嵌入式系统设计与应用

第二章 ARM处理器组成结构（1）

Pipeline & Memory hierarchy

西安交通大学电信学院

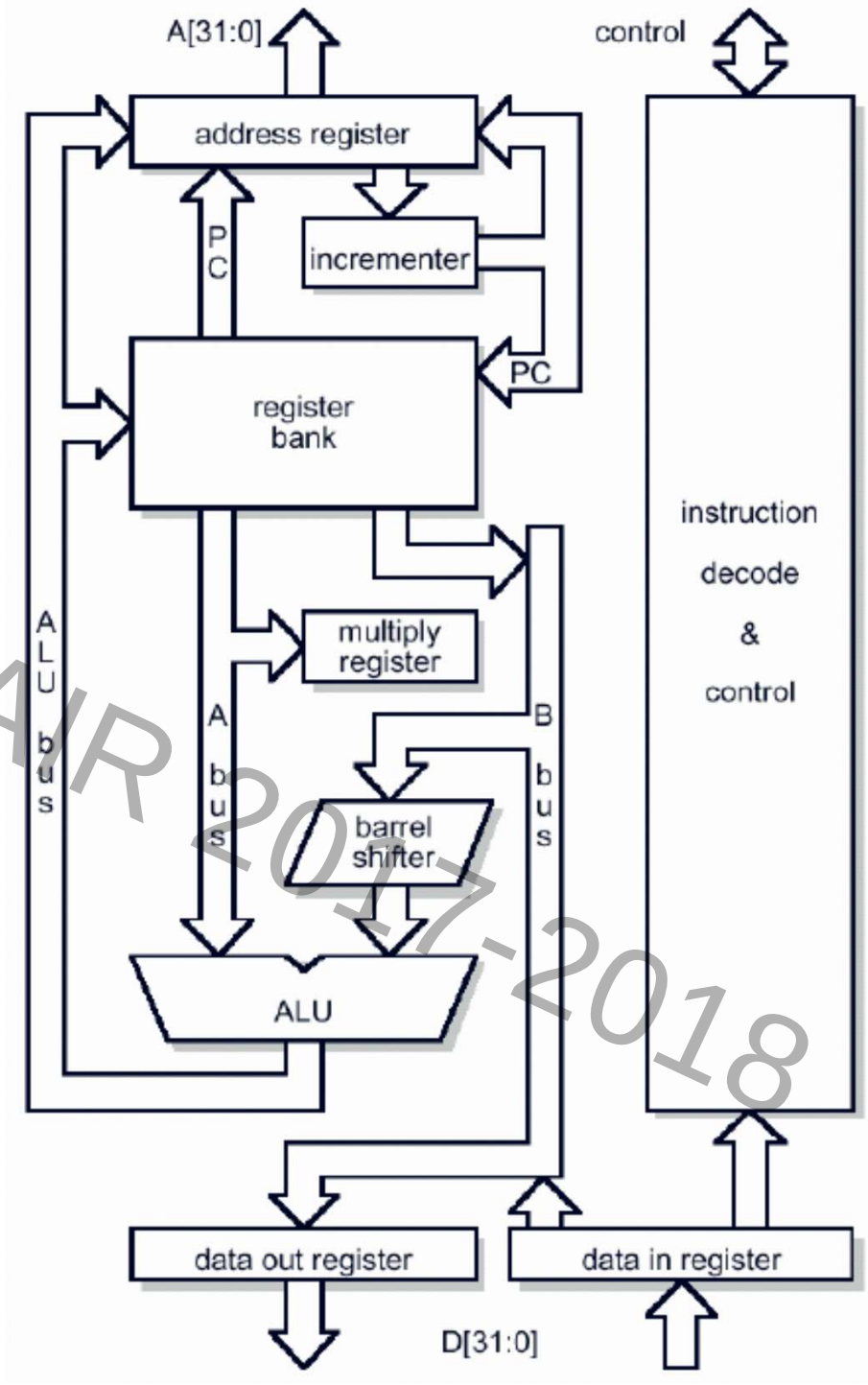
任鹏举

1 ARM处理器内核结构

- 83~85年：第一款ARM处理器采用3微米工艺；
- 90~95年：ARM6/ARM7，采用三级流水线结构；
- 95~02年：ARM9，采用五级流水线，程序/数据存储器分开；
- 02~12年：ARM Cortex系列处理器，ARMv7-A结构，ARM Cortex-A7,A8（单核），ARM Cortex-A9,A15（多核）
- 12年底：ARM Cotex-A50系列，64位ARMv8结构，进军PC和服务器市场。
- 本课程讲解内容主要基于ARM7.

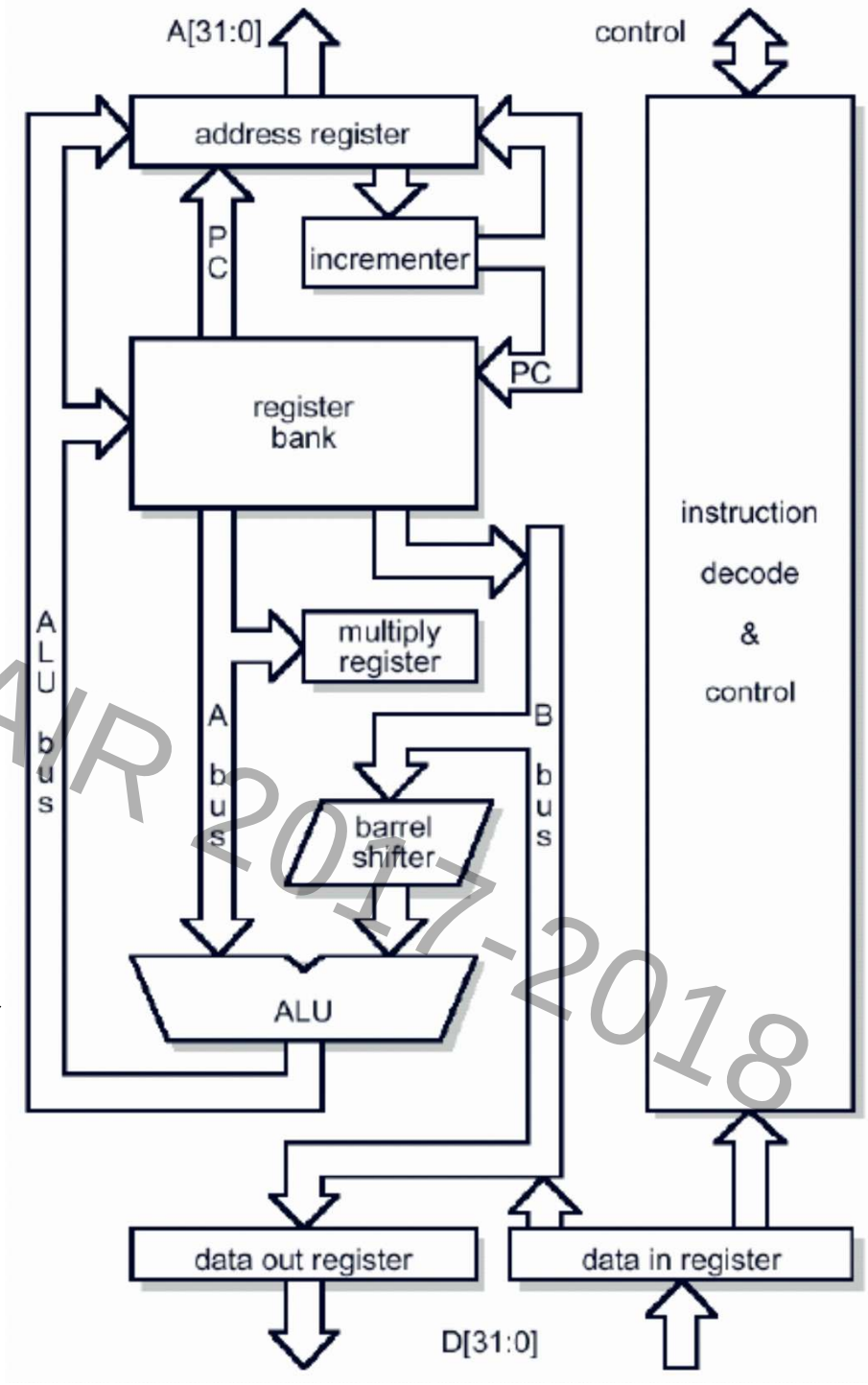
ARM内部组成

- 两个主要模块：数据通路（**data path**）与解码器和控制器（**decoder/Controller**）。
- 寄存器堆（**r0~r15**）
  - 两个读端口**A-bus/B-bus**
  - 一个写端口**ALU-bus**
  - 额外的**r15**读写端口
- 桶形移位寄存器：对第二操作数移位或循环移位若干数据位**(Barrel shifter)**。
- **ALU**执行算术逻辑运算。

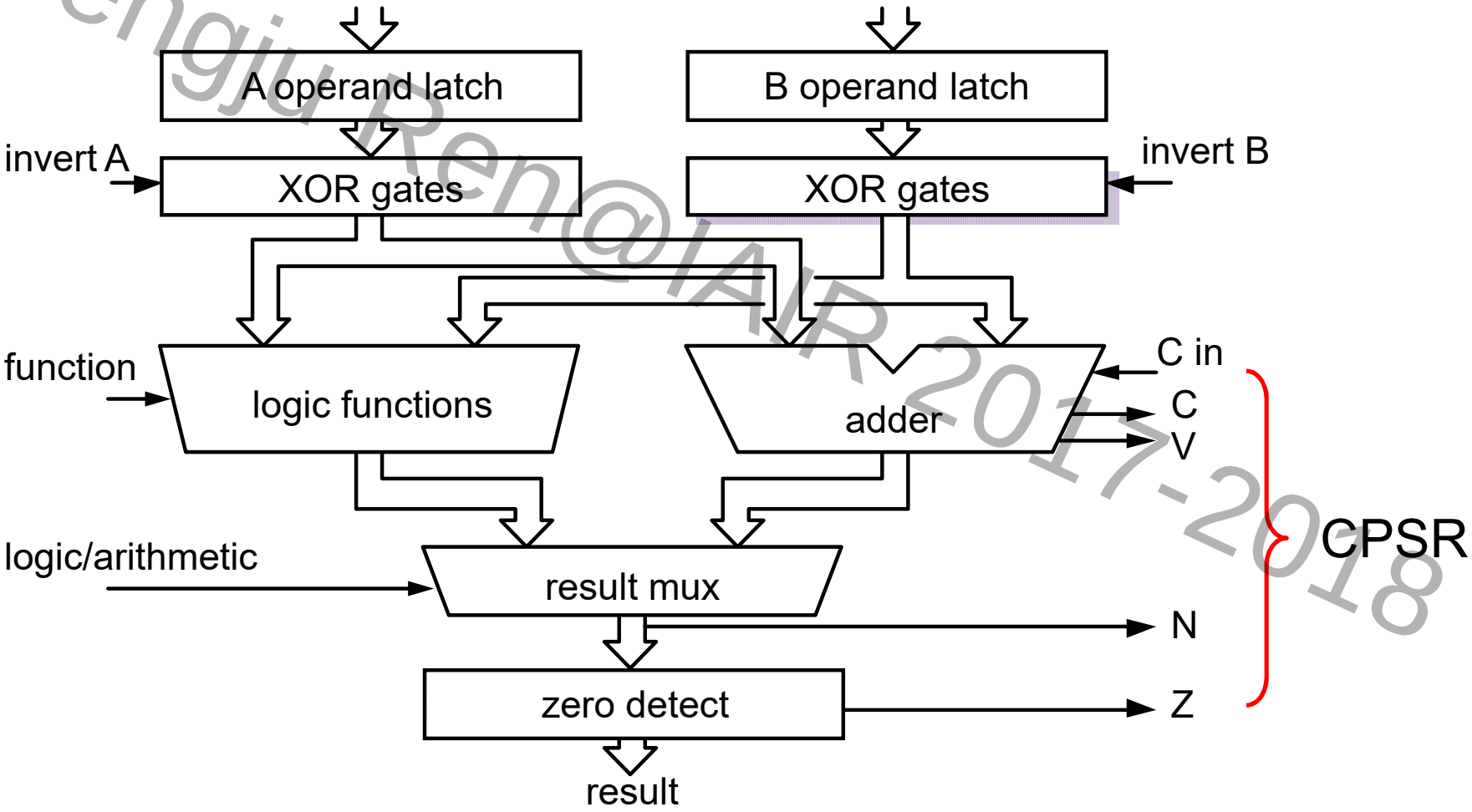


ARM内部组成(续)

- 专门的PC累加器
- 地址寄存器：从ALU或PC
- 数据寄存器：保存对存储器写或读的数据
- 指令译码器对指令的机器码译码，产生数据通路所需的控制信号。
- 数据处理指令需要1个时钟周期，数据由A-bus&B-bus读，结果经ALU-bus写回寄存器堆。



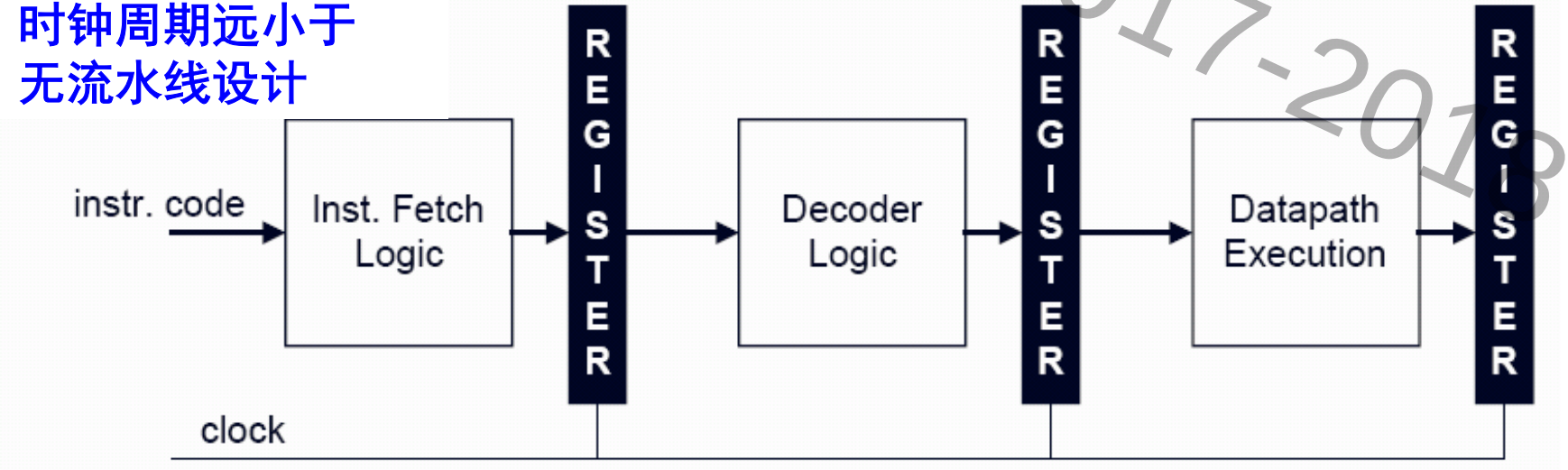
ARM的ALU



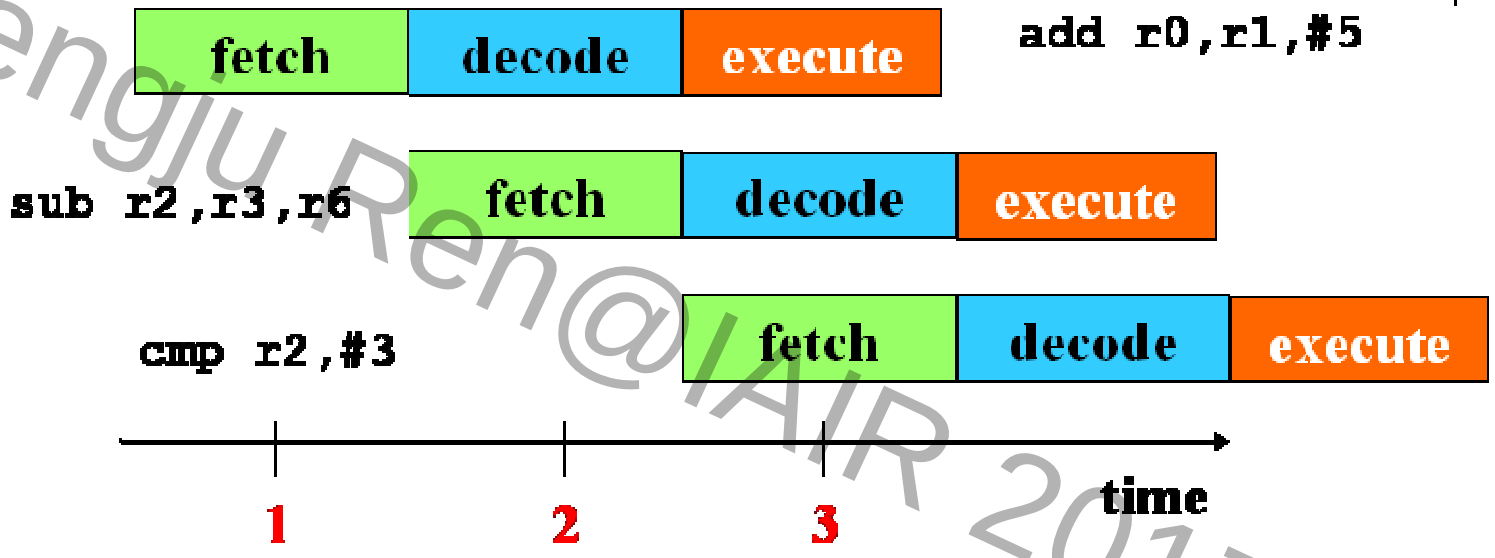
ARM7的流水线

- **ARM7使用三级流水线(ARM9使用五级流水线结构)**
  - 取指：从存储器中获取指令代码到指令流水线。
  - 译码：指令译码，通过控制信号操作数据通路准备执行。
  - 执行：指令获得数据通路“使用权”：读寄存器, 移位操作, **ALU**运算和写回。
- 每一级的结果存储在寄存器中。

时钟周期远小于
无流水线设计

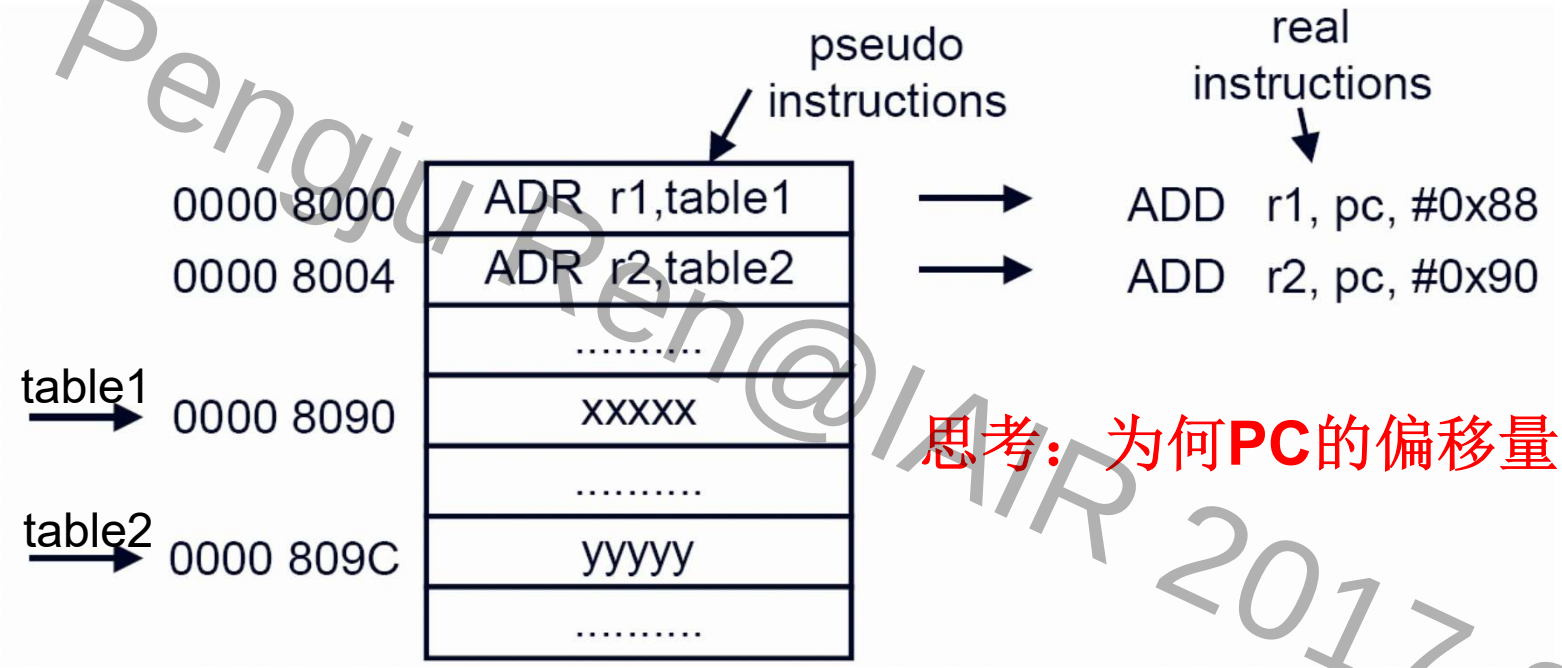


ARM的流水线（续）



- 任意时刻，有三条不同的指令占据着流水线的不同级
- 完成一条指令需要三个时钟周期，称之为三周期延迟
- 流水线装满后，处理器一个时钟周期可以完成一条指令。因此，流水线的吞吐率是每周期一条指令。
- 流水线提高了吞吐率，但无法降低延迟。

数据传送指令 – ADR伪指令(回顾)



思考：为何PC的偏移量是0x88?

- 程序计数器相对寻址（**PC-relative addressing**）
- **Example:**

PC=PC+8 ? Why

start MOV r0, #10

ADR r4, start => **SUB r4, pc, #0x0c**

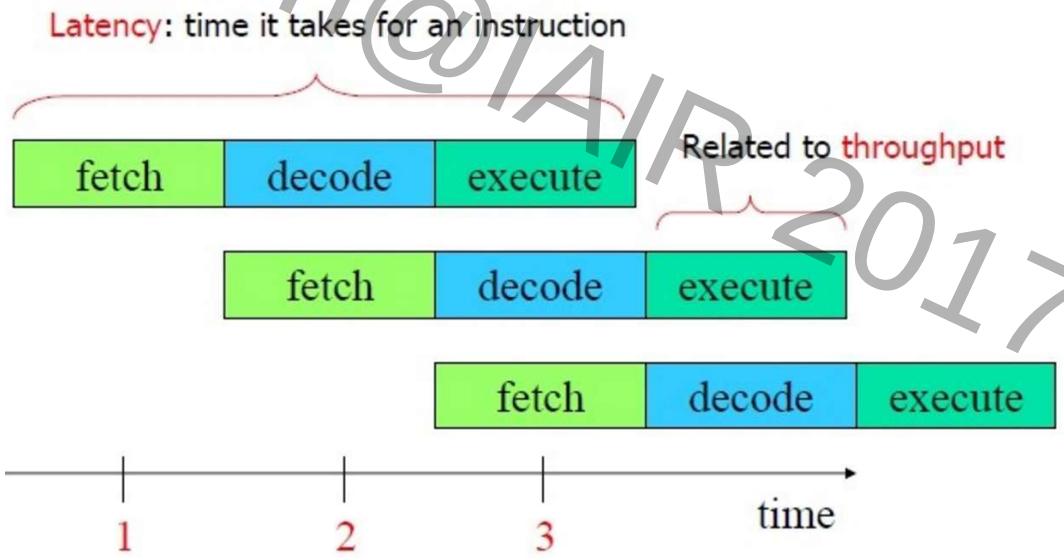
PC=PC+8 ? Why ?

```
start  MOV r0, #10
       ADR  r4, start
       ADD  r0, r1, #5
       SUB  r2, r3, r6
```

含伪代码

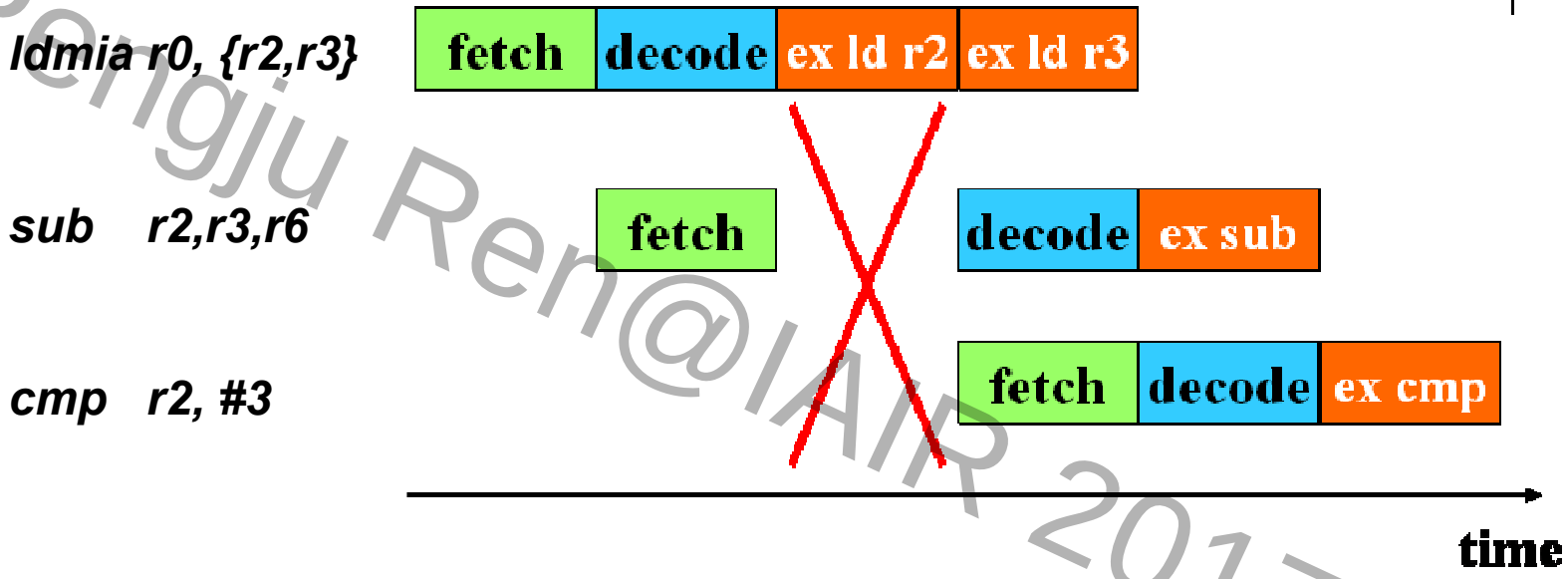
```
start  MOV r0, #10
       SUB  r4, pc, #0x0c
       ADD  r0, r1, #5
       SUB  r2, r3, r6
```

不含伪代码



当PC-relative计算执行的时候，PC已经指向其后的第二条指令 (PC+8)
Rather than pointing to the instruction being executed, the PC points to the instruction being fetched.

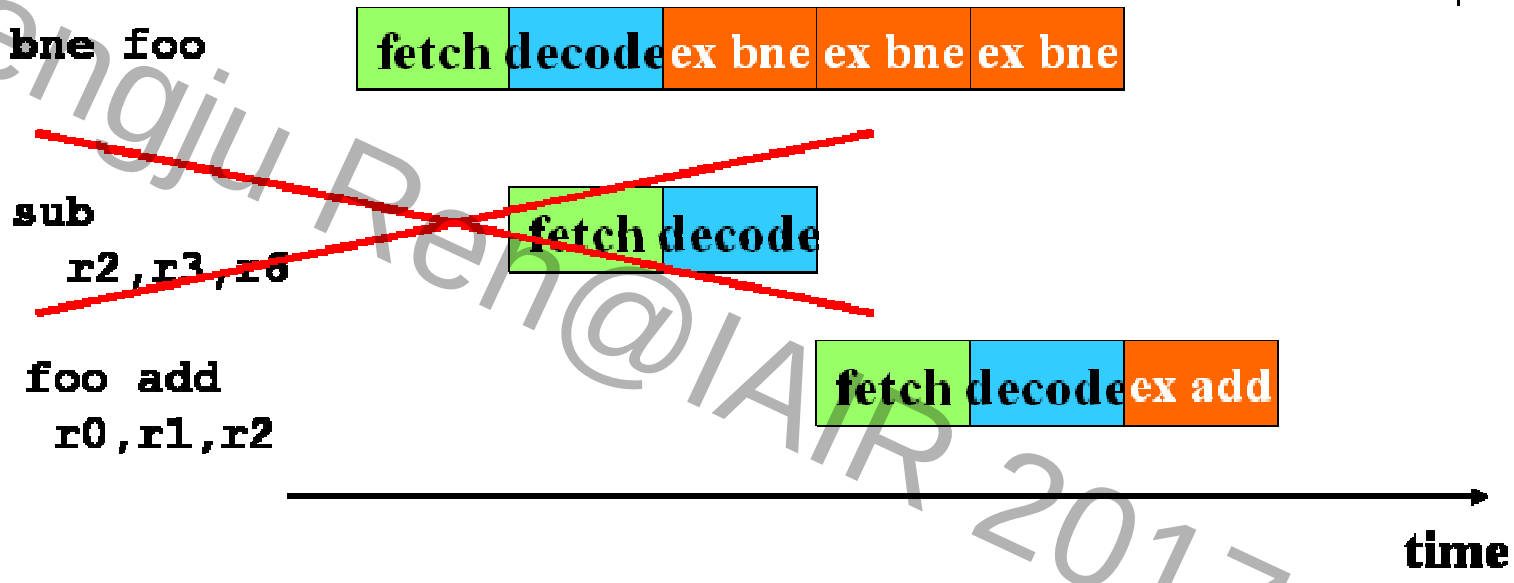
ARM的流水线数据阻滞



- 如果流水线的某一级无法在一个时钟周期内完成，将会导致**流水线阻滞**。
- 例如多寄存加载指令（**LDMIA**），将导致**数据阻滞（Data hazard）**。

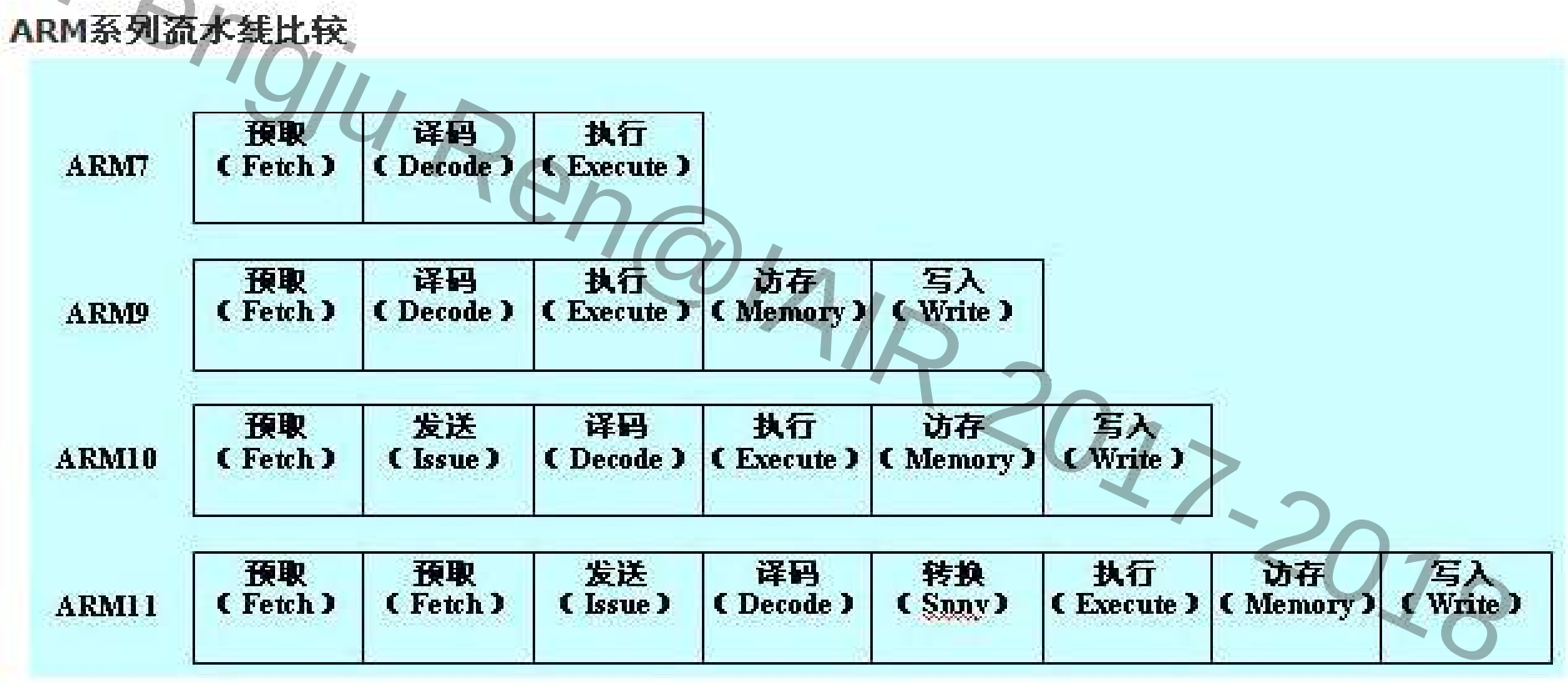
译码逻辑负责产生下一个时钟需要的控制信号，对于多阶段执行的语句，必须要记住已经译码的指令。

ARM的流水线控制阻滞

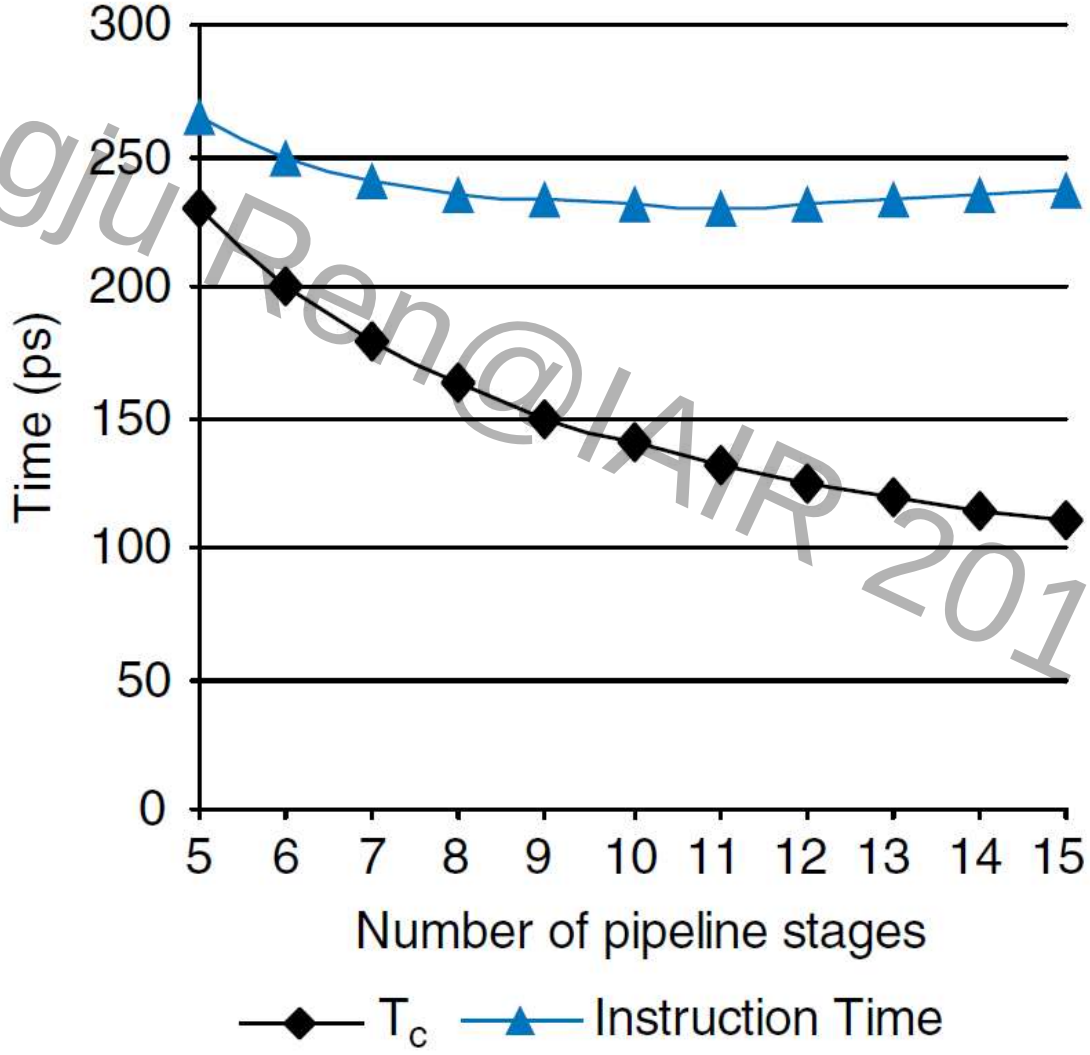


- 分支指令将导致流水线产生**控制阻滞**，也被称为**分支损失（Branch Hazard）**。
- 是否执行分支指令**BNE**要在第三个时钟周期才能确定，因此，浪费两个时钟周期。

ARM的流水线比较



更深的流水线是否更好？



延迟分支（delayed branch）

- 为改善流水线效率，可使用**延迟分支**技术。
- 延迟分支要求分支指令后面的n条指令总是执行，无论分支指令执行与否。这样，在分支指令执行期间CPU能够让流水线满负载运行。
- 但是，分支指令后面可能是空操作（NOP）。因为处于延迟分支窗口（branch delay slot）中的指令对于两条执行路径都要有效。如没有没有足够的有效指令填充，则需使用NOP。

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- “Squash” instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

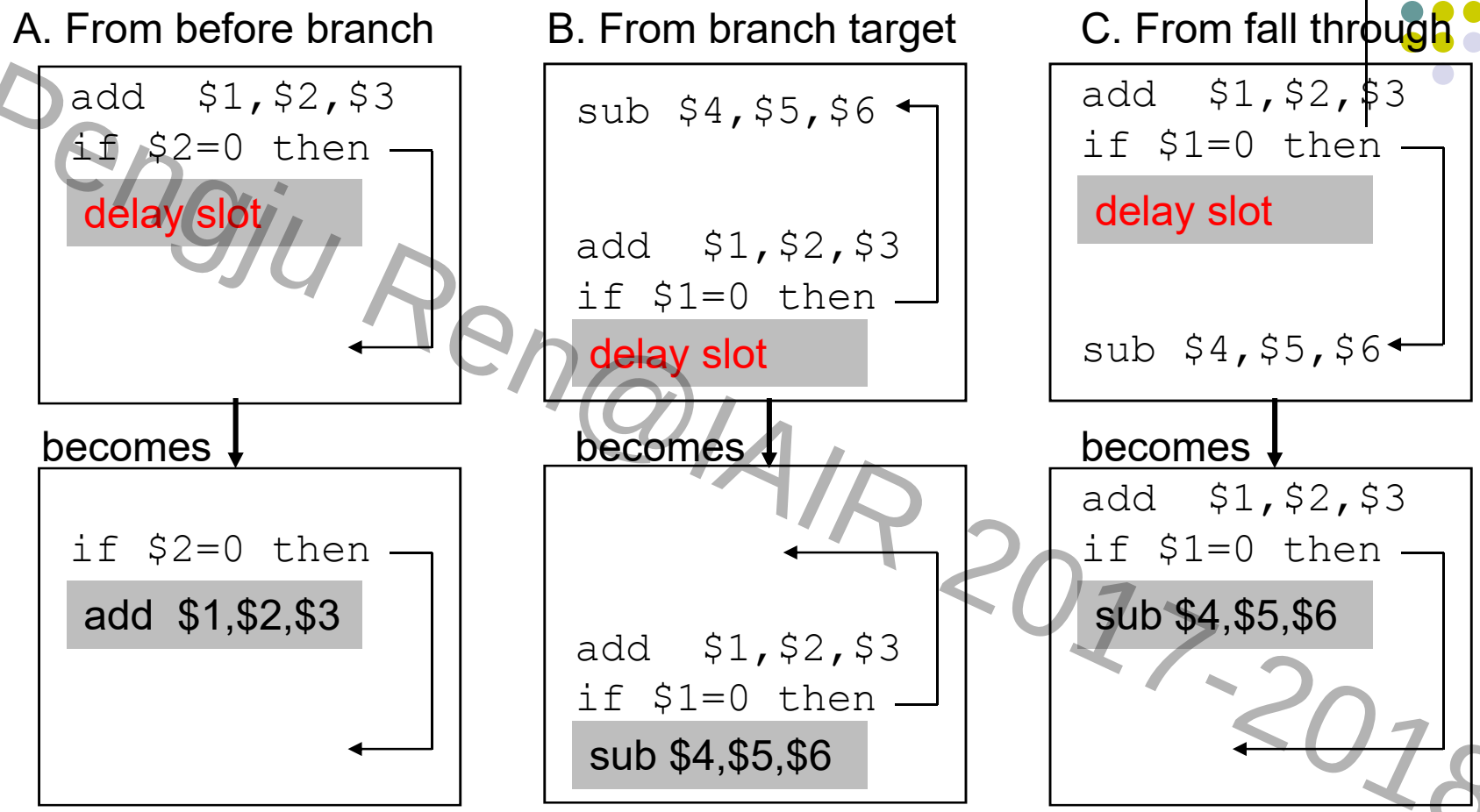
#4: Delayed Branch

- Define branch to take place AFTER a following instruction

Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction **(#4)**
  - From the target address: only valuable when branch taken **(#3)**
  - From fall through: only valuable when branch not taken **(#2)**
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar) **(#1)**

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot & reduces instruction count (IC) **(#1)**
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

范例：ARM中for循环的执行时间

- 决定FIR滤波器应用的执行时间：
  - for (I = 0; i < N; i++)
  - f = f + c[i] * x[i];

FIR滤波器汇编程序

```
; loop initiation code
MOV r0, #0        ; use r0 for i, set to 0
MOV r8, #0        ; use a separate index for arrays
ADR  r2, N        ; get address for N
LDR  r1, [r2]     ; get value of N for loop termination test
MOV r2, #0        ; use r2 for f, set to 0
ADR  r3, c        ; load r3 with address of base of c array
ADR  r5, x        ; load r5 with address of base of x array
          ; loop body
loop      LDR r4, [r3, r8]        ; get value of c[i]
          LDR r6, [r5, r8]        ; get value of x[i]
          MUL r4, r4, r6          ; compute c[i]*x[i]
          ADD r2, r2, r4          ; add into running sum
          ; update loop counter and array index
          ADD r8, r8, #4          ; add one to array index
          ADD r0, r0, #1          ; add 1 to i
          ; test for exit
          CMP r0, r1
          BLT loop                ; if i < N, continue loop
loopend ...
```

FIR滤波器的性能

Block	Variable	# instructions	# cycles
Initialization	t_{init}	7	7
Body	t_{body}	4	4
Update	t_{update}	2	2
Test	t_{test}	2	[2,4]

$$t_{loop} = t_{init} + N(t_{body} + t_{update}) + (N-1)\, t_{test,worst} + t_{test,best}$$

继续循环是最坏情况

退出循环是最好情况

判断一定跳转？？？

思考

- 思考前面课程提到的问题：
  - “条件执行指令只有在条件序列小于等于3的时候才会达到高效，当大于时，要使用条件转移指令。” ---为什么???

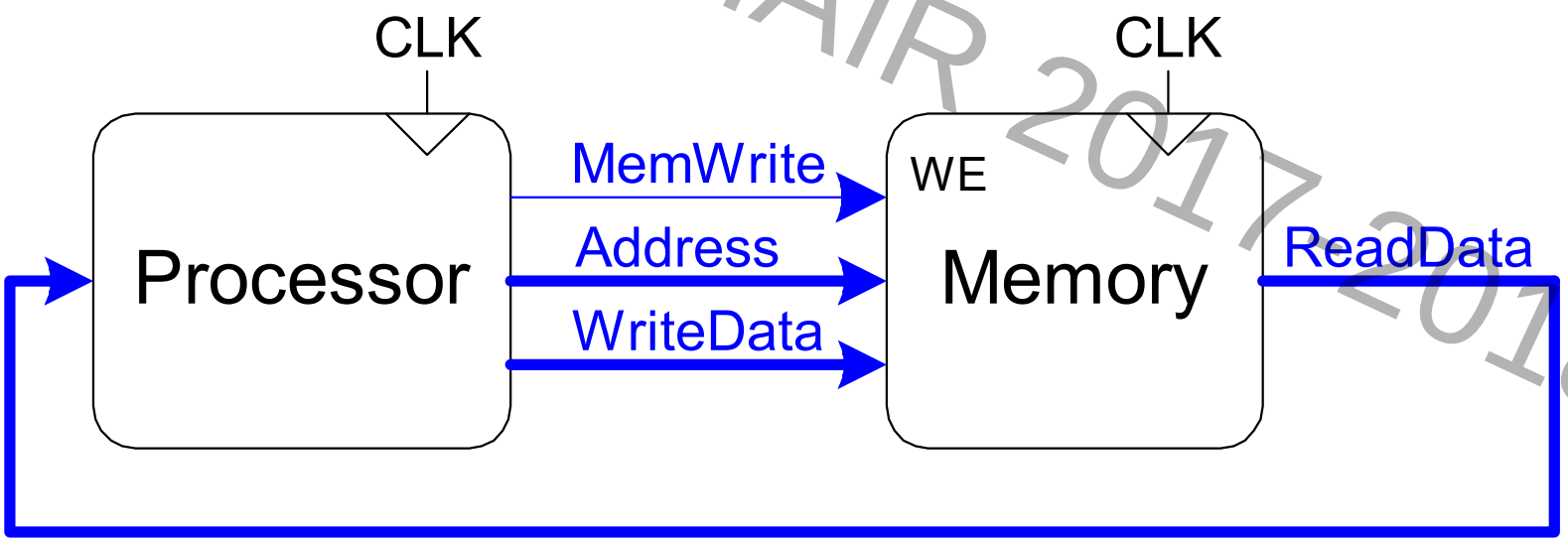
```
                CMP     r0, #5          ; if (r0 != 5) then
                ADDNE   r1, r1, r0      ;     r1 := r1 + r0 - r2
                SUBNE   r1, r1, r2
BYPASS          .....
```

ADDNE和SUBNE是条件执行指令，只有CMP指令是非零结果的时候才执行。

2 高速缓存

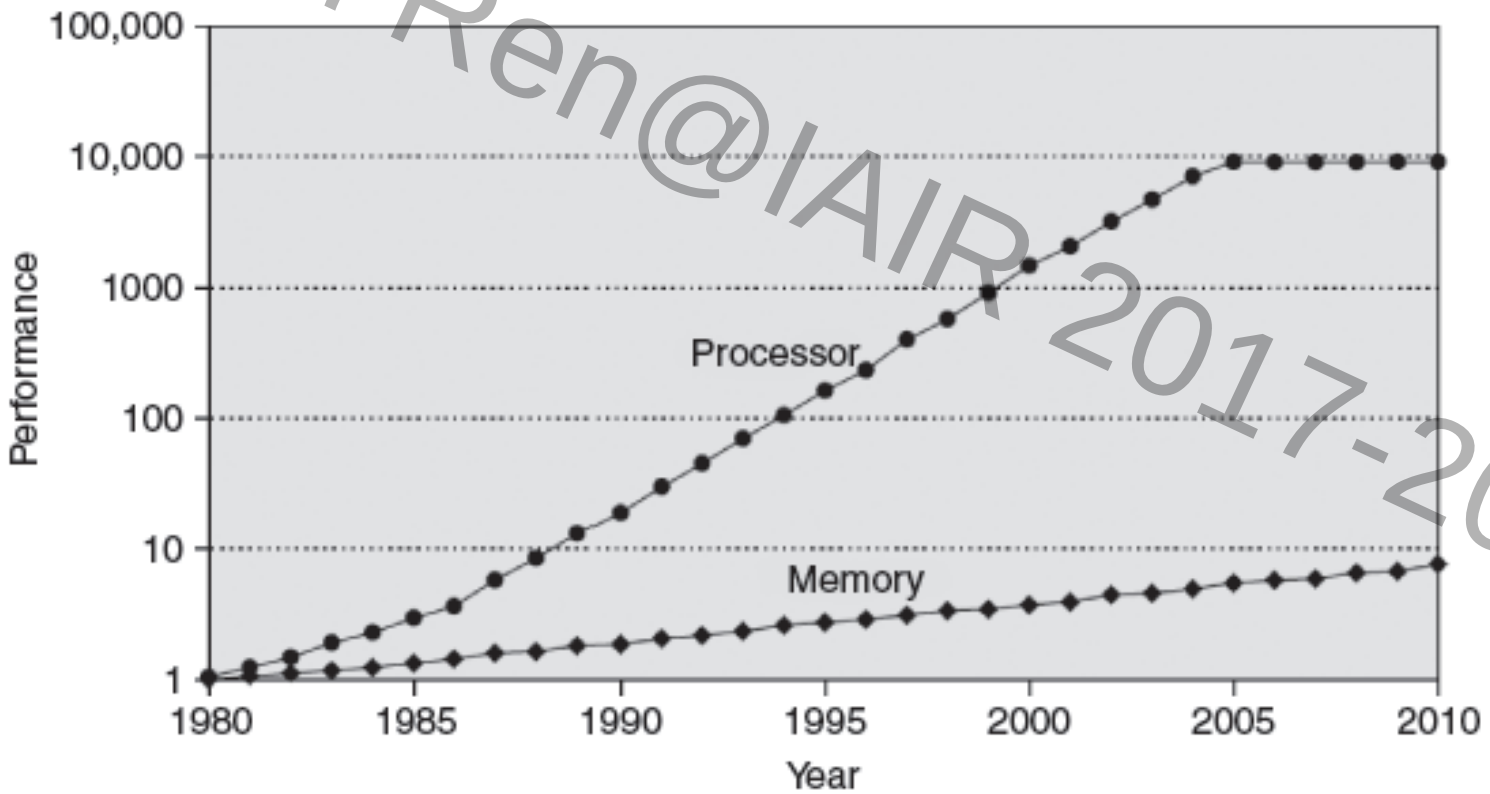
- Computer performance depends on:
  - Processor performance
  - Memory system performance

Memory Interface



2 高速缓存(Processor-Mem Gap)

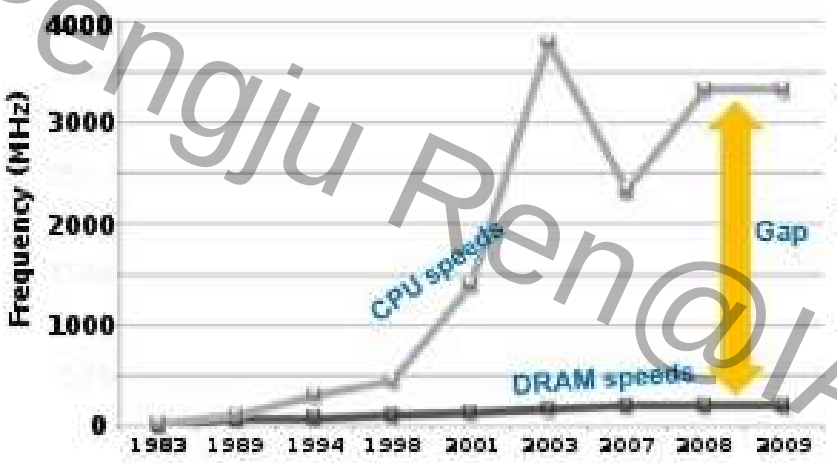
In prior chapters, assumed access memory in 1 clock cycle – but hasn't been true since the 1980's

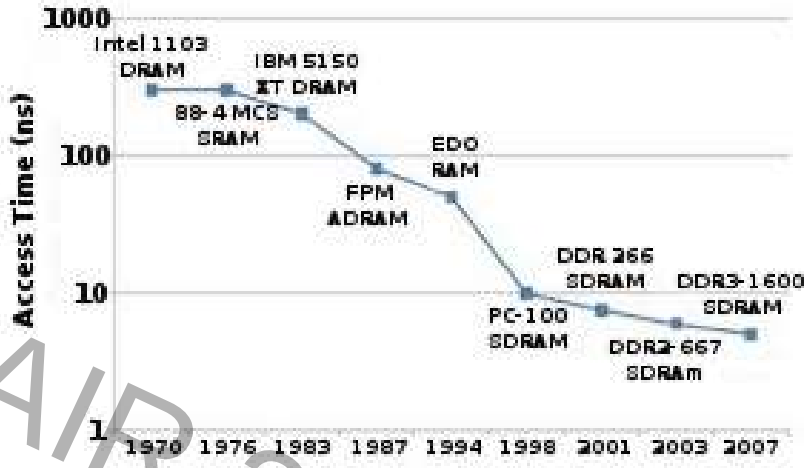


2 高速缓存

- 静态随机存储器（**SRAM**）：依靠一对反相器存储数值，速度很快，但面积占用比**DRAM**高。
- 动态随机存储器（**DRAM**）：依靠电容电荷实现数据存储（必须刷新），很小但比**SRAM**速度慢（**5~10**倍）
- 存储器性能 **vs** 逻辑性能：
  - 存储器比逻辑电路的性能提升慢得多（“存储墙”问题）。
  - 因此，速度非常快的存储器是很昂贵的。
  - 同时，应用对存储器的需求量越来越大。
- 用户需要大容量且快速的存储资源。
  - **SRAM**访问延迟**:2~25ns**，耗费**:$100~$250 per MB**
  - **DRAM**访问延迟**:60~120ns**，耗费**:$0.05~$0.1 per MB**
  - **Disk**访问延迟**:10~20million ns**，耗费**:$0.05~$0.1 per MB**

“存储墙”问题

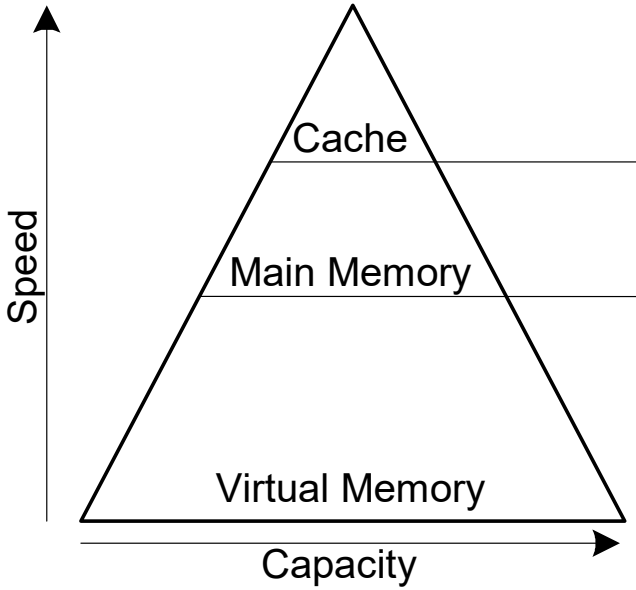




- 随着集成电路工艺尺寸的缩小，处理器的速度按照“摩尔定律”每隔**18**个月就翻一番。
- 存储器的速度每年仅增长**7%**。
- 处理器与存储器之间的性能鸿沟每**21**个月就翻一番。

存储系统（Memory Hierarchy）

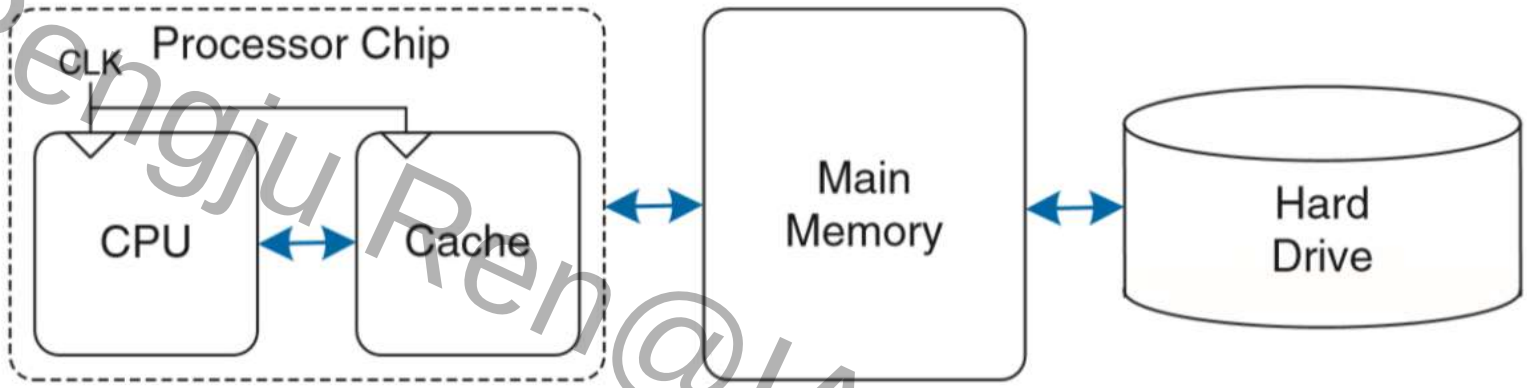
- 如何组织存储器资源，但又不显著提高耗费？
- 答案是：构建存储系统。

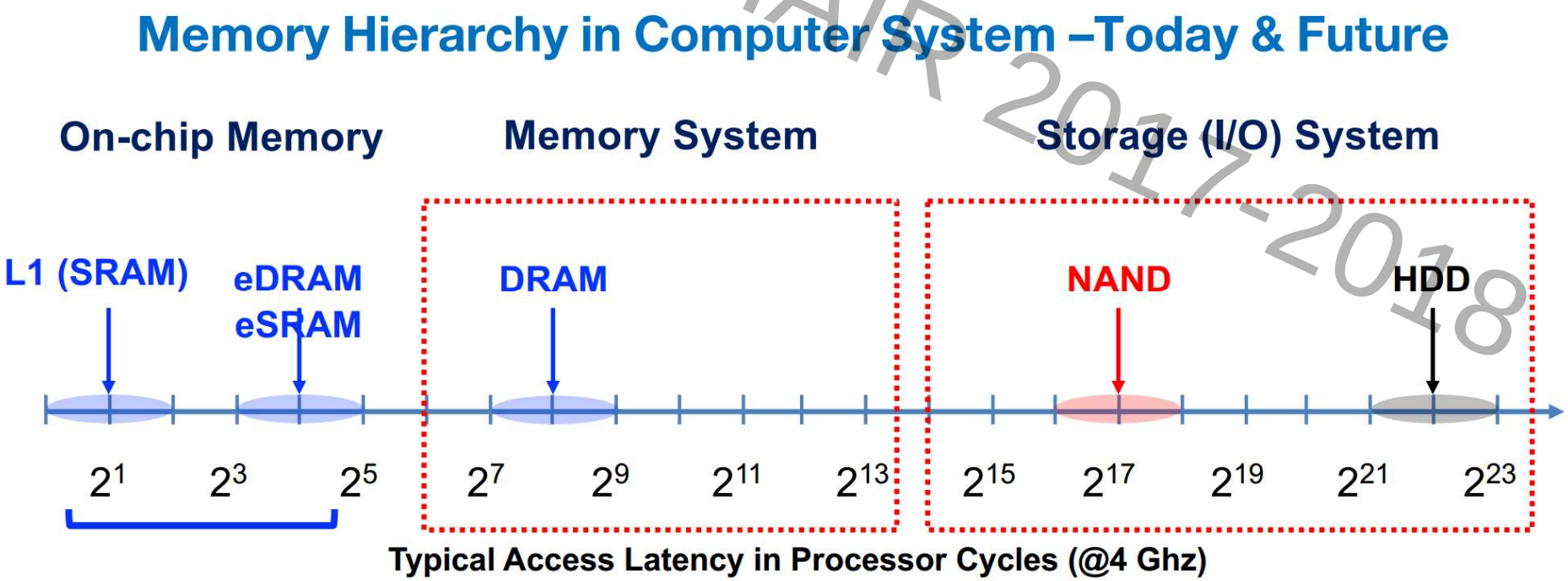


Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

高速缓存保存主存中最近用到的内容的拷贝。

存储系统（Memory Hierarchy）





缓存Caches

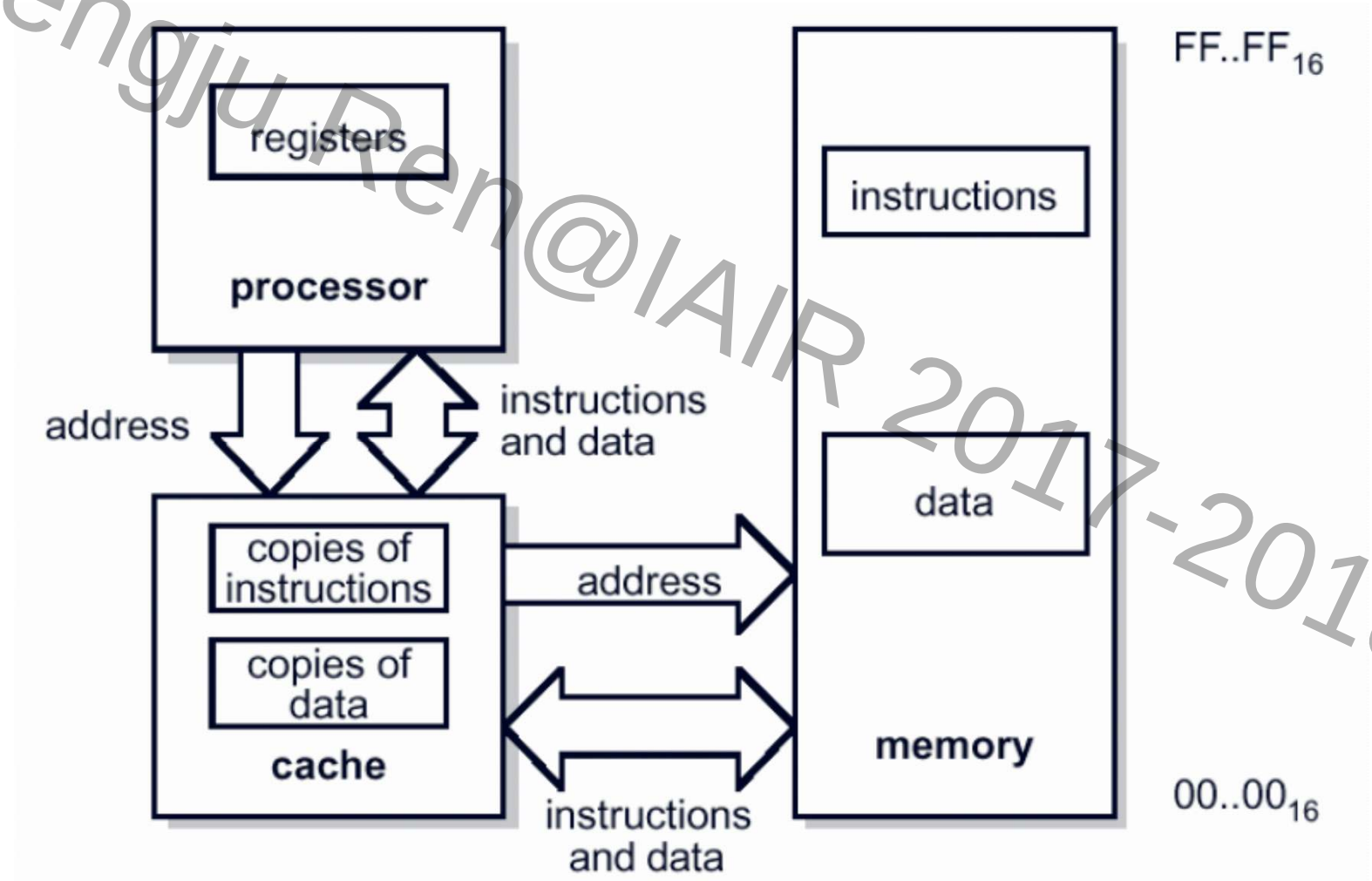
- A cache holds commonly used memory data. The number of data words that it can hold is called the capacity, C. Because the capacity of the cache is smaller than that of main memory, the computer system designer must choose what subset of the main memory is kept in the cache.
- 高速缓存保存常用的内存数据。 它所能容纳的数据字的数量称为容量. 由于缓存的容量小于主存的容量，因此计算机系统设计者必须选择在缓存中保留主存的哪个子集。
- When the processor attempts to access data, it first checks the cache for the data. If the cache hits, the data is available immediately. If the cache misses, the processor fetches the data from main memory and places it in the cache for future use.
- 当处理器试图访问数据时，它首先检查缓存中的数据。 如果缓存命中，数据立即可用。 如果高速缓存未命中，处理器将从主内存中提取数据并将其放入高速缓存中以供将来使用。

局部性原理

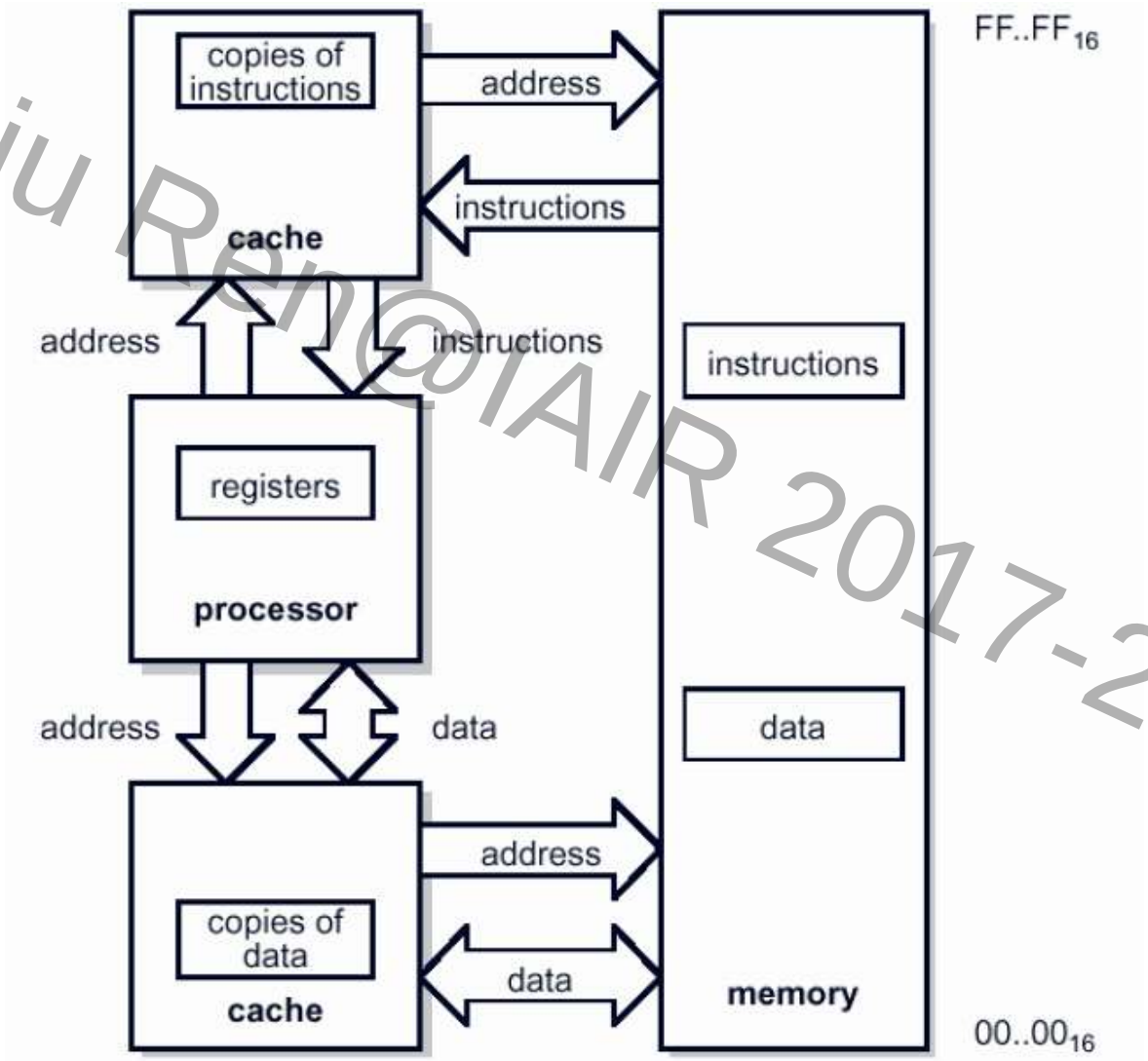
- 局部性原理是存储系统层级构成的基础，将有可能再次访问的数据放在存储结构顶层
  - 时域局部性：当前被访问的数据有可能很快再次被访问； 将最近访问的数据保存在存储系统层级高层
  - 空域局部性：当前被访问数据地址相邻的数据有可能很快被访问，将临近访问的数据保存存储系统层级高层
- 程序为何会具有局部性？
- 高速缓存减少了内存的平均访问时间，但也提高了内存访问时间的可变性，值得注意！

统一的高速缓存

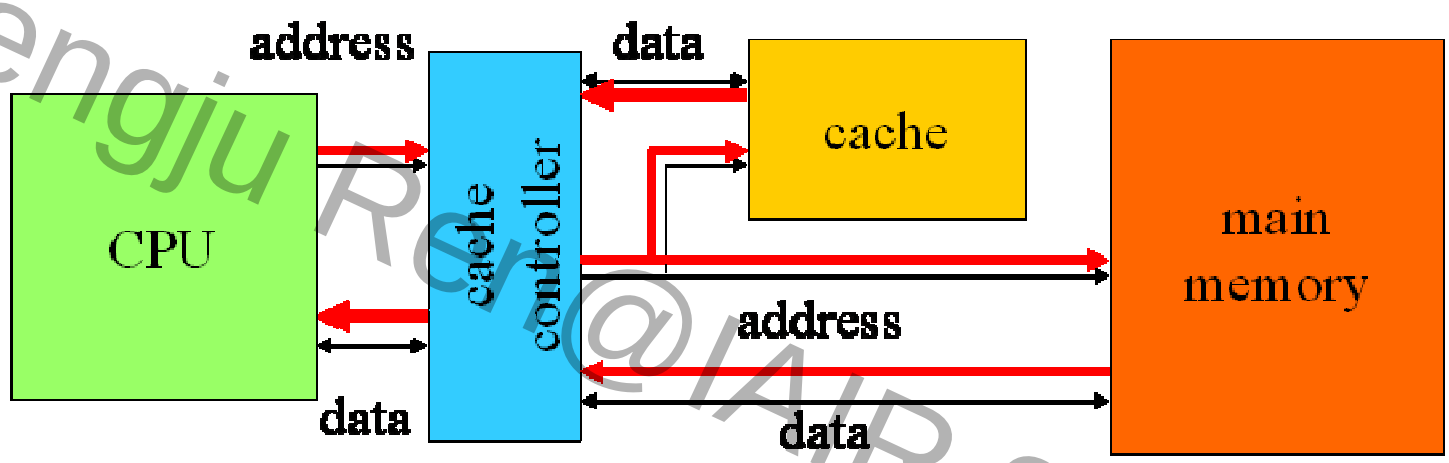
数据与指令之间共享同一个高速缓存。



哈佛高速缓存



Cache的基本术语



- 数据块（data block）：数据传送的最小单元 – 也被称为cache line；
- 高速缓存命中（cache hit）：被请求的数据在高速缓存中；
- 高速缓存未命中（cache miss）：被请求的数据不在高速缓存中。
- 工作集（working set）：CPU在一段时间内访问的活动单元集合。

未命中的类型划分

- 强制性未命中（compulsory miss），也称为冷未命中（cold miss），发生在单元第一次被访问时。
- 容量未命中（capacity miss）的发生是由于工作集过大。
- 冲突未命中（conflict miss）的发生是由于两个地址映射到高速缓存的同一个单元。
- 平均内存访问时间计算公式：

$$t_{av} = t_{cache} + (1-h)t_{main}$$, h代表命中率

Cache的性能-举例1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- **What are the hit and miss rates for the cache?**

Cache的性能-举例1

- A program has 2,000 loads and stores
- 1,250 of these data values in cache
- Rest supplied by other levels of memory hierarchy
- **What are the hit and miss rates for the cache?**

Hit Rate = 1250/2000 = **0.625**

Miss Rate = 750/2000 = **0.375** = 1 – Hit Rate

Cache的性能-举例2

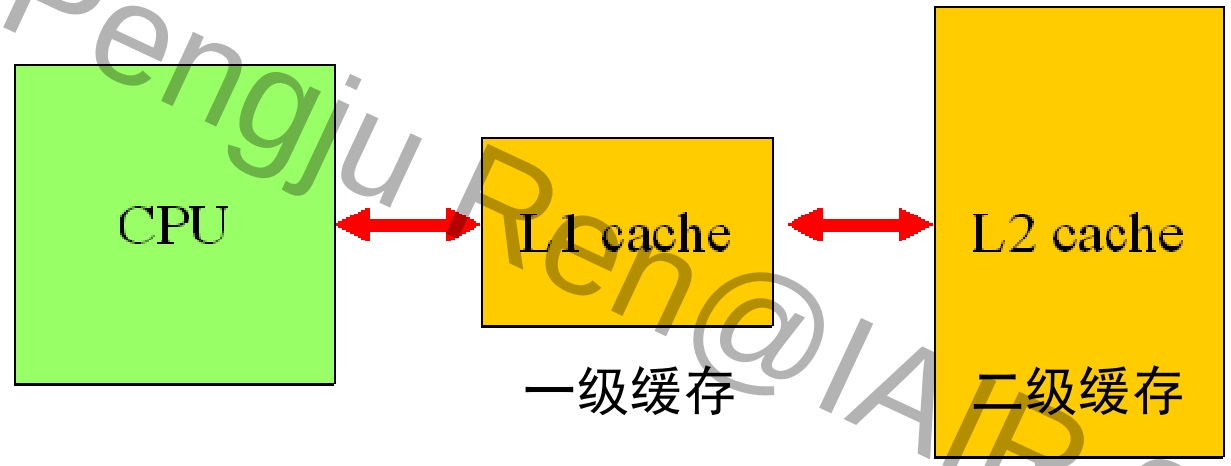
- Suppose processor has 2 levels of hierarchy: cache and main memory
- t_{cache} = 1 cycle, t_{MM} = 100 cycles(Main Memory)
- **What is the average data access time of the program from Example 1?**

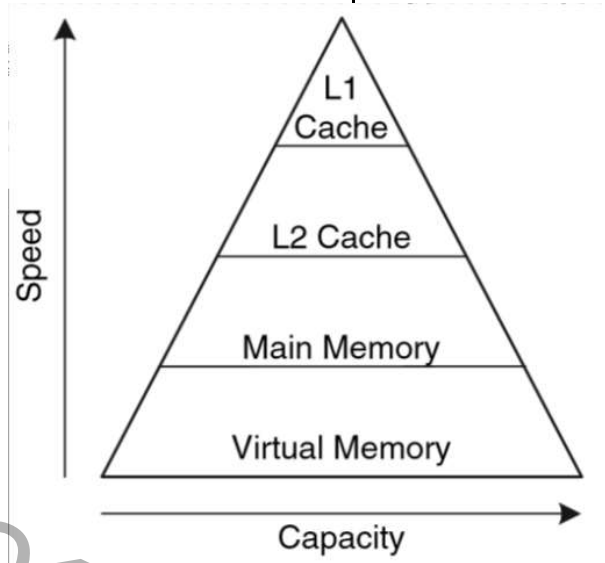
Cache的性能-举例2

- Suppose processor has 2 levels of hierarchy: cache and main memory
- $t_{\text{cache}} = 1$ cycle, $t_{MM} = 100$ cycles (Main Memory)
- **What is the average data access time of the program from Example 1?**

$$\begin{aligned} \boldsymbol{Tav} &= t_{\text{cache}} + MR_{\text{cache}}(t_{MM}) \\ &= [1 + 0.375(100)] \text{ cycles} \\ &= \mathbf{38.5\ cycles} \end{aligned}$$

多级高速缓存





- 平均内存访问时间计算公式：

$$t_{av} = t_{L1} + (1-h_1)t_{L2} + (1-h_1-h_2)t_{main}$$

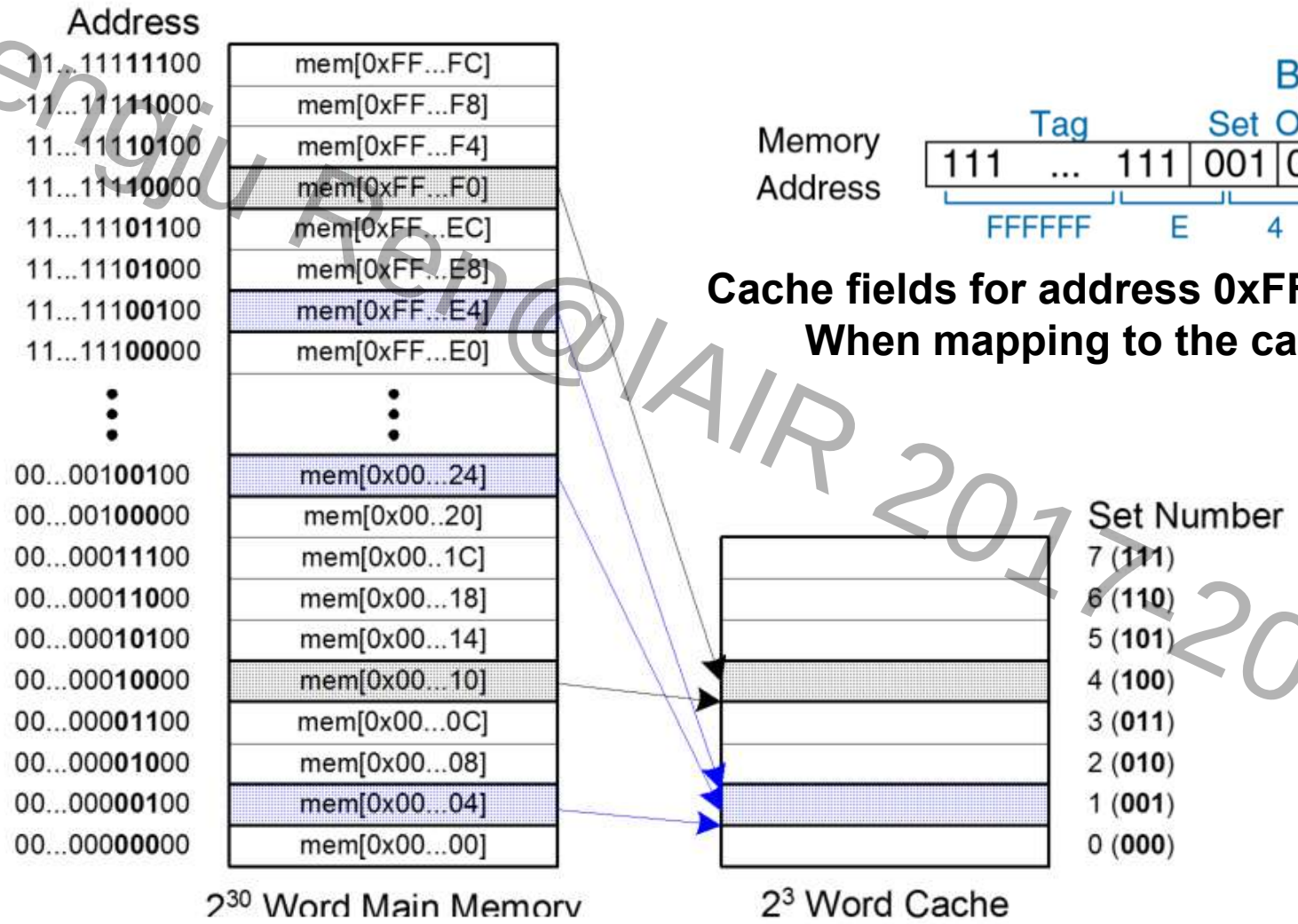
h_1表示一级缓存命中率，h_2表示命中二级缓存而未命中一级缓存的概率。

高速缓存设计

- 让我们思考三个问题：
  - **(1) What data is held in the cache?**
  - 我们如何知道需要的数据是否在高速缓存中？（命中还是未命中？）
  - **(2) How is data found?**
  - 如果在高速缓存中，我们如何找到它？
  - **(3) What data is replaced to make room for new data when the cache is full?**
  - 如果缓存满了，我们该替换掉哪些数据为新的数据提供空间？
- 我们先看一个最简单的例子：
  - 数据块的大小为一个字节；
  - **直接映射（direct mapped）**高速缓存

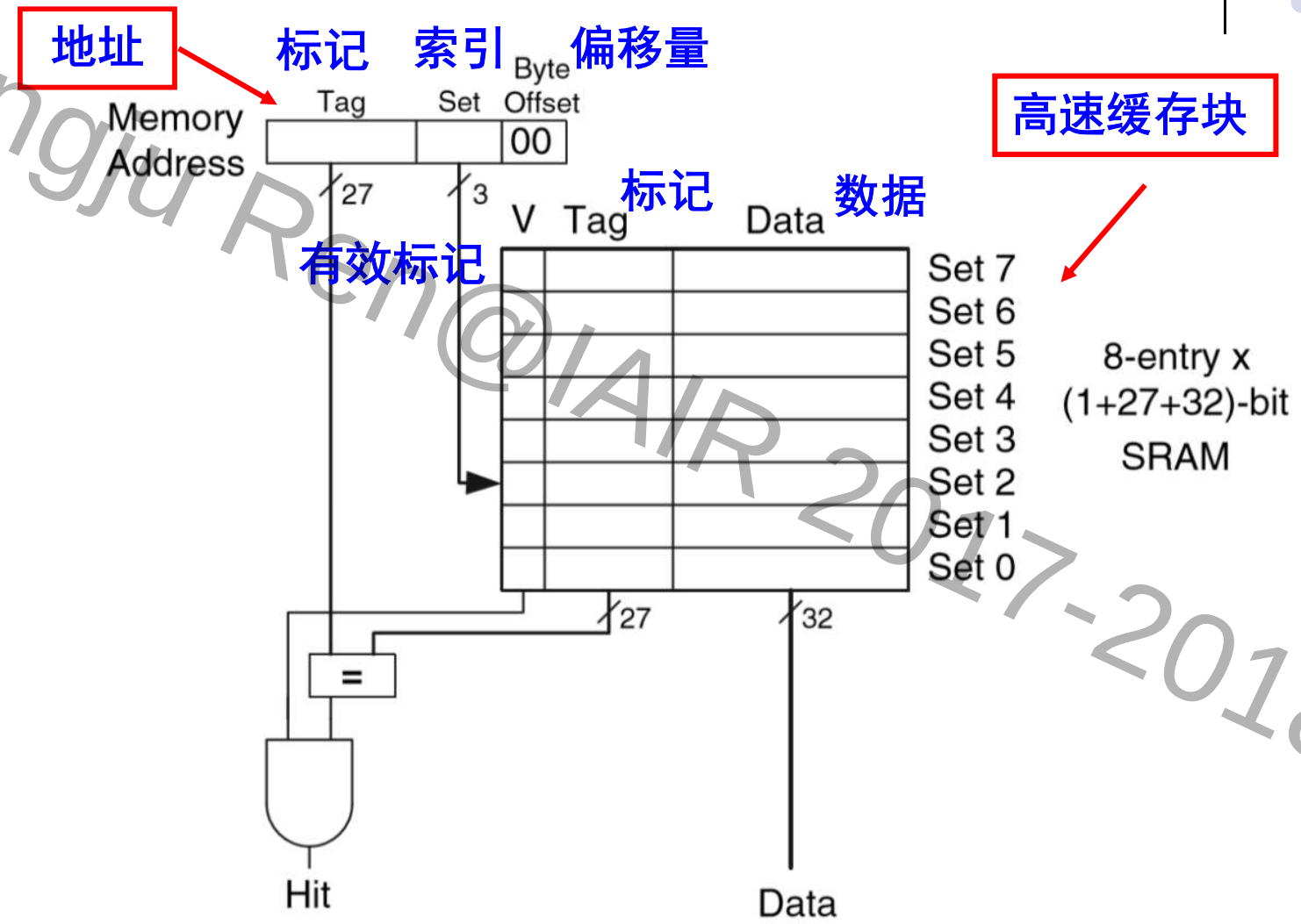
直接映射指低一级存储中的每一个数据在高速缓存中的映射都是唯一确定的地址。

直接映射高速缓存图示



Cache fields for address 0xFFFFFFE4
When mapping to the cache

直接映射高速缓存结构



只有一个块需要检查，索引唯一地确定了要检查的块。

直接映射高速缓存操作简述

- A particular memory item is stored in a unique location in the cache.
- To check if a particular memory item is in cache, the relevant address bits are used to access the cache entry.
- The top address bits are then compared with the stored tag. If they are equal, we have got a hit.
- Two items with the same cache address field will contend for use of that location.
- Only those bits of the address which are not used to select within the line or to address the cache RAM need be stored in the tag field.
- When a miss occurs, data cannot be read from the cache. A slower read from the next level of memory must take place, incurring a miss penalty.
- A cache line is typically more than one word. It shows 4 words in the diagram here. A large cache line exploits principle of spatial locality - more hits for sequential access. It also incurs higher miss penalty.

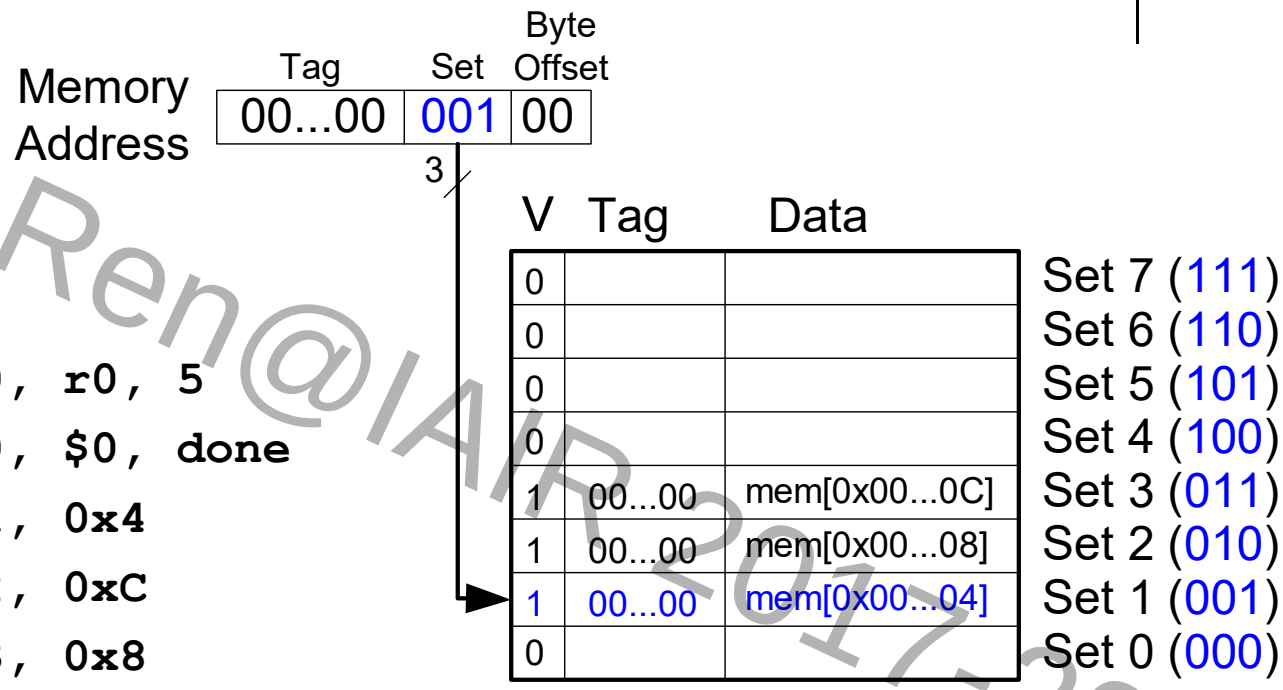
写操作

- 写操作比读操作复杂，因为我们要更新高速缓存和主存（或下一级存储）的内容。
- 两种写操作方式:
  - 通写（**write through**）：每次写操作都将同时改变高速缓存和主存单元。这种模式保证了高速缓存和主存的一致性，但会产生额外的主存通信。
  - 回写（**write back**）：只是在将某一单元从高速缓存中移出时才进行写操作（利用**dirty bit**来判断）。这种模式可以减少那些该单元移出高速缓存之前对它进行的多次写操作。

命中 vs. 未命中

- 读命中：
  - 所希望的情况：高速缓存直接传送数据到处理器。
- 读未命中：
  - 暂停**CPU**，从主存中获取数据块，传送到高速缓存，重新启动
- 写命中：
  - 可以替换高速缓存和主存中的数据（通写）
  - 只将数据写入高速缓存（稍后写入主存）
- 写未命中：
  - 将整个数据块读入高速缓存，然后写某个字。

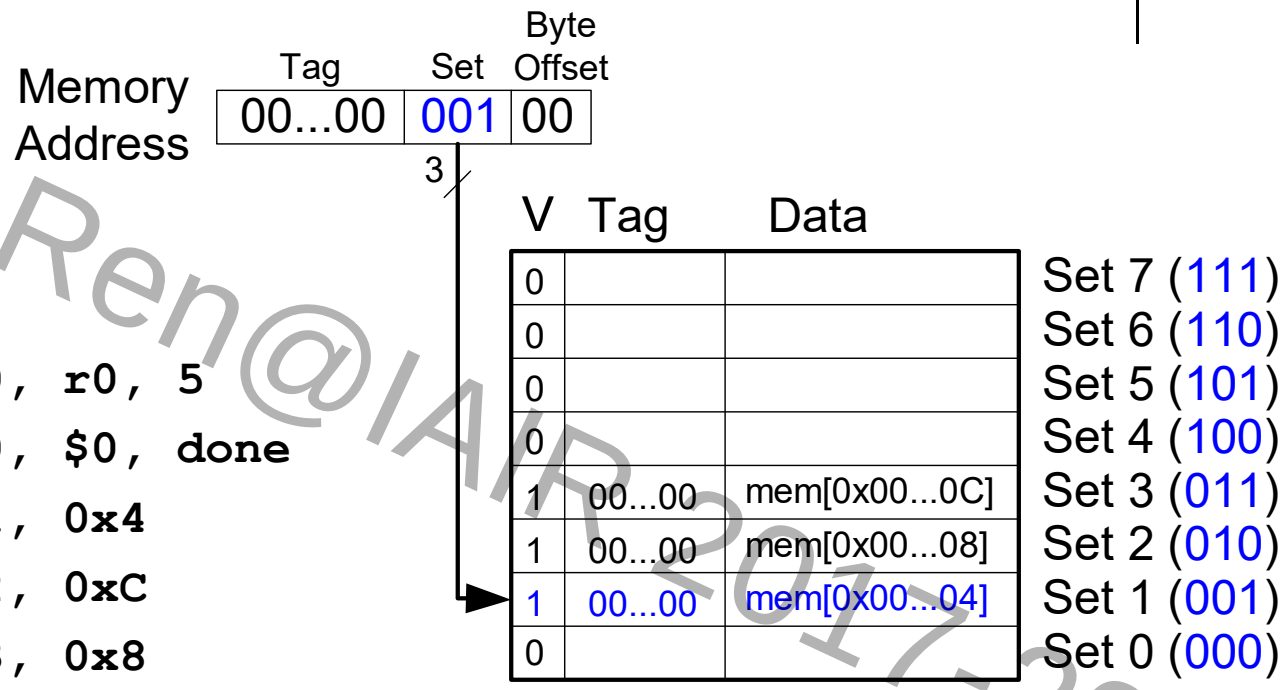
直接映射的性能




```
        addi r0, r0, 5
Loop:   beq  r0, $0, done
        ldr  r1, 0x4
        ldr  r2, 0xC
        ldr  r3, 0x8
        add  r0, r0, -1
        bl    loop
done:
```

Miss Rate = ?

直接映射的性能




```
        addi r0, r0, 5
Loop:   beq  r0, $0, done
        ldr  r1, 0x4
        ldr  r2, 0xC
        ldr  r3, 0x8
        add  r0, r0, -1
        bl    loop
done:
```

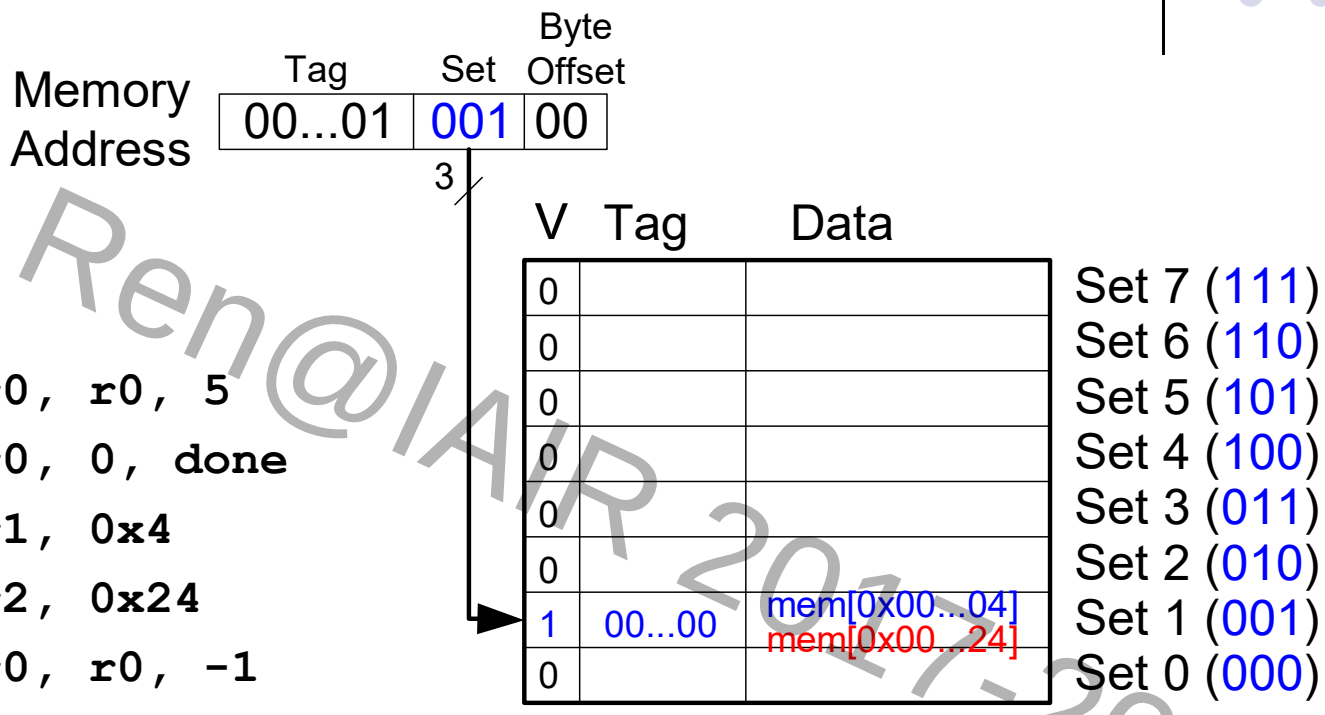
Miss Rate = 3/15

= 20%

Temporal Locality

Compulsory Misses

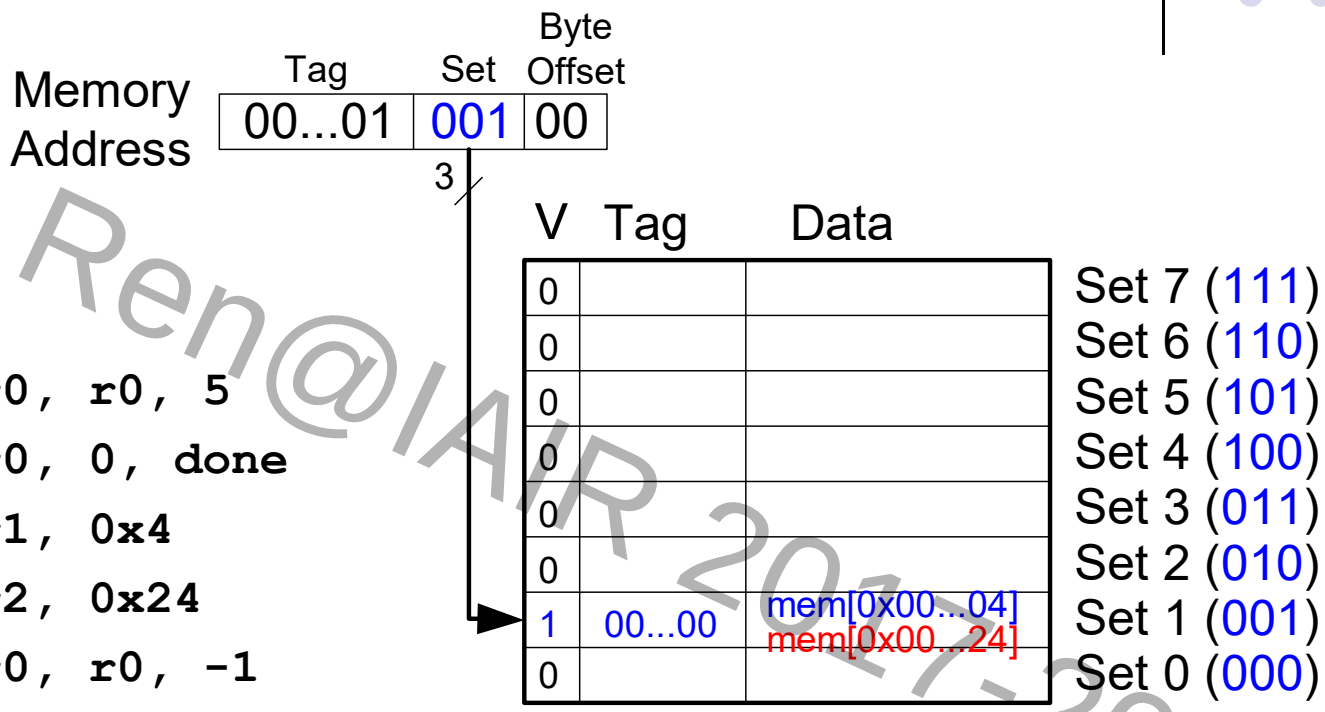
直接映射的冲突




```
        addi r0, r0, 5
Loop:   beq  r0, 0, done
        ldr  r1, 0x4
        ldr  r2, 0x24
        add  r0, r0, -1
        bl    loop
done:
```

Miss Rate = ?

直接映射的冲突




```
        addi r0, r0, 5
Loop:   beq  r0, 0, done
        ldr  r1, 0x4
        ldr  r2, 0x24
        add  r0, r0, -1
        bl    loop
done:
```

Miss Rate = 10/10

= 100%

Conflict Misses

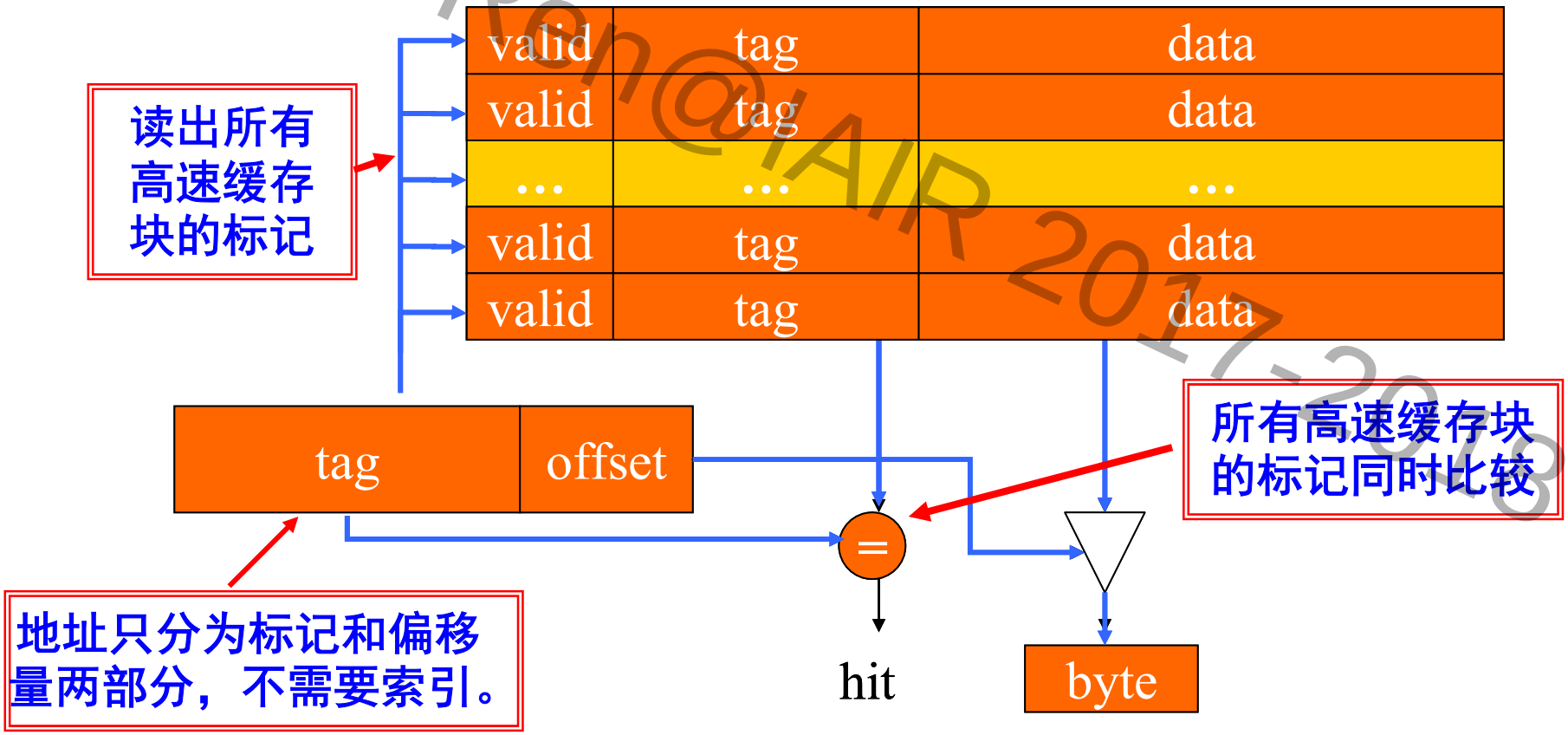
直接映射的局限性

- 直接映射高速缓存速度快，耗费相对较低。
- 直接映射高速缓存映射策略简单，有一定的局限性。
- 容易造成冲突未命中的情况：
  - **Array a[] uses locations 0, 1, 2, …**
  - **Array b[] uses locations 1024, 1025, 1026, …**
  - 计算 **a[i] + b[i]** 将产生冲突未命中的情况.
- 如果频繁访问的单元正好被映射到同一个高速缓存块，我们将无法充分利用高速缓存的优势。

如何解决？？？

全相联高速缓存

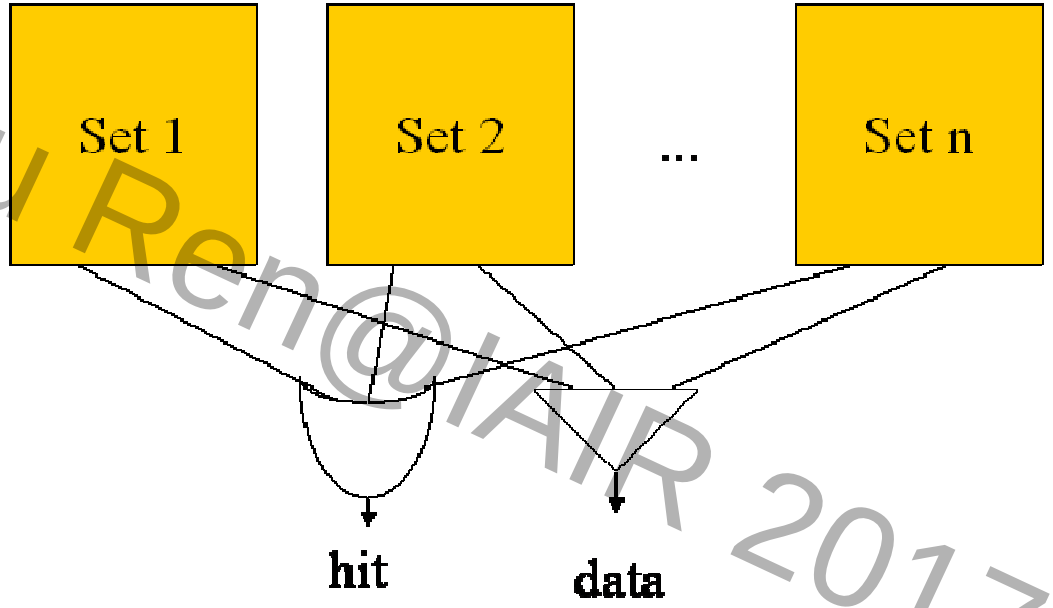
- 全相联（**fully-associative**）高速缓存：任意个内存数据块可以映射到高速缓存中的任意个地址。



全相联的局限性

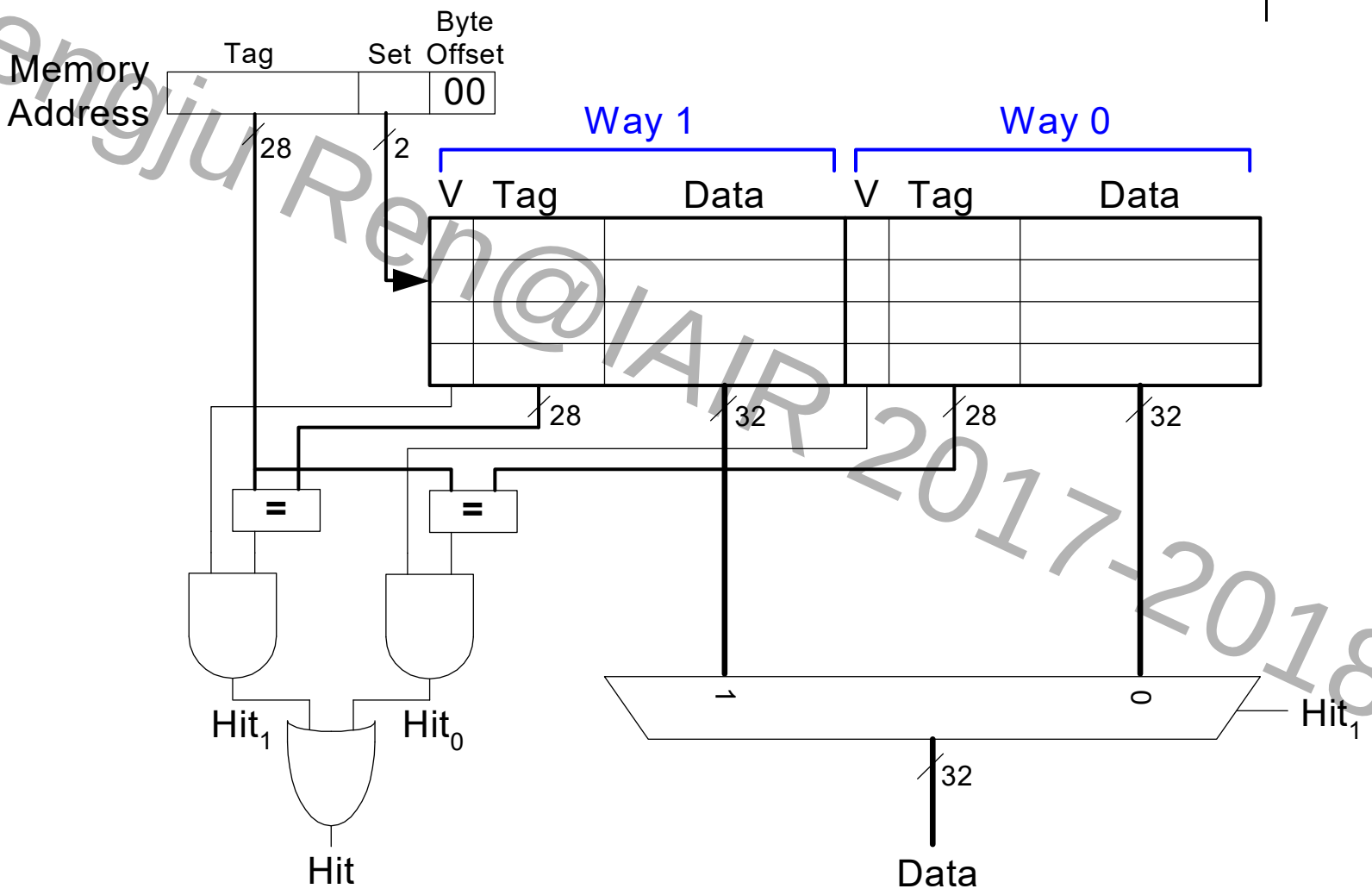
- 全相联不限制主存数据在高速缓存中的存放位置，使映射具有最大的灵活性，命中率最高。
- 但是，由于全相联通常需要采用内容可寻址存储器（**CAM**），耗费很高。同时，要在每次数据访问时读出并比较所有标记单元，速度慢，能耗也非常大。
- 因此，全相联高速缓存一般只用于设计小容量缓存器。

组相联高速缓存



- 组相联（**Set Associative**）是直接映射和全相联之间的折中。组（**Set**）由共享同一索引的所有块组成。
- 由每个组的高速缓存块个数来标识，**N**路组相联。
- 高速缓存访问请求按照索引广播给组内所有缓存块，判断是否命中。每个组由一个全相联高速缓存实现。

组相联高速缓存



两路组相联高速缓存性能举例

```
        addi r0, r0, 5
Loop:   beq  r0, 0, done
        ldr  r1, 0x4
        ldr  r2, 0x24
        add  r0, r0, -1
        bl    loop
done:
```

Miss Rate = ?

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
0			0			Set 1
0			0			Set 0

两路组相联高速缓存性能举例

```
        addi r0, r0, 5
Loop:   beq  r0, 0, done
        ldr  r1, 0x4
        ldr  r2, 0x24
        add  r0, r0, -1
        j    loop
done:
```

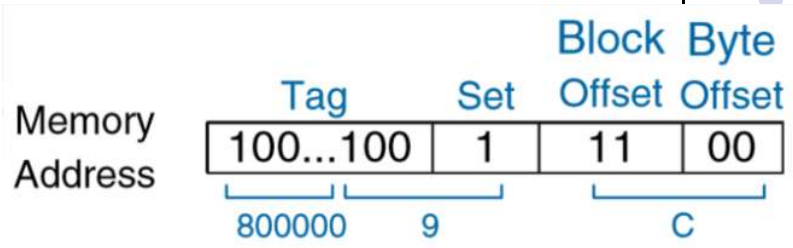
Miss Rate = 2/10

= 20%

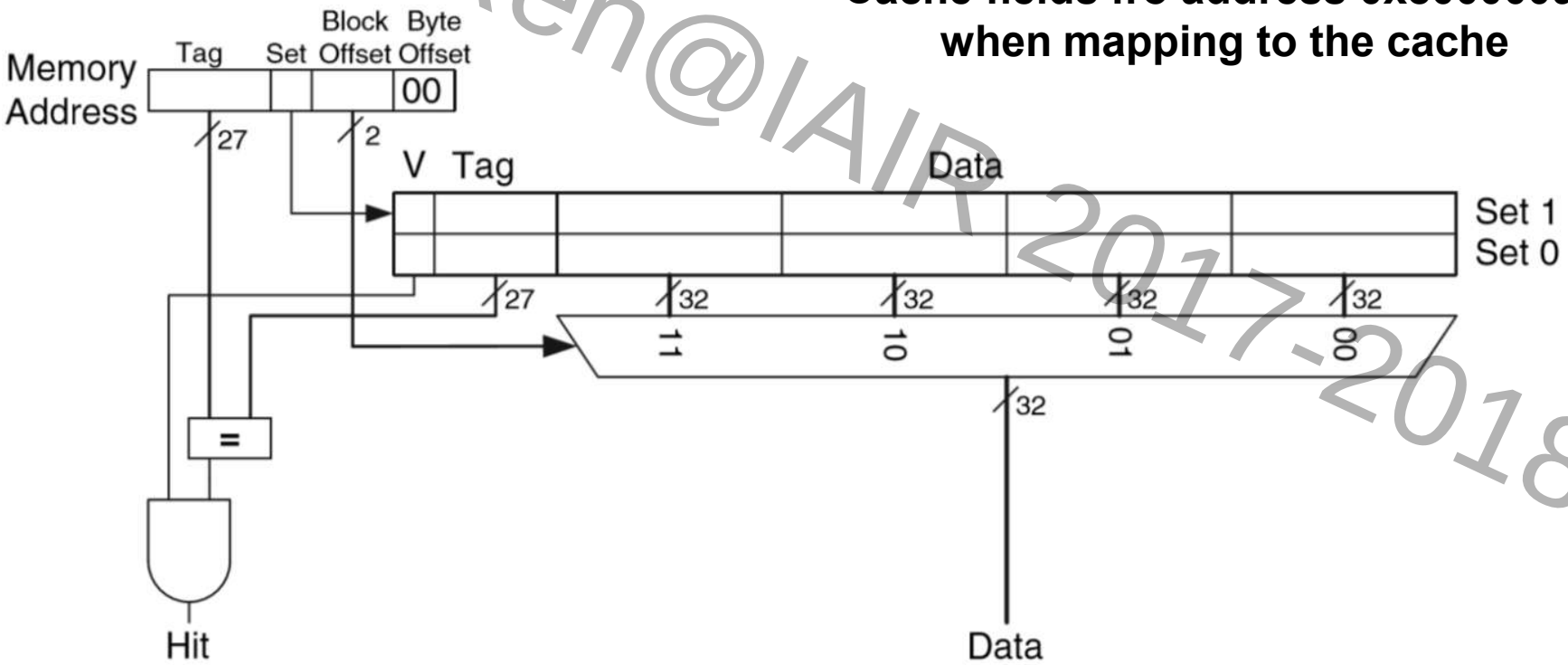
Associativity reduces conflict misses

Way 1			Way 0			
V	Tag	Data	V	Tag	Data	
0			0			Set 3
0			0			Set 2
1	00...10	mem[0x00...24]	1	00...00	mem[0x00...04]	Set 1
0			0			Set 0

Cache(空间相关性)



Cache fields fro address 0x8000009C when mapping to the cache



Direct mapped cache with two sets and a four-word block size

Cache：Put-it-together

- **Caches are organized as two-dimensional arrays. The rows are called sets, and the columns are called ways. Each entry in the array consists of a data block and its associated valid and tag bits.**
- 缓存被组织为二维数组。 这些行被称为**集(Set)**，而这些列被称为**组(Way)**。 阵列中的每个条目由一个**数据块(cache block)**及其相关的**有效(Valid)**和**标签位(tag)**组成。

Cache address format

[tag | index(set) | block or line offset | byte offset]

Quiz 1: for a 256-byte cache block and 4-byte accesses the block offset would be ? bits and the byte offset ? bits.

4-byte access (word access)
256/4=64 so, 6 bits block offset
2bits byte-offset to specific which byte inside a word

Quiz 2: for direct-mapped cache 128KB in which each block has 8 32-bit words, How many bits are needed for the tag and index fields, assuming a 32-bit address ?

(128K*8)/(8*32)=4096, so Index = 2^12
Block offset = 3bits, byte offset = 2bits
Tag bits = 32-12-3-2=15

Quiz 3: for 4-ways cache 128KB in which each block has 4 32-bit words, How many bits are needed for the tag and index fields, assuming a 32-bit address ?

替换策略

- 替换策略：在高速缓存“满”且需要载入新的数据块时，需要考虑何种旧数据块需要被移出高速缓存。
- 直接映射不存在替换策略问题，因为每个数据块对应唯一映射地址，其替换的数据块也是确定的。
- 组相联与全相联高速缓存需要考虑替换策略：
  - 随机替换（**Random**）：在组内或全相联高速缓存中随机选择一个数据块被替换。优点：硬件简单，耗费低。缺点：命中率相对较低。
  - 最近最少使用策略（**Least Recently Used, LRU**）：将组内或全相联高速缓存中最近最少使用的数据块。

    优点：命中率高。缺点：硬件耗费大。
  - 先入先出策略（**First In First Out, FIFO**）：折中！**Trade off**

LRU 替换策略

```
ldr r0, #0x04
ldr r1, #0x24
ldr r2, #0x54
```

Way 1				Way 0			
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	0	00...010	mem[0x00...24]	1	00...000	mem[0x00...04]	Set 1 (01)
0	0			0			Set 0 (00)

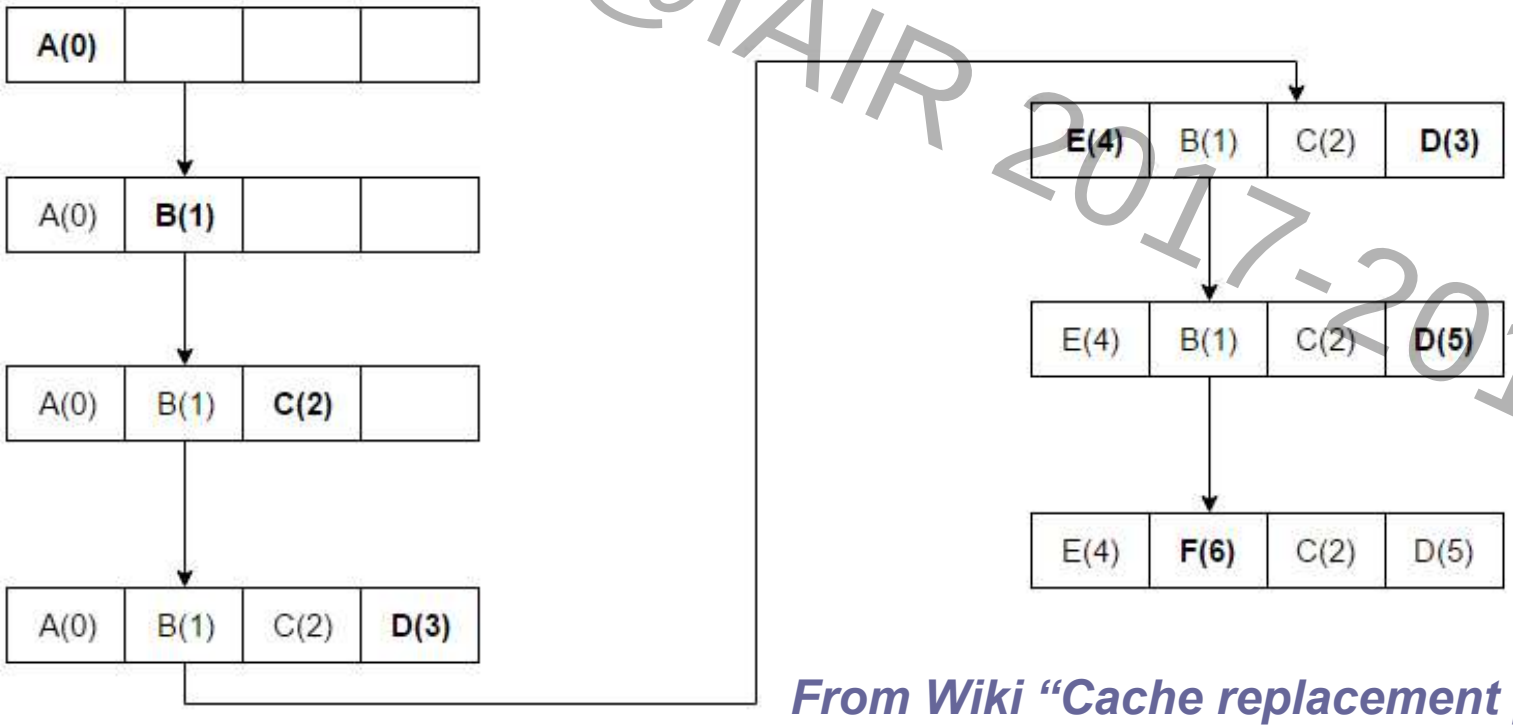
(a)

Way 1				Way 0			
V	U	Tag	Data	V	Tag	Data	
0	0			0			Set 3 (11)
0	0			0			Set 2 (10)
1	1	00...010	mem[0x00...24]	1	00...101	mem[0x00...54]	Set 1 (01)
0	0			0			Set 0 (00)

(b)

LRU 替换策略

Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards *the* least recently used item. General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits. In such an implementation, every time a cache-line is used, the age of all other cache-lines changes.



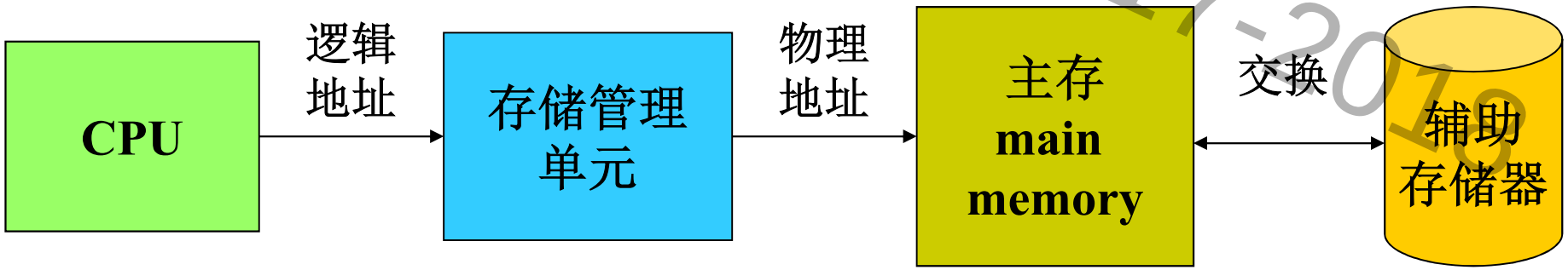
From Wiki "Cache replacement policies"

Cache Summary

- **What data is held in the cache?**
  - Recently used data (temporal locality)
  - Nearby data (spatial locality)
- **How is data found?**
  - Set is determined by address of data
  - Word within block also determined by address
  - In associative caches, data could be in one of several ways
- **What data is replaced?**
  - Least-recently used way in the set (LRU)

3 存储管理单元和地址转换

- 存储管理单元在**CPU**和物理主存之间，实现逻辑/虚拟地址到物理地址之间的转换。转换过程也称内存映射。
  - 物理地址（**physical address**）：用于内存芯片级的单元寻址，与实际的**RAM**单元对应。
  - 逻辑地址（**logical address/virtual address**）：程序的抽象地址空间而不与实际的**RAM**单元对应。

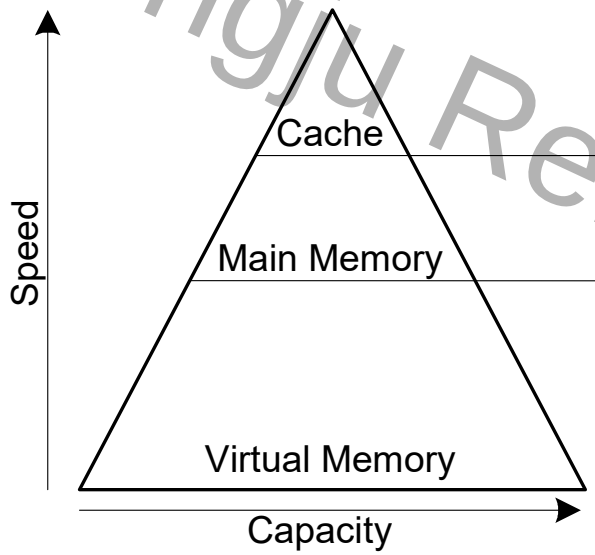


（**Memory Management Unit, MMU**）

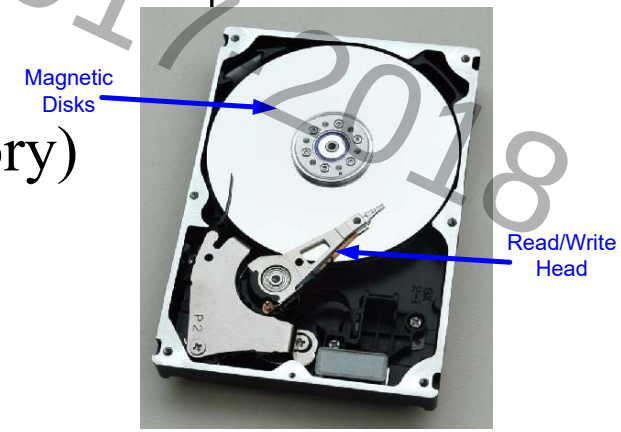
虚拟内存

- 什么是虚拟内存？？？
- 虚拟内存能做什么？？？

虚拟内存



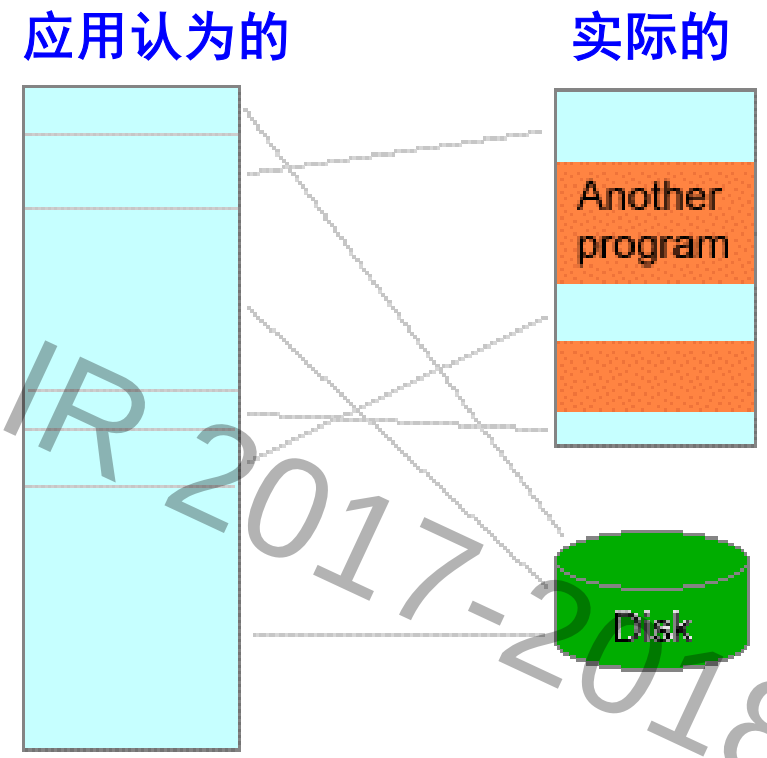
Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1



- **Physical Memory:** DRAM (Main Memory)
- **Virtual Memory:** Hard drive
  - Slow, Large, Cheap

虚拟内存与虚拟寻址

- 虚拟内存是计算机系统内存管理的一种技术。它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间），而实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换。
  -- from wikipedia



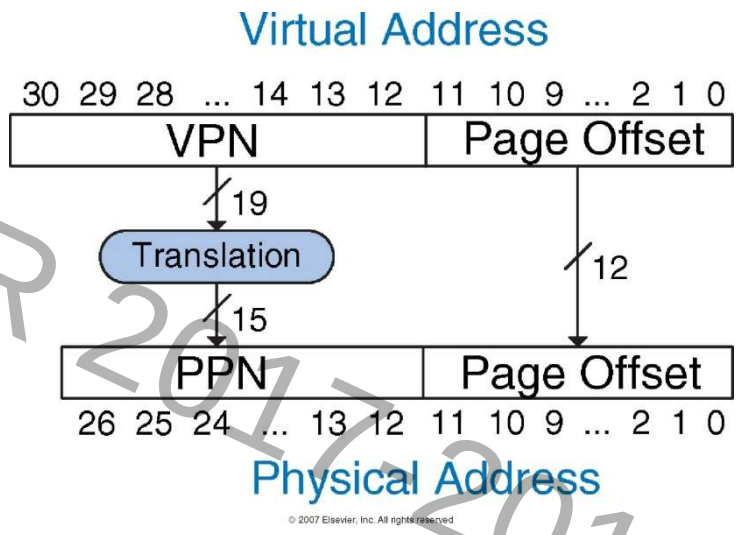
虚拟寻址（**virtual addressing**）:将逻辑地址转换成物理地址的寻址模式。

虚拟内存

- **Virtual addresses**
  - Programs use virtual addresses **(**程序使用虚拟内存**)**
  - Entire virtual address space stored on a hard drive
  - Subset of virtual address data in DRAM
  - CPU translates virtual addresses into ***physical addresses*** (DRAM addresses)
  - Data not in DRAM fetched from hard drive
- **Memory Protection**
  - Each program has own virtual to physical mapping **(V to P)**
  - Two programs can use same virtual address for different data
  - Programs don't need to be aware others are running
  - One program (or virus) can't corrupt memory used by another

虚拟内存 与 Cache类比

Cache	Virtual Memory
Block	Page
Block Size	Page Size
Block Offset	Page Offset
Miss	Page Fault
Tag	Virtual Page Number



Physical memory acts as cache for virtual memory

虚拟内存

- **Page size:** amount of memory transferred from hard disk to DRAM at once **(类比Cache Block size)**
- **Address translation:** determining physical address from virtual address
- **Page table:** lookup table used to translate virtual addresses to physical addresses

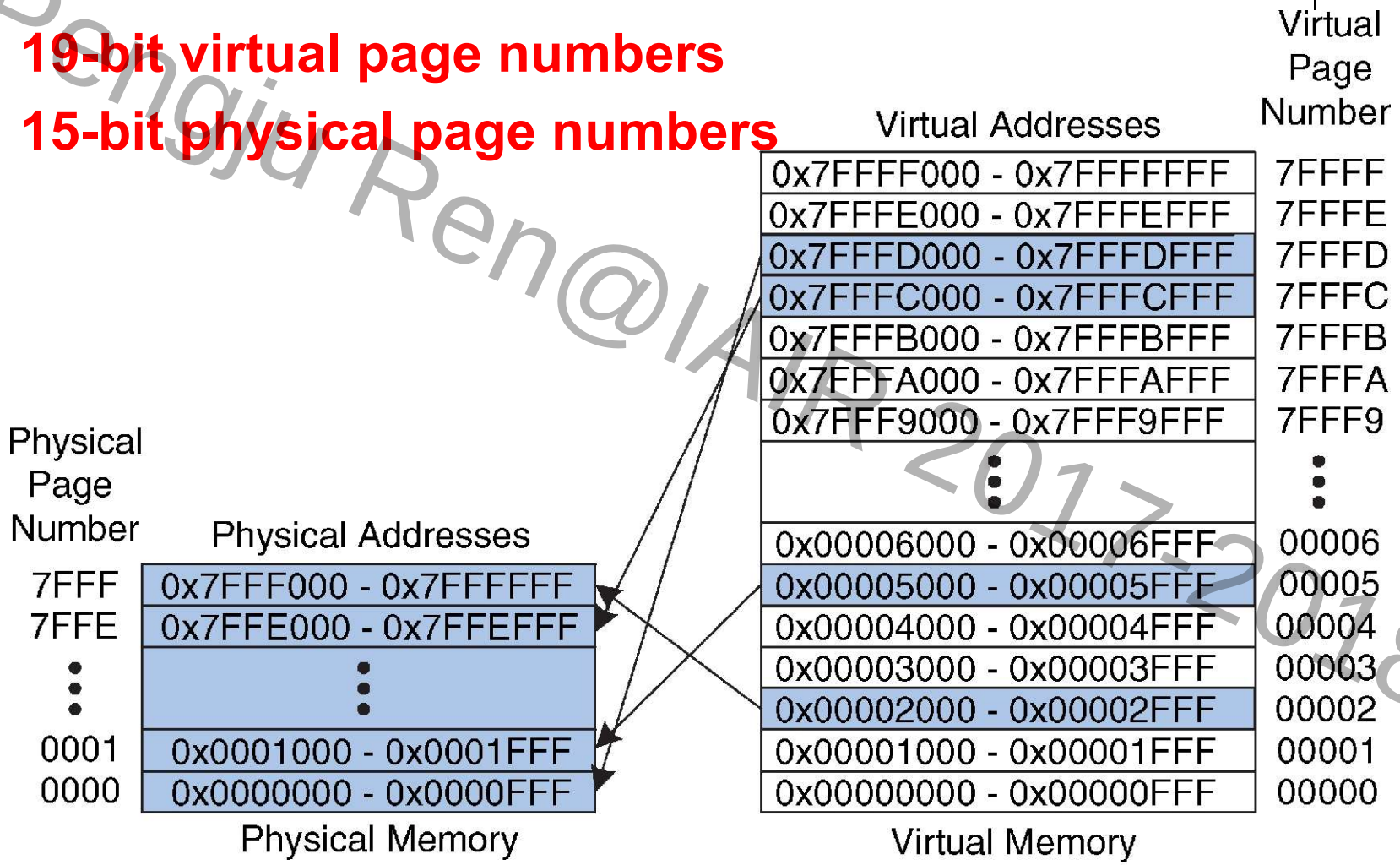
- **System:**
  - Virtual memory size: 2 GB = 2^{31} bytes
  - Physical memory size: 128 MB = 2^{27} bytes
  - Page size: 4 KB = 2^{12} bytes

虚拟内存

- **System:**
  - Virtual memory size: 2 GB = 2^{31} bytes
  - Physical memory size: 128 MB = 2^{27} bytes
  - Page size: 4 KB = 2^{12} bytes
- **Organization:**
  - Virtual address: **31** bits
  - Physical address: **27** bits
  - Page offset: **12** bits
  - # Virtual pages = $2^{31}/2^{12}$ = $\mathbf{2^{19}}$ (VPN = 19 bits)
  - # Physical pages = $2^{27}/2^{12}$ = $\mathbf{2^{15}}$ (PPN = 15 bits)

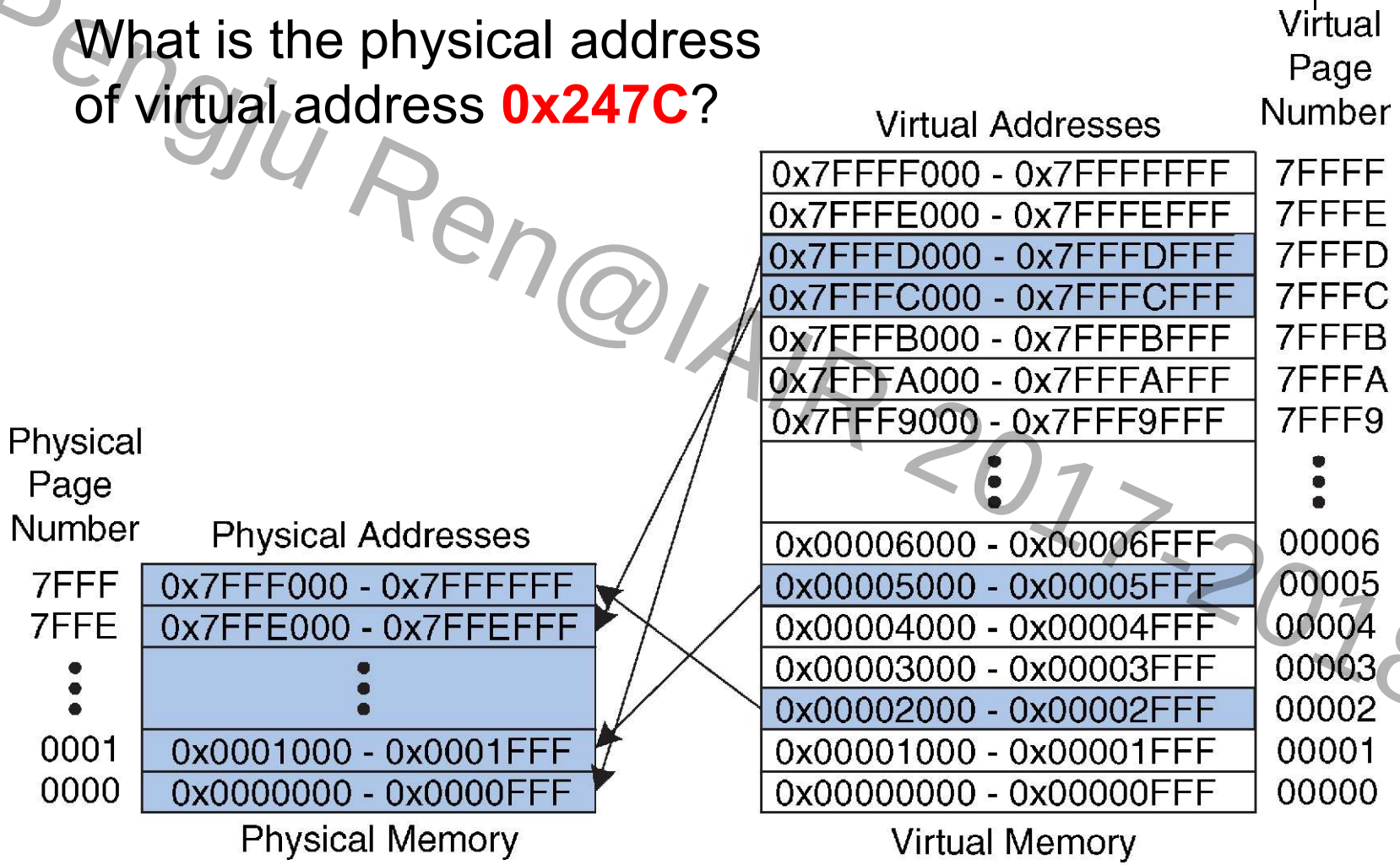
虚拟内存

- **19-bit virtual page numbers**
- **15-bit physical page numbers**



虚拟内存

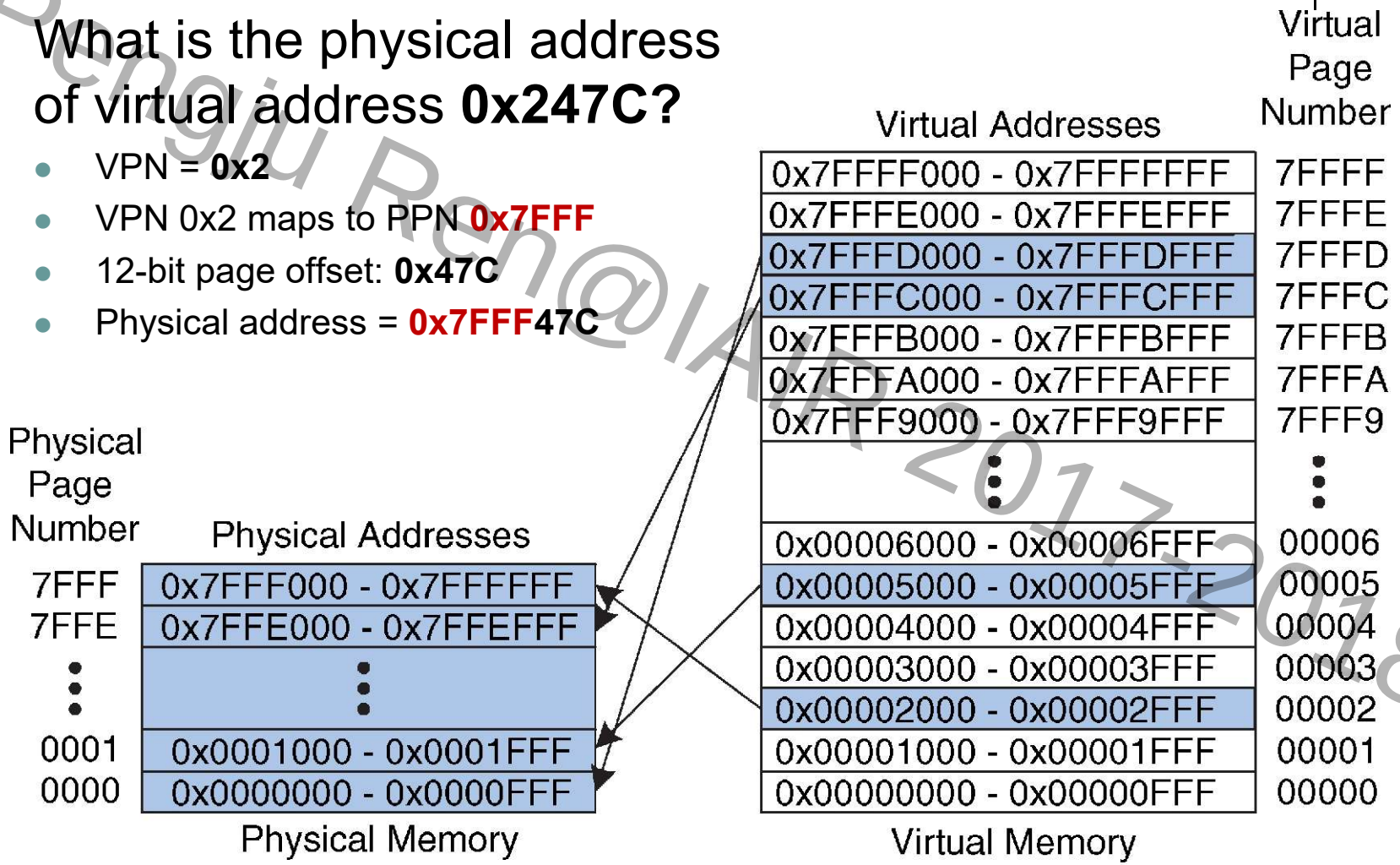
What is the physical address of virtual address **0x247C**?



虚拟内存

What is the physical address of virtual address **0x247C?**

- VPN = **0x2**
- VPN 0x2 maps to PPN **0x7FFF**
- 12-bit page offset: **0x47C**
- Physical address = **0x7FFF47C**

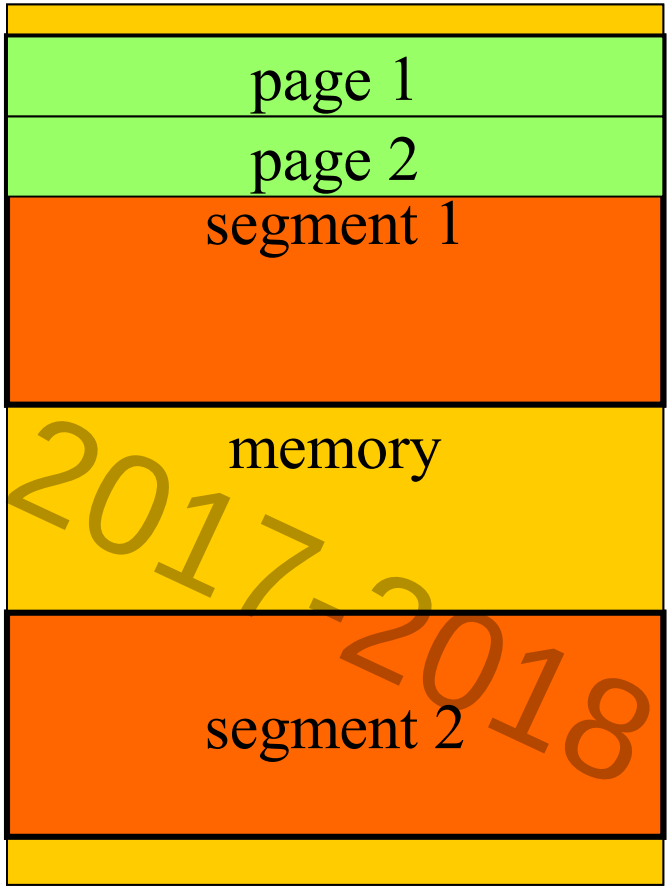


存储管理单元

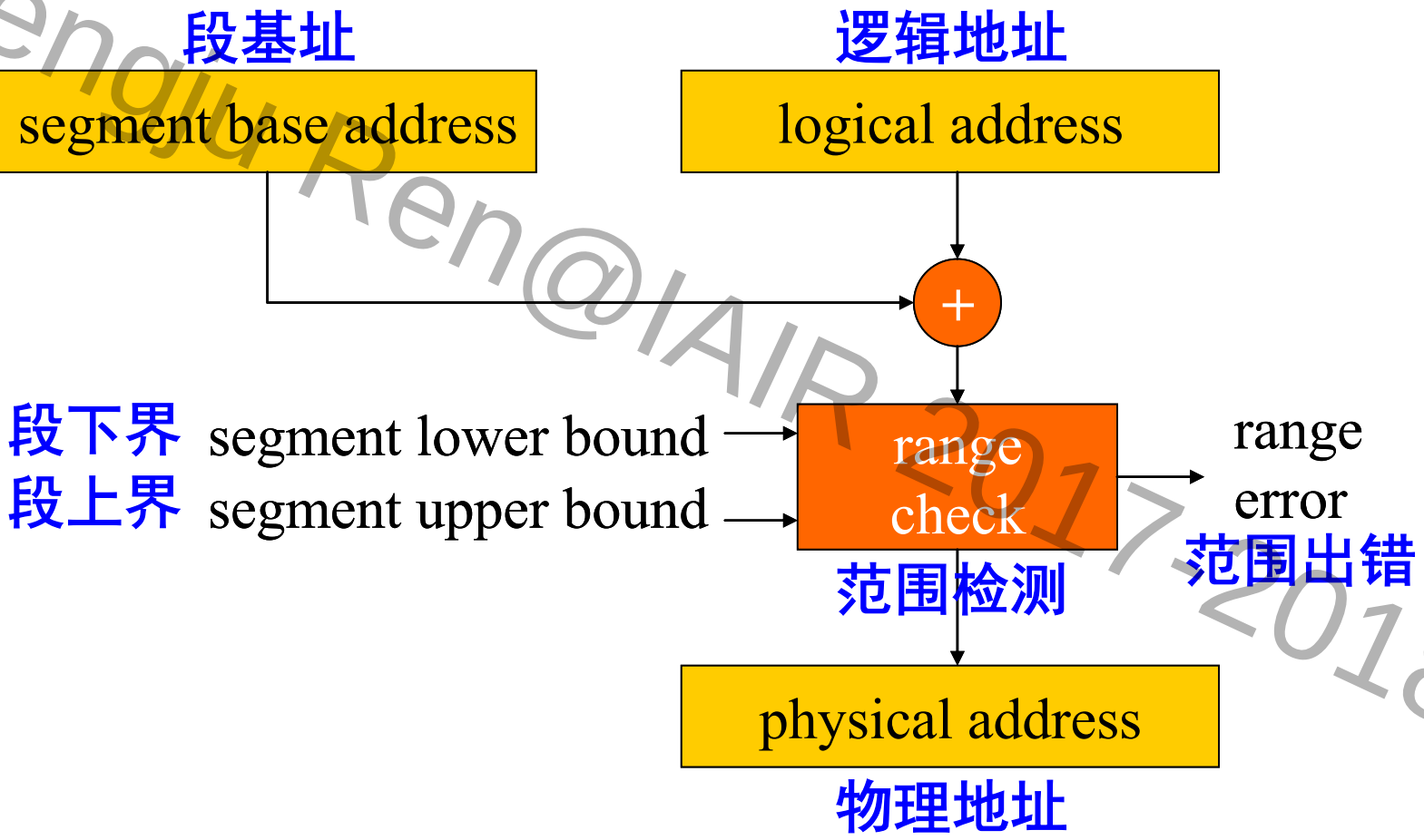
- 存储管理单元实现的功能：
  - 实现虚拟寻址：用表把逻辑地址转换成物理地址。
  - 可以将部分程序移入辅助存储器，当程序执行需要时，再将镜像调入主存。
  - **缺页（page fault）**异常：**CPU**请求一个不在主存的地址。
- 缺页异常重新启动的条件：
  - 所有内容已读到主存中；
  - **MMU**表已经进行了相应的更新。
- **LRU**替换策略。

地址转换方法

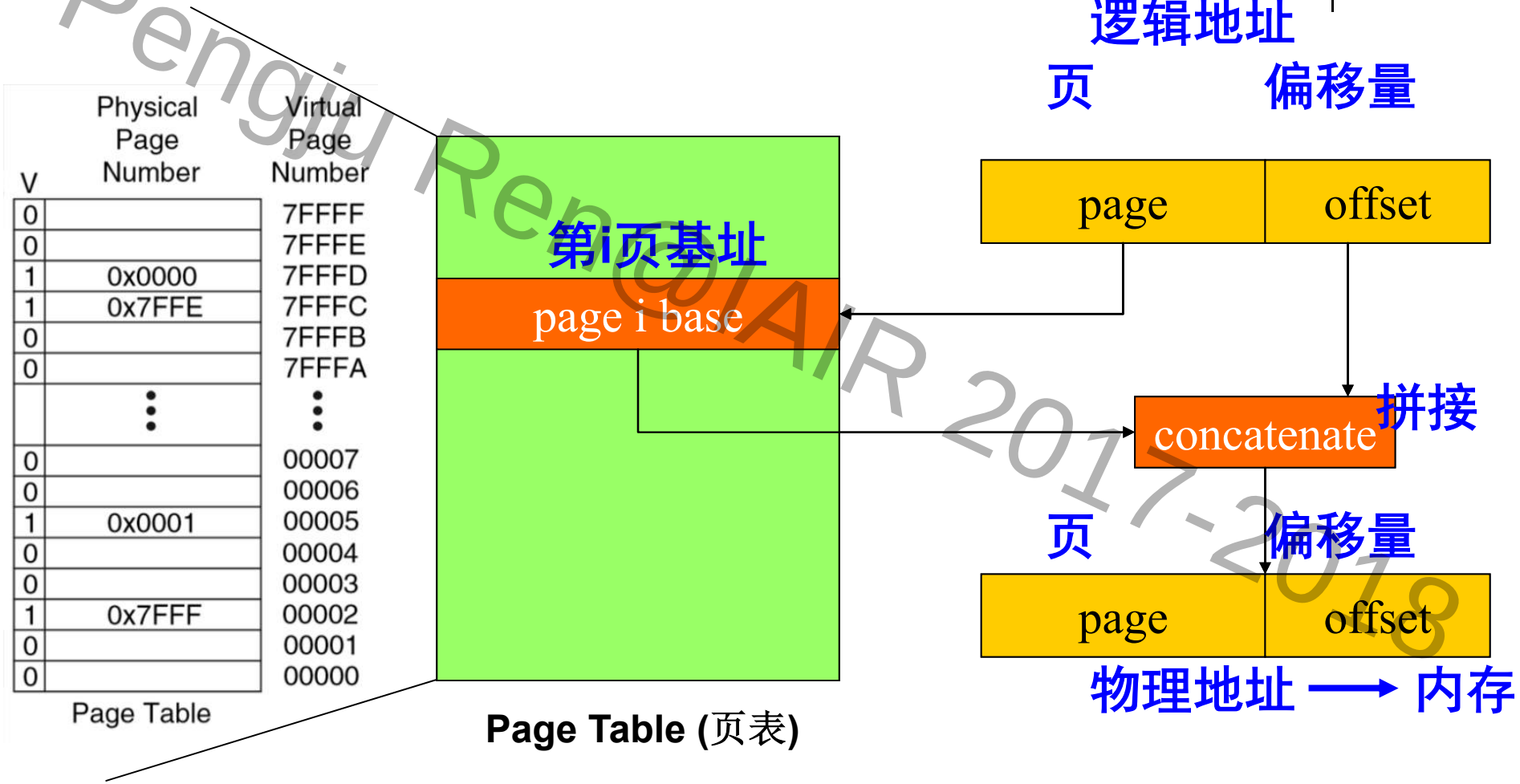
- 需要某种寄存器或表来实现逻辑地址到物理地址之间的任意转换。
- 两种基本地址转换方法：
  - 分段：支持较大的、大小可不等的内存区域。
    - 起始地址+大小。
  - 分页：支持较小的、大小相等的内存区域。
- 段页式寻址模式：将每个段分成页并且其地址转换分成两步的方法来建立。



段地址转换

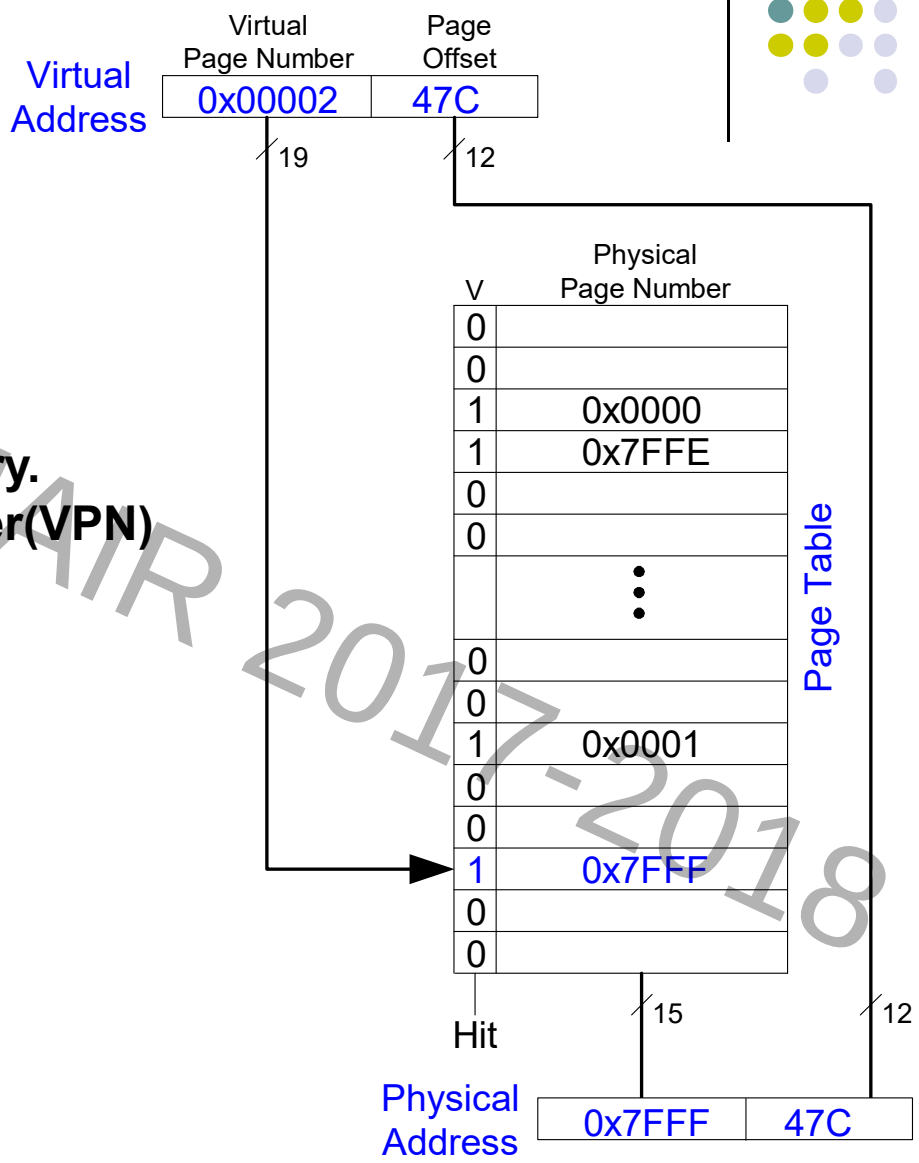


页地址转换



页地址转换

Page Table is stored in Physical Memory.
It is indexed by the Virtual Page Number(VPN)



页地址转换

What is the physical address of virtual address **0x5F20**?

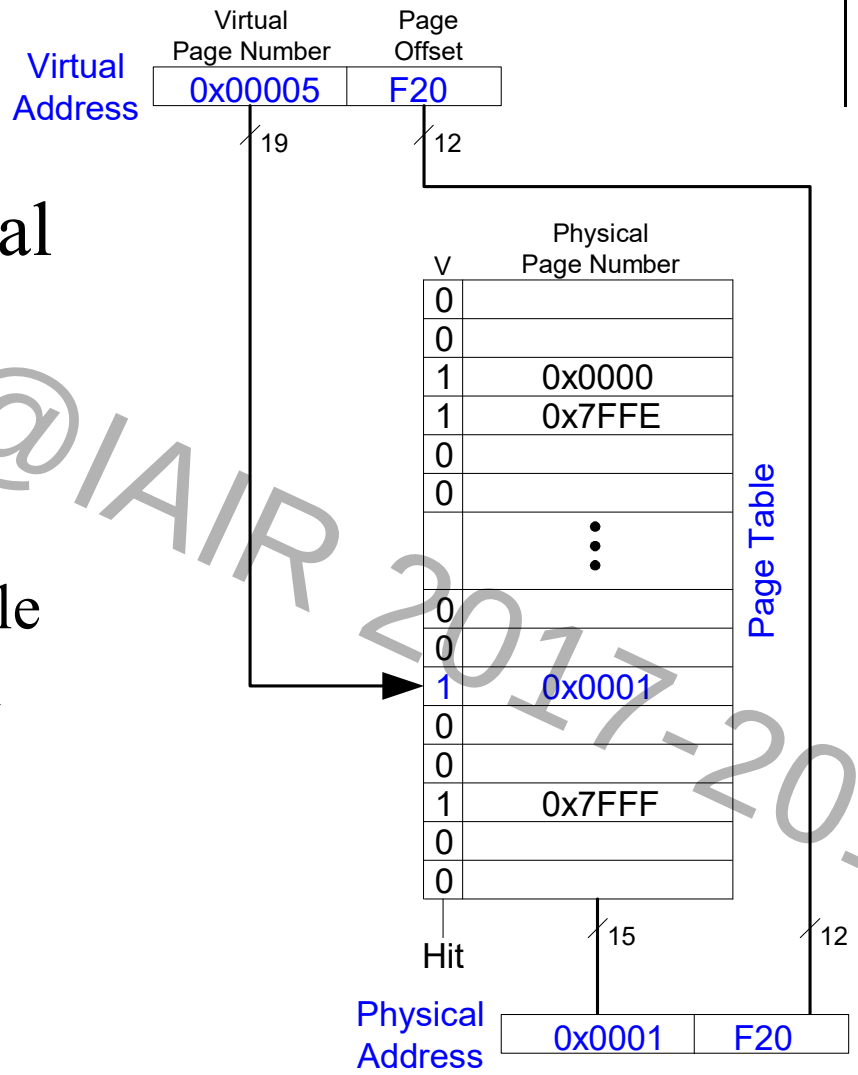
V	Physical Page Number
0	
0	
1	0x0000
1	0x7FFE
0	
0	
	⋮
0	
0	
1	0x0001
0	
0	
1	0x7FFF
0	
0	

Page Table

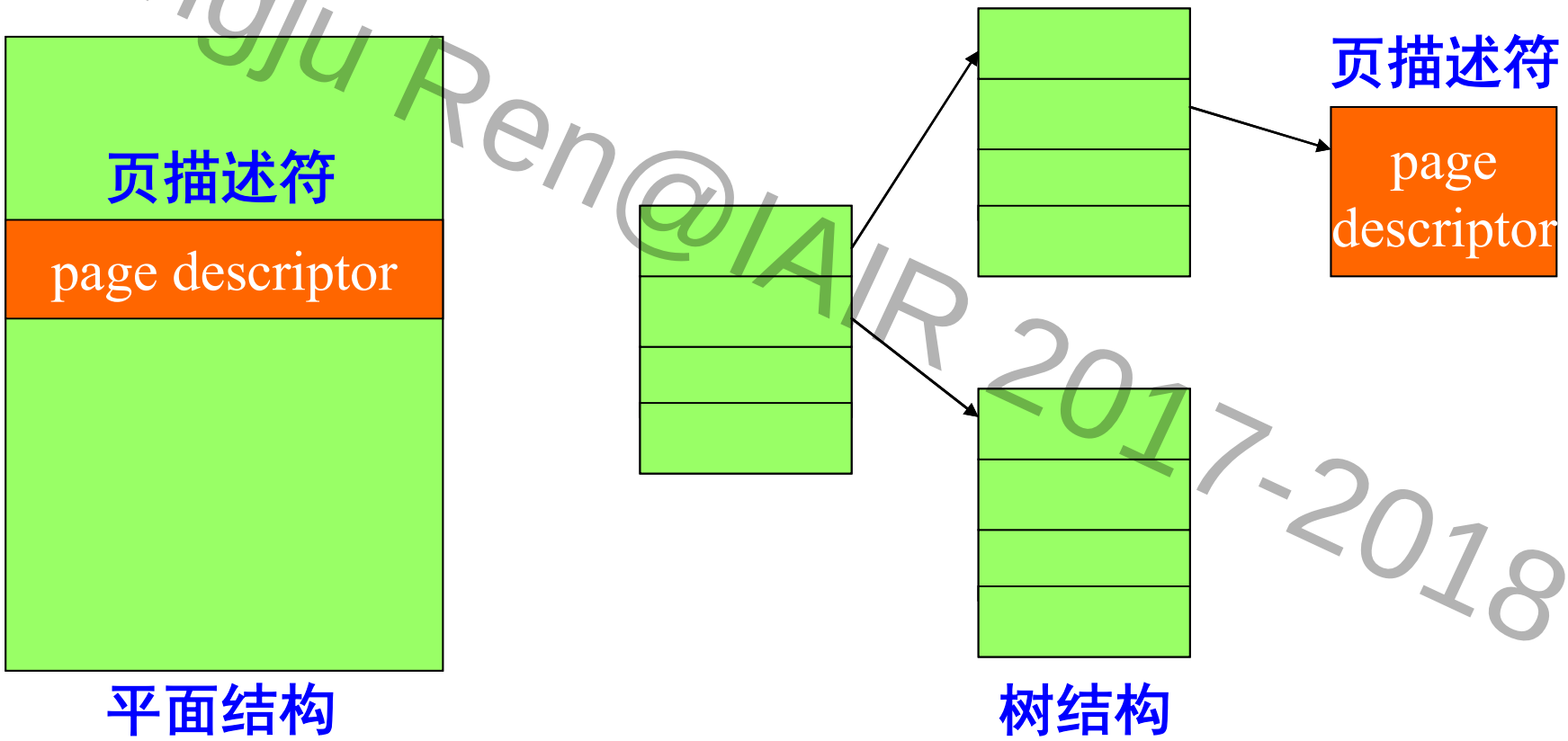
页地址转换

What is the physical address of virtual address **0x5F20**?

- VPN = **5**
- Entry 5 in page table VPN 5 => physical page **1**
- Physical address: **0x1F20**



页表组织模式



转换后援缓冲区

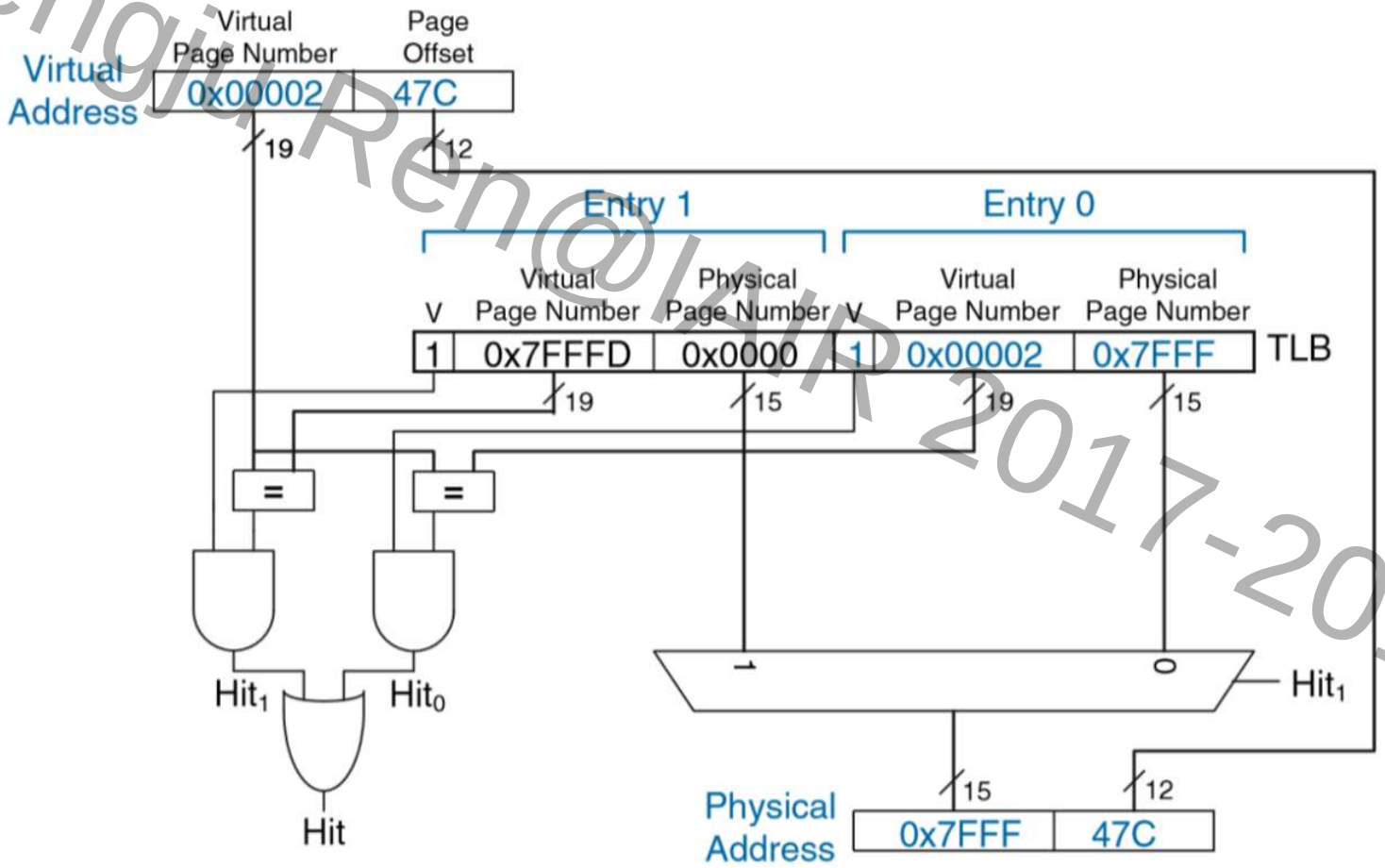
- 大的转换表需要访问主存。
- 转换后援缓冲区（**Translation Lookaside Buffer, TLB**）：缓存转换表，通常很小。

Virtual Address	Physical Address	Dirty	Ref	Valid	Access

  - **Valid**：表示逻辑段或逻辑页当前是否在物理内存。
  - **Dirty**：表示该段或该页是否已被改写。
  - **Ref&Access**：权限位。

TLB(Translation Look-aside Buffer)

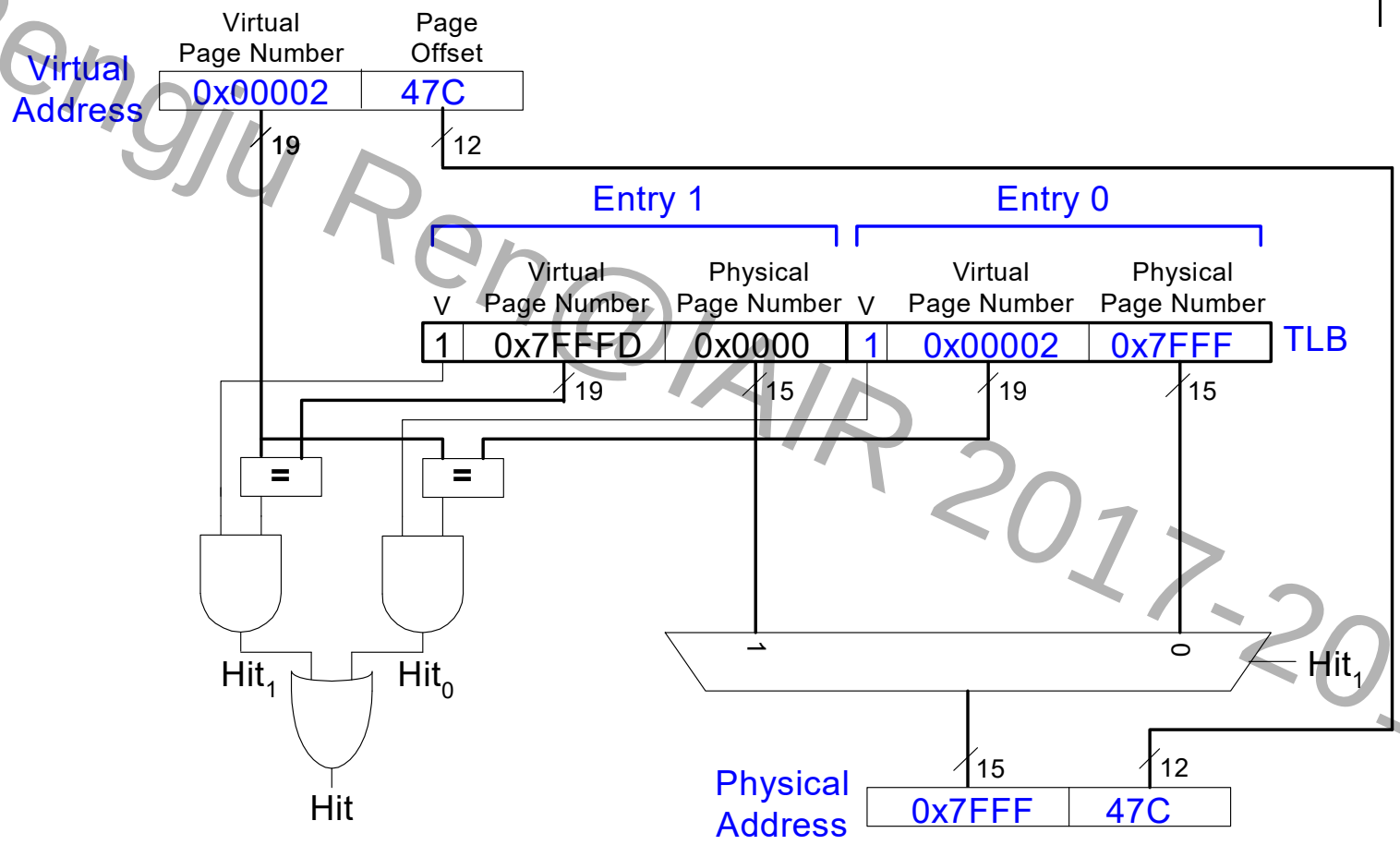
使用Cache结构存放最近使用的页表项。



TLB(Translation Look-aside Buffer)

- Page table accesses: high temporal locality
  - Large page size, so consecutive loads/stores likely to access same page
- TLB
  - Small: accessed in < 1 cycle
  - Typically 16 - 512 entries
  - **Fully associative**（全相联）
  - > 99 % hit rates typical
  - Reduces # of memory accesses for most loads/stores from 2 to 1

TLB(Translation Look-aside Buffer)



2-Entry TLB 示例

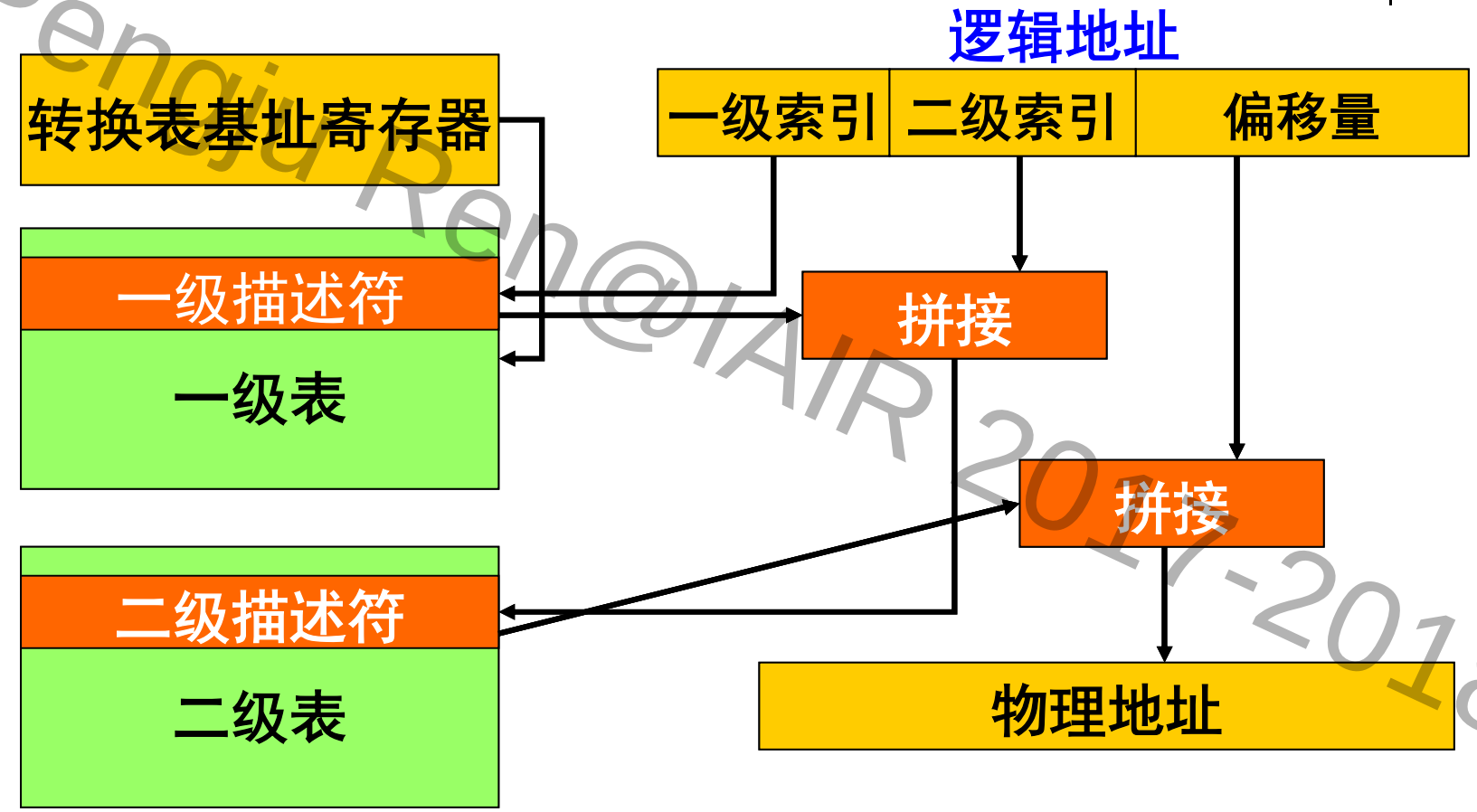
虚拟内存小结

- Virtual memory increases **capacity**
- A subset of virtual pages in physical memory
- **Page table** maps virtual pages to physical pages – address translation
- A **TLB** speeds up address translation
- Different page tables for different programs provides **memory protection**

ARM的内存管理

- 存储管理单元是一个可选部件，支持虚拟地址转换和内存保护。**ARM的MMU**支持下列内存区域类型：
  - **1MB**内存块大小的段
  - **64KB**的大页**(16bits)**
  - **4KB**的小页**(12bits)**
- 地址可以在选择段映射或者页映射。
- 采用二级地址转换模式。

ARM两级页表地址转换



ARM两级页表地址转换(续)