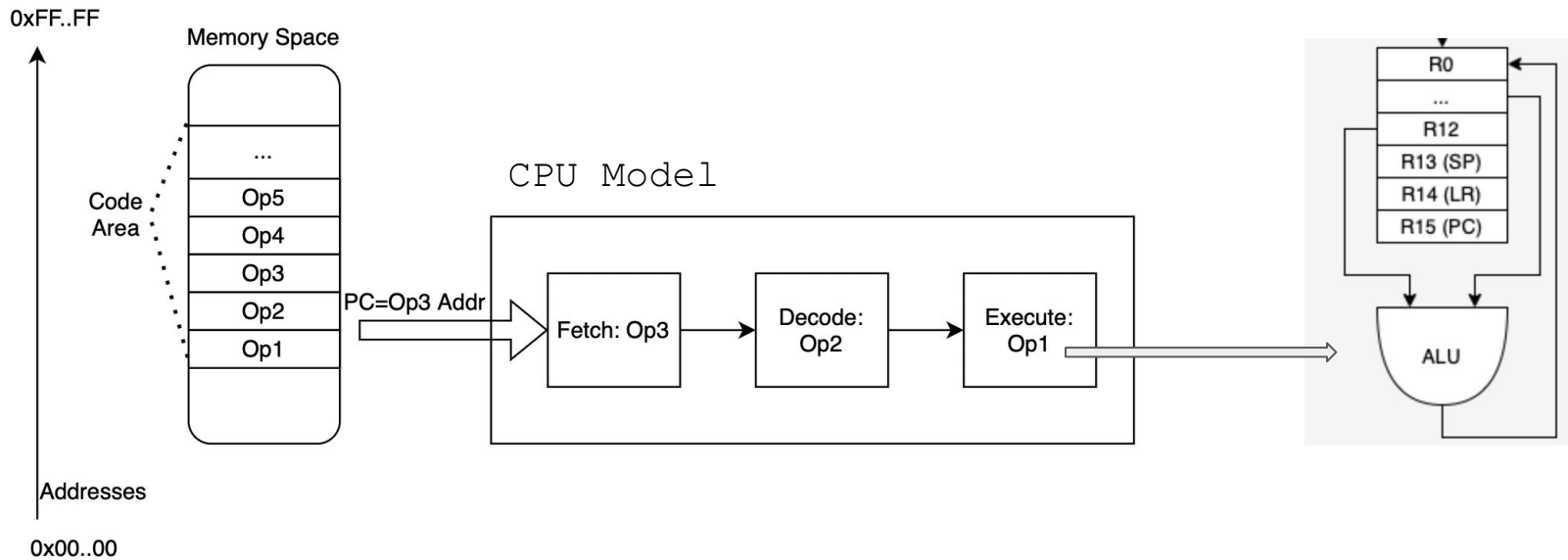


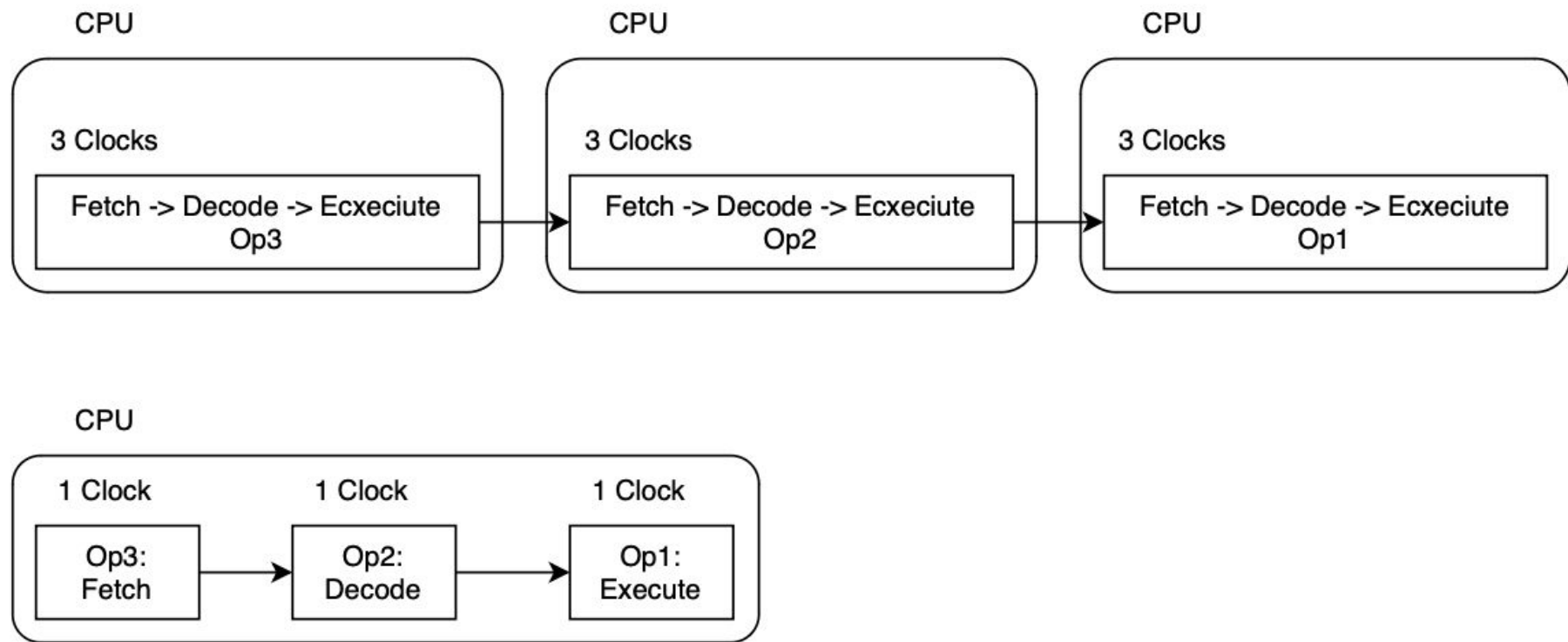
# Архитектура ARM Cortex-M и ее сравнение с другими ARM архитектурами



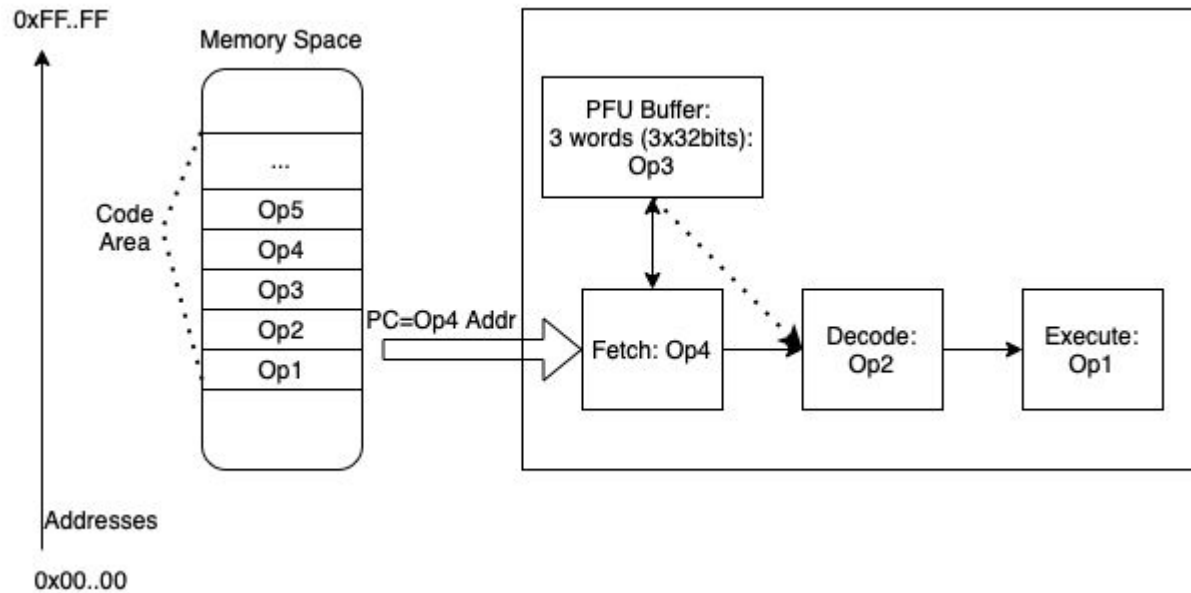
- SP - Stack Pointer Register  
(Вершина стека)
- LR - Link Register (Адрес возврата)
- PC - Program Counter Register  
(Адрес следующей выполняемой инструкции)




- Fetch: Instruction fetched from memory
- Decode: Decoding of registers used in instruction
- Execute: Registers read, ALU operation, write registers back to register bank
- **PC register** always contains the address of the instruction currently being fetched



- Prefetch Unit (PFU) buffer, zero-wait flash or 64-bit flash bus



Fetch	Decode	Execute
movs r1, #1		
bl label1	movs r1, 1	
and r0, r0, r1	bl label 1	movs r1, 1
movs r2, 1	and r0, r0, r1	bl label 1
		
sub r0, r0, r1	stall	stall
	sub r0, r0, r1	
		sub r0, r0, r1

```

func:
...
movs r1, #1
bl label1
and r0, r0, r1
movs r2, 1
...
label1:
sub r0, r0, r1
...

```


Как улучшить ?

Определять адрес для prefetching  
следующей инструкции как можно  
раньше!

\*на ARM9TDMI так и было;

\*на cortex-m0 так;

*Flush Pipeline при выполнении branch инструкции* 6

Fetch	Decode	Execute
movs r1, #1		
bl label1	movs r1, 1	
and r0, r0, r1	bl label 1	movs r1, 1
		
sub r0, r0, r1	stall	bl label 1
	sub r0, r0, r1	stall
		sub r0, r0, r1

```

func:
...
movs r1, #1
bl label1
and r0, r0, r1
movs r2, 1
...
label1:
sub r0, r0, r1
...

```

- Что если у нас branch с условием ? Например: Перейти, если предыдущая операция сравнения закончилась как равно (equal) - *BLEQ*;

Не Перешли по branch, без PFU буфера, zero-wait flash or 64-bit flash bus

Fetch	Decode	Execute
<code>sub r0, r0, r1</code>	<code>bleq func2</code>	<code>cmp r1, r0</code>
<code>adds r0, r0, #20</code>	<code>sub r0, r0, r1</code>	<code>bleq func2</code>
<code>sub r2, r2, r0</code>	<b>stall because of reload</b>	<code>sub r0, r0, r1</code>
	<code>sub r2, r2, r0</code>	<b>stall because of reload</b>
		<code>sub r2, r2, r0</code>

```
func2:
    adds r0, r0, #20
    sub r1, r1, r2
    ...

func:
    ...
    cmp r1, r0 ;compare R0 and R1, set APSR flags
    bleq func2 ;branch if equal
    sub r0, r0, r1
    sub r2, r2, r0
    ...
```

From <https://developer.arm.com/documentation/ddi0337/e/BABBCJII> :

Non-taken branches are 1 cycle total . Taken branches with an immediate are normally 1 cycle of pipeline reload (2 cycles total) .



- С PFU буфером

Не Перешли по branch, zero-wait flash or 64-bit flash bus

Fetch	Decode	Execute
sub r2, r2, r0	bleq func2	cmp r1, r0
adds r0, r0, #20	sub r0, r0, r1	bleq func2
	sub r2, r2, r0 уже запрефетчена из PFU буфера	sub r0, r0, r1
		sub r2, r2, r0

```
func2:
    adds r0, r0, #20
    sub r1, r1, r2
    ...

func:
    ...
    cmp r1, r0 ;compare R0 and R1, set APSR flags
    bleq func2 ;branch if equal
    sub r0, r0, r1
    sub r2, r2, r0
    ...
```

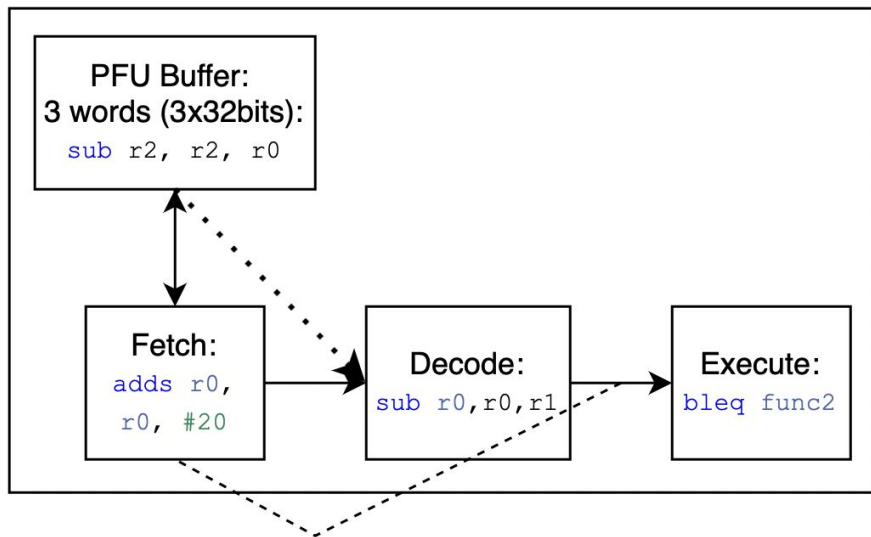
From <https://developer.arm.com/documentation/ddi0337/e/BABBCJII> :

Non-taken branches are 1 cycle total. Taken branches with an immediate are normally 1 cycle of pipeline reload (2 cycles total) .

ARM ommunity answer:

<https://community.arm.com/support-forums/f/architectures-and-processors-for-um/3190/cortex-m3-pipeline-stages-branch-prediction>

- Что если у нас branch с условием ? Например: Перейти, если предыдущая операция сравнения закончилась как равно (equal) - *BLEQ*;



Speculative fetch instruction from func2 address label

```
func2:
  adds r0, r0, #20
  sub r1, r1, r2
  ...

func:
  ...
  cmp r1, r0 ;compare R0 and R1, set APSR flags
  bleq func2 ;branch if equal
  sub r0, r0, r1
  sub r2, r2, r0
  ...
```

При условном branch процессор спекулятивно префетчит инструкции по адресу на который должен быть сделан бранч. И если условие не срабатывает, то эти инструкции будут отброшены (удалены из pipeline до Execute).

- Что если у нас branch с условием ? Например: Перейти, если предыдущая операция сравнения закончилась как равно (equal) - *BLEQ*;

Перешли по **branch**, zero-wait flash or 64-bit flash bus

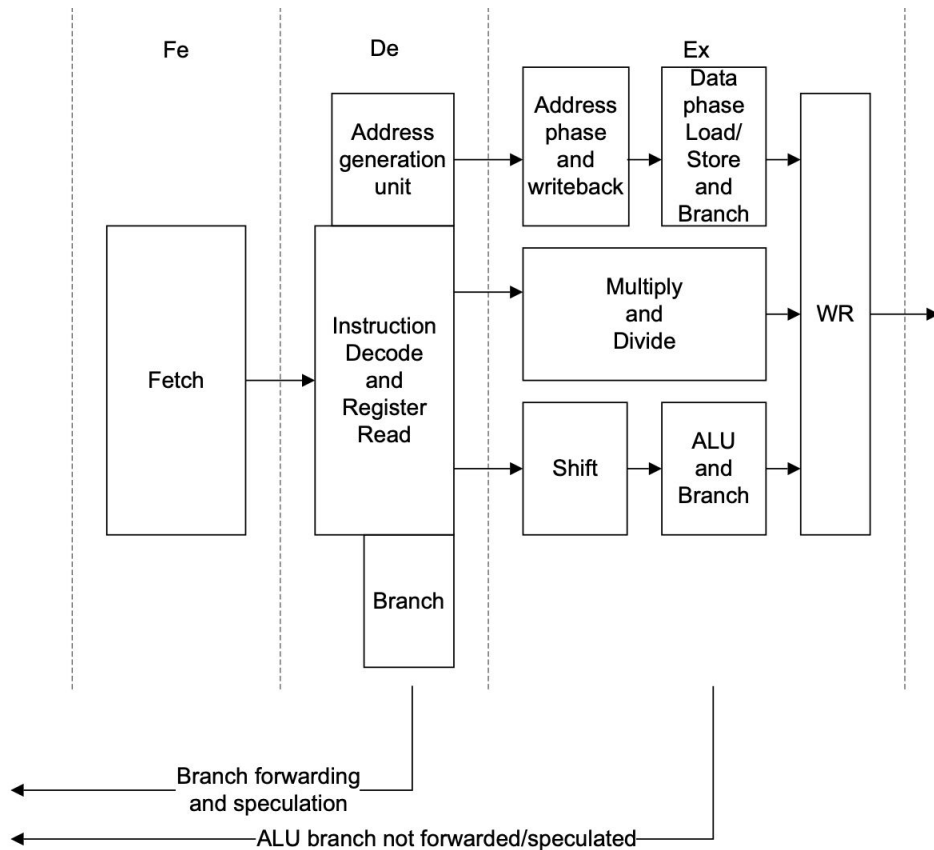
Fetch	Decode	Execute
<code>sub r0, r0, r1</code>	<code>bleq func2</code>	<code>cmp r1, r0</code>
<code>adds r0, r0, #20</code>	<b>stall because of reload</b>	<code>bleq func2</code>
<code>sub r1, r1, r2</code>	<code>adds r0, r0, #20</code>	<b>stall because of reload</b>
	<code>sub r1, r1, r2</code>	<code>adds r0, r0, #20</code>

```
func2:
    adds r0, r0, #20
    sub r1, r1, r2
    ...

func:
    ...
    cmp r1, r0 ;compare R0 and R1, set APSR flags
    bleq func2 ;branch if equal
    sub r0, r0, r1
    sub r2, r2, r0
    ...
```

From <https://developer.arm.com/documentation/ddi0337/e/BABBCJII> :

Non-taken branches are 1 cycle total . Taken branches with an immediate are normally 1 cycle of pipeline reload (2 cycles total) . Taken branches with register operand are normally 2 cycles of pipeline reload (3 cycles total) .

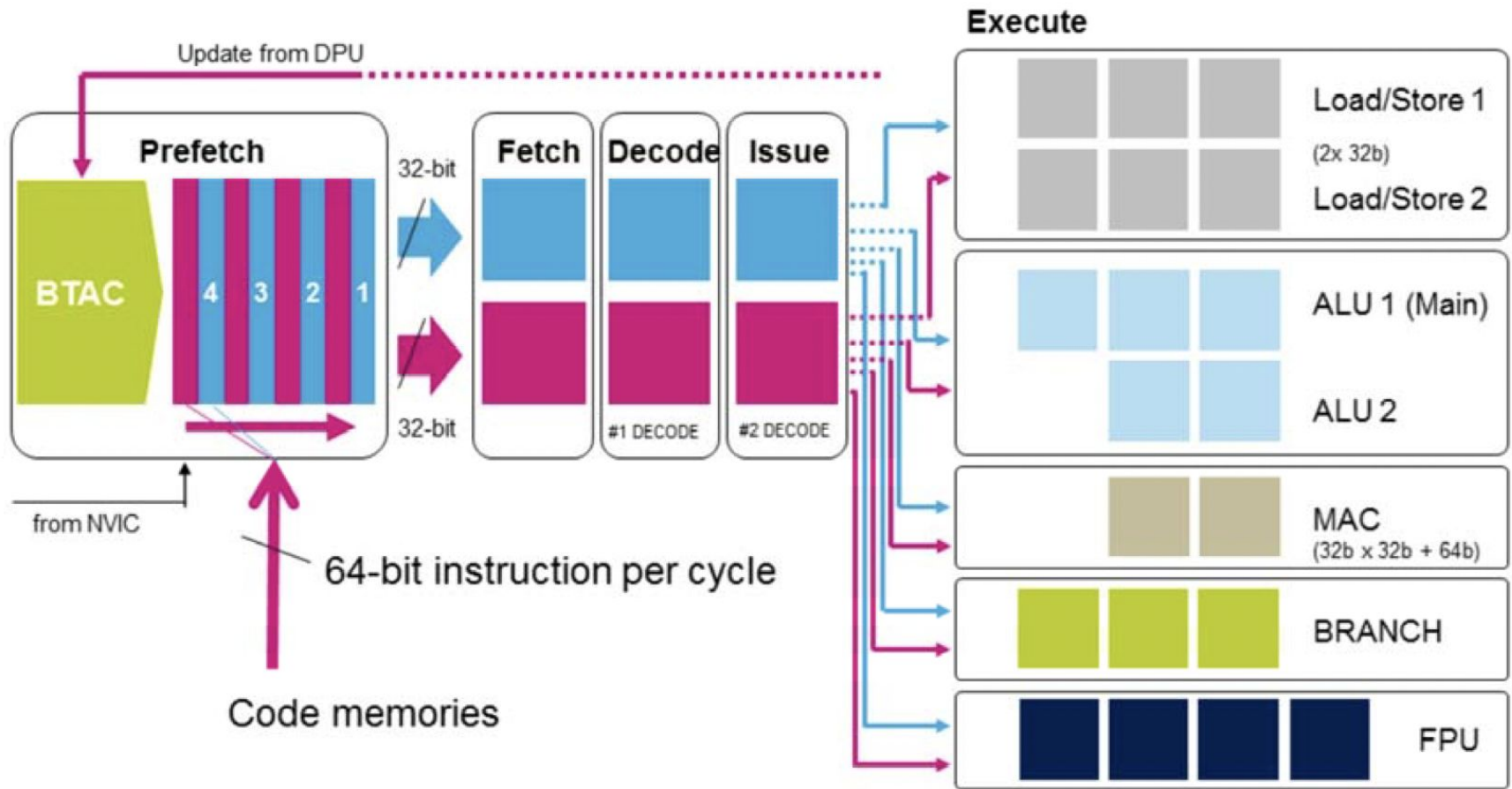


Generally, load-store instructions take two cycles: Address phase и Data Phase. But neighboring load and store single instructions can pipeline their address and data phases. This enables these instructions to complete in a **single execution cycle**.

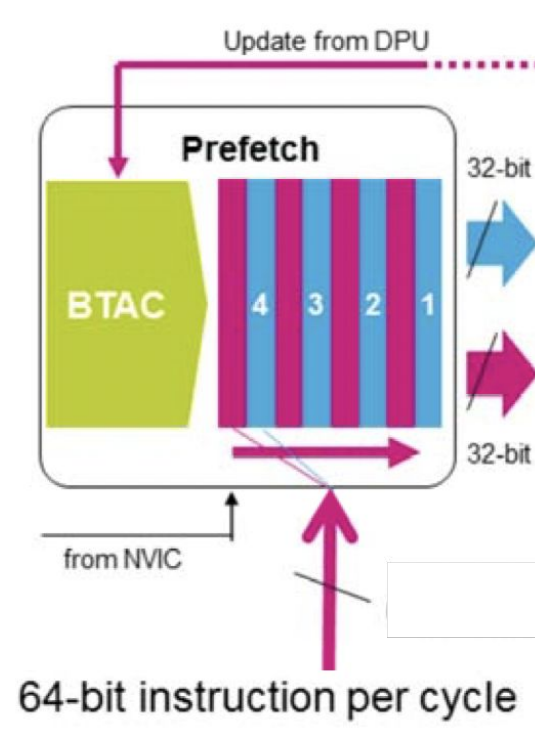
Можем ли мы выполнять load/store и ALU операции параллельно ?

для Cortex M3 нет, но да для Cortex M7

[https://www.keil.com/dd/docs/datashts/arm/cortex\\_m3/r1p1/ddi0337e\\_cortex\\_m3\\_r1p1\\_trm.pdf](https://www.keil.com/dd/docs/datashts/arm/cortex_m3/r1p1/ddi0337e_cortex_m3_r1p1_trm.pdf)




- Что нового в Prefetch ?



- Prefetch 64-bit за цикл: 2 32-битные инструкции или 4 16-битные;
- PFU буфер 4 x 64bit: 8 32-битных инструкций;
- Появился Branch Prediction: Branch Target Address Cache (BTAC) ( или Branch Target Buffer (BTB) ) - **64 entry**;

Упоминание о BTB в Cortex M7:

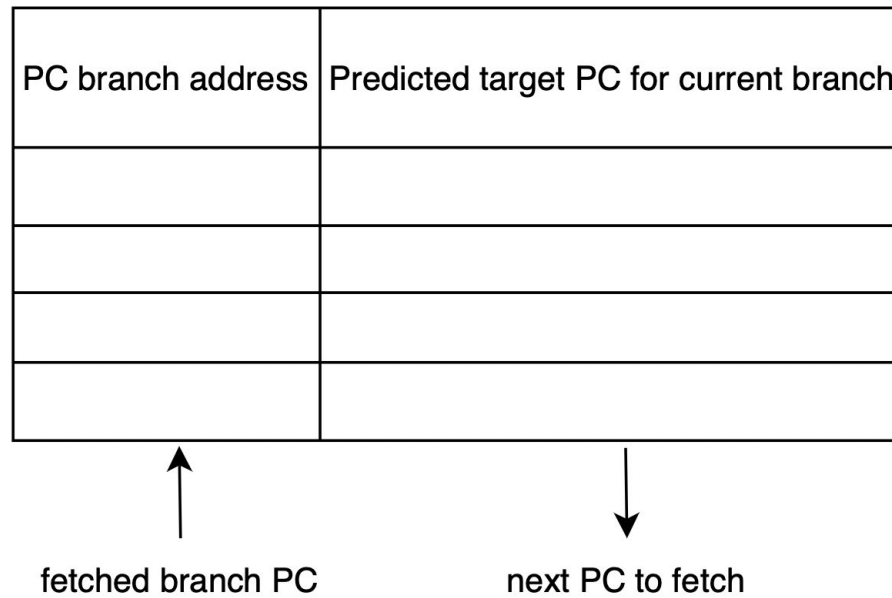
<https://semiaccurate.com/2015/04/30/arm-goes-great-detail-m7-core/>

Fetch	Decode	Execute
movs r1, #1		
bl label1	movs r1, 1	
and r0, r0, r1	bl label 1	movs r1, 1
		
sub r0, r0, r1	stall	bl label 1
	sub r0, r0, r1	stall
		sub r0, r0, r1

```

func:
...
movs r1, #1
bl label1
and r0, r0, r1
movs r2, 1
...
label1:
sub r0, r0, r1
...

```



- При первом переходе по branch в таблицу записывается адрес операции и куда (адрес перехода) этот branch; При повторном выполнении этой же операции, адрес следующей операции для prefetch определяется из таблицы;
- ВТАС расположен на первой стадии pipeline (prefetch) что позволяет предугадывать адрес следующей инструкции без простоя pipeline;
- Таблица содержит 64 вхождения;



## Cortex M7 (*bx label1* в BTAC Branch Cache)

Fetch	Decode	Decode 2 (Issue)	Execute (> 1 cycle)
<code>movs r1, #1</code>			
<code>bl label1</code>	<code>movs r1, 1</code>		
<code>sub r0, r0, r1</code>	<code>bl label 1</code>	<code>movs r1, 1</code>	
	<code>sub r0, r0, r1</code>	<code>bl label 1</code>	<code>movs r1, 1</code>
		<code>sub r0, r0, r1</code>	<code>bl label 1</code>

func:

```

...
movs r1, #1
bl label1
and r0, r0, r1
...
label1:
sub r0, r0, r1
...

```

## Cortex M3

Fetch	Decode	Execute
<code>movs r1, #1</code>		
<code>bl label1</code>	<code>movs r1, 1</code>	
<code>and r0, r0, r1</code>	<code>bl label 1</code>	<code>movs r1, 1</code>
<code>sub r0, r0, r1</code>	<b>stall</b>	<code>bl label 1</code>
	<code>sub r0, r0, r1</code>	
		<code>sub r0, r0, r1</code>

**NOTE:** Данная схема Cortex M7 pipeline не учитывает детали pipeline и суперскалярность Cortex M7 (выполнение и префетчинг 2 инструкций за такт параллельно); **Данная схема показывает только отличия branch префетчинга Cortex M7 и Cortex M3 !!!**

*Cortex M7 vs M3, выполнение branch инструкции 17*

# Cortex M7, Dual Issue

Fetch	Decode	Decode 2 (Issue)	Execute(> 1 cycle)
<div>movs r1, #1</div> <div>bl label1</div>			
<div>sub r0, r0, r1</div> <div>...</div>	<div>movs r1, #1</div> <div>bl label1</div>		
	<div>sub r0, r0, r1</div> <div>...</div>	<div>movs r1, #1</div> <div>bl label1</div>	
		<div>sub r0, r0, r1</div> <div>...</div>	<div>movs r1, #1</div> <div>bl label1</div>
			<div>sub r0, r0, r1</div> <div>...</div>

```
func:
...
movs r1, #1
bl label1
and r0, r0, r1
...
label1:
sub r0, r0, r1
...
```

operand forwarding ??  
Используем R1 из предыдущей операции

Extensive forwarding logic to minimize interlocks

```

int calculate(uint32_t repeat,
             uint32_t x, uint32_t y) {
    int max = 10;
    int res = 0;
    for (int i = 0; i < repeat; ++i) {
        int sum = x + y;
        int mul = x * y;

        res += sum + mul;
        x = (sum < max) ? sum : max;
    }
    return res;
}

void main(void) {
    int res = calculate(10, 2, 4);
}

```

Могут ли вычисления sum и mul выполняться параллельно ?

На Cortex M3 Нет;

На Cortex M7:

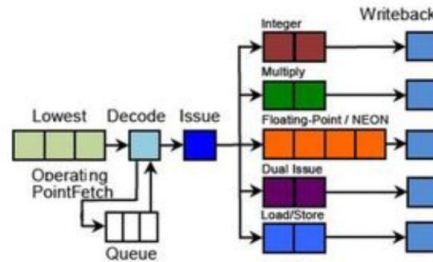
Теоретически, Да;

Практически, надо смотреть на машинный код, который генерирует компилятор;

# ARM A7 & A15

## A7

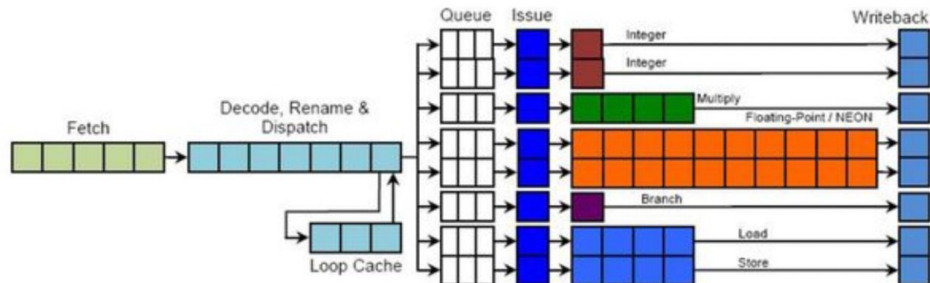
- Patial Dual Issue
- 8 stage integer pipeline



Cortex-A7 Pipeline

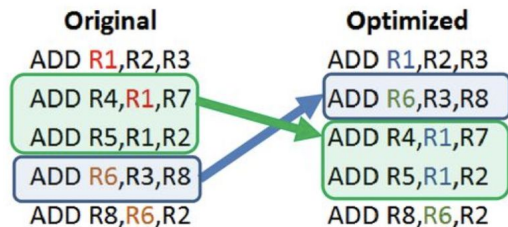
## A15

- 3 instruction issue
- 15 stage integer pipeline



Cortex-A15 Pipeline

- Out Of Order Execution – позволяет выполнять операции не в том порядке в котором они расположены в сгенерированном машинном коде. Но только при условии что процессор учитывает все зависимости, что позволит программе выдать ожидаемый результат;

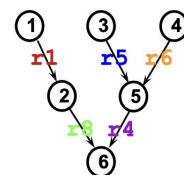


```

(1) r1 ← r4 / r7 /* assume division takes 20 cycles */
(2) r8 ← r1 + r2
(3) r5 ← r5 + 1
(4) r6 ← r6 - r3
(5) r4 ← r5 + r6
(6) r7 ← r8 * r4

```

Data Flow Graph



In-order execution

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

In-order (2-way superscalar)

1	2	4	5	6
	3			

Out-of-order execution

1					
3	5		2	6	
4					

- Процессор ищет зависимости в последовательных операциях и если в очереди есть операции которые не зависят друг от друга начинает выполнять их;

- RAW (Read After Write) Dependency:

```
LDR R1, [R2]      ; Write R1
LDR R3, [R1]      ; Read R1
```

- WAR (Write After Read) Dependency:

```
LDR R1, [R2]      ; Read R2
ADD R2, R3, R4     ; Write R2
```

- WAW (Write After Write) Dependency:

```
LDR R1, [R2]      ; Write R1
LDR R1, [R3]      ; Write R1
```

*Устраняются с помощью  
Register Renaming*

## REGISTER RENAMING

- WAR (Write After Read) Dependency:

```
LDR R1, [R2]      ; Read R2
ADD R2-shadow, R3, R4 ; Write R2 заменена на Write R2-shadow
```

- WAW (Write After Write) Dependency:

```
LDR R1-shadow, [R2] ; Write R1 заменен на Write R1-shadow
LDR R1, [R3]        ; Write R1
```

- ARM - **weakly-ordered** model of memory. In general terms, this means that the order of memory accesses is not required to be the same as the program order for load and store operations;
- X86/64 - **TSO (Total Store Order)** model. Only "store, load" pattern can be reordered.



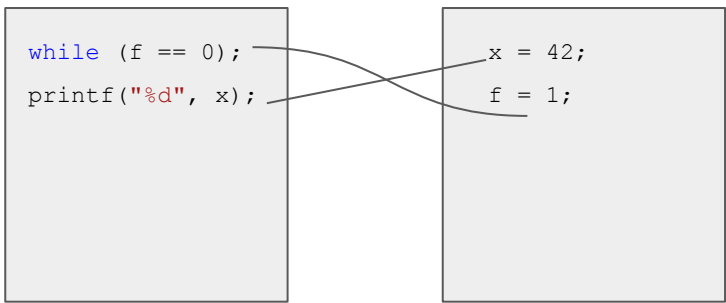
Type	Alpha	ARMv7	MIPS	RISC-V		PA-RISC	POWER	SPARC			x86 [a]
				WMO	TSO			RMO	PSO	TSO	
Loads can be reordered after loads	Y	Y	depend on implementation	Y		Y	Y	Y			
Loads can be reordered after stores	Y	Y		Y		Y	Y	Y			
Stores can be reordered after stores	Y	Y		Y		Y	Y	Y	Y		
Stores can be reordered after loads	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y

Thread 1,  
на CPU1

```
while (f == 0);  
printf("%d", x);
```

Thread 2,  
на CPU2

```
x = 42;  
f = 1;
```



*Out Of Order Execution* может  
привести к тому что, в Thread 2:  
***f=1 запишется раньше чем x=42.***

***Что приведет к тому что Thread 1  
проскочит до printf(x) еще до того  
как x выставлен в 42 !!!***

Out Of Order Execution может переставлять запись/чтение переменных **если они не зависимы**. В однопоточной программе или на одноядерном CPU у вас не будет проблем с этим. Но в многопоточной программе это может приводить к не очевидным ошибкам !!!

Запись/чтение могут переставляться местами, например если одна переменная лежит в кэше, а другая нет. Тогда завершить операцию с той переменной что в кэше быстрее.

<https://preshing.com/20121019/this-is-why-they-call-it-a-weakly-ordered-cpu/>



Thread 1,  
на CPU1

```
while (f == 0);  
DMB  
printf("%d", x);
```

Thread 2,  
на CPU2

```
x = 42;  
DMB  
f = 1;
```

*Out Of Order Execution* может привести к тому что, в Thread 2: ***f=1*** запишется раньше чем ***x=42***.

**Что приведет к тому что Thread 1 проскочит до printf(x) еще до того как x выставлен в 42 !!!**

Чтобы пофиксить такую проблему, между операциями, которые всегда должны выполняться в правильном порядке, нужно вставить операцию Data Memory Barrier: **DMB** - гарантирует что операции доступа к памяти, расположенные до этого DMB, будут завершены до DMB:

<https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/miscellaneous-instructions/dmb--dsb--and-isb>

<https://preshing.com/20121019/this-is-why-they-call-it-a-weakly-ordered-cpu/>

store to X  
load from Y

Processor 1	Processor 2
mov [X], 1 mov r1, [Y]	mov [Y], 1 mov r2, [X]

store to Y  
load from X

```
void *thread1Func(void *param)
{
    for (;;)
    {
        sem_wait(&beginSema1); // Wait for signal

        // ----- THE TRANSACTION! -----
        X = 1;
#ifdef USE_CPU_FENCE
        asm volatile("mfence" ::: "memory"); // Prevent CPU reordering
#endif
        r1 = Y;

        sem_post(&endSema); // Notify transaction complete
    }
    return NULL; // Never returns
};
```

```
void *thread2Func(void *param)
{
    for (;;)
    {
        sem_wait(&beginSema2); // Wait for signal

        // ----- THE TRANSACTION! -----
        Y = 1;
#ifdef USE_CPU_FENCE
        asm volatile("mfence" ::: "memory"); // Prevent CPU reordering
#endif
        r2 = X;

        sem_post(&endSema); // Notify transaction complete
    }
    return NULL; // Never returns
};
```

```
while (1)
{
    // Reset X and Y
    X = 0;
    Y = 0;
    // Signal both threads
    sem_post(&beginSema1);
    sem_post(&beginSema2);
    // Wait for both threads
    sem_wait(&endSema);
    sem_wait(&endSema);
    // Check if there was a simultaneous reorder
    if (r1 == 0 && r2 == 0)
    {
        detected++;
        printf("%d reorders detected after %d iterations\n", detected, iterations);
    }
    iterations++;
}
```

<https://preshing.com/20120515/memory-reordering-caught-in-the-act/>

<https://github.com/badembed/OutOfOrderExecutionTest>

Что вынести из лекции:

- PC
- Что из себя представляет Pipeline процессора;
- Что такое Branch Prediction и Superscalar Pipeline в Cortex M7;
- Что такое Out Of Order (OOO) Execution и Register Renaming в Cortex A;

Задание:

- Разобраться что происходит в коде из предыдущего слайда - описать это в формате в каком вам удобно и прислать мне - нужно разобраться как работают семафоры для синхронизации, в каком случае печатается "reorders detected ..." и будет ли это печататься при USE\_CPU\_FENCE и почему. Полный код доступен на <https://github.com/badembed/OutOfOrderExecutionTest>

Если у вас Linux, то можно все это скомпилировать и запустить.

[https://github.com/badembed/branch\\_prediction\\_example](https://github.com/badembed/branch_prediction_example)

<https://www.youtube.com/watch?v=g-WPhYREFjk>

```
7 void BM_branch_predicted(benchmark::State& state) {
8     srand(1);
9     const unsigned int N = state.range(0);
10
11     int *in1 = (int *)malloc(N * sizeof(int));
12     int *in2 = (int *)malloc(N * sizeof(int));
13     int *cond = (int *)malloc(N * sizeof(int));
14     for (size_t i = 0; i < N; ++i) {
15         in1[i] = 3;
16         in2[i] = 3;
17         cond[i] = 1; // always true
18     }
19
20     for (auto _ : state) {
21         unsigned long a1 = 0, a2 = 0;
22         for (size_t i = 0; i < N; ++i) {
23             if (cond[i]) {
24                 a1 += in1[i];
25             } else {
26                 a1 += in2[i];
27             }
28         }
29         benchmark::DoNotOptimize(a1);
30         benchmark::DoNotOptimize(a2);
31         benchmark::ClobberMemory();
32     }
33     state.SetItemsProcessed(N*state.iterations());
34 }
```

```
7 void BM_branch_not_predicted(benchmark::State& state) {
8     srand(1);
9     const unsigned int N = state.range(0);
10
11     int *in1 = (int *)malloc(N * sizeof(int));
12     int *in2 = (int *)malloc(N * sizeof(int));
13     int *cond = (int *)malloc(N * sizeof(int));
14     for (size_t i = 0; i < N; ++i) {
15         in1[i] = 3;
16         in2[i] = 3;
17         cond[i] = rand() & 0x1; // random
18     }
19
20     for (auto _ : state) {
21         unsigned long a1 = 0, a2 = 0;
22         for (size_t i = 0; i < N; ++i) {
23             if (cond[i]) {
24                 a1 += in1[i];
25             } else {
26                 a1 += in2[i];
27             }
28         }
29         benchmark::DoNotOptimize(a1);
30         benchmark::DoNotOptimize(a2);
31         benchmark::ClobberMemory();
32     }
33     state.SetItemsProcessed(N*state.iterations());
34 }
```

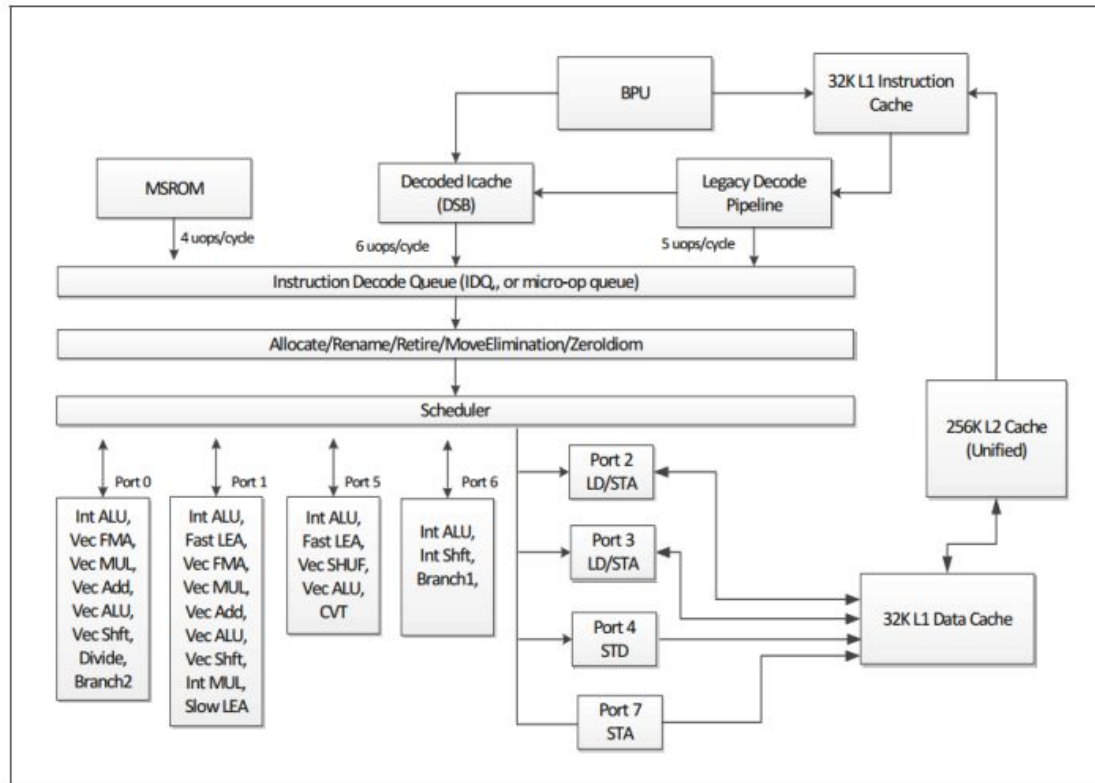


Figure 14: Block diagram of a CPU Core in the Intel Skylake Microarchitecture. © Image from [Int, 2020].