嵌入式系统设计与应用
复习课

西安交通大学电信学院
任鹏举

Pengju Ren@IAIR 2017-2018

考察范围：

第一章： 1.1-1.2（了解）, 1.3.1-1.3.5
第二章： 2.1, 2.2, 2.3
第三章： 3.1, 3.2, 3.3, 3.4, 3.5, 3.6,3.7 其中 3.2.4 的 5（ C55X 中断） 6 PIC16F 中断不做要求
第四章： 4.1 – 4.5，4.6（了解）， 4.7
第五章： 5.1 –5.9
第六章： 6.1 – 6.7

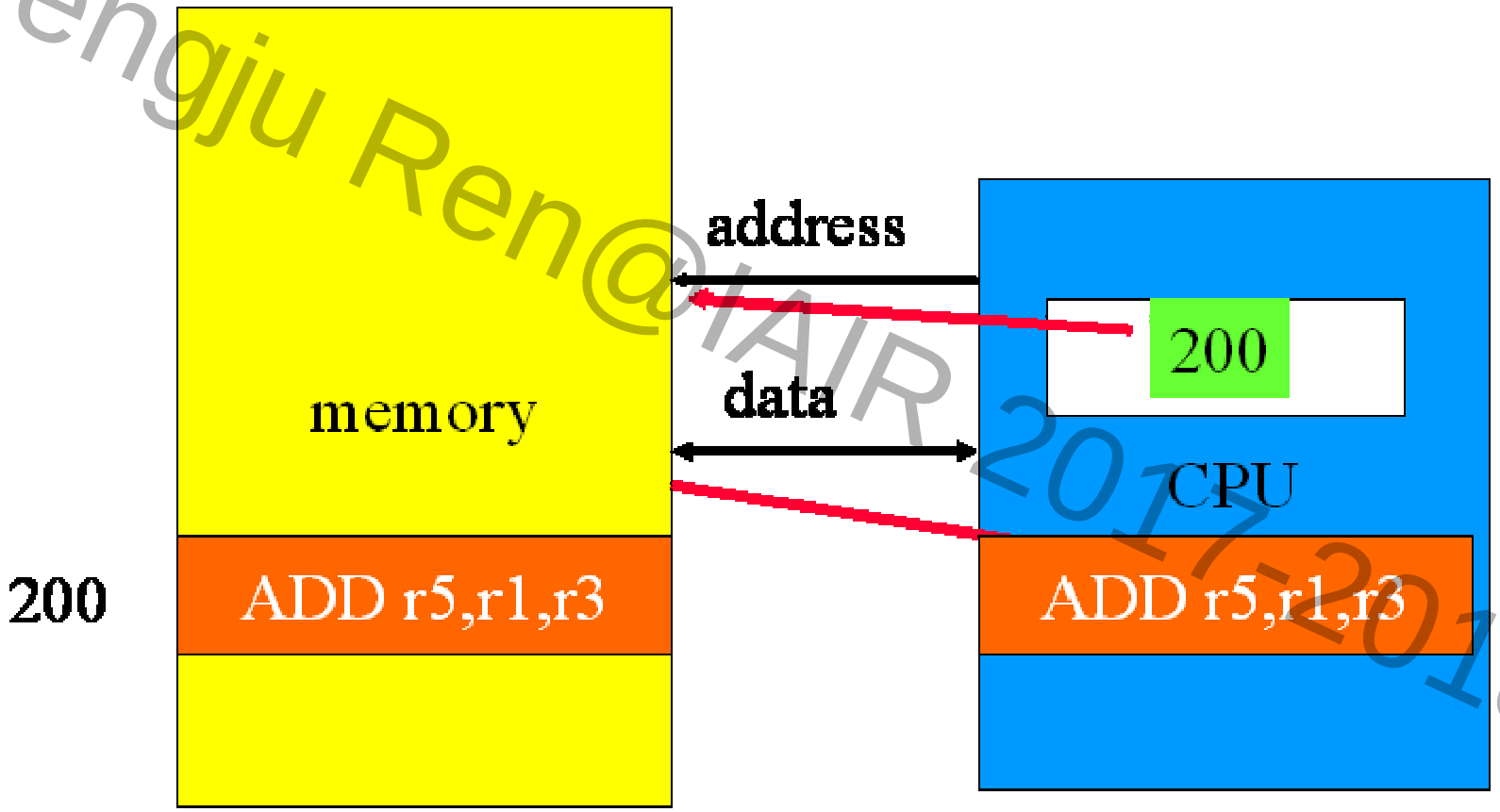
Pengju Ren@IAIR 2017-2018

考题种类

- 嵌入式部分卷面 50 分，其中：
  - 不定项选择题 10 分 （ 10题）
  - 问答题20分（ 3题）
  - 计算题10分（ 2 题）
  - 编程题10分（ 2 题， C 和汇编各 1，难度不超过教材以及课后习题）

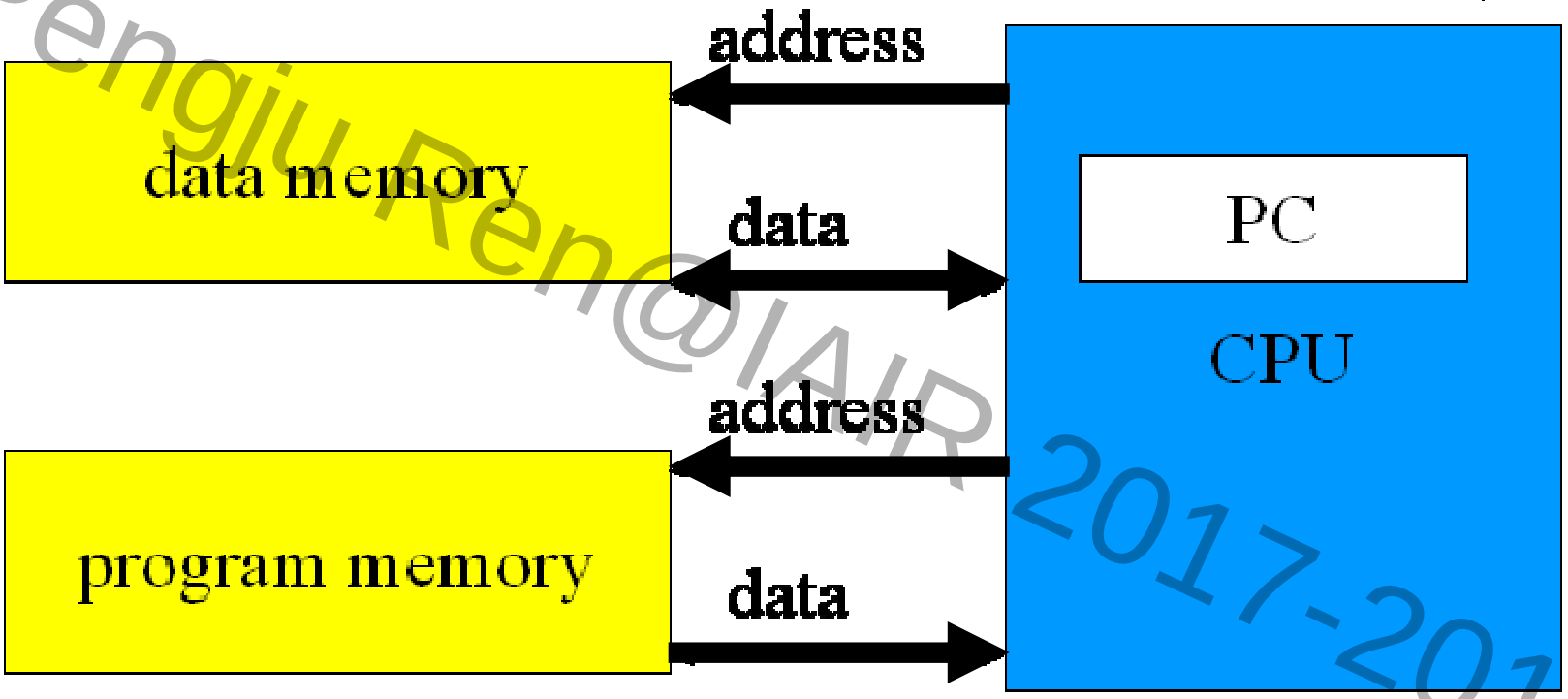
Pengju Ren@IAIR 2017-2018

冯·诺伊曼体系结构



Pengju Ren@IAIR 2017-2018

哈佛体系结构



Pengju Ren@IAIR 2017-2018

独立的程序和数据存储器避免了存储端口的竞争，为数字信号处理提供了较高的性能，因此大部分**DSP**都采用哈佛体系结构。

CISC vs. RISC

- **CISC（Complex Instruction Set Compuer）**复杂指令集计算机，诞生于**20**世纪**70**年代，是**1980**年前指令集设计的主要趋势。
  - 背景：处理器比主存储器速度快。
  - 采用大量变长形式的不同指令，指令执行周期差别较大。
  - **Intel**的**x86**系列**CPU**是**CISC**的代表。
- **RISC（Reduced Instruction Set Computer）**精简指令集计算机，诞生于**1980**年**--**指令集日益复杂的时刻。

Peng Peng@IAR 2017-2018

RISC的优点与缺点

- 芯片面积小：RISC CPU设计简单，电路规模小。
- 开发时间短：简单的处理器占用设计力量小，设计费用低。
- 性能高：流水线处理、高时钟频率和单周期执行。

- 代码密度低：指令集固定长度，占用更大的存储资源和带宽，造成更高的功耗。
- RISC不能执行x86代码：在一段时间里，由于x86强大的生态环境，RISC主要应用在x86不涉及的领域。

Pengju Ren@IXIR 2017-2018

ARM编程模型

- 用户级程序：**15**个通用**32**位寄存器、程序计数器（**r15**）和当前状态寄存器（**CPSR**，**Current Program Status Register**）。
- **CPSR**在用户级编程时用于存储条件码（自动设置）。
  - **N**：负数。改变标志位的最后一次**ALU**操作的结果为负。
  - **Z**：零。改变标志位的最后一次**ALU**操作的结果为**0**。
  - **C**：进位。改变标志位的最后一次**ALU**操作产生进位输出。
  - **V**：溢出位。改变标志位的最后一次算术**ALU**操作产生符号位的溢出
  - **IF**：中断使能。
  - **T**：指令集选择。
  - **Mode:**控制处理器的模式。

Pengju Ren@IWR 2017-2018

ARM编程模型

- **N**：当用两个补码表示的带符号数进行运算时，**N=1**表示运算的结果为负数；**N=0**表示运算的结果为正数或零。
- **Z**：**Z=1**表示运算结果为零；**Z=0**表示运算结果为非零。
- **C**：可以有**4**种方法设置**C**的值
  - 加法运算（包括**CMN**）：产生进位**C=1**，否则**C=0**。
  - 减法运算（包括**CMP**）：**产生借位C=0，否则C=1。**
  - 包含**RRX**的非加/减运算指令：**C**为移出值的最后一位。
  - 对于其他非加减运算指令，**C**值通常保持不变。
- **V**：当操作数和运算结果为二进制的补码表示的带符号数时，**V=1**表示符号位溢出。对于其他非加减运算，**V**的值通常保持不变。

Pengju Ren@IWR 2011-2018

ARM的Load-Store体系结构

- **ARM**指令都属于下列三种类型之一：
  - **数据处理指令：**只能使用或改变寄存器中的值。
    - 加、减、逻辑与、逻辑或
  - **数据传送指令：**把存储器中的值拷贝到寄存器中，或者相反。
    - **Load**、**Store**、存储器和寄存器值交换（系统）
  - **控制流指令：**使指令执行切换到不同的地址
    - 转移、转移和链接或陷入系统代码

Pengju Ren@XIR 2017-2018

基本的汇编范式

if statements

```
if [C] then [T] else [E]

        [C]
        BNE else
        [T]
        B   endif
else:
        [E]
endif:
```

repeat loops

```
do {[S]} while ([C])

loop:
        [S]
        [Test C]
        BEQ   loop
endw:
```

Pengju Ren@IAIR 2017-2018

基本的汇编范式

while loops

```
while ( C ) { S }
```

```
loop:   Test C
        BNE endw
        S
        B   loop
endw:
```

```
        B test
loop:   S
test:   Test C
endw:
```

Pengju Ren@IAIR 2017-2018

基本的汇编范式

for loops

```
for ( I ; C ; A ) { S }
```

```
for (i=0; i<10; i++)
  { a[i]:=0; }
```

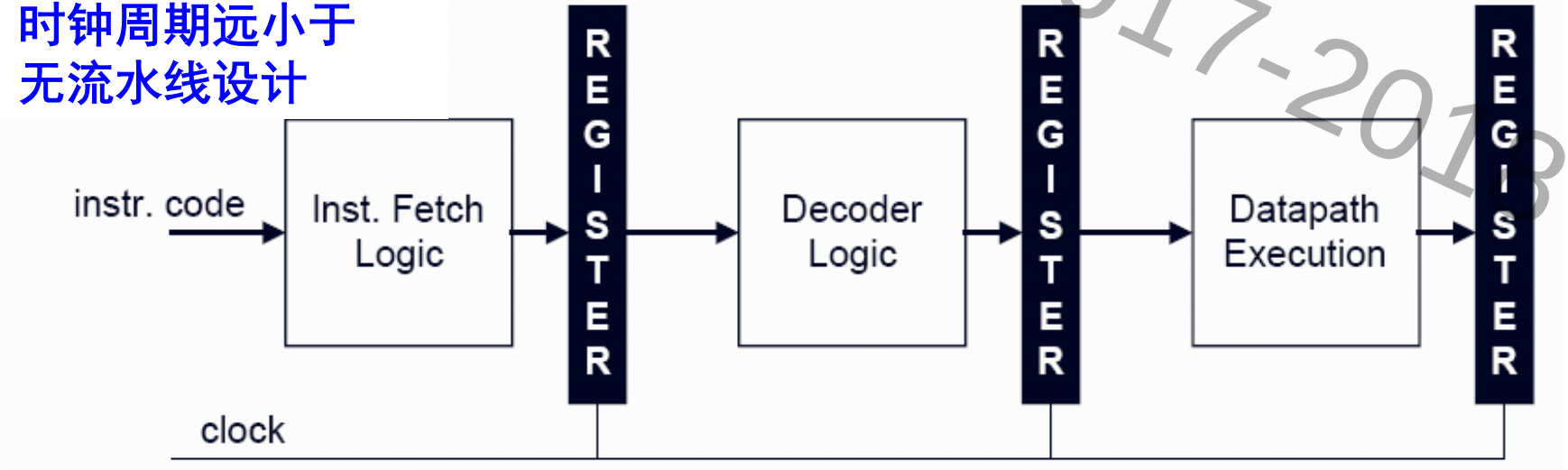
```
        I
loop:   C
        BNE endfor
        S
        A
        B   loop
endfor:
```

Pengju Ren@IAIR 2017-2018

ARM7的流水线

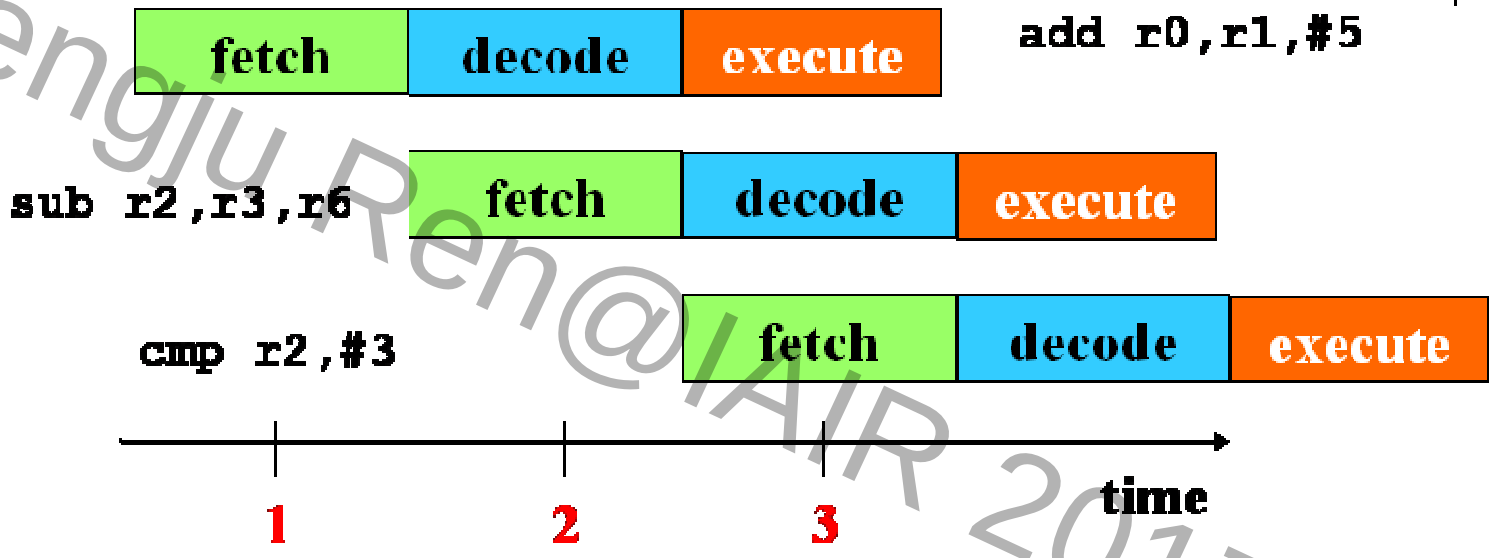
- **ARM7使用三级流水线(ARM9使用五级流水线结构)**
  - 获取：从存储器中获取指令代码到指令流水线。
  - 译码：指令译码，通过控制信号操作数据通路准备执行。
  - 执行：指令获得数据通路“使用权”：读寄存器, 移位操作, **ALU**运算和写回。
- 每一级的结果存储在寄存器中。

时钟周期远小于
无流水线设计



Pengju Ren@IAIR 2017-2018

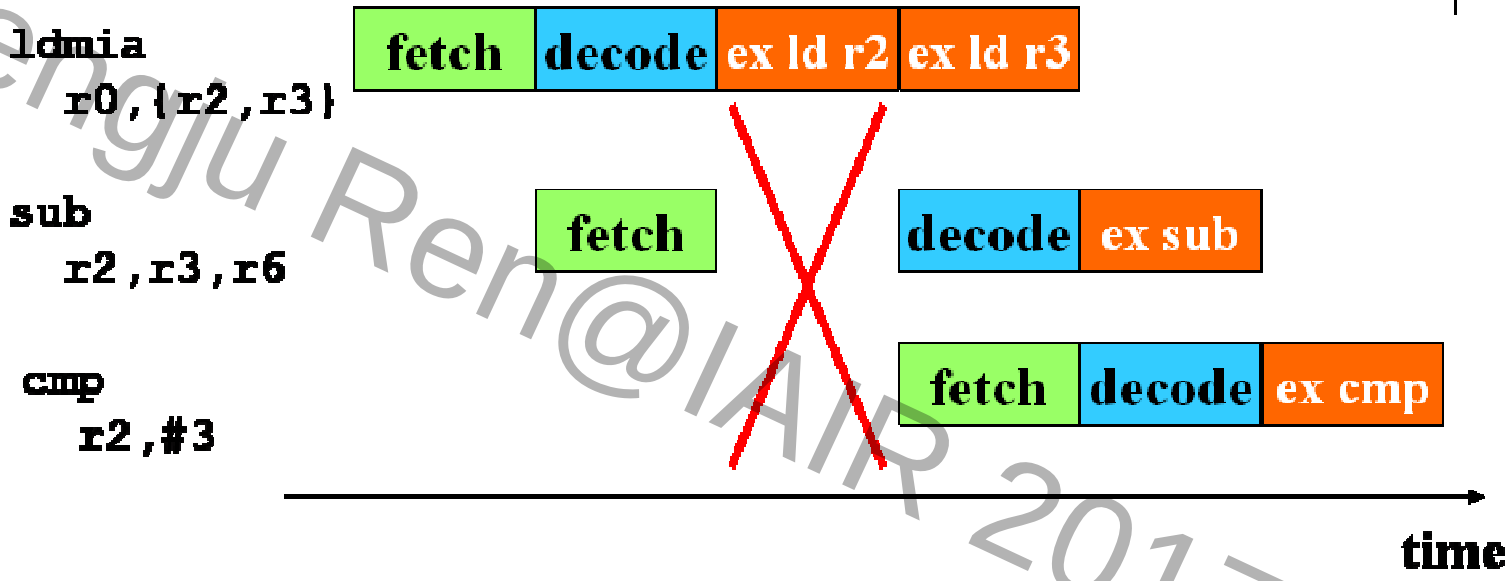
ARM的流水线（续）



- 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。

Pengju Ren@IAIR 2017-2018

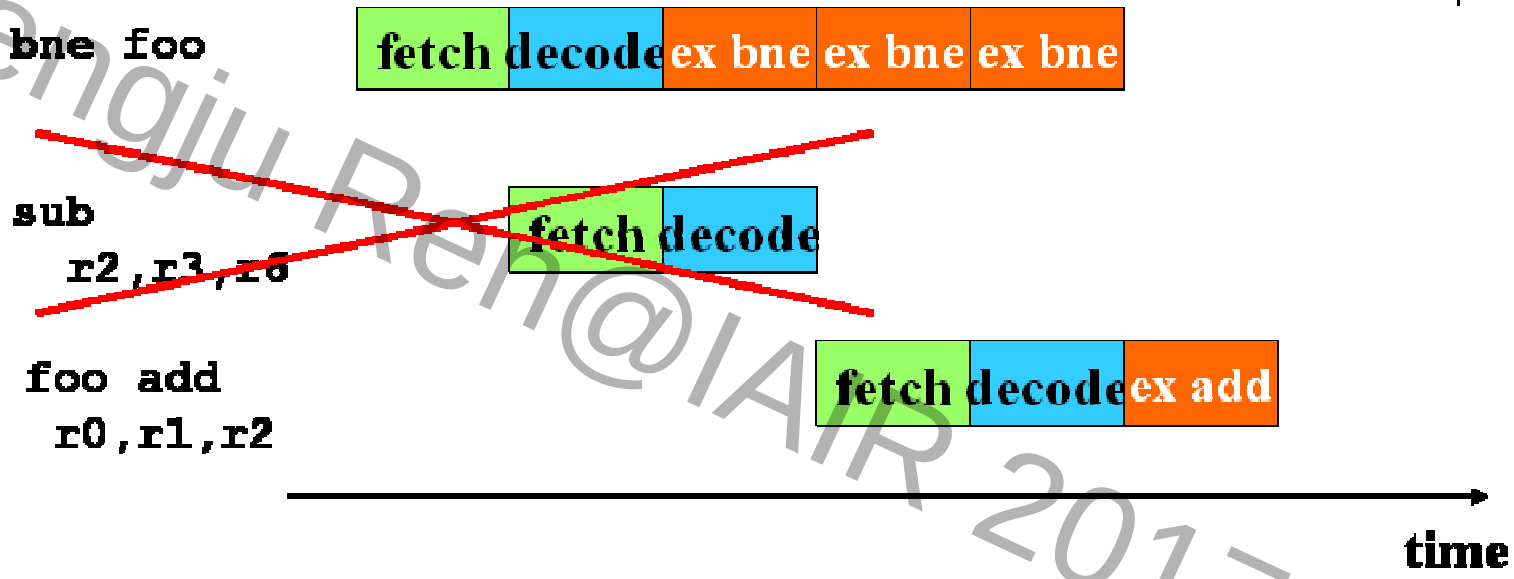
ARM的流水线数据阻滞



- 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）**。

Pengju Ren@IAIR 2017-2018

ARM的流水线控制阻滞



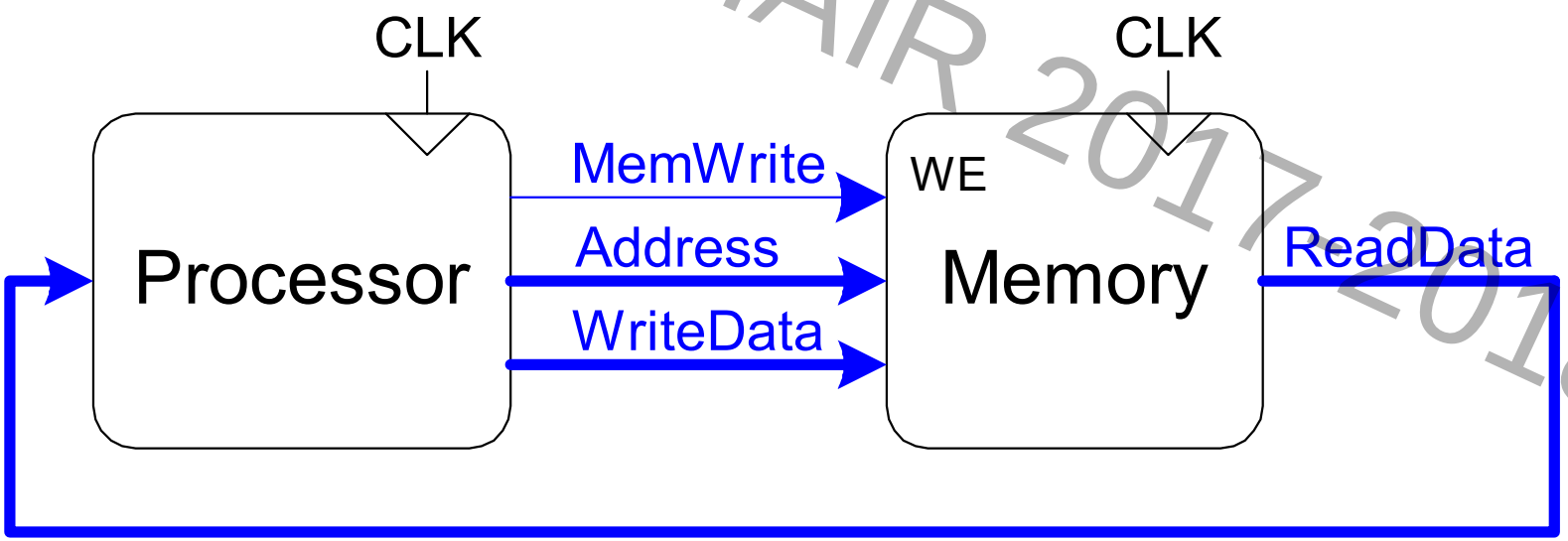
- 分支指令将导致流水线产生**控制阻滞**，也被称为**分支损失（Branch Hazard）**。
- 是否执行分支指令**BNE**要在第三个时钟周期才能确定，因此，浪费两个时钟周期。

Pengju Ren@IAIR 2017-2018

2 高速缓存

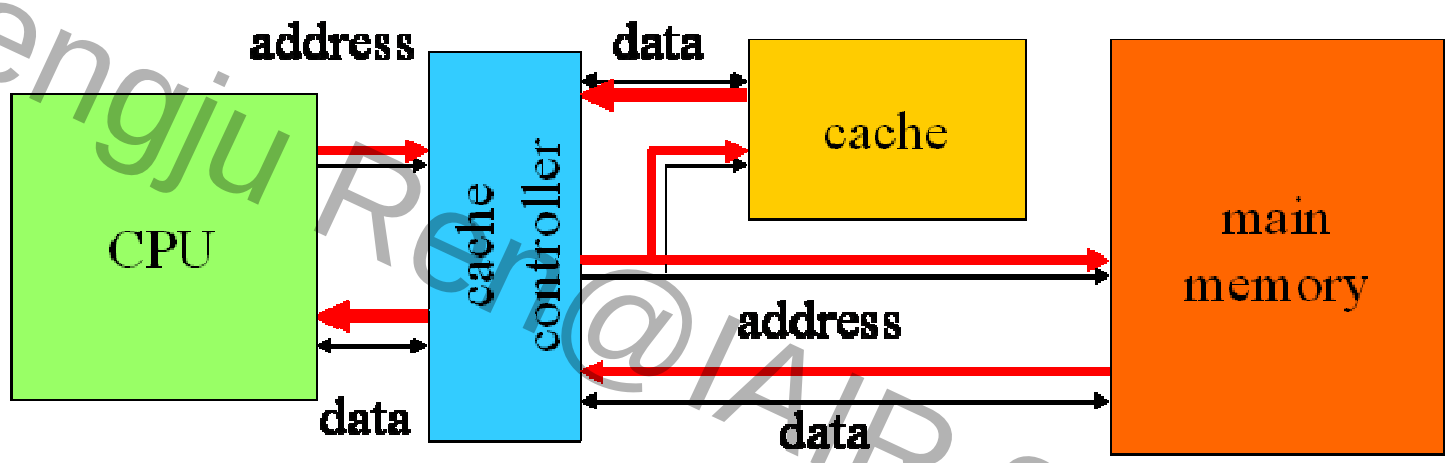
- Computer performance depends on:
  - Processor performance
  - Memory system performance

Memory Interface



Pengju Ren@IAIR 2017-2018

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。

Pengju Ren@IWR 2017-2018

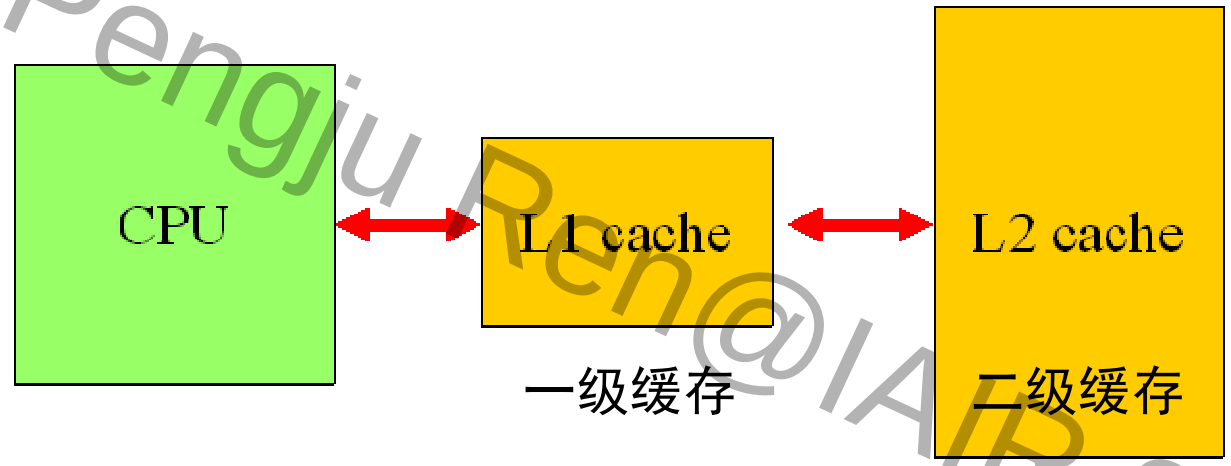
未命中的类型划分

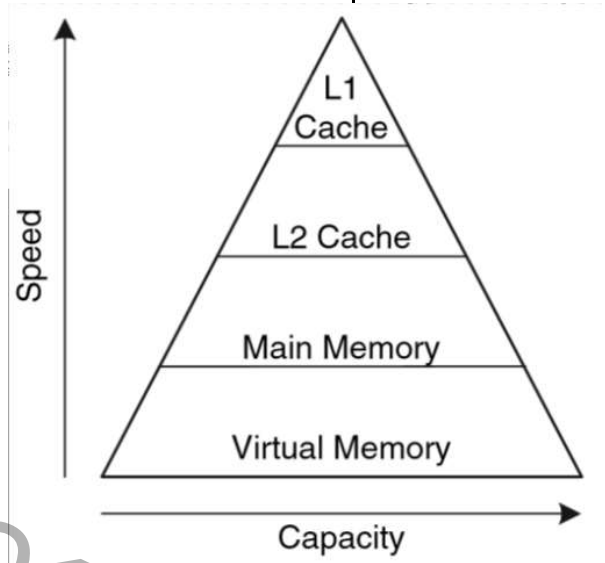
- 强制性未命中（compulsory miss），也称为冷未命中（cold miss），发生在单元第一次被访问时。
- 容量未命中（capacity miss）的发生是由于工作集过大。
- 冲突未命中（conflict miss）的发生是由于两个地址映射到高速缓存的同一个单元。
- 平均内存访问时间计算公式：

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

Pengju Ren@IAIR 2017-2018

多级高速缓存





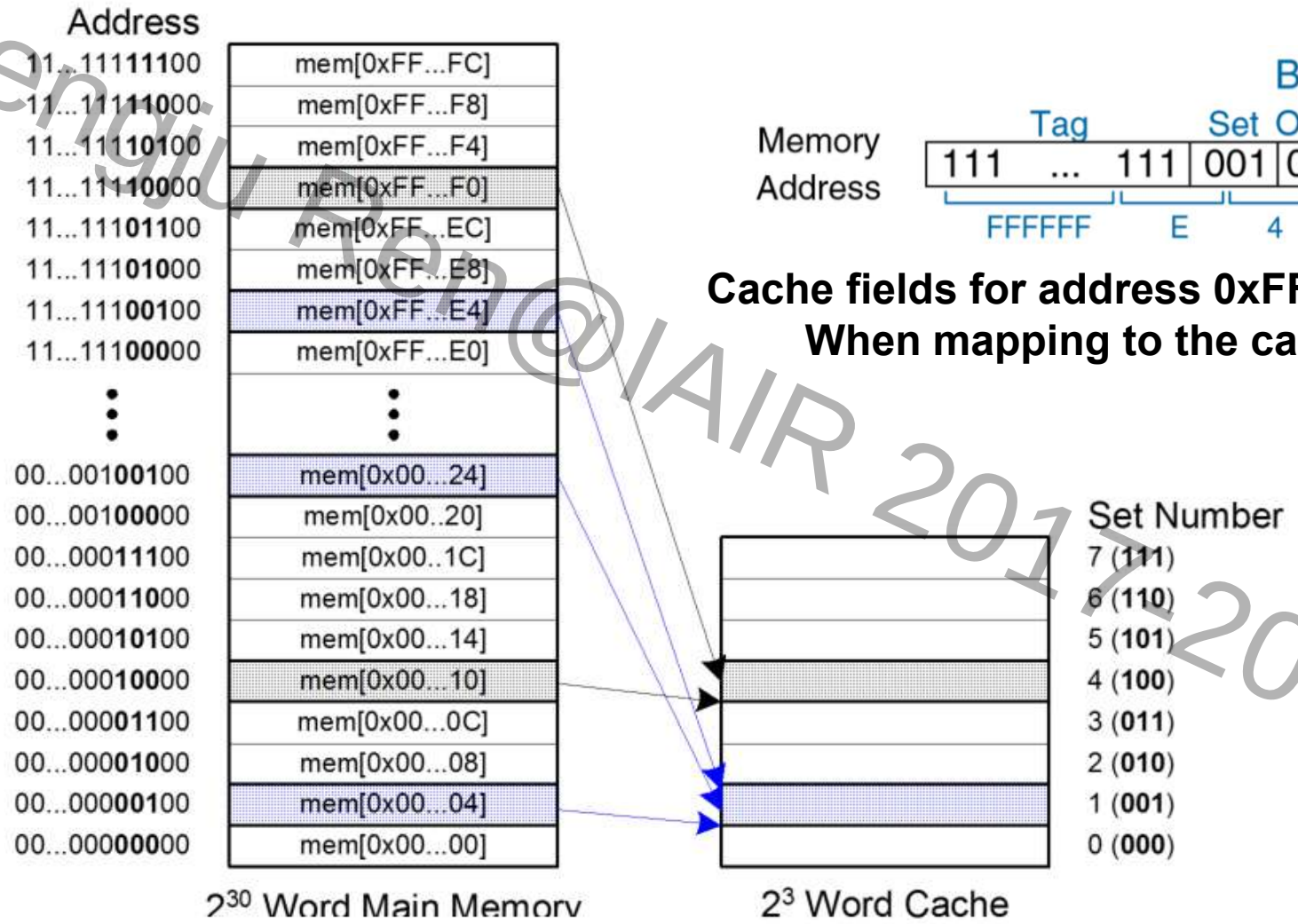
- 平均内存访问时间计算公式：

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

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。

Pengju Ren@IAIR 2017-2018

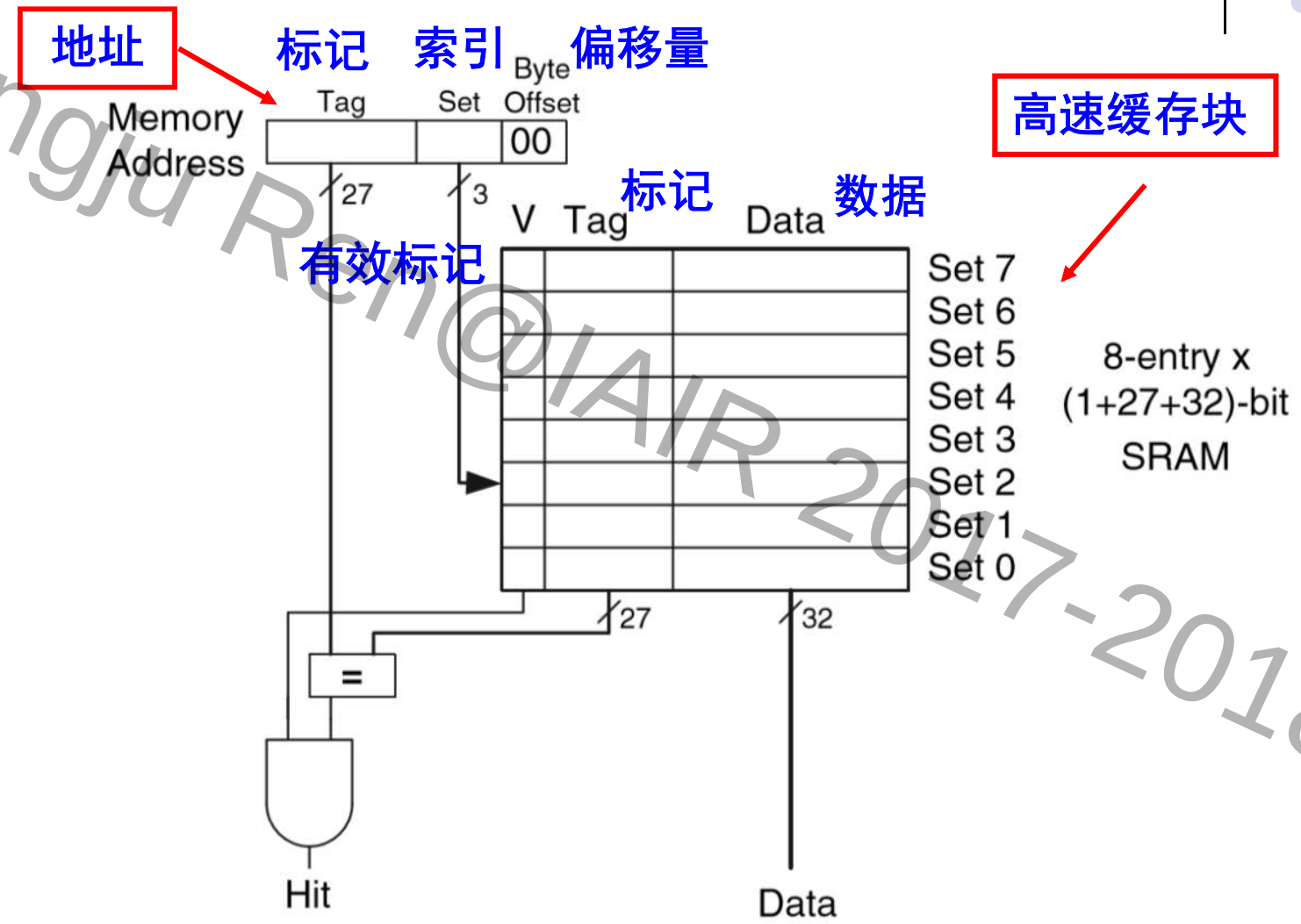
直接映射高速缓存图示



Cache fields for address 0xFFFFFFE4
When mapping to the cache

Pengju Ren@IAIR 2017-2018

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。

Pengju Ren@IAIR 2017-2018

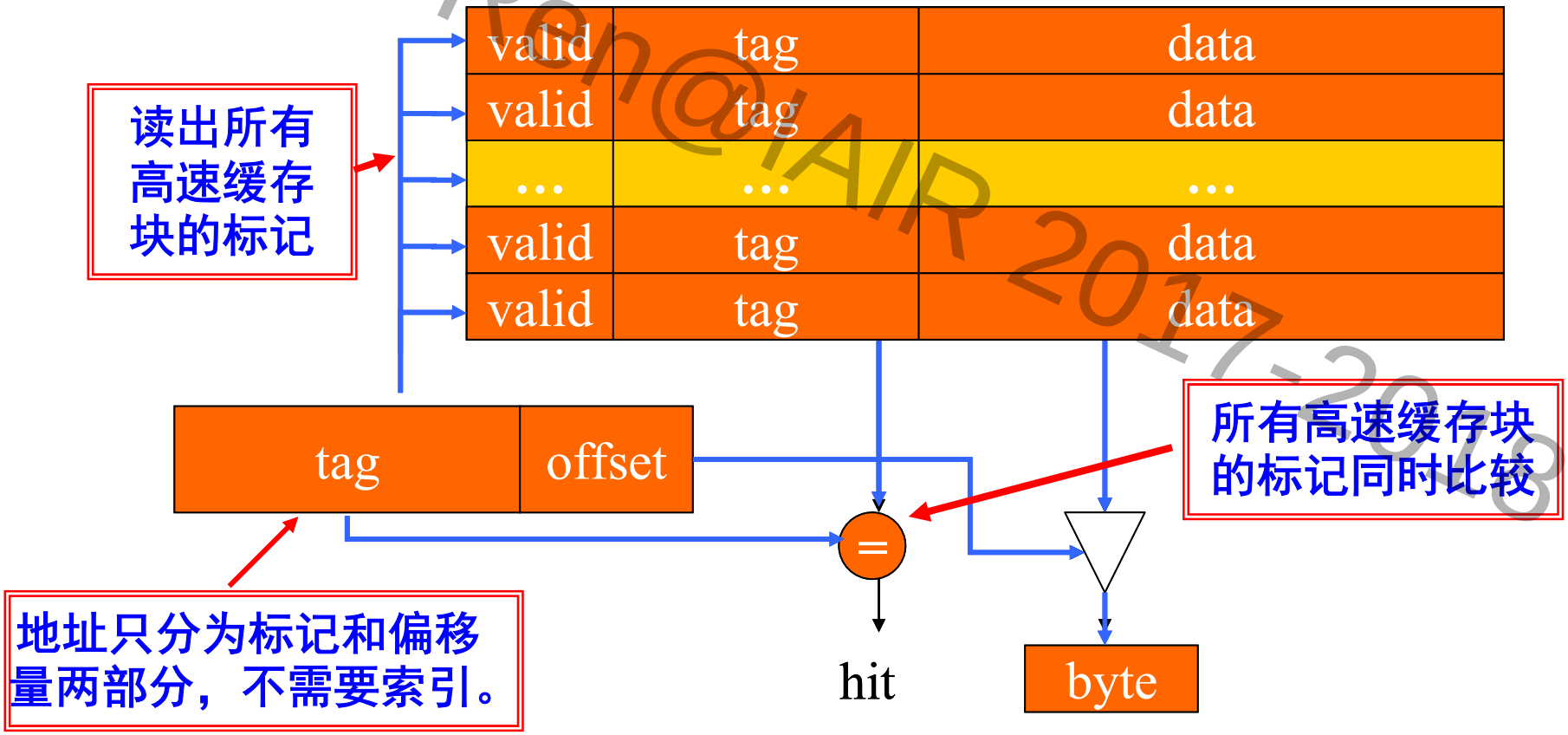
写操作

- 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。

Pengju Ren@IAIR 2017-2018

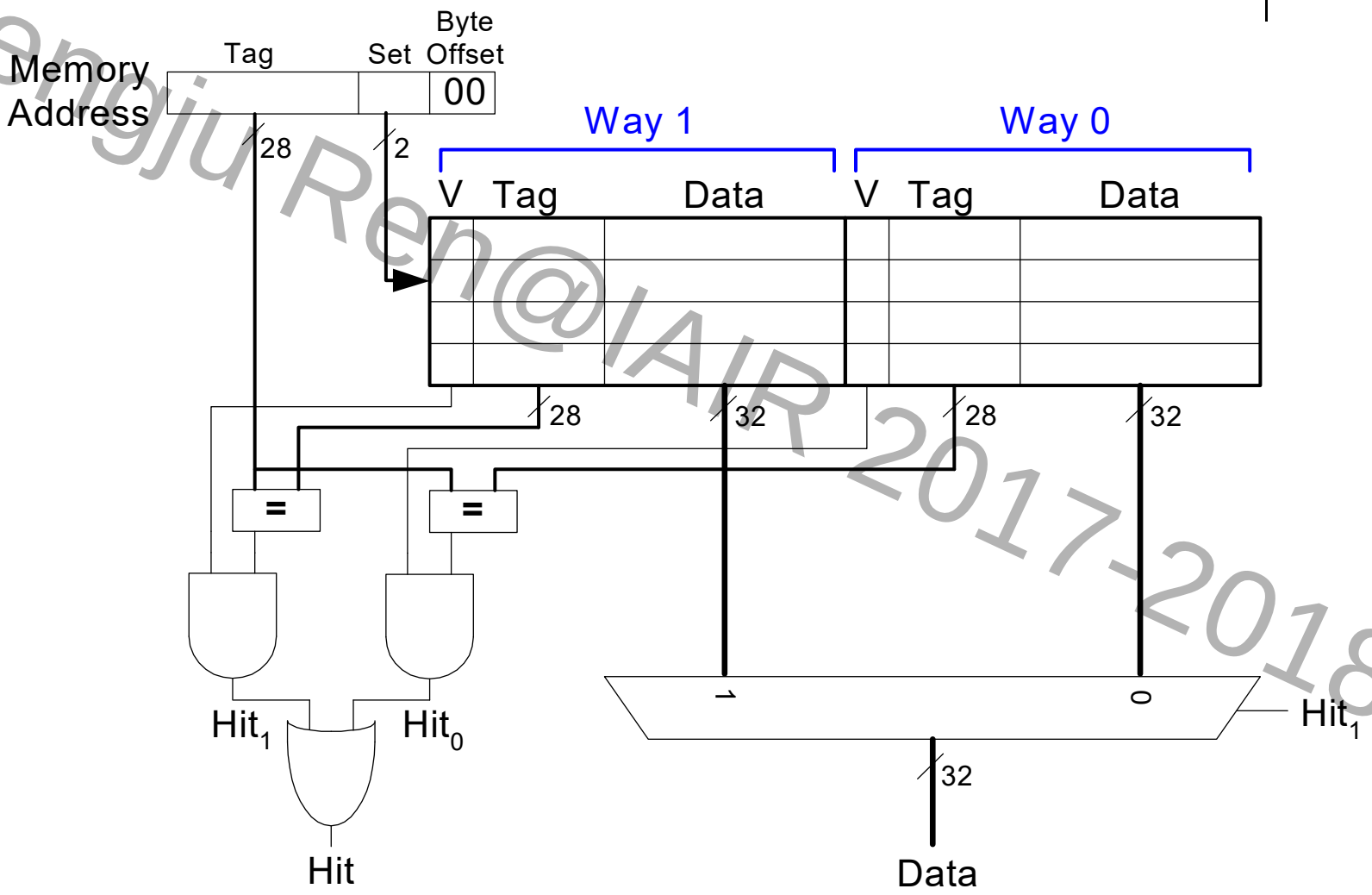
全相联高速缓存

- 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。



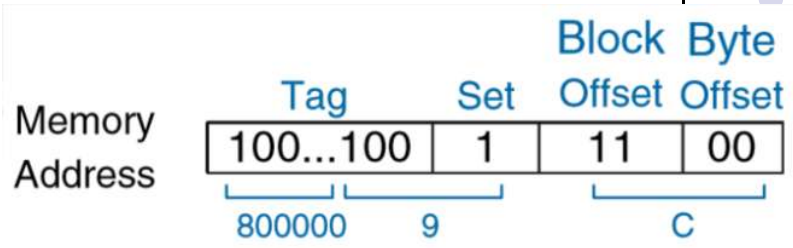
Pengju Ren@IAIR 2017-2018

组相联高速缓存

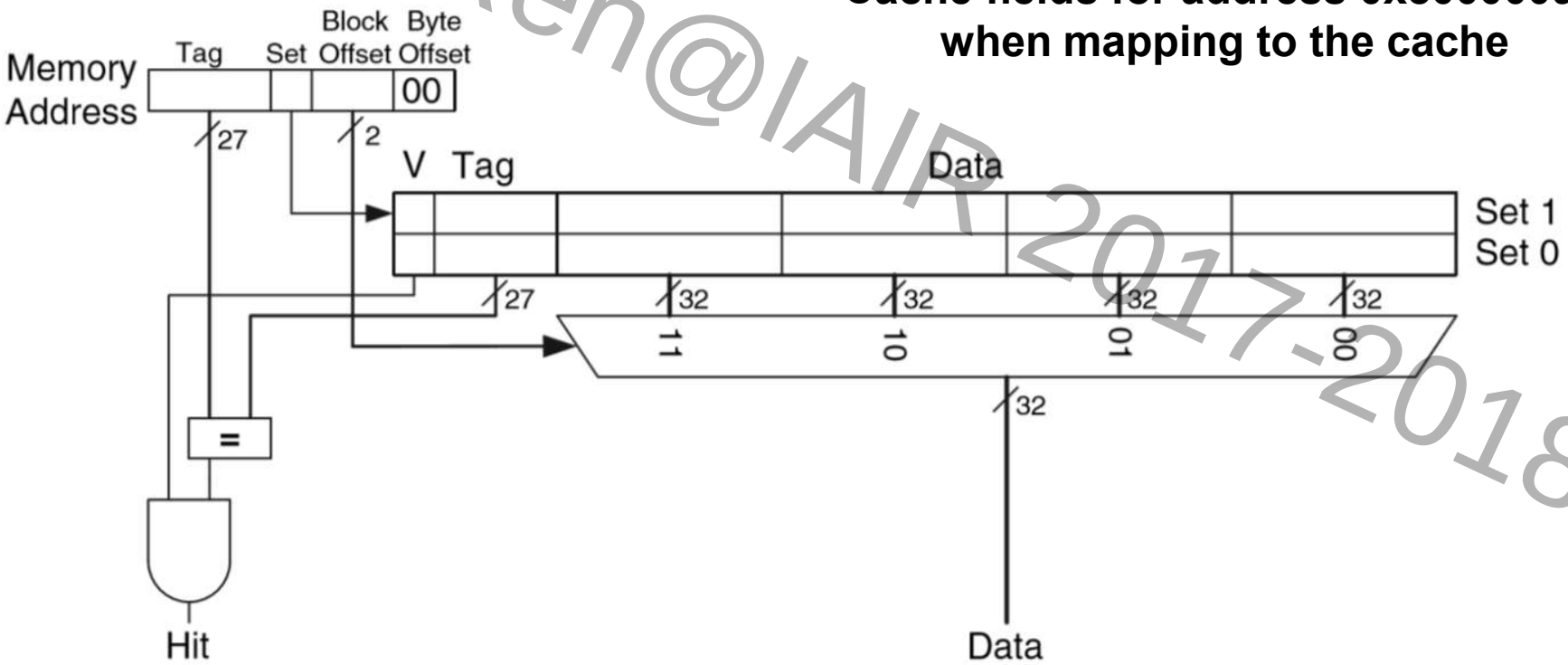


Pengju Ren@IAIR 2017-2018

Cache(空间相关性)



Cache fields for address 0x8000009C when mapping to the cache



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

Pengju Ren@IAIR 2017-2018

替换策略

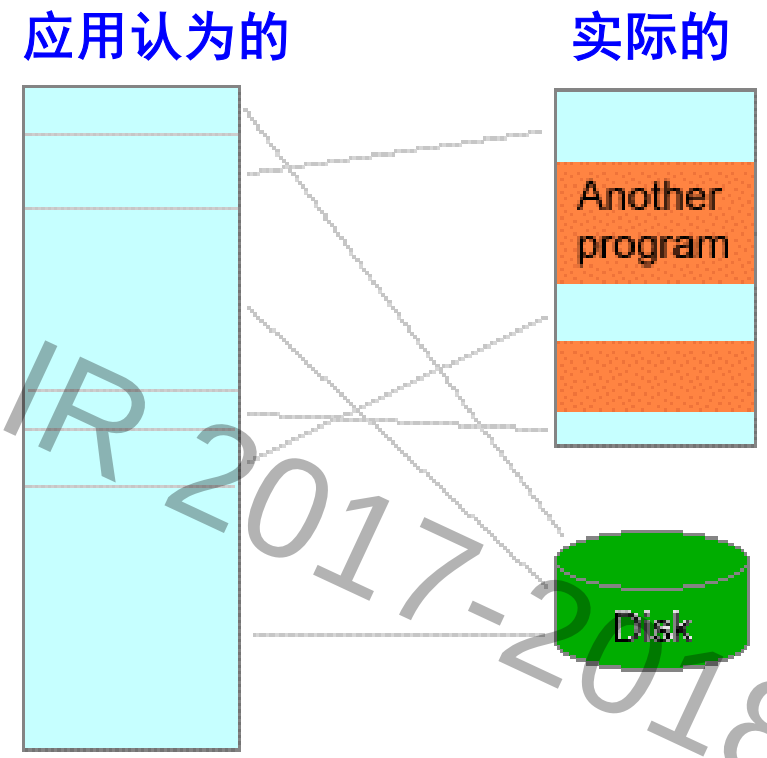
- 替换策略：在高速缓存“满”且需要载入新的数据块时，需要考虑何种旧数据块需要被移出高速缓存。
- 直接映射不存在替换策略问题，因为每个数据块对应唯一映射地址，其替换的数据块也是确定的。
- 组相联与全相联高速缓存需要考虑替换策略：
  - 随机替换（**Random**）：在组内或全相联高速缓存中随机选择一个数据块被替换。优点：硬件简单，耗费低。缺点：命中率相对较低。
  - 最近最少使用策略（**Least Recently Used, LRU**）：将组内或全相联高速缓存中最近最少使用的数据块。

    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**

Peng Ju Rh@ZIR 2017-2018

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

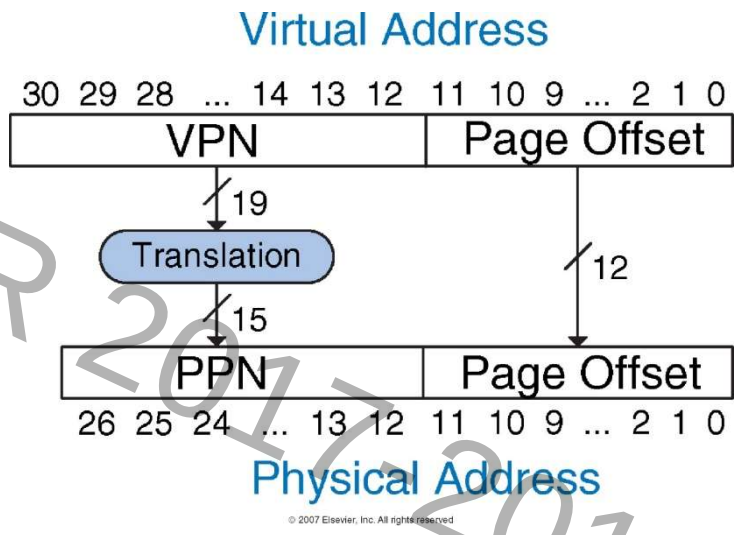


虚拟寻址（**virtual addressing**）:将逻辑地址转换成物理地址的寻址模式。

Pengju Ren@IAIR 2017-2018

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

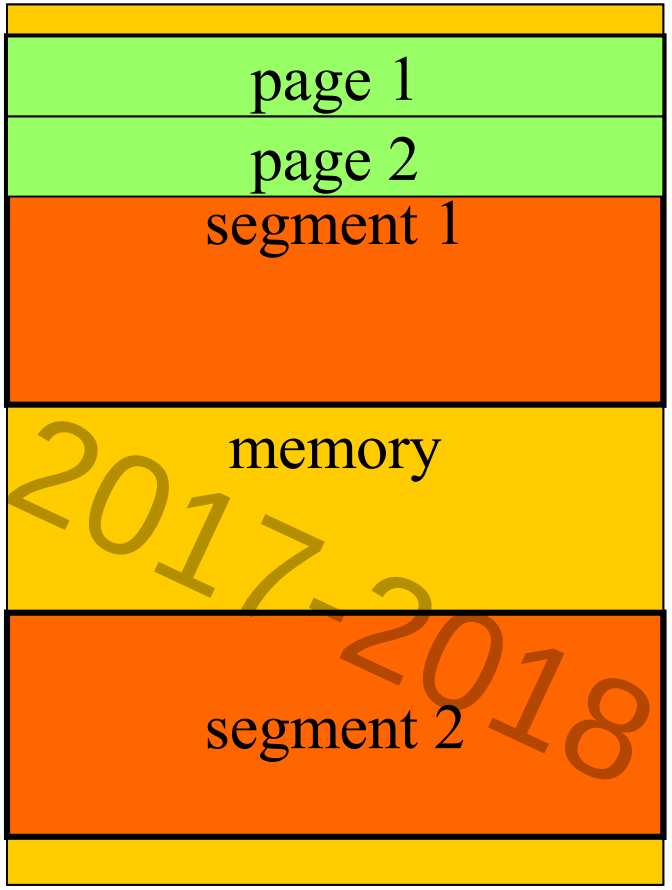


Physical memory acts as cache for virtual memory

Pengju Ren@IAIR 2017-2018

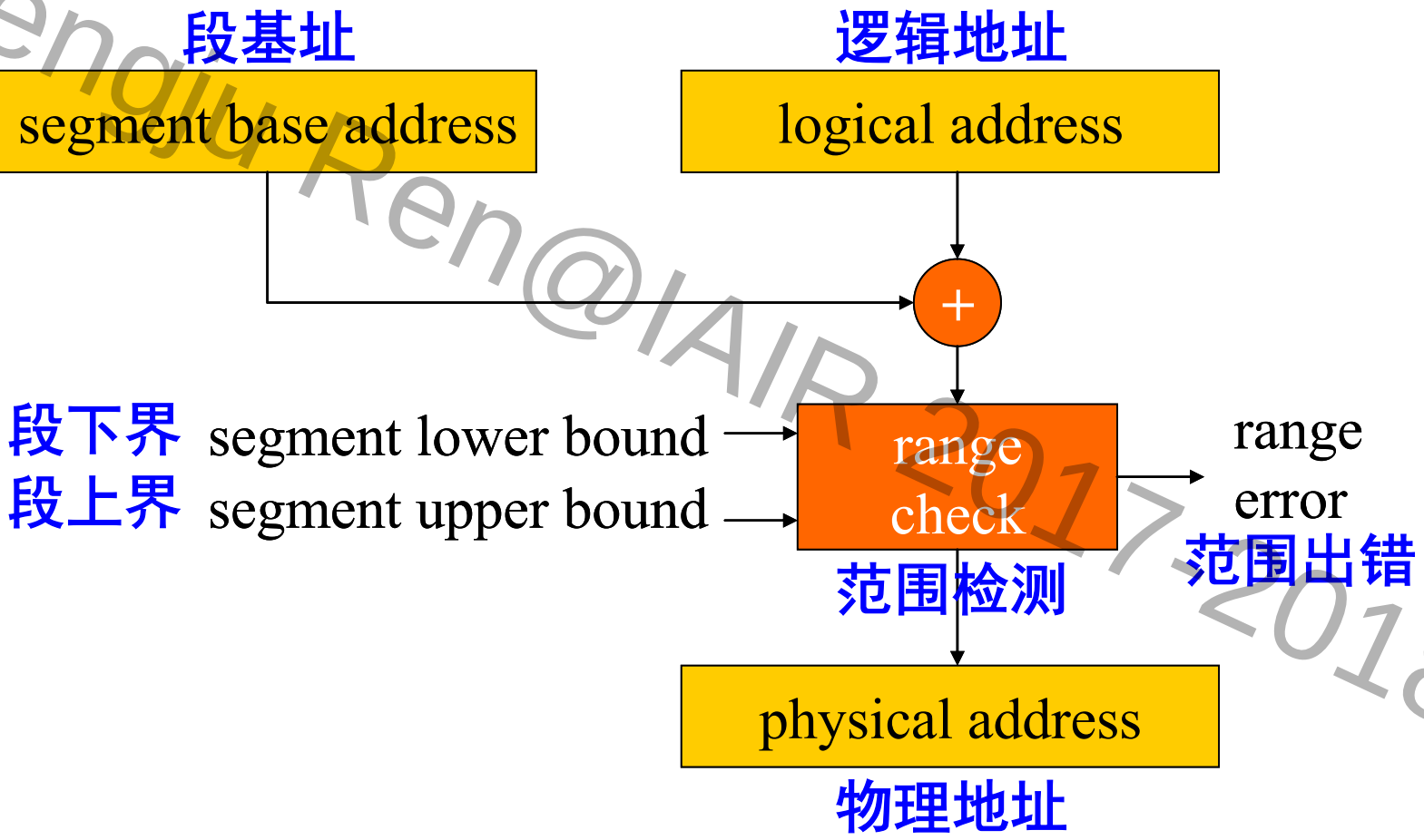
地址转换方法

- 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。



Penju Ren@IAIR 2017-2018

段地址转换

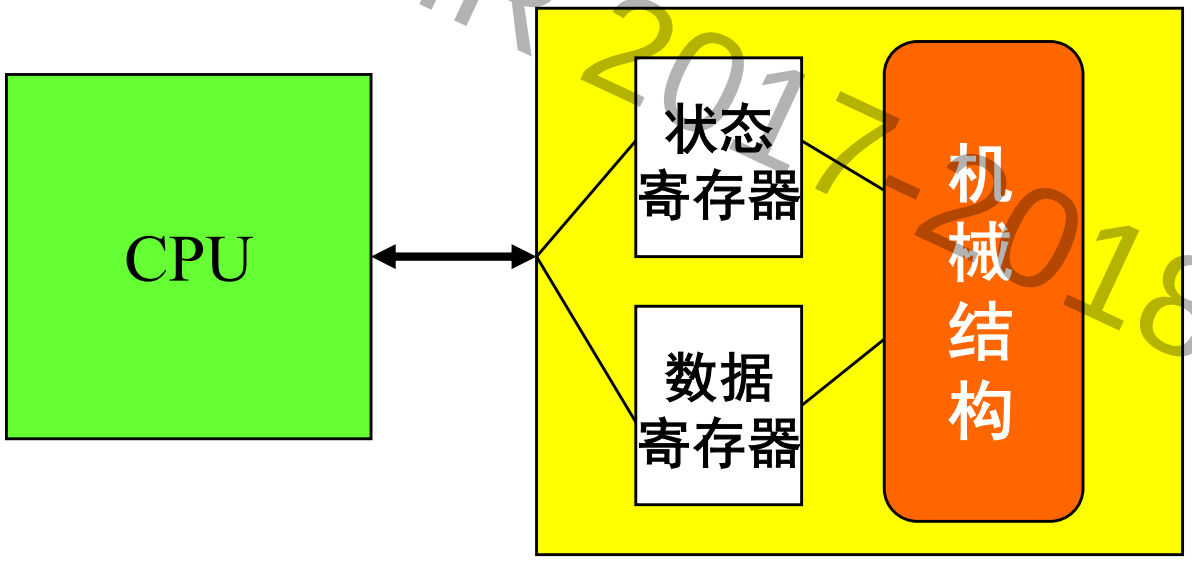


Pengju Ren@IAIR 2017-2018

1 输入/输出编程

- **输入输出设备（I/O Devices）：**通常有一些模拟元件或非电子元件（比如磁盘驱动器）。
- 但与CPU的接口通常是数字接口---一组寄存器。
  - 数据寄存器：通过设备将这些值作为数据保存，如磁盘读写
  - 状态寄存器：提供设备运行的信息，如当前时间完成查询

寄存器可以是只读的，也可以是可读可写的。



Peiqu Peng@WIR 2017-2018

ARM内存映射I/O

- 定义I/O设备地址:

```
DEV1 EQU 0x1000
```

- 读写设备寄存器代码:

```
LDR r1,#DEV1 ; set up device addrs
LDR r0,[r1]      ; read DEV1
LDR r0,#8        ; set up value(8) to write
STR r0,[r1]      ; write value(8) to device
```

Pengju Ren@IAIR 2017-2018

高级语言直接写I/O设备

- 在C语言中定义和使用一个变量时，编译器隐藏了变量地址。可以使用指针来操纵I/O设备的地址。

```
int peek(char *location) { return *location; }
#define DEV1 0x1000
……
dev_status = peek(DEV1);  /*读设备寄存器*/

void poke(char *location, char newval) {
   (*location) = newval; }
poke(DEV1, 8);                /*写设备寄存器*/
```

Peiqun Ren@IAIR 2017-2018

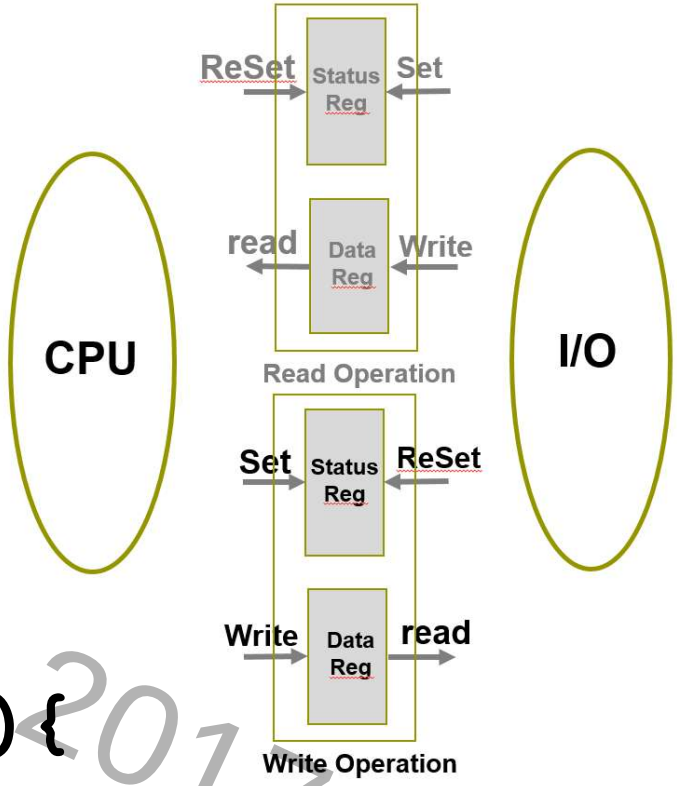
忙等I/O

- 忙等I/O是程序中访问设备的最基本方法。
- 设备一般比CPU慢，需要多周期来完成一项操作。
- 通过读I/O设备状态寄存器来询问I/O设备是否空闲通常被称为轮询（polling）。

Pengju Ren@IAIR 2017-2018

忙等I/O编程（1）

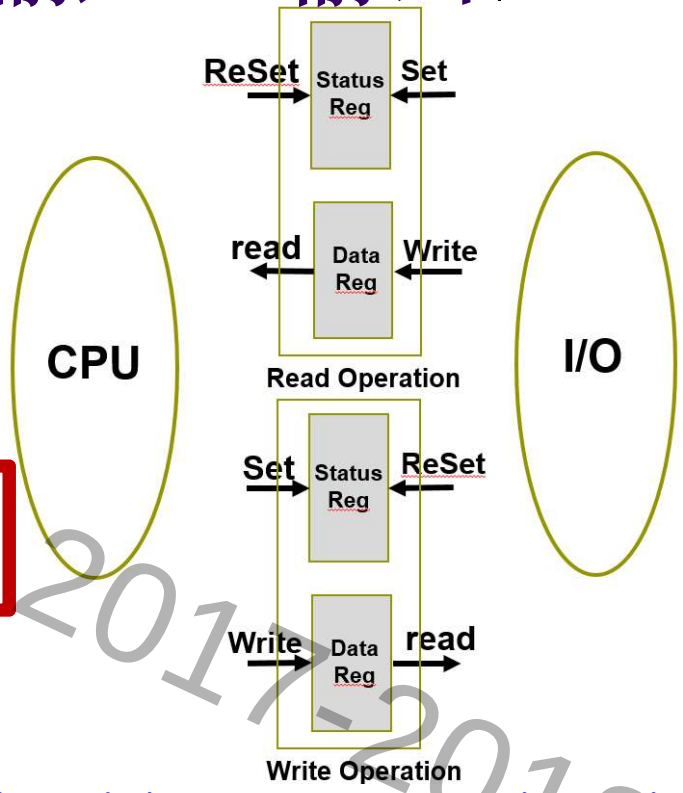
```
#define  OUT_CHAR     0x1000
#define  OUT_STATUS   0x1001
char  *mystring = "Hello world."
char  *current_char;
      current_char = mystring;
      while (*current_char != '\0') {
            poke(OUT_CHAR,*current_char);
            poke(OUT_STATUS,1);
            while (peek(OUT_STATUS) != 0);
            current_char++;
            }
```



Pengju Ren@IAIR 2017-2018

忙等I/O编程(2)-不断将输入->输出

```
#define  IN_DATA      0x1000
#define  IN_STATUS    0x1001
#define  OUT_DATA     0x1100
#define  OUT_STATUS 0x1101
while (TRUE) {
    /* read */
    while (peek(IN_STATUS) == 0);
    achar = (char)peek(IN_DATA);
    poke(IN_STATUS,0);
    /* write */
    poke(OUT_DATA,achar);
    poke(OUT_STATUS,1);
    while (peek(OUT_STATUS) != 0);
    }
```




```
while(peek(IN_STATUS)==1)
{ achar = (char)peek(IN_DATA)
```

Pengju Ren@IWR 2017-2018

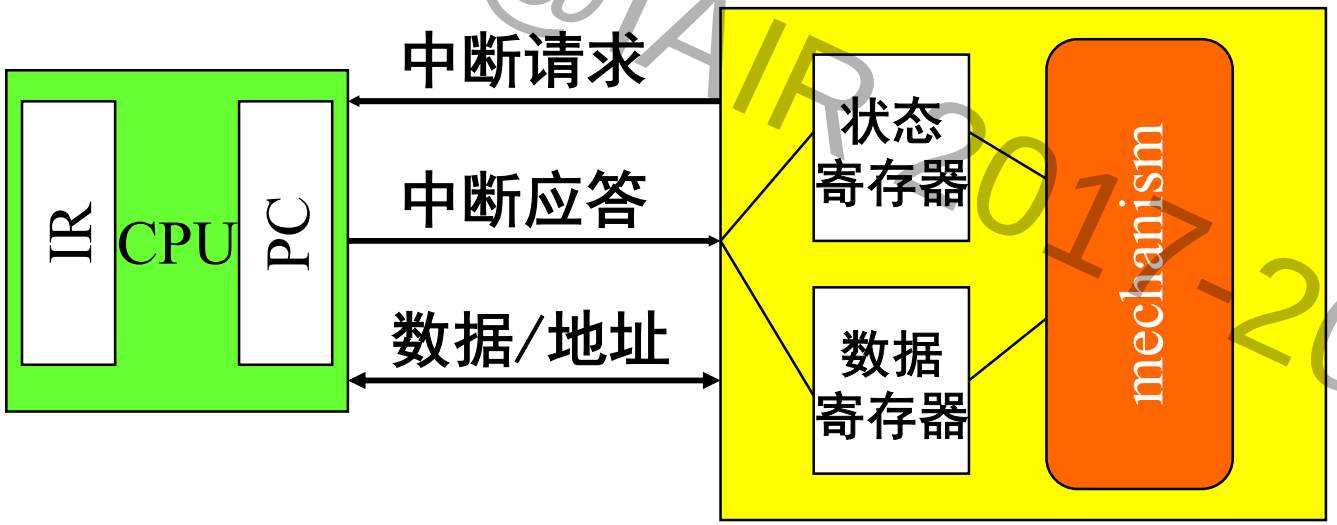
中断I/O

- 忙等I/O的效率非常低：
  - I/O事务未完成时，CPU除测试设备状态什么都不能做。
  - 无法将其他操作与I/O事务并行执行。
- 中断机制能允许设备发送信号到CPU，强制执行一段特定代码。
  - 中断发生时，PC指向I/O设备中一个管理设备的中断处理子程序（中断服务程序或设备驱动程序）。
  - 中断机制保留了系统被中断时的PC值，以便CPU能够返回到被中断的程序。
- 中断机制使得CPU能够在不同的上下文中间切换。

Peng Rong@HIT 2017-2018

中断I/O接口

- 当需要CPU服务时，I/O设备就发送**中断请求**信号。
- 当CPU准备处理I/O设备的请求时，CPU便发送**中断应答**信号。



Pengju Ren@IR 2017-2018

中断处理范例(1)不断将输入->输出

输入输出处理程序

```
void input_handler() {
    achar = peek(IN_DATA);
    gotchar = TRUE;
    poke(IN_STATUS,0);
}

void output_handler() {
}
```

主程序

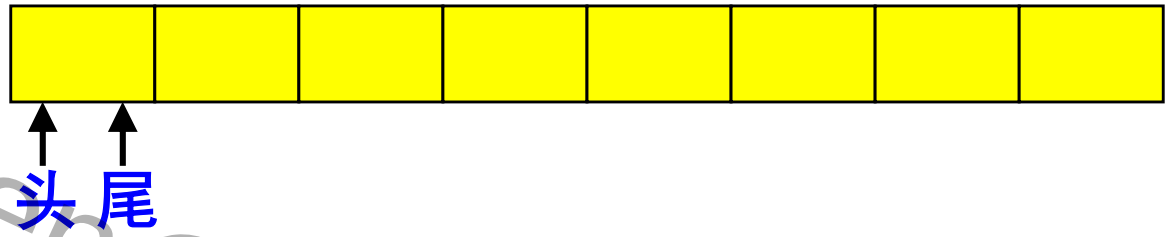
```
main() {
    while (TRUE) {
        if (gotchar) {
        poke(OUT_DATA,achar);
        poke(OUT_STATUS,1);
        While(
         peek(OUT_STATUS)!=0);
        gotchar = FALSE;
        }//end if
    }//end while
}
```

中断的使用使主程序稍微变简单，但是仍然没有让前台程序做有用的工作。

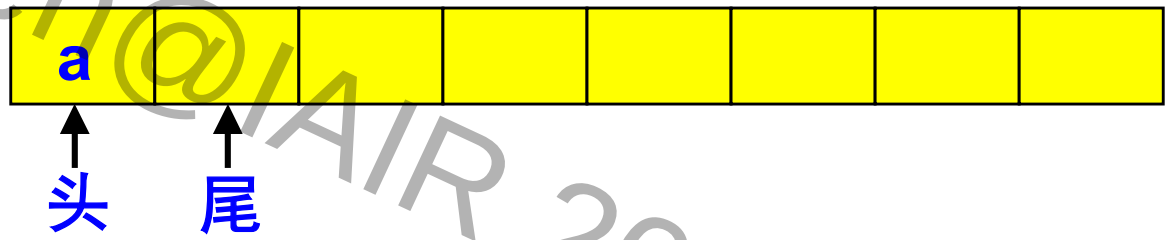
Pengju Ren@IAIR 2017-2018

中断处理范例（2）- 1

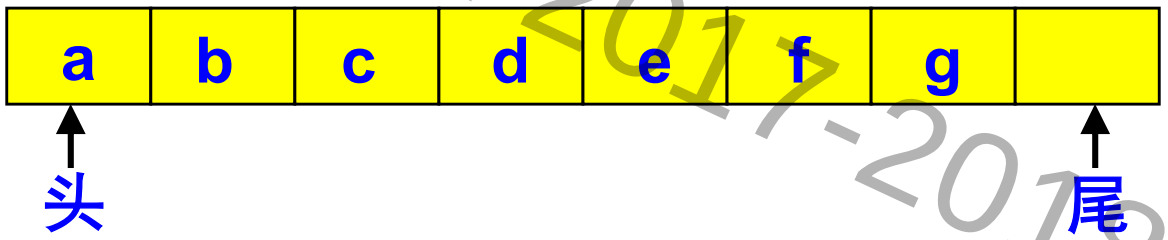
队列为空



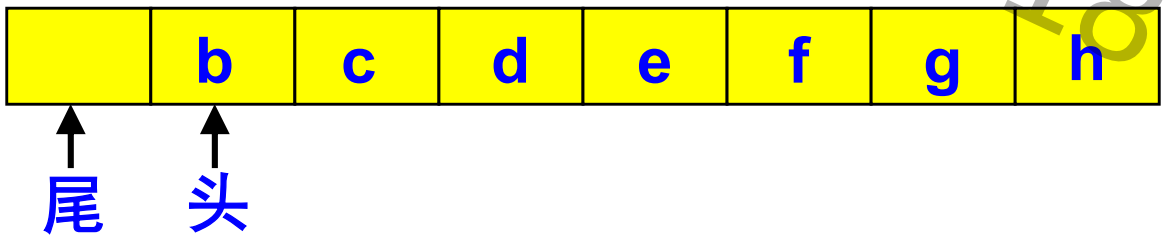
读入第一个字符



缓冲区满，最后一个字符位不使用



循环缓冲区



Pengju Ren@IAIR 2017-2018

中断处理范例（2）- 2

```
#define BUF_SIZE 8
char io_buf[BUF_SIZE];
int   buf_head = 0, buf_tail = 0;
int   error = 0;
bool empty_buffer() {retun buf_head == buf_tail;}
bool full_buffer() {
        return (buf_tail+1)%BUF_SIZE == buf_head;}
int   nchars(){
        if(buf_tail>=buf_head) return buf_tail-buf_head;
        else return BUF_SIZE + buf_tail – buf_head;
}
```

Pengju Ren@IAIR 2017-2018

中断处理范例（2）- 3

```
void add_char(char achar) {
   io_buf[buf_tail++] = achar;
   if(buf_tail == BUF_SIZE)
        buf_tail = 0;
}
char remove_char() {
   char achar;
   achar = io_buf[buf_head++];
   if(buf_head == BUF_SIZE)
        buf_head = 0;
}
```

Pengju Ren@IAIR 2017-2018

中断处理范例（2）- 4

```
#define IN_DATA    0x1000
#define IN_STATUS 0x1001
void input_handler() {
   char achar;
   if (full_buffer()) error = 1;     /*出错*/
   else {                  achar = peek(IN_DATA);
                           add_char(achar);
   }
   poke(IN_STATUS,0);
   if (nchars == 1 && peek(OUT_STATUS)==0)
       { poke(OUT_DATA,remove_char());
                           poke(OUT_STATUS,1); }
}
```

Pengju Ren@IAIR 2017-2018

中断处理范例（2）- 5

```
#define OUT_DATA     0x1100
#define OUT_STATUS   0x1101
void output_handler() {
        if(!empty_buffer())
            {poke(OUT_DATA, remove_char());
             poke(OUT_STATUS, 1);}
        }
```

Pengju Ren@IAIR 2017-2018

CPU的功耗

- 能量 vs. 功率
  - 功率是指单元时间内的能量消耗，功耗决定产生的热量
  - 能量消耗在很大程度上决定着电池的寿命。
  - 一般**功耗**作为**能量消耗**和**功率消耗**的简称，只有必要时才区分。
- **CMOS VLSI**功耗的计算公式：

$$P_{total} = P_{dynamic} + P_{static} \qquad P_{dynamic} = \lambda C V_{DD}^2 f$$

λ电平变化比率，f频率，V_{DD}供电电压，C负载电容

- 动态功耗：电压降和切换（频率和有效切换）
- 静态功耗：漏电，I_{DD} 称为 $P_{static} = I_{DD} V_{DD}$

Pengju Ren@IAIR 2017-2018

CPU的功耗

- **Estimate the power consumption of a wireless handheld computer**
  - V_{DD} = 1.2 V
  - C = 20 nF
  - f = 1 GHz
  - I_{DD} = 20 mA

Pengju Ren@IAIR 2017-2018

CPU的功耗

- **Estimate the power consumption of a wireless handheld computer**
  - V_{DD} = 1.2 V
  - C = 20 nF
  - f = 1 GHz
  - I_{DD} = 20 mA

$$P = \tfrac{1}{2}CV_{DD}^2 f + I_{DD}V_{DD}$$

$$= \tfrac{1}{2}(20\text{ nF})(1.2\text{ V})^2(1\text{ GHz}) + (20\text{ mA})(1.2\text{ V})$$

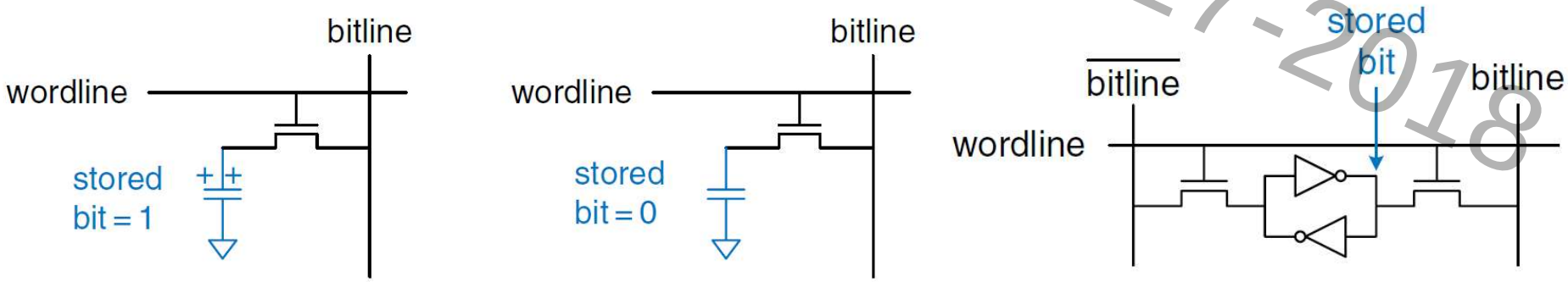
$$\mathbf{= (14.4 + 0.024)\text{ W} \approx 14.4\text{ W}}$$

Pengju Ren@IAIR 2017-2018

随机存储器（RAM）

- 随机存储器（**RAM**）可以被读和写，地址可以按任意顺序读取。
  - 静态随机存储器（**SRAM**）：小容量快速存储器。
  - 动态随机存储器（**DRAM**）：密度大、需要周期刷新。
- 同步动态随机存储器（**Synchronous DRAM**），使用时钟来提升**DRAM**的性能（最常见）。

Peng Ren@IAIR 2017-2018



DRAM cell　　SRAM cell

带宽（bandwidth）

- 带宽指数据传输的速率，涉及到嵌入式计算系统中的多个组件：
  - 总线
  - 存储器
  - **CPU**获取
- 系统的不同部分采用不同的时钟频率
- 系统不同部分的接口数据宽度也不一同
- 必须经过时钟周期和宽度换算，为每一部分的性能评估提供正确的参考。

Pengju Ren@IAIR 2017-2018

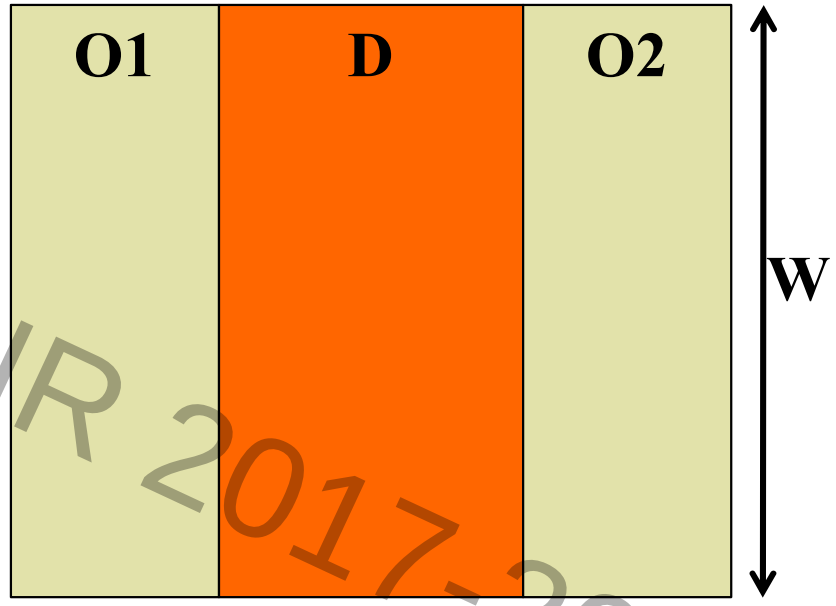
带宽与数据传输

- 以传输视频数据为例：
- 视频帧数据 **Video frame:**
  - **320 x 240 x 3 = 230,400 bytes.**
  - **Transfer in 1/30 sec == 0.033 sec per frame.**
- 假设传输速度 **1 byte/usec**（等价于**1MHz**）
  - **0.23 sec per frame.**
  - **Too slow.**
- 两个方面可以提高带宽:
  - 提高总线的宽度：每次传**4**字节，传输时间降至**0.058s**。
  - 提高总线的时钟频率：**2MHz**，传输时间降至**0.029S**。

Pengju Ren@IAIR 2017-2018

总线带宽

- T: # bus cycles.
- P: time/bus cycle.
- Total time for transfer:
  - t = TP.
- D: data payload length.
- O1 + O2 = overhead O.

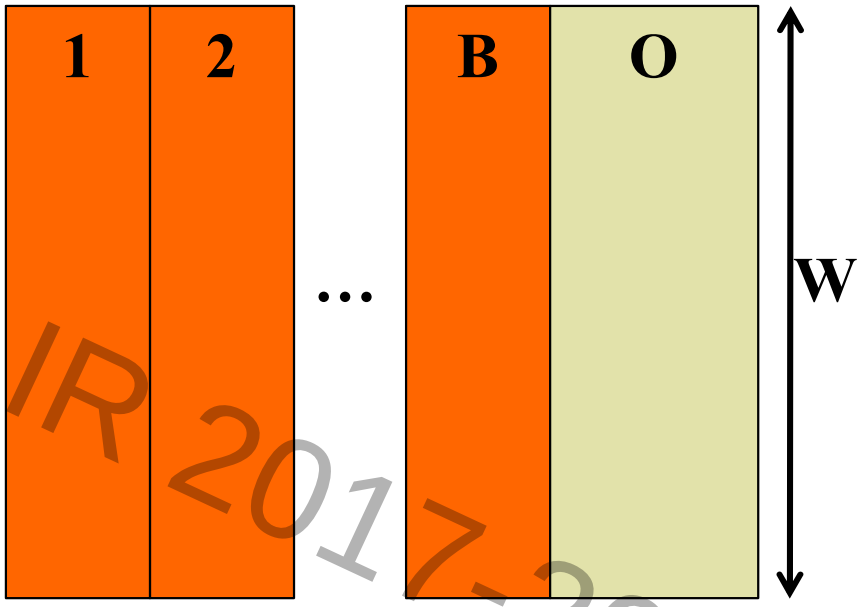


$$T_{basic}(N) = (D+O)N/W$$

Pengju Ren@IAIR 2017-2018

突发总线带宽

- T: # bus cycles.
- P: time/bus cycle.
- Total time for transfer:
  - t = TP.
- D: data payload length.
- O1 + O2 = overhead O.

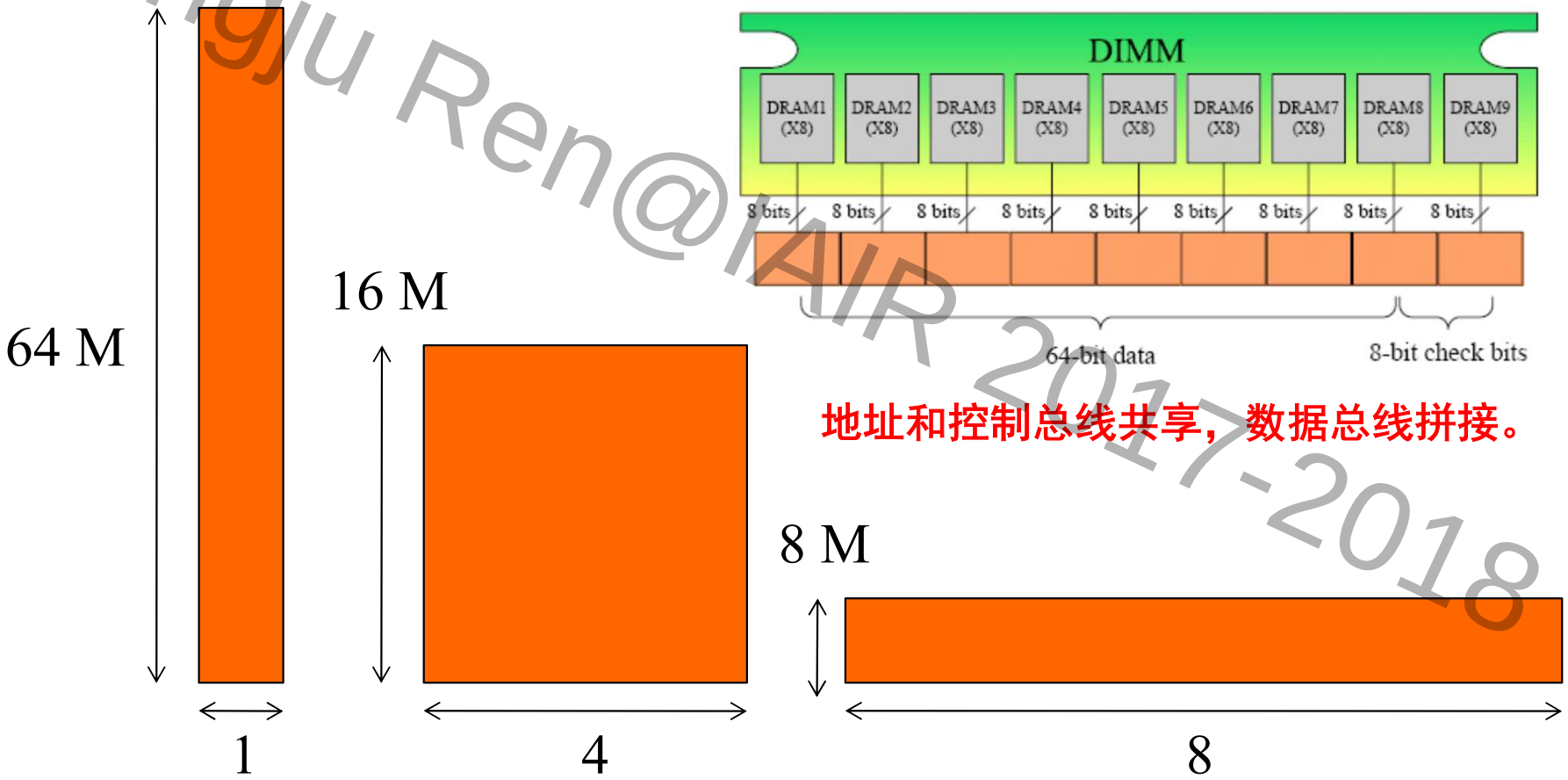


$$T_{burst}(N) = (BD+O)*(N/BW)$$

Pengju Ren@IAIR 2017-2018

存储器长宽比（aspect ratio）

存储器宽度也是数据带宽的一种形式。存储器没有非常宽的长宽比，可以组合构建，选择SIMM或DIMM。



地址和控制总线共享，数据总线拼接。

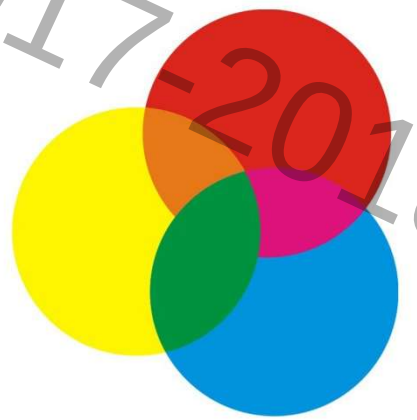
若用**16M×4**位存储器构造**32M×16**位存储器，需要几个？

Renju Ren@IAIR 2017-2018

带宽分析

- 当数据类型无法自然对应到存储器宽度上时，情形会变得复杂。
- 每个标准像素是**3x8=24bit**（红、绿、蓝）。
- 一个**24**位宽的存储器允许我们每次访问一个完整像素。
- 一个**32**位宽的存储器？**Two choice**
  - 每次传输浪费**1**个字节
  - 不浪费，将像素拼接起来

$$\lceil (E \times w)/W \rceil$$



E为传输数据个数；**w**为数据位宽，**W**为存储器位宽

Peng Ren@HIR 2017-2018

带宽瓶颈

- 假设以30帧/秒的速度在总线传输320x240的视频，带宽需求是612000字节/秒。
- 带宽计算：
- 总线:
  - 总线时钟频率1MHz（周期10^{-6}秒），宽度2字节，D=1，O=3.
  - T_{basic} = (1+3)612,000/2 = 1,224,000 cycles = 1.224 sec.
- 存储器:
  - 提供突发模式B=4，宽度仅为4bit（W=0.5），D=1，O=4，时钟频率10MHz。
  - Tmem = (4*1+4)612,000/(4*0.5) = 2,448,000 cycles = 0.2448 sec.

Peng Ren@IMR 2017-2018

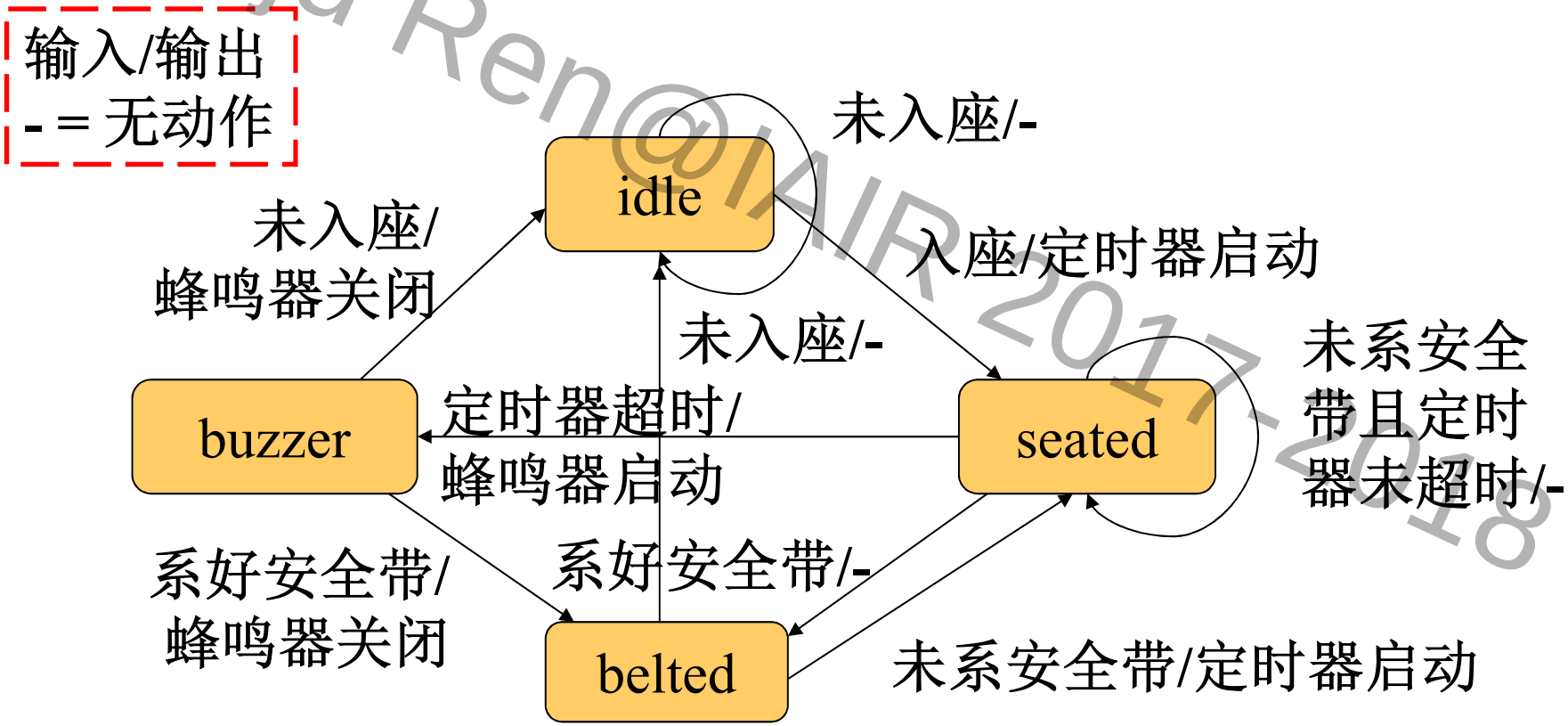
状态机（1）

- 反应系统（**reactive system**）：响应外部事件的系统。
  - 外部输入是间歇到达
  - 适合使用状态机描述
- 有限状态机是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。
  - **Moore**机：输出只由当前状态确定
  - **Mealy**机： 输出依赖于当前状态和输入

Peng Ren@IAIR 2017-2018

状态机（2）

- 例子：一个简单的座位安全带控制器



Pengju Ren@IAIR 2017-2018

状态机（3）

Inputs：seat, belt, timer
Outputs: buzzer

```
#define IDLE 0
#define SEATED 1
#define BELTED 2
#define BUZZER 3
switch (state) {
    case IDLE: if (seat) { state = SEATED; timer_on = TRUE; }
            break;
    case SEATED: if (belt) state = BELTED;
                    else if (timer) state = BUZZER;
            break;
    case BELTED: if (!seat) state = IDLE;
                    else if (!belt) state = SEATED;
            break;
    case  BUZZER: if (belt) state = BELTED;
                    else if (!seat) state = IDLE;
            break;
}
```

Renqiu Ren@IAIP 2017-2018

控制/数据流图

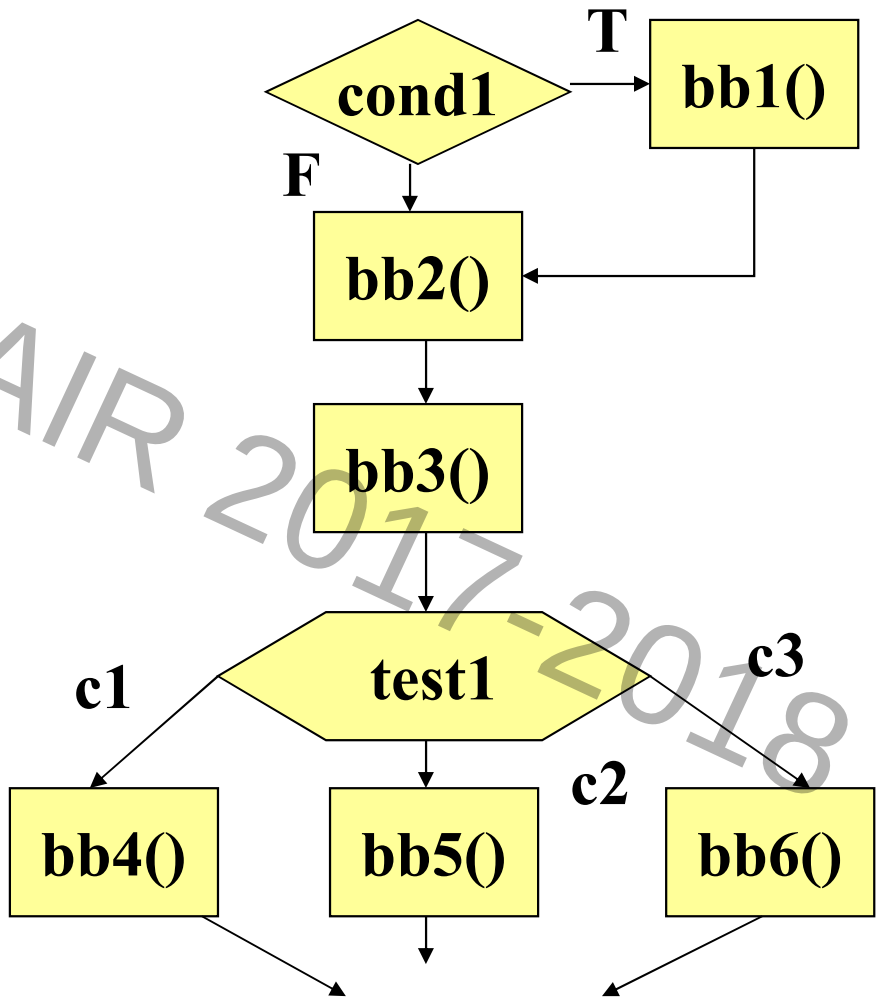
- **CDFG**用数据流图作为一个元素，添加描述控制的结构。

- 基本的**CDFG**有两种节点：
  - 判定节点**(Decision node)**
  - 数据流节点 **(data flow node)**：包括一个完整的表示基本语句块的数据流图

Pengju Ren@IAIR 2017-2018

CDFG示例

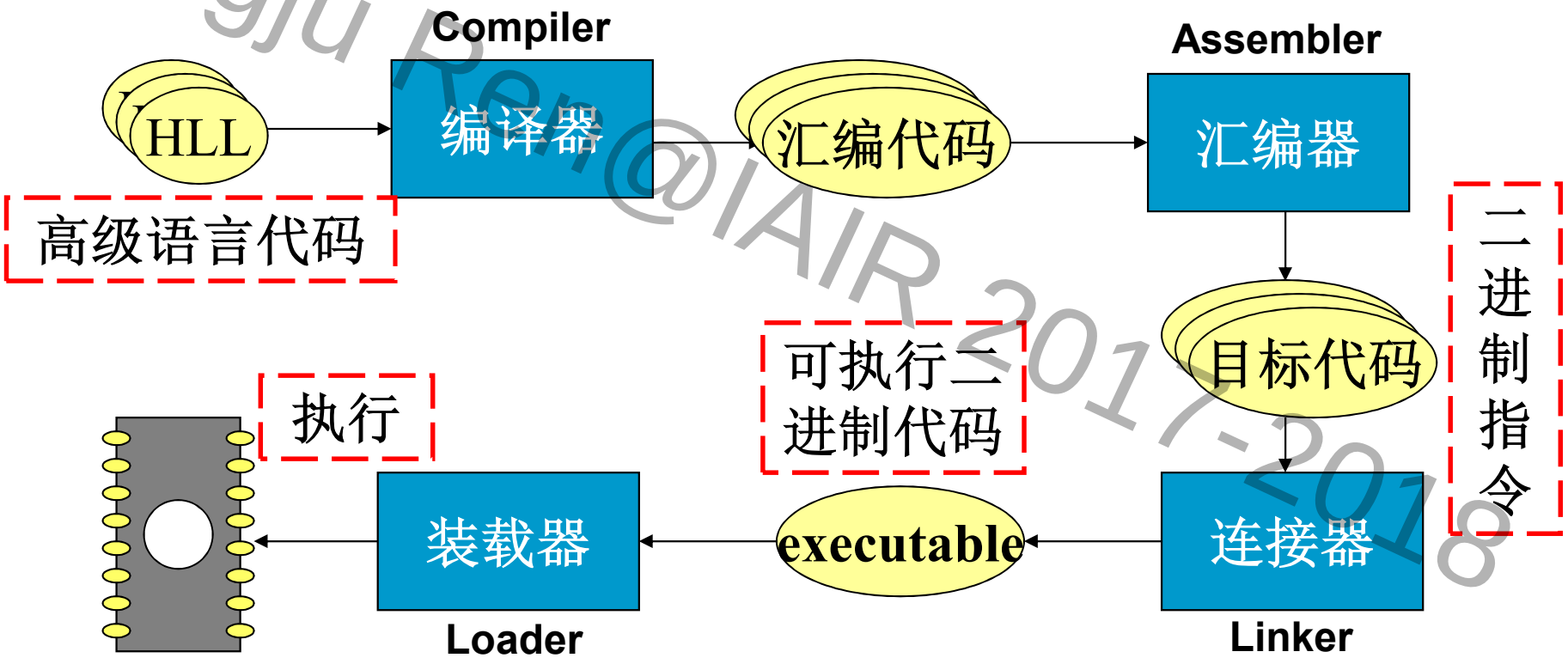
```
if (cond1) bb1();
 else bb2();
bb3();
switch (test1) {
    case c1: bb4(); break;
    case c2: bb5(); break;
    case c3: bb6(); break;
}
```



Pengju Ren@IAIR 2017-2018

3 汇编、连接和装载

- 汇编和连接是编译处理过程的最后几步:



从编译到装载的程序创建过程

Pengju Ren@IAIR 2017-2018

过程链接

- 过程定义包括:
  - 生成处理过程调用与返回的代码
  - 传递参数和返回值
- 过程链接机制允许参数和返回值传递，并保护和恢复修改过的寄存器。
- 帧栈的具体构造和数据结构取决于不同的**CPU (Machine dependent)**
- 参数和返回值主要通过栈（**stack**）来传递
  - 少量参数也可以通过寄存器来传递

Pengju Ren@IAIR 2017-2018

5 程序优化

- 表达式简化
- 无效代码的清除
- 过程内嵌
- 循环变换
- 寄存器分配
- 调度
- 指令选择

Pengju Ren@IAIR 2017-2018

表达式简化(Expression simplificaction)

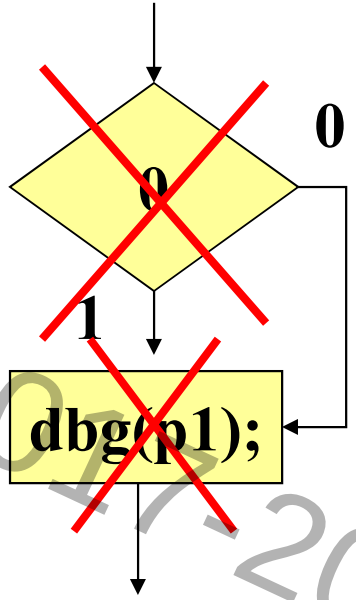
- 表达式简化是对独立于机器的变换很有效。
- 利用代数法则：
  - **a*b + a*c = a*(b+c)**　分配律使3操作简化成2操作
- 常量叠算（**constants folding**）：
  - **8+1 = 9**
  - **for (i=0; i<8+1; i++)**　　**for (i=0; i<NOPS+1; i++)**
- 强度化简:
  - **a*2 = a<<1**

Pengju Ren@IAIR 2017-2018

无效代码清除(Dead code elimination)

- 无效代码:

```
#define DEBUG 0
if (DEBUG) dbg(p1);
```



- 可以通过控制流分析、常量叠算来消除。

- 有些无效代码是编译器引入的。

Pengju Ren@IAIR 2017-2018

过程内嵌(procedure inlining)

- 过程内嵌属于独立于机器的变换，可以去除过程链接造成的耗费。

```
int foo(a,b,c) { return a + b - c;}
z = foo(w,x,y);
```

⇨

```
z = w + x + y;
```

- 是否使用过程内嵌需要评估，不一定有益：
  - 函数多份拷贝可能造成cache冲突，减慢指令的存取速度
  - 增加代码量，增加存储器开销

Peng Ren@IAIR 2017-2018

循环变换(Loop transformations)

- 循环虽然在源代码中描述的很紧凑，但通常占用大量的计算时间。

- 有很多优化循环技术，其目标：
  - 减少循环的耗费；
  - 增加流水线执行的机会；
  - 改善存储系统的性能。

Pengju Ren@IAIR 2017-2018

循环展开(Loop unrolling)

- 减少循环的开销，使能后续的优化，比如指令并行。

```
for (i=0; i<4; i++)
    a[i] = b[i] * c[i];
```

⇨

```
for (i=0; i<2; i++) {
    a[i*2] = b[i*2] * c[i*2];
    a[i*2+1] = b[i*2+1] * c[i*2+1];
}
```

```
a[0] = b[0] * c[0];
a[1] = b[1] * c[1];
a[2] = b[2] * c[2];
a[3] = b[3] * c[3];
```

Pengju Ren@IAIR 2017-2018

循环合并与分解(Loop fusion and distribution)

- 循环合并将两个或多个循环合并成一个：

```
for (i=0; i<N; i++) a[i] = b[i] * 5;
for (j=0; j<N; j++) w[j] = c[j] * d[j];
⇨ for (i=0; i<N; i++) {
    a[i] = b[i] * 5; w[i] = c[i] * d[i];
  }
```

- 循环分解将一个循环分解成多个。

Pengju Ren@IAIR 2017-2018

循环分块(Loop tiling)

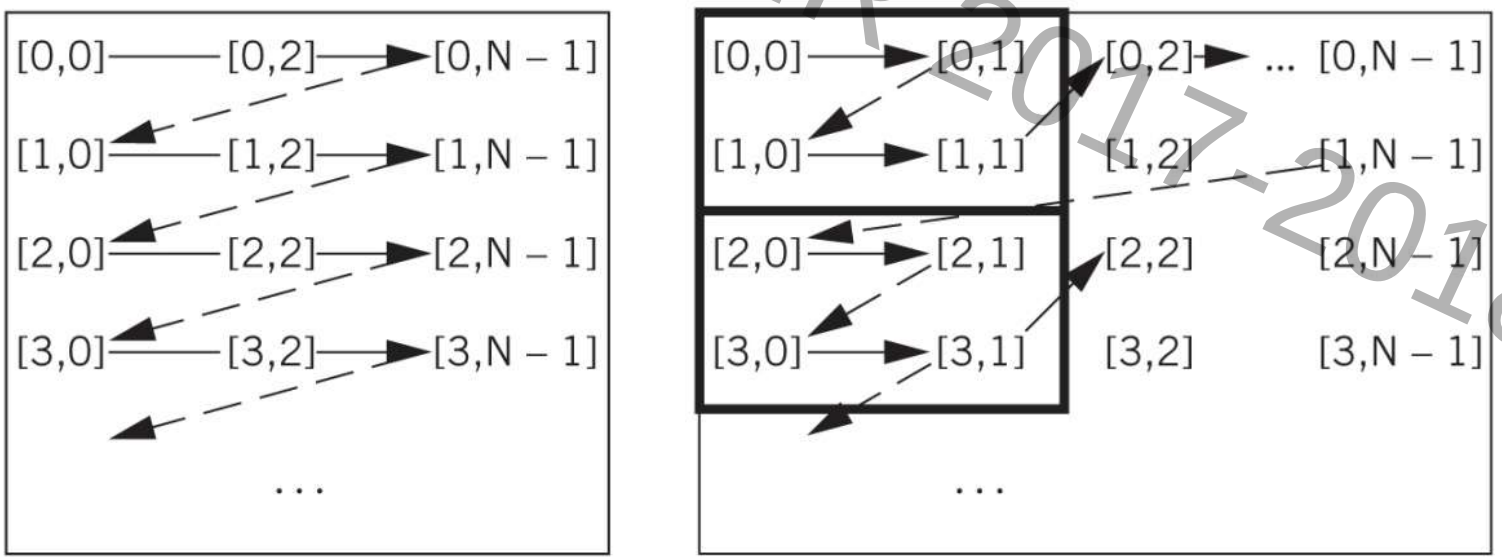
- 将一个循环拆分成一系列嵌套循环，通过改变数组中的访问顺序，改善cache性能

Code

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        c[i] = a[i,j] * b[i];
```

```
for (i = 0; i < N; i += 2)
    for (j = 0; j < N; j += 2)
        for (ii = i; ii < min(i + 2,N); i++)
            for (jj = j; jj < min(j + 2,N); j++)
                c[ii] = a[ii,jj] * b[ii];
```

Access pattern in a array



Before **After**

Fengju Ren@IAIR 2017-2018

数组填充(Array padding)

- 向循环中添加哑数据元素，改变数组在高速缓存中的布局，降低高速缓存冲突。

a[0,0]	a[0,1]	a[0,2]
a[1,0]	a[1,1]	a[1,2]

before

a[0,0]	a[0,1]	a[0,2]	a[0,2]
a[1,0]	a[1,1]	a[1,2]	a[1,2]

after

Pengju Ren@IAIR 2017-2018

寄存器分配(Register allocation)

- 目标:
  - 选择寄存器来保存每个变量
  - 决定变量在寄存器中的生命周期
  - 控制变量（包括声明的和临时的）在寄存器上的分配以最小化所需的寄存器总数。

- 用一个基本语句块的例子来说明

Pengju Ren@IAIR 2017-2018

寄存器生命周期图 lifetime graph

w = a + b;

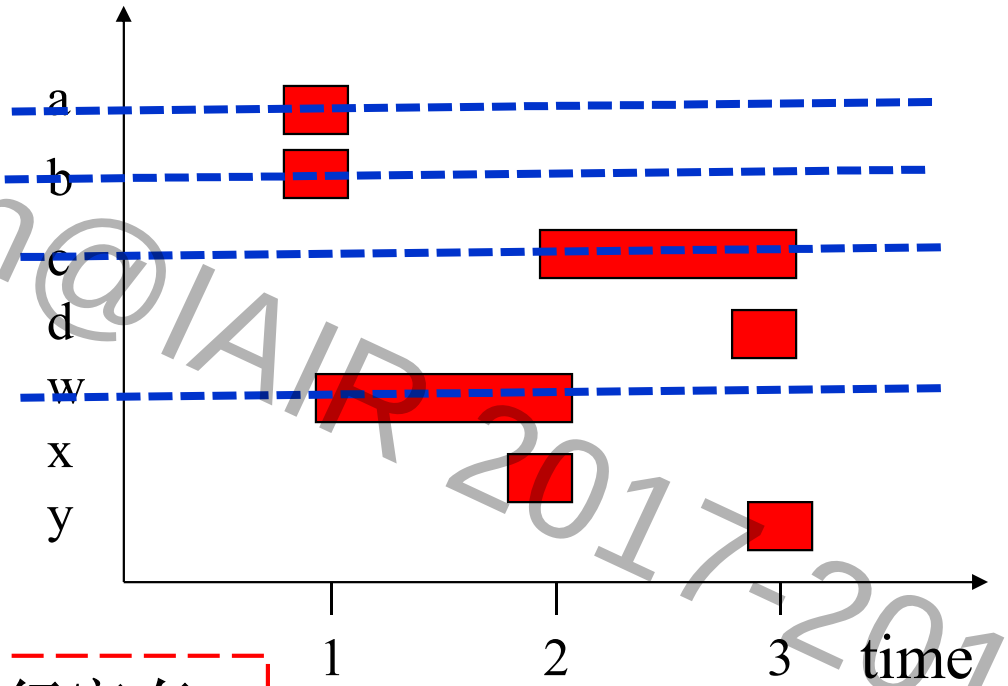
x = c + w;

y = c + d;

t=1

t=2

t=3



如果按照变量的个数进行寄存器分配，将每个变量分配给独立的寄存器，需要7个寄存器。

Pengju Ren@IAIR 2017-2018

优化的寄存器分配

A	r0
B	r1
C	r2
D	r0
W	r3
X	r0
Y	r3

```
LDR r0, [p_a]
LDR r1, [p_b]
ADD r3, r0, r1
STR r3, [p_w]
LDR r2, [p_c]
ADD r0, r2, r3
STR r0, [p_x]
LDR r0, [p_d]
ADD r3, r2, r0
STR r3, [p_y]
```

通过重用寄存器，我们可以编写需要不到**4**个寄存器的代码。

Pengju Ren@IAIR 2017-2018

7 软件性能优化

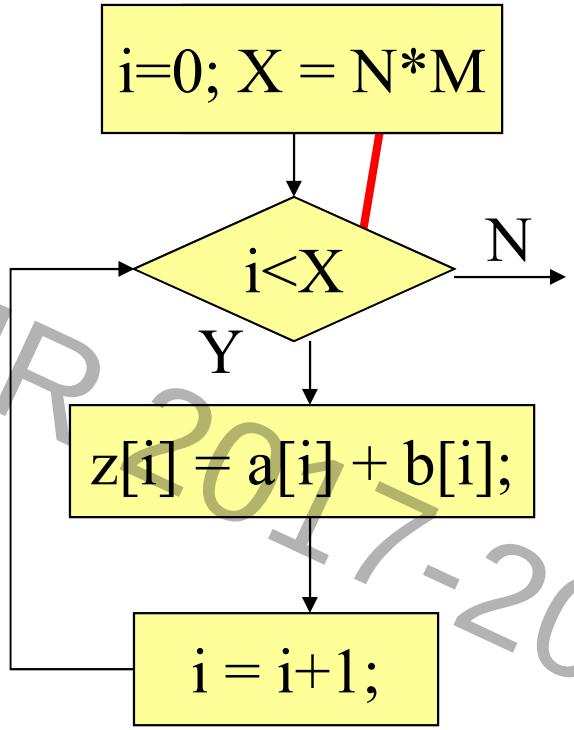
- 循环是优化的重要对象，带有循环结构的程序需要花费较长的执行时间。

- 基本的循环优化技术:
  - 代码移出（**code motion**）
  - 归纳变量消除（**induction-variable elimination**）
  - 强度削减（**strength reduction (x*2 -> x<<1)**）

Pengju Ren@IAIR 2017-2018

代码移出(Code motion)

```
for (i=0; i<N*M; i++)
    z[i] = a[i] + b[i];
```



Pengju Ren@IAIR 2017-2018

归纳变量消除(induction variable elimination)

- 归纳变量源于循环变量的值: **loop index.**

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      z[i,j] = b[i,j];
```

zbinduct = i*M + j;
***(zptr + zbinduct) = *(bptr + zbinduct);**

- 改变每次都重新计算**i*M+j**，利用数组的步进顺序遍历.

```
for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      *(zptr + zbinduct) = *(bptr + zbinduct);
      zbinduct ++;
```

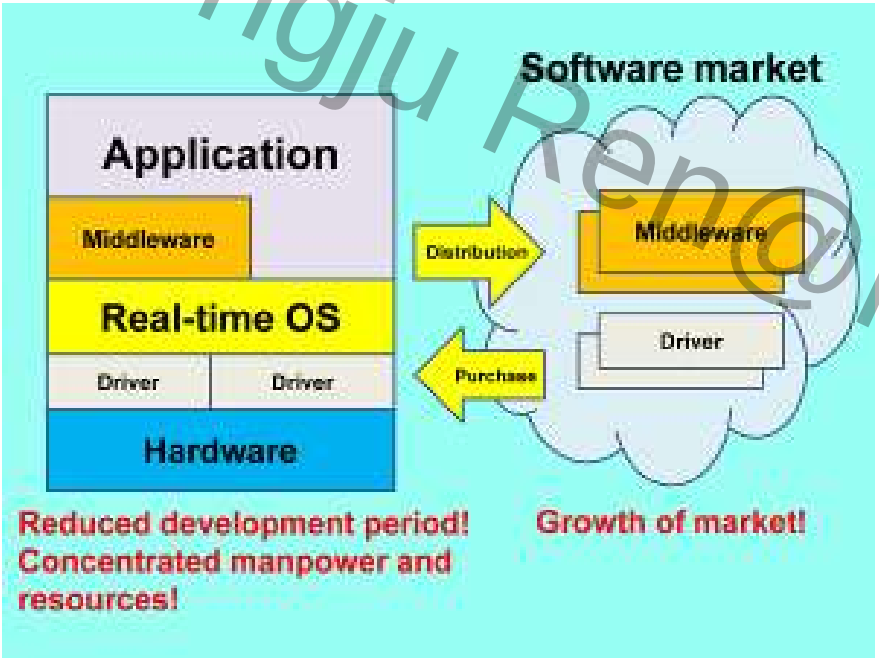
Pengju Ren@IAIR 2017-2018

强度削减(Strength reduction)

- **y = x*2;**
- 用左移替代乘**2**

Pengju Ren@IAIR 2017-2018

进程和操作系统

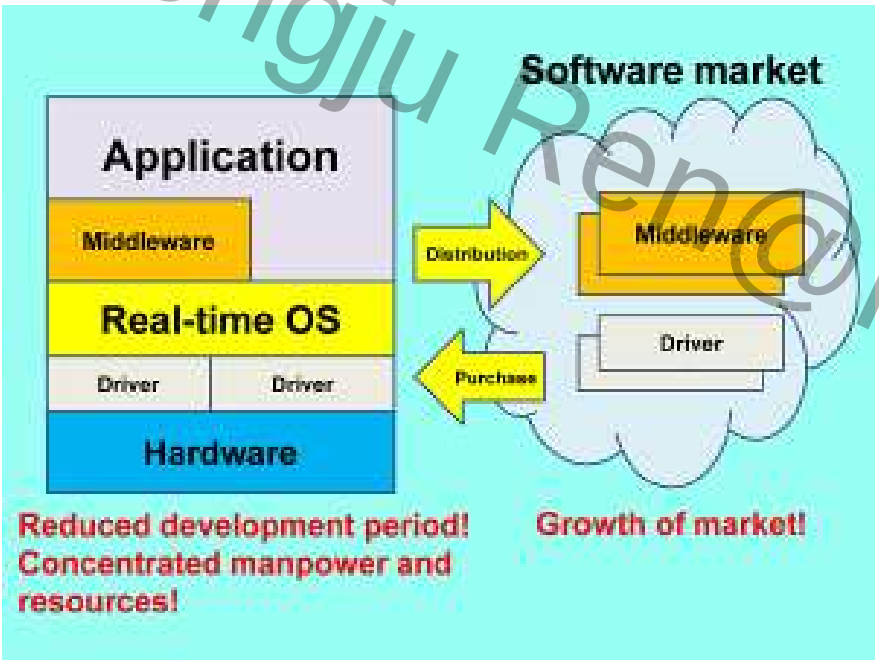


An **operating system** (**OS**) is system software that manages computer hardware and software resources and provides common services for computer programs. All computer programs, excluding firmware, require an operating system to function. *-----From Wikipedia*

操作系统（**OS**）是管理计算机硬件和软件资源并为计算机程序提供公共服务的系统软件。 所有计算机程序（不包括固件）都需要操作系统才能运行。

Pengju Ren@IAIR 2017-2018

进程和操作系统



In computing, a **process** is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system(OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

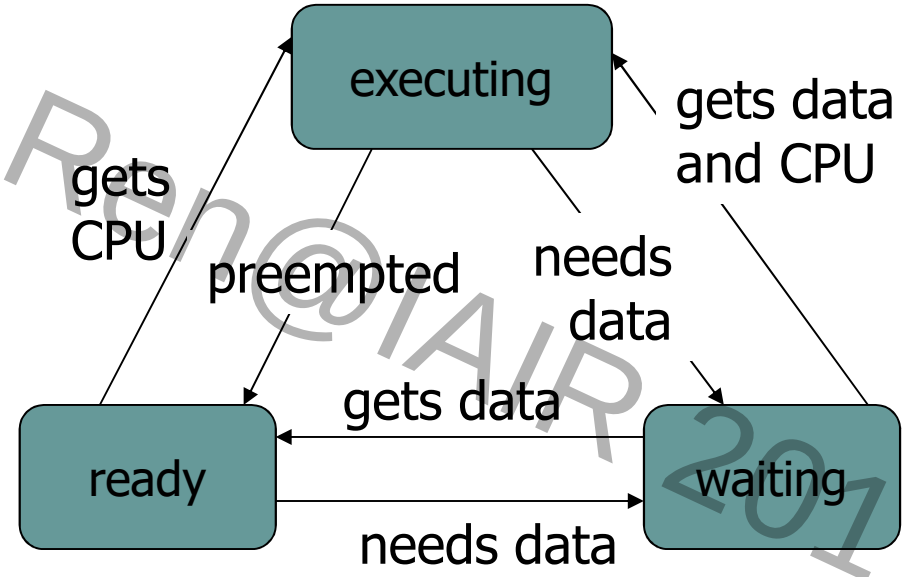
-----From Wikipedia

在计算中，进程是正在执行的计算机程序的一个实例。 它包含程序代码及其当前活动。 根据操作系统（**OS**），一个进程可以由多个并行执行指令的执行线程组成。

- 进程明确定义了当前执行程序的状态
- 操作系统则提供了进程间切换执行的机制

Pengju Ren@IAIR 2017-2018

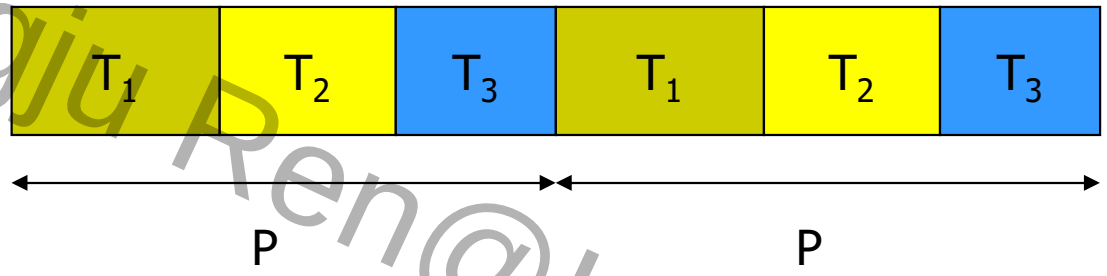
进程的调度状态



Pengju Ren@IAIR 2017-2018

就绪 任务已经获得除处理器之外的一切需要的资源，等待任务调度
执行 任务正在运行中，它得到了所有需要的资源
等待 任务缺少某些必须的运行条件或资源而不能参与任务调度

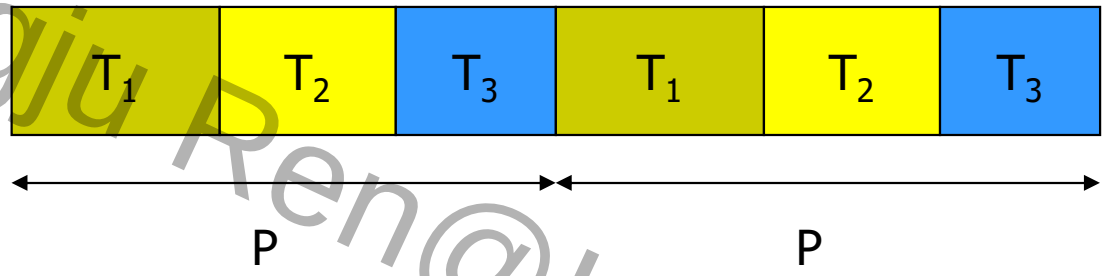
静态循环（Cyclostatic）/时分（TDMA）



- 基于时间片的调度：进程总在相同的时间片内运行。
- CPU的利用率受时间片的数量和用于有用工作的每个时间片影响。

Pengju Ren@IAIR 2017-2018

静态循环（Cyclostatic）/时分（TDMA）



- 基于时间片的调度：进程总在相同的时间片内运行。
- CPU的利用率受时间片的数量和用于有用工作的每个时间片影响。

Pengju Ren@IAIR 2017-2018

运行周期性进程(1)

- 需要代码来控制进程的执行
- 简单的实现方式：进程 = 子例程
- **最简单的实现方式只有1层循环**
- **没有控制各个进程的执行速率，以同速率执行**

```
while (TRUE) {
    p1();
    p2();
}
```

while 循环实现方式

Pengju Ren@IAIR 2017-2018

运行周期性进程(2)

- 将具有相同速率的进程封装在一个函数中
- 使用定时器来控制任务的执行

```
void pall(){
    p1();
    p2();
}
```

定时循环实现

Pengju Ren@IAIR 2017-2018

运行周期性进程(3)

- 用一个函数收集以某一速率运行的所有进程
- 每个任务都有自己的定时器：可能没有足够的定时器支持

```
void pA(){ /* rate A */
    p1();
    p3();
}
void B(){ /* rate B */
    p2();
    p4();
    p5();
}
```

多个定时器实现多速率系统

Pengju Ren@IAIR 2017-2018

运行周期性进程(4)

- 使用软件计数器划分定时器
- 仅允许执行各自速率具有整数倍差异的多进程
- 对不是简单比例关系的多速率系统，进程计数实现可能非常复杂，还可能导致故障。

```
int p2count = 0;
void pall(){
    p1();
    if (p2count >= 2) {
        p2();
        p2count = 0;
    }
    else p2count++;
    p3();
}
```

定时器+计算器的实现方式

Pengju Ren@IAIR 2017-2018

基于优先级的调度

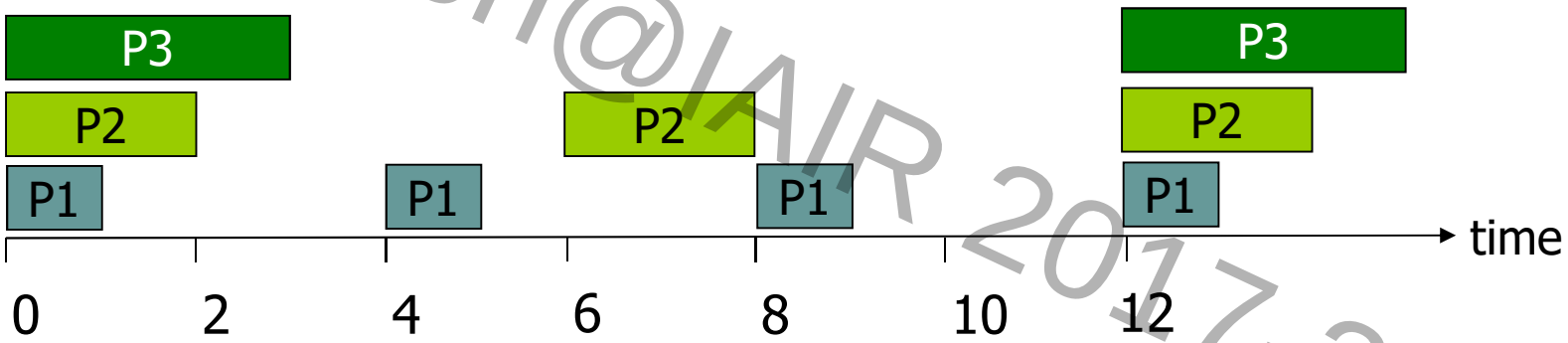
优先级分配的两种方法：

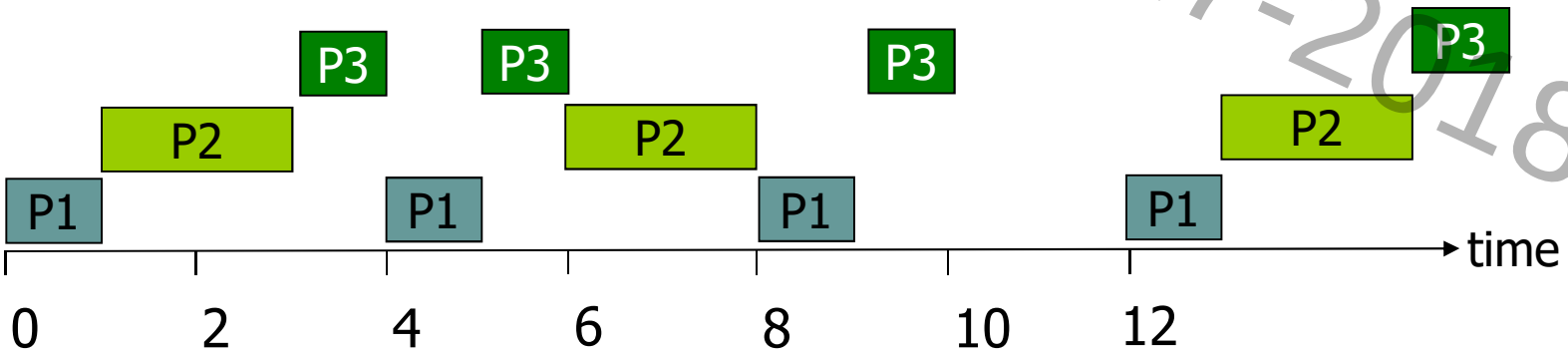
- 静态优先级：整个执行过程中优先级始终不变
  举例：单一速度调度(RMS)
- 动态优先级：在执行过程中优先级发生变化
  举例：最早截止时间优先调度(EDF)

Pengju Ren@IAIR 2011-2018

单一速率调度示例 1（展开调度）

进程	执行时间(Ti)	周期(τ_i)
P1	1	4
P2	2	6
P3	3	12



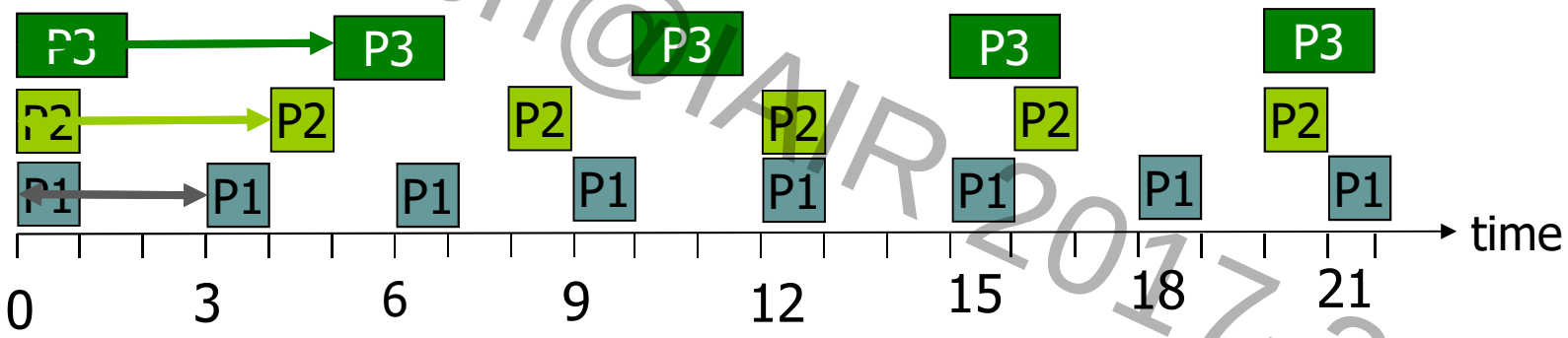


CPU利用率：1/4+2/6+3/12 = 10/12 =83.33%

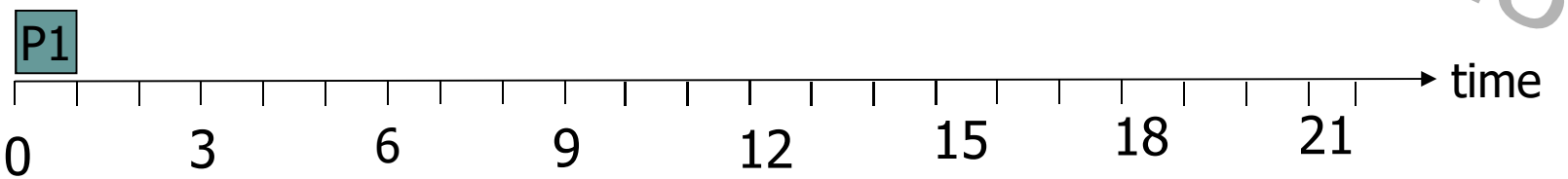
Pengju Ren@IAIR 2017-2018

EDF调度示例-Step1 （展开调度）

进程	执行时间(Ti)	周期(τ_i)	LCM(60)执行次数
P1	1	3	20次
P2	1	4	15次
P3	2	5	12次



Min(3,4,5)

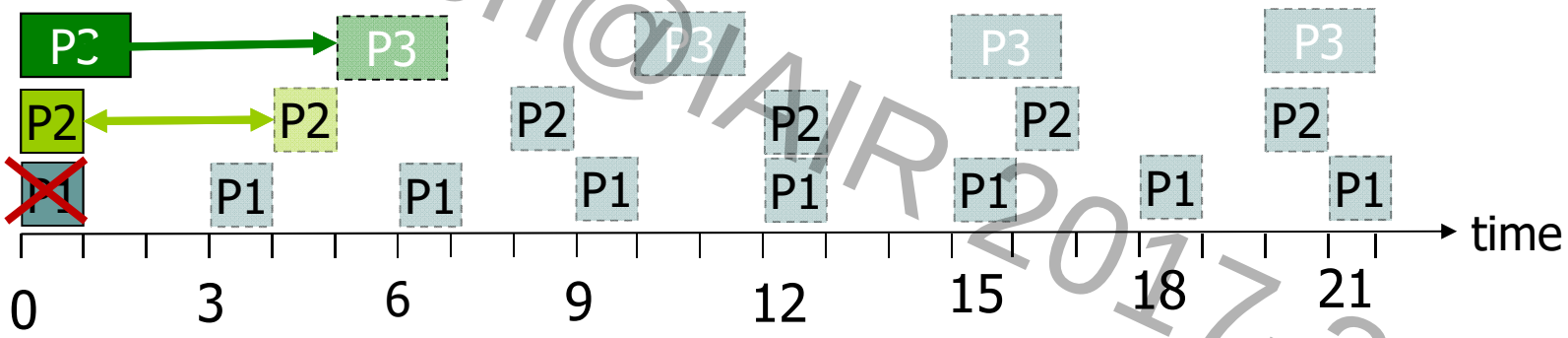


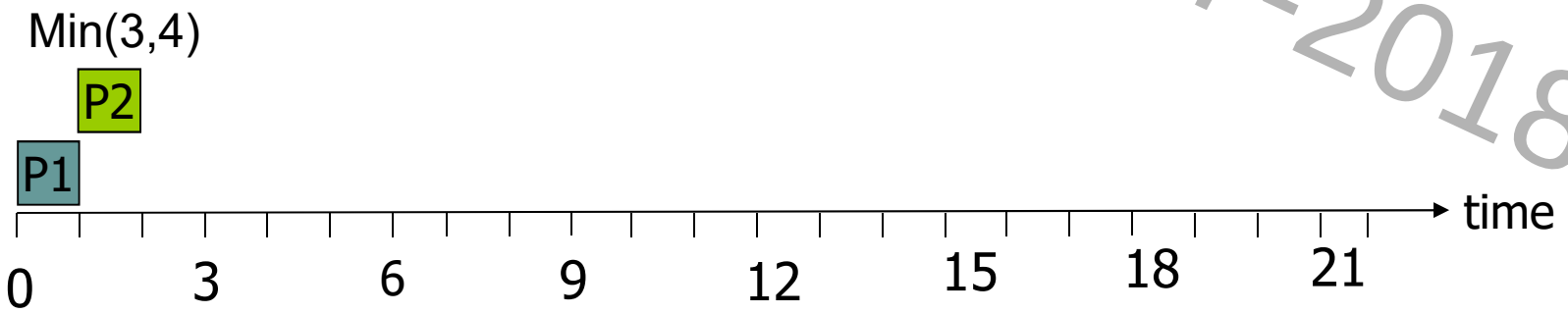
如果在某时刻，多个进程deadline一样，如果包含正在执行的未完成进程，则继续执行当前进程；否则按P1 P2 P3的执行优先顺序。

Pengju Ren@IAIR 2017-2018

EDF调度示例-Step2 （展开调度）

进程	执行时间(Ti)	周期(τ_i)	LCM(60)执行次数
P1	1	3	20次
P2	1	4	15次
P3	2	5	12次



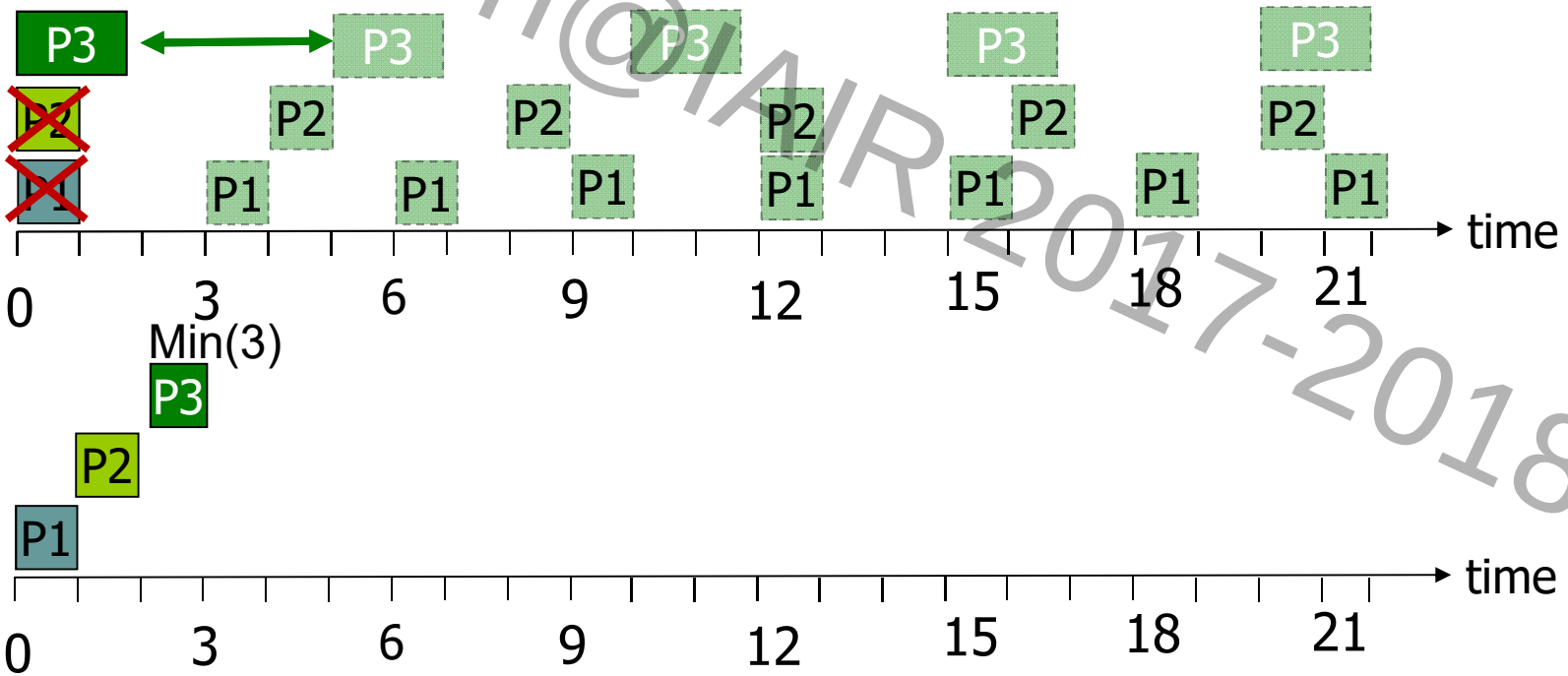


如果在某时刻，多个进程deadline一样，如果包含正在执行的未完成进程，则继续执行当前进程；否则按P1 P2 P3的执行优先顺序。

Pengju Ren@IAIR 2017-2018

EDF调度示例-Step3 （展开调度）

进程	执行时间(Ti)	周期(τ_i)	LCM(60)执行次数
P1	1	3	20次
P2	1	4	15次
P3	2	5	12次

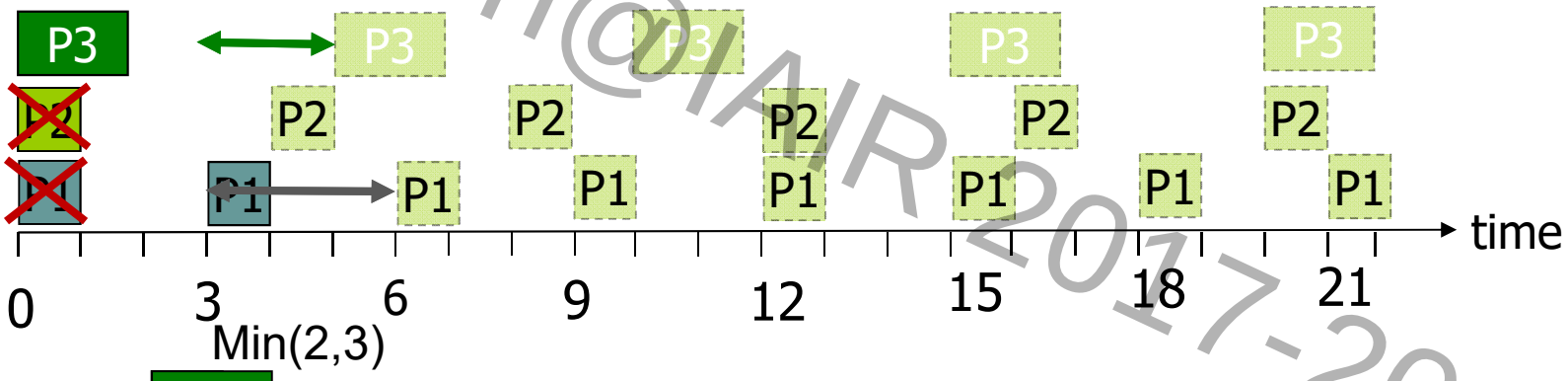


如果在某时刻，多个进程deadline一样，如果包含正在执行的未完成进程，则继续执行当前进程；否则按P1 P2 P3的执行优先顺序。

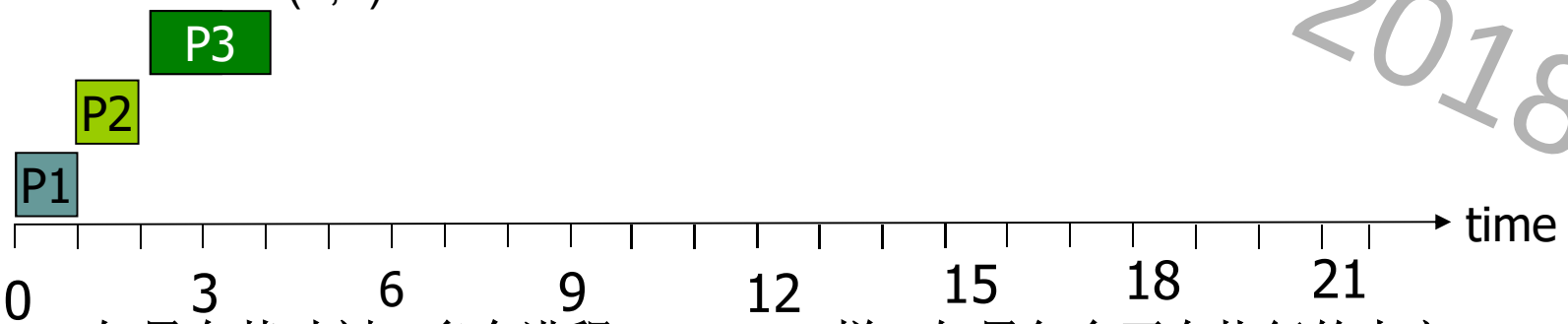
Pengju Ren@IIR 2017-2018

EDF调度示例-Step4 （展开调度）

进程	执行时间(Ti)	周期(τ_i)	LCM(60)执行次数
P1	1	3	20次
P2	1	4	15次
P3	2	5	12次



Min(2,3)

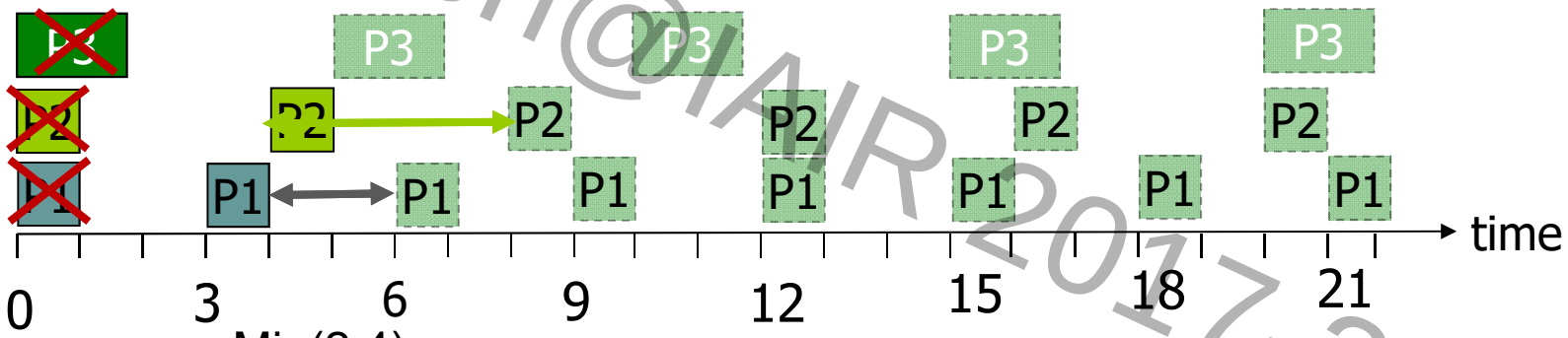


如果在某时刻，多个进程deadline一样，如果包含正在执行的未完成进程，则继续执行当前进程；否则按P1 P2 P3的执行优先顺序。

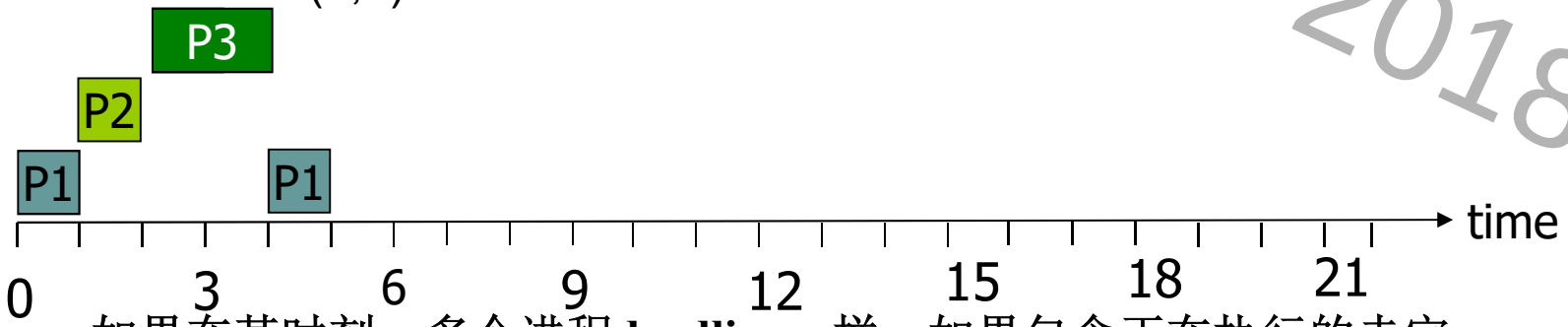
Pengju Ren@IAIR 2017-2018

EDF调度示例-Step5 （展开调度）

进程	执行时间(Ti)	周期(τ_i)	LCM(60)执行次数
P1	1	3	20次
P2	1	4	15次
P3	2	5	12次



Min(2,4)

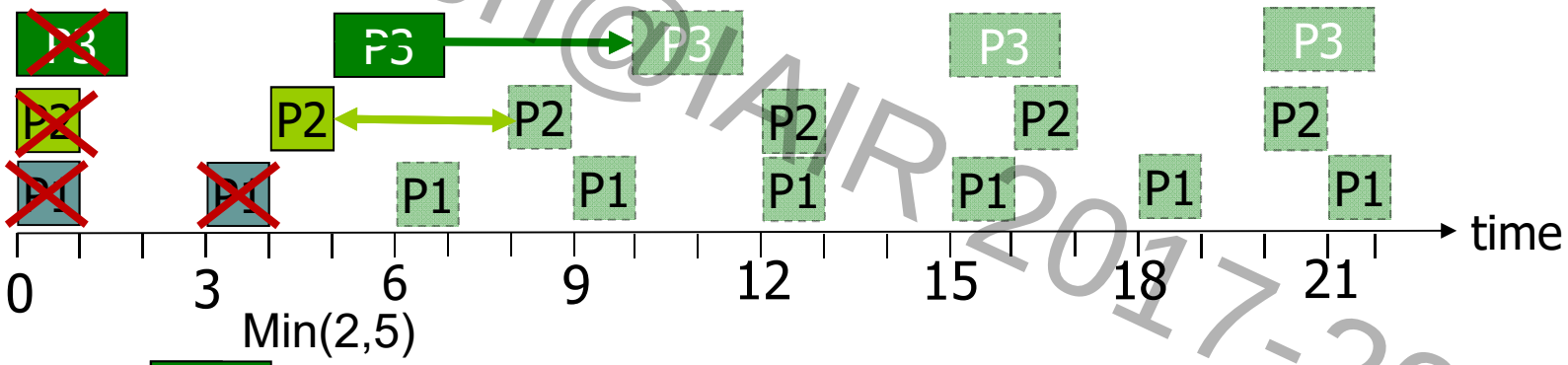


如果在某时刻，多个进程deadline一样，如果包含正在执行的未完成进程，则继续执行当前进程；否则按P1 P2 P3的执行优先顺序。

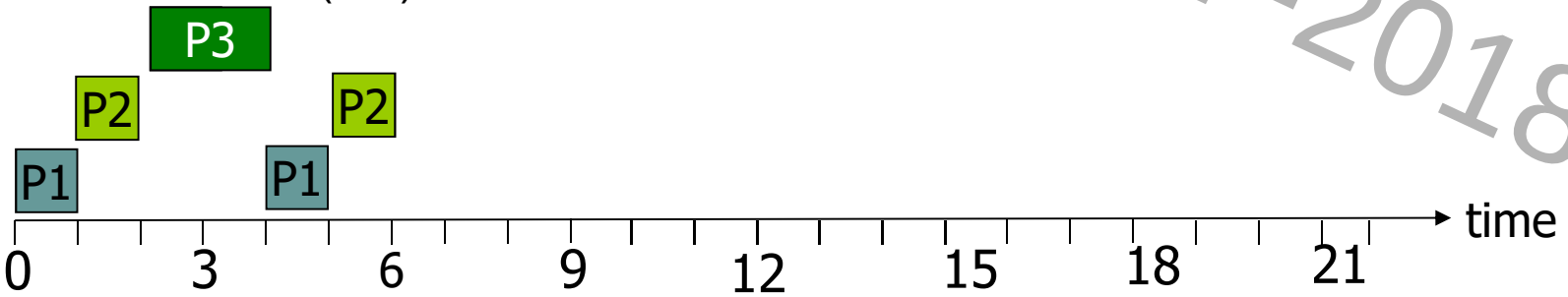
Pengju Ren@IAIR 2017-2018

EDF调度示例-Step6 （展开调度）

进程	执行时间(Ti)	周期(τ_i)	LCM(60)执行次数
P1	1	3	20次
P2	1	4	15次
P3	2	5	12次



Min(2,5)



如果在某时刻，多个进程deadline一样，如果包含正在执行的未完成进程，则继续执行当前进程；否则按P1 P2 P3的执行优先顺序。

Pengju Ren@IAIR 2017-2018

示例6.4　EDF调度

进程	执行时间	周期
P1	1	3
P2	1	4
P3	2	5

进程超周期为60，则P1执行20次（deadline：3,6,9,12等），P2执行15次（deadline：4,8,12等），P3执行12次（deadline：5,10,15,20等）。1-X表示第1次执行，X表示完成。

t	R&E	截止时限	选择
0	P1 P2 P3	3 4 5	P1 1-X
1	P2 P3	3 4	P2 1-X
2	P3	3	P3 1-1
3	P1 P3	3 2	P3 1-X
4	P1 P2	2 4	P1 2-X
5	P2 P3	3 5	P2 2-X
6	P1 P3	3 4	P1 3-X

t	R&E	截止时限	选择
7	P3	3	P3 2-1
8	P2 P3	4 2	P3 2-X
9	P1 P2	3 3	P1 4-X
10	P2 P3	2 5	P2 3-X
11	P3	4	P3 3-1
12	P1 P2 P3	3 4 3	P3 3-X
13	P1 P2	2 3	P1 5-X

如果在某时刻，多个进程deadline一样，如果包含正在执行的进程，则继续执行当前进程；否则按P1 P2 P3的执行优先顺序。

Pengju Ren@IAIR 2017-2018

示例6.4　EDF调度

t	R&E	截止时限	选择
14	P2	1	P2 4-X
15	P1 P3	3 5	P1 6-X
16	P2 P3	4 4	P2 5-X
17	P3	3	P3 4-1
18	P1 P3	3 2	P3 4-X
19	P1	2	P1 7-X
20	P2 P3	4 5	P2 6-X
21	P1 P3	3 4	P1 8-X
22	P3	3	P3 5-1
23	P3	2	P3 5-X
24	P1 P2	3 4	P1 9-X
25	P2 P3	3 5	P2 7-X
26	P3	4	P3 6-1

t	R&E	截止时限	选择
27	P1 P3	3 3	P3 6-X
28	P1 P2	2 4	P1 10-X
29	P2	3	P2 8-X
30	P1 P3	3 5	P1 11-X
31	P3	4	P3 7-1
32	P2 P3	4 3	P3 7-X
33	P1 P2	3 3	P1 12-X
34			
35			
36			
37			
38			
39			

Pengju Ren@IR 2017-2018

示例6.3 RMS调度

进程	执行时间	周期
P1	1	4
P2	2	6
P3	3	12

进程超周期为12，则P1执行3次（deadline：4,8,12），P2执行2次（deadline：6,12），P3执行1次（deadline：12）。1-X表示第1次执行，X表示完成。

时间	就绪&正执行	进程选择
0	P1 P2 P3	P1 1-X
1	P2 P3	P2 1-1
2	P2 P3	P2 1-X
3	P3	P3 1-1
4	P1 P3	P1 2-X
5	P3	P3 1-2
6	P2 P3	P2 2-1

时间	就绪&正执行	进程选择
7	P2 P3	P2 2-X
8	P1 P3	P1 3-X
9	P3	P3 1-X
10		
11		

Pengju Ren@IAIR 2017-2018

考试时间：1月2日

答疑时间：12月29日下午2：00-4：00，科学馆211房间

Pengju Ren@IAR 2017-2018