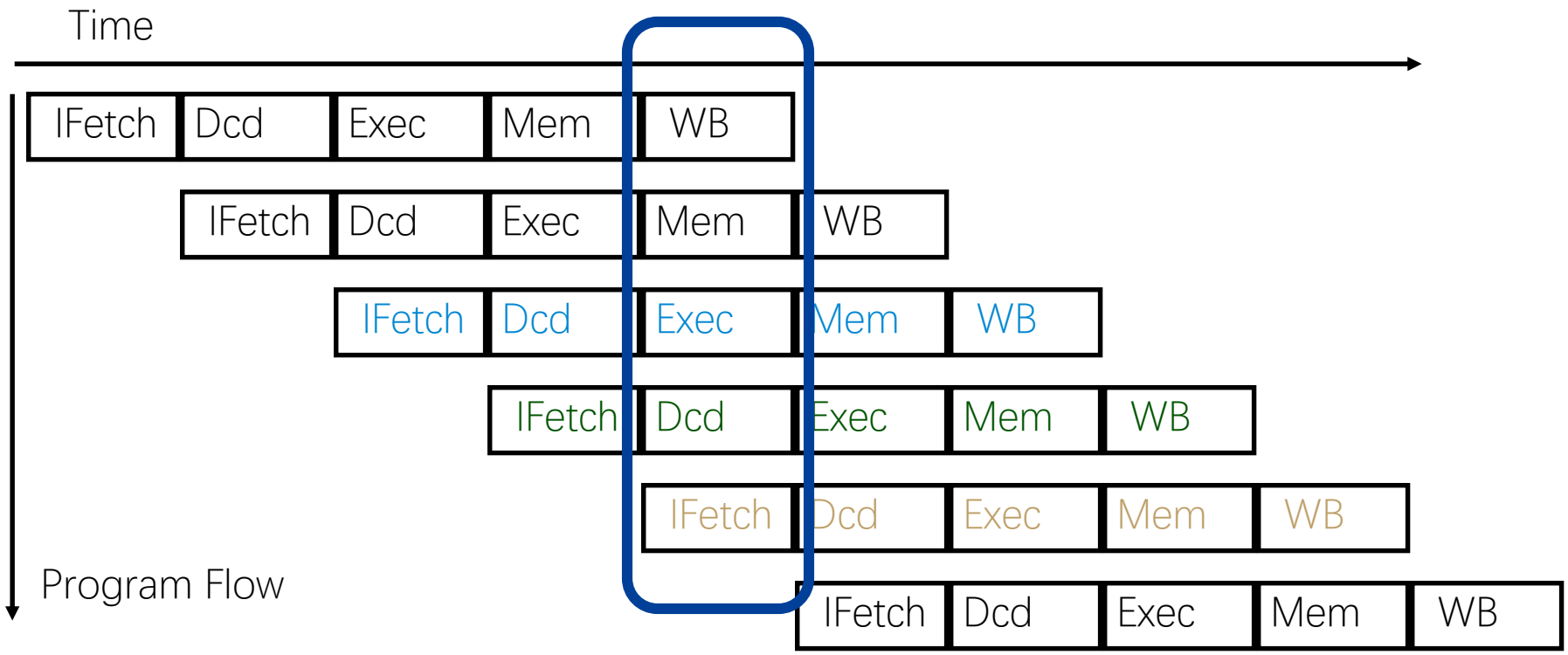
流水线的优化

主讲人: 邓倩妮

上海交通大学

回顾：五阶段指令流水线



理想情况：流水线不停顿

- 每一个周期完成一条指令
- CPI=1；
- CPI：完成一条指令所花的周期数

实际情况：各种情况导致停顿

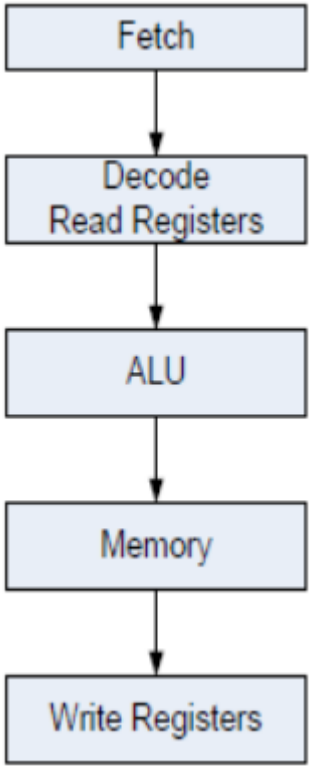
五阶段流水线处理器

优点：

- $CPI_{ideal}=1$
- 简单，容易实现
  - 基本思想仍然用于ARM和MIPS处理器中

缺点：

- 性能的上限：CPI=1
- 延迟高的指令难处理，例如：
  - 乘法指令
- 紧耦合流水线
  - 一条指令停顿，导致后面的指令全部停顿



流水线处理器的性能

- CPU性能公式：
- CPU执行时间 =指令数目 x 平均每条指令所花的时钟周期（CPI）
  x 一个时钟周期长度
- $CPI = CPI_{ideal} + CPI_{stall}$
- $CPI_{ideal} = 1$
- CPI_{stall} 发生的原因
  - 数据相关、
  - 结构冒险
  - 控制冒险
  - 访存延迟

流水线处理器的性能优化（1）

- CPU执行时间 =指令数目 x 平均每条指令所花的时钟周期（CPI）
  x 一个时钟周期长度
- 策略1：减少一个时钟周期的长度（提高时钟频率）
- 超级流水(superpipelining)
  - 增加流水线的段数，提升时钟频率 、从而提高指令吞吐率

时钟周期=200ps + 50ps(流水段寄存器的延迟） = 250ps

时钟周期=100ps+ 50ps=150ps

流水线的深度

- 流水线的级数是越多越好吗？ 当然不！

时钟周期：200ps+50ps=250ps
单条指令的延迟：1250ps
流水段寄存器延迟所占比例：50ps / 250ps = 20%

时钟周期：100ps+50ps=150ps
单条指令的延迟：1500ps （增加了！）
流水段寄存器延迟所占比例：50ps / 150ps =33%

原因： 流水线越深，流水段寄存器延迟就越多，一条指令的延迟就越大！

深度流水导致的问题

- 性能下降
  - 流水段寄存器个数增加，一条指令的延迟就越大
  - 重叠执行的指令越多 → 可能出现的相关性越多 → 停顿的可能性就越大 →CPI 增加
- 开销增加
  - 复杂度：流水段越多，前向通路越多
- 功耗
  - 时钟频率高，功耗越大

$$\text{Power}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$$

处理器流水线深度的变化

- 1986年，MIPS R2000: 5级
- 1988年，MIPS R3000 5级
- 1991年，MIPS R4000（64位）: 8级
- 1997年，ARM9: 5级
- 2002年，ARM11: 8级
- 2009年，Cortex-A8：13级
- 2011年，Cortex-A15：15级
- 2013年，Cortex-A57：15级

- 1993年，Pentium: 5级
- 1995年，Pentium Pro: 12级
- 2004年，Pemtium 4(Prescott)：**31级**
- 2006年，Core 2 Duo(Merom)：14级
- 2008年，Core i7(Nehalem)：16级
- 2013年，Core i7(Haswell)：14级

取指1	取指2	译码	发射	执行1	执行2	执行3	写回

ARM11: 8级流水线

流水线处理器的性能优化（2）

- CPU执行时间 =指令数目 x 平均每条指令所花的时钟周期（CPI）
  x 一个时钟周期长度
- 策略2：进一步减小CPI
- 多发射：让多条指令在流水线中并行执行
  - 一次取出并执行多条指令

```
OpA   F  D  A0 A1 W
OpB   F  D  B0 B1 W
OpC      F  D  A0 A1 W
OpD      F  D  B0 B1 W
OpE         F  D  A0 A1 W
OpF         F  D  B0 B1 W
```

- 双发射流水线
- 一个周期能发射两条指令，完成两条指令
- CPI_{ideal} =0.5

Multiple Issue 多发射

- Multiple Issue Processor(多发射处理器)
- 利用指令级并行性（Instruction Level Parallism: ILP ）获得 CPI < 1
  - static multiple issue (静态多发射：编译时确定多发射)
    - 编译器在编译时，决定哪些指令可以同时发射
  - dynamic multiple issue (动态多发射)
    - 超标量处理器 （ superscalar processors ）
    - 执行时确定哪些指令并行发射，
      - 可以是按序（ in-order ）执行的
      - 也可以是乱序（ out-of-order ）的超标量处理器
        - 猜测执行

术语

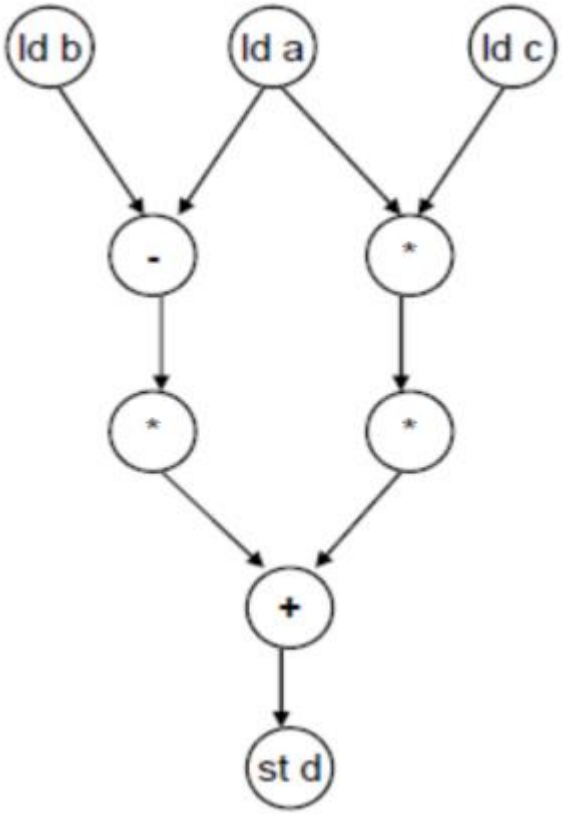
- Instruction Level Parallism（指令级并行性: ILP ）
- Multiple-Issue (多发射)
- 静态多发射 =
- Very long instruction word (超长指令字)
- 动态多发射 =
- Superscalar (超标量)
  - 按序执行
  - Out of order execution (乱序执行)
- Register renaming (寄存器换名)
- In order commit （按序提交）
- Branch prediction (分支预测)
- Speculation（猜测执行）

指令级并行性: ILP

$$D = 3(a-b) + 7ac$$

Sequential execution order

ld a
ld b
sub a-b
mul 3(a-b)
ld c
mul ac
mul 7ac
add 3(a-b)+7ac
st d



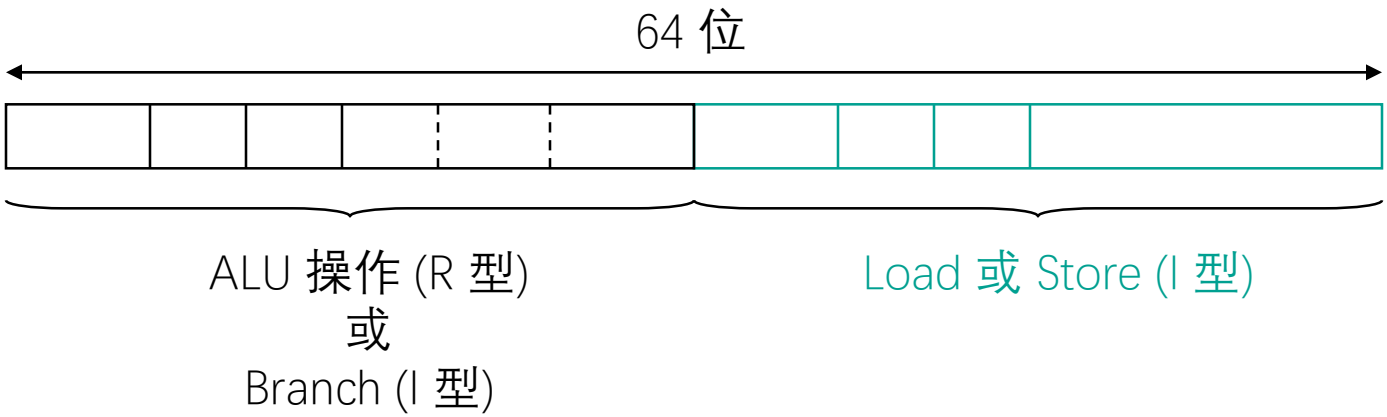
静态多发射(Static multiple-issue)处理器

- 又称为“超长指令字处理器” (VLIW：Very Long Instruction Word)
  - 编译器决定哪些指令并行发射和同时执行
  - Issue packet – 将多条可以并行发射的指令，看做一条长指令，在同一周期中发射。
  - 在一条长指令中，混合的指令类型和条数是有限制的。 – a single “instruction” with several predefined fields
  - 编译器同时也进行静态转移预测和代码调度– reduce (control) or eliminate (data) hazards

- 支持 VLIW 的处理器
  - 多个功能部件 （ functional units ）
  - 多端口寄存器（ Multi-ported register files ）
  - 更宽的存储器总线 （ Wide program bus ）

一个 VLIW MIPS 处理器实例

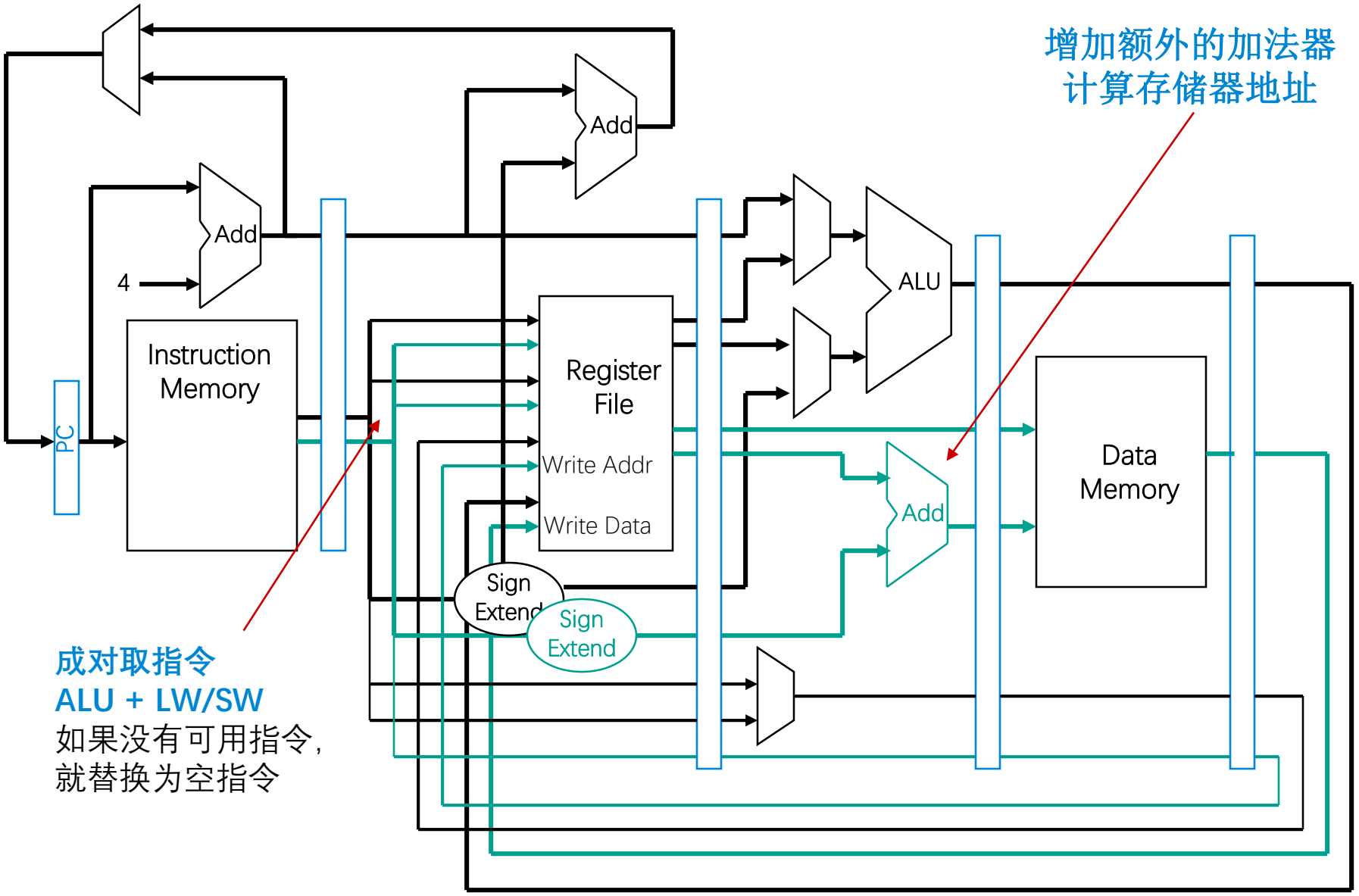
- 双发射：2条MIPS 指令构成一个issue packet



- 指令都是成对的取指（fetched），译码（ decoded ），发射 （ issued ）
  如果没有可用指令，就替换为 noop 空指令

- 寄存器文件：
  - 4 读端口 （ read ports ）
  - 2 写端口 （ write ports ）
  - 1个独立的存储器地址加法器 （ memory address adder ）

MIPS VLIW (2-发射) 数据通路



代码调度实例：

- 考虑以下循环语句：

```
lp:   lw    $t0,0($s1)    # $t0=数组元素
      addu  $t0,$t0,$s2   # 加上$s2中的某个标量
      sw    $t0,0($s1)    # 将运算结果写入存储器
      addi  $s1,$s1,-4    # 重新计算数组元素的地址
      bne   $s1,$0,lp     # if $s1 != 0 跳转
```

- 指令调度“schedule”
  - 避免流水线停顿
  - 在一个指令束中的指令必须不相关

Lw, addu 指令间有 load use 相关性

有Load-use相关的指令之间必须间隔一个周期

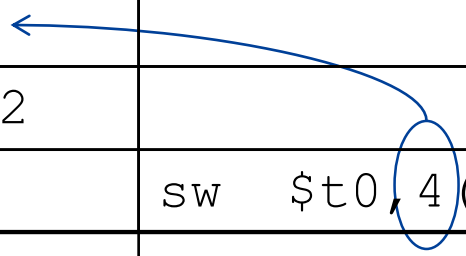
addu与sw、 addi和bne之间有数据相关性,

假设转移指令总能被正确的预测

被调度的代码(循环未展开)

```
lp:     lw      $t0,0($s1)      # $t0=数组元素
        addu    $t0,$t0,$s2     # 加上$s2中的某个标量
        sw      $t0,0($s1)      # 将运算结果写入存储器
        addi    $s1,$s1,-4      # 重新计算数组元素的地址
        bne     $s1,$0,lp
```

	ALU or branch	Data transfer	CC
lp:		lw $t0,0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$0,lp	sw $t0,4($s1)	4
			5



- 4个周期执行 5 条指令
- CPI = 0.8 (理想情况是 0.5)
  - IPC（每个周期完成的指令=1.25 (理想情况 2.0)
  - 插入的空指令，对性能改善无帮助!!

循环展开（Loop Unrolling）

- 循环展开
  - 在不同循环体中的指令可以并行调度，同时执行，以提高指令级并行性（ILP）
- 调度
  - 消除不必要的、会增加循环开销的指令
  - 避免 load-use 相关性
- 编译器
  - 寄存器换名 （register renaming）
  - 消除非“真正”的相关性
- 本例
  - 循环展开4次，对展开后的代码进行调度

初始的代码：

```
lp:     lw      $t0,0($s1)      # $t0=数组元素
        addu    $t0,$t0,$s2     # 加上$s2中的某个标量
        sw      $t0,0($s1)      # 将运算结果写入存储器
        addi    $s1,$s1,-4      # 重新计算数组元素的地址
        bne     $s1,$0,lp
```

循环展开、调度指令的顺序、消除不必要的指令、寄存器换名：

```
lp:     lw      $t0,0($s1)      # $t0=array element
        lw      $t1,-4($s1)     # $t1=array element
        lw      $t2,-8($s1)     # $t2=array element
        lw      $t3,-12($s1)    # $t3=array element
        addu    $t0,$t0,$s2     # add scalar in $s2
        addu    $t1,$t1,$s2     # add scalar in $s2
        addu    $t2,$t2,$s2     # add scalar in $s2
        addu    $t3,$t3,$s2     # add scalar in $s2
        sw      $t0,0($s1)      # store result
        sw      $t1,-4($s1)     # store result
        sw      $t2,-8($s1)     # store result
        sw      $t3,-12($s1)    # store result
        addi    $s1,$s1,-16     # decrement pointer
        bne     $s1,$0,lp       # branch if $s1 != 0
```

被调度的代码(循环展开4次)

	ALU or branch	Data transfer	CC
lp:	addi $s1,$s1,-16	lw $t0,0($s1)	1
		lw $t1,12($s1)	2
	addu $t0,$t0,$s2	lw $t2,8($s1)	3
	addu $t1,$t1,$s2	lw $t3,4($s1)	4
	addu $t2,$t2,$s2	sw $t0,16($s1)	5
	addu $t3,$t3,$s2	sw $t1,12($s1)	6
		sw $t2,8($s1)	7
	bne $s1,$0,lp	sw $t3,4($s1)	8

- 8个时钟周期完成 14 条指令
- CPI = 0.57 (理想情况为：0.5)
- IPC = 1.8 (理想情况：2.0)

编译器支持的 VLIW 处理器

- 总结： 对VLIW 处理器，编译器负责的工作包括：
  - 将不相关的指令打包成为一束长指令
  - 通过展开循环开发指令级并行性（ILP）
    - 展开循环后，需要减少条件转移指令的数目
  - 通过寄存器换名消除名字引起的相关性
  - 主要通过编译器进行分支预测（branch prediction）
    - 将 if-then-else branch 结构替换为预测的指令
  - 预测存储器访问，最小化存储体访问冲突

VLIW 优点& 缺点

- 优点：
  - 硬件简单 、功耗低
- 缺点：
  - 复杂的编译器，编译时间长
  - 二进制代码不兼容
  - 代码量膨胀
    - 插入的空指令，是对程序存储空间的浪费
    - 循环展开导致程序体变长
  - 无法解决“存储器别名”（“memory aliasing”问题）
    - 避免对loads 和 stores 指令的顺序进行调整
  - 无法对执行时的情况进行预测，例如 cache miss

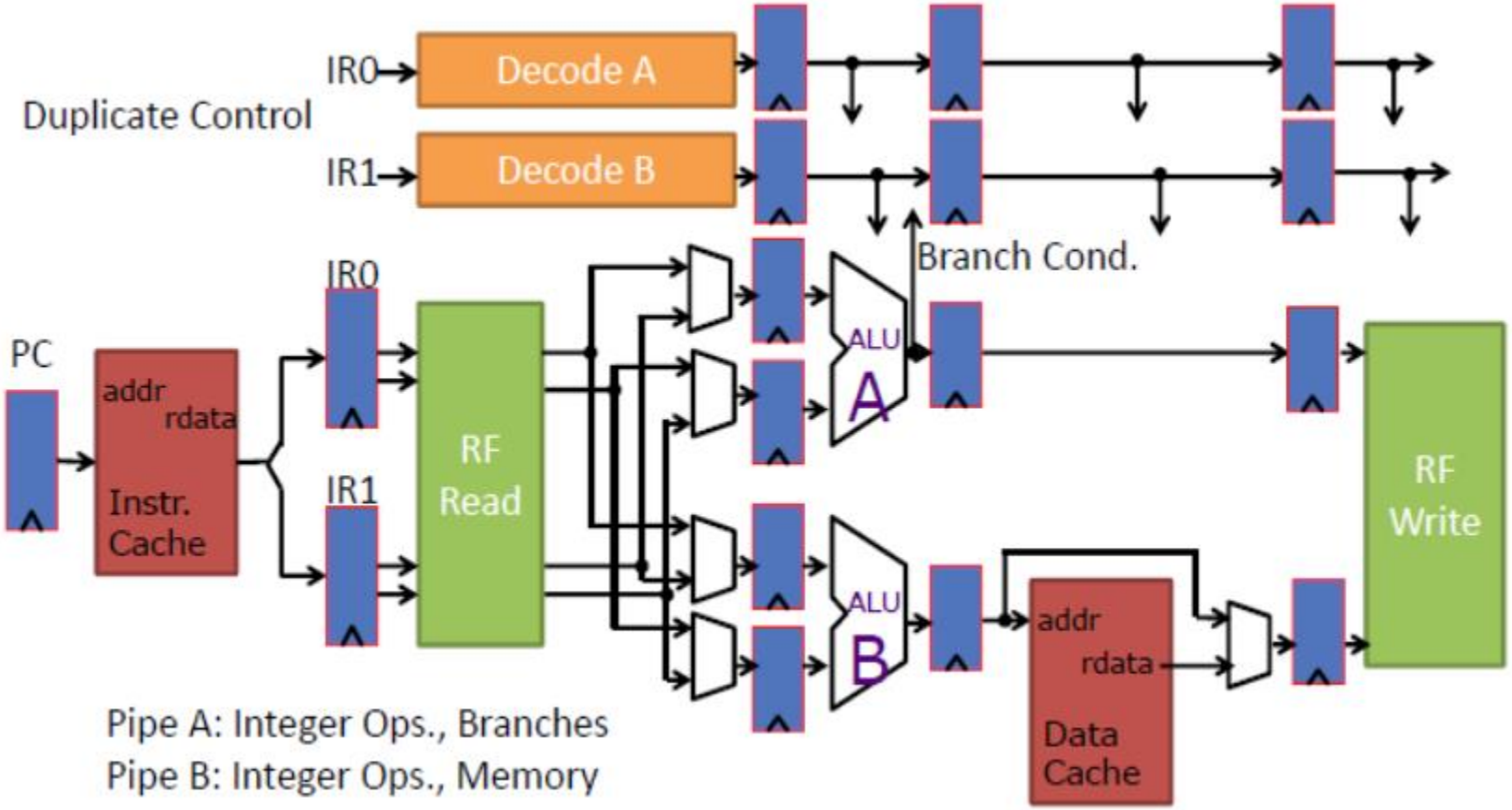
动态多发射处理器 (Dynamic multiple-issue)

- 又称为超标量处理器(SuperScalar)
  - 使用硬件，在运行时动态决定哪些指令可以并行发射、同时执行

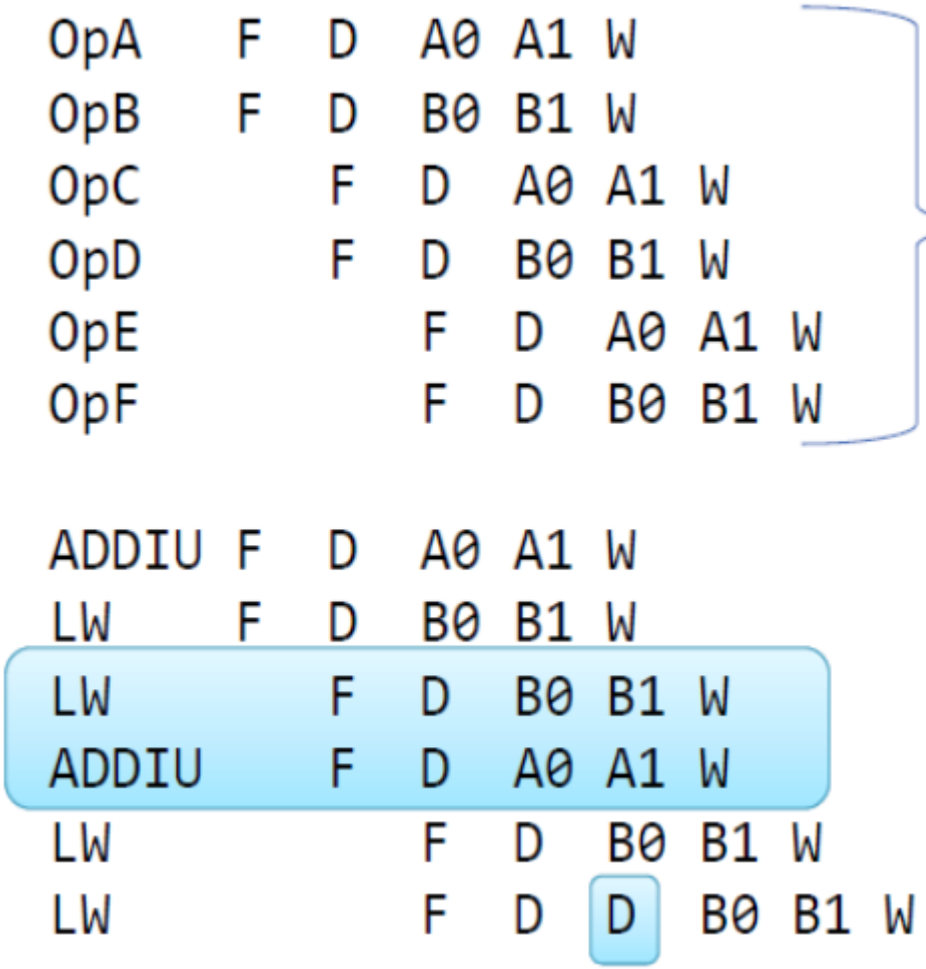
指令执行过程一般有以下步骤：

- 取指令（Instruction-fetch）和发射（issue）
  - 取指令
  - 译码
  - 发射：将指令分配给某个功能单元（FU）并等待执行
- 指令执行（Instruction-execution）
  - 按序执行(In- Order），
    - 例如： ARM Cortex A8
  - 乱序执行（Out-of-Order， OOO）
    - 准备好的指令可以先执行，
    - 例如： ARMA9, A15； Intel Core i3,i5,i7
- 指令提交（Instruction-commit）
  - 为了安全性和正确性，一般为按序提交（commit）

举例：一个双发射、按序执行的超标量处理器



双发射按序超标量处理器：流水线时空图（1）



CPI = 0.5 (IPC = 2)

双发射
一次最多执行两条指令：
Line A: ALU op /Branch 指令
Line B: Load/Store 指令

指令发射逻辑：
根据指令类型选择功能部件

结构冒险导致停顿

双发射按序超标量处理器：数据相关性

无前向通路（no forwaridng):

```
ADDIU R1,R1,1   F   D   A0 A1 W
ADDIU R3,R4,1   F   D   B0 B1 W
ADDIU R5,R6,1       F   D   A0 A1 W
ADDIU R7,R5,1       F   D   D   D   D   A0 A1 W
```

有前向通路（forwaridng):

```
ADDIU R1,R1,1   F   D   A0 A1 W
ADDIU R3,R4,1   F   D   B0 B1 W
ADDIU R5,R6,1       F   D   A0 A1 W
ADDIU R7,R5,1       F   D   D   A0 A1 W
```

按序超标量的局限之处

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [r1] ➜ r2	F	D	X	M_1	M_2	W							
add r2 + r3 ➜ r4	F	D	d*	d*	d*	X	M_1	M_2	W				
xor r4 ^ r5 ➜ r6		F	D	d*	d*	d*	X	M_1	M_2	W			
ld [r7] ➜ r8		F	D	p*	p*	p*	X	M_1	M_2	W			

- 按序流水线 F(取指）D（译码）X（执行） M1 M2（访存） W（写回）
- 相关性：load-use 相关性、数据相关性
- 停顿：三个周期

乱序超标量

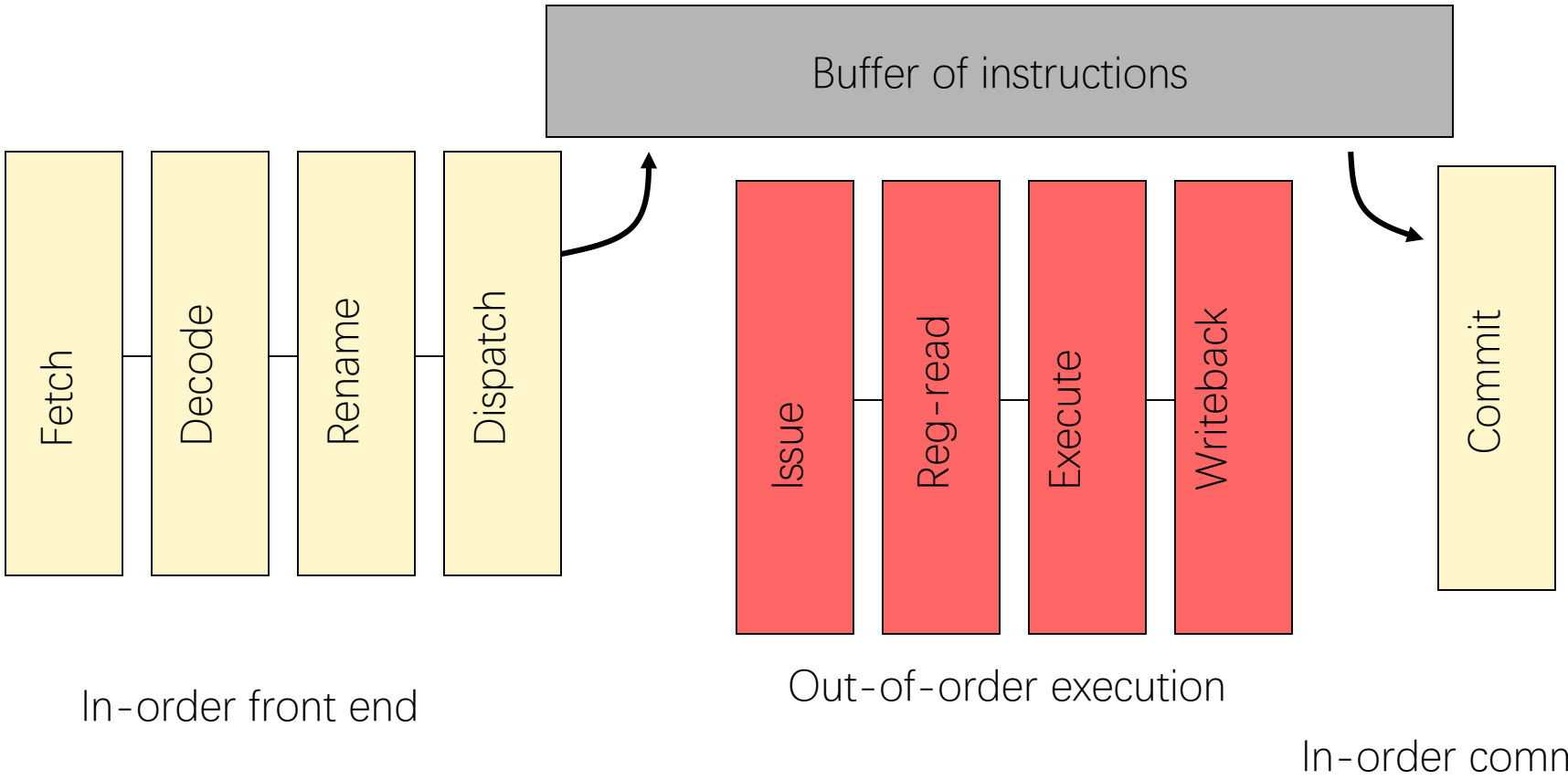
	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [r1] ➜ r2	F	D	X	M_1	M_2	W							
add r2 + r3 ➜ r4	F	D	**d***	**d***	**d***	X	M_1	M_2	W				
xor r4 ^ r5 ➜ r6		F	D	**d***	**d***	**d***	X	M_1	M_2	W			
ld [r7] ➜ r8		F	D	X	**M_1**	**M_2**	**W**						

- 乱序流水线 ：允许不相关的、后发射的指令先执行
- 例如：ld [r7] ➜ r8 先执行，
- 减少了延迟时间

乱序超标量处理器的结构

- 指令执行过程一般有以下步骤：
  - 取指令（Instruction-fetch）和发射（issue）
    - 必须是“按序”的，这样才能记录指令的顺序
    - 指令超前（Instruction lookahead） 能力 – 可以提前取指、译码和发射指令
  - 指令执行（Instruction-execution）
    - 乱序：只要操作数和功能单元准备好，就可以开始执行
    - 处理超前 （processor lookahead）能力– 后发射的指令可以先完成计算
  - 指令提交（Instruction-commit）
    - 必须按照取指顺序提交，以应对：
      - 中断：维持精确中断，执行某指令时发生中断时，能保证：所有该指令之前的指令都已经提交其状态；所有后续指令（包括发生中断的指令）没有改变任何机器的状态
      - 转移预测错误：因为预测错误而发射和执行的指令不会影响机器的状态

Out-of-order Pipeline



乱序执行引发的相关性：输出相关

- 写后写（write after write） 数据相关性

```
lw    $t0,0($s1)
addu  $t0,$t1,$s2
. . .
sub   $t2, $t0, $s2
```

如果 lw 写$t0 发生在 addu 写之后，会导致sub 获得错误的$t0值

addu 与lw 有一个输出相关性（output dependency）

乱序执行引发的反相关

- 反相关（anti-dependencies）– 顺序排在后面(先执行)的指令产生的数据结果，是另一条顺序排在前面（后执行）的指令的数据来源。

```
R3 := R3 * R5
R4 := R3 + 1
R3 := R5 + 1
```

Antidependency

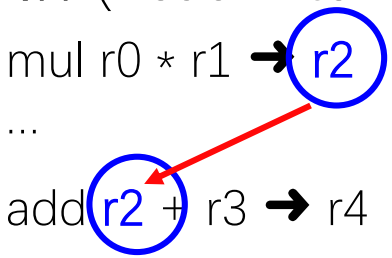
True data dependency

Output dependency

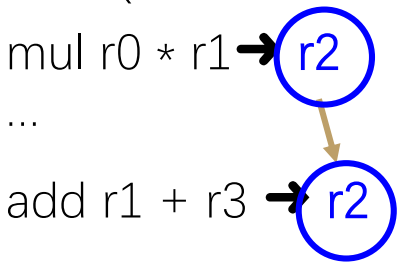
- 除了真相关，反相关和输出相关可以通过换名消除

Dependence types

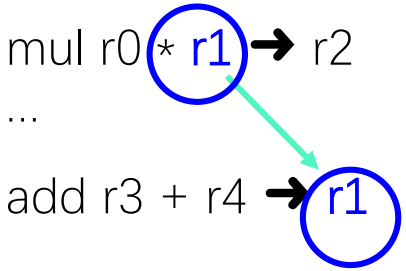
- **RAW** (Read After Write) = "true dependence" (true)



- **WAW** (Write After Write) = "output dependence" (false)



- **WAR** (Write After Read) = "anti-dependence" (false)



WAW & WAR are "false", Can be **totally eliminated** by "renaming"

寄存器换名 Register Renaming

- Register renaming – the processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

```
R3 := R3 * R5          R3b := R3a * R5a
R4 := R3 + 1     ⇒     R4a := R3b + 1
R3 := R5 + 1           R3c := R5a + 1
```

Step#1: 寄存器换名（Register Renaming）

- 消除寄存器冒险
- “结构” vs “物理” 寄存器
  - Names: **r1,r2,r3**
  - Locations: **p1,p2,p3,p4,p5,p6,p7**
  - Original mapping: **r1→p1**, **r2→p2**, **r3→p3**, **p4–p7** are “available”

Time

MapTable

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6

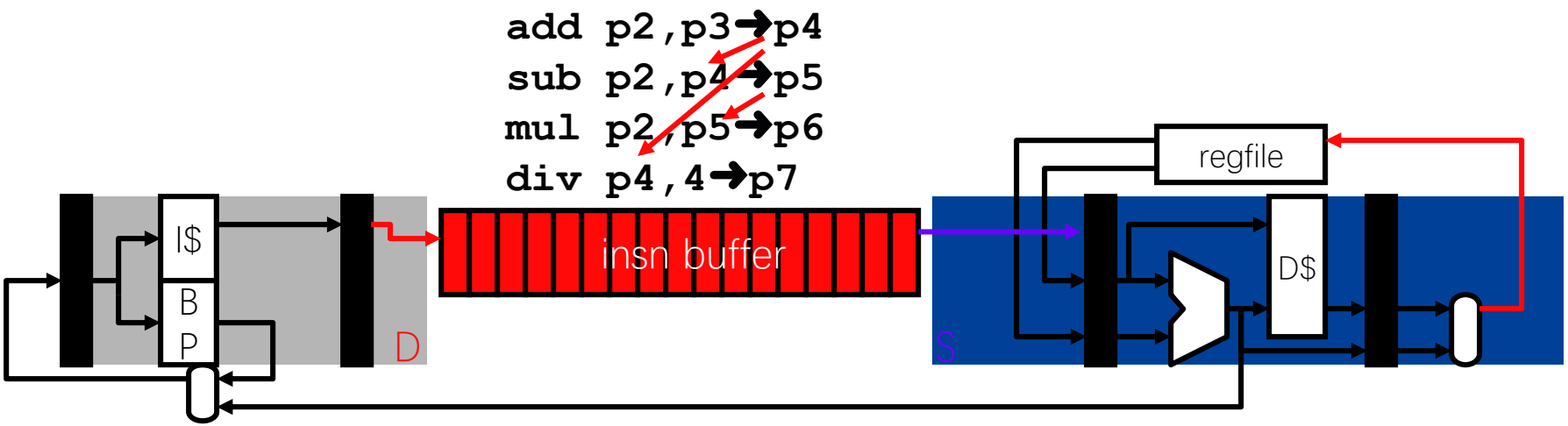
FreeList

p4,p5,p6,p7
p5,p6,p7
p6,p7
p7

Original insns	Renamed insns
add r2,r3➜r1	add p2,p3➜p4
sub r2,r1➜r3	sub p2,p4➜p5
mul r2,r3➜r3	mul p2,p5➜p6
div r1,4➜r1	div p4,4➜p7

通过换名，消除反相关、输出相关，保留真相关

Step #2: 动态调度（Dynamic Scheduling）



Ready Table

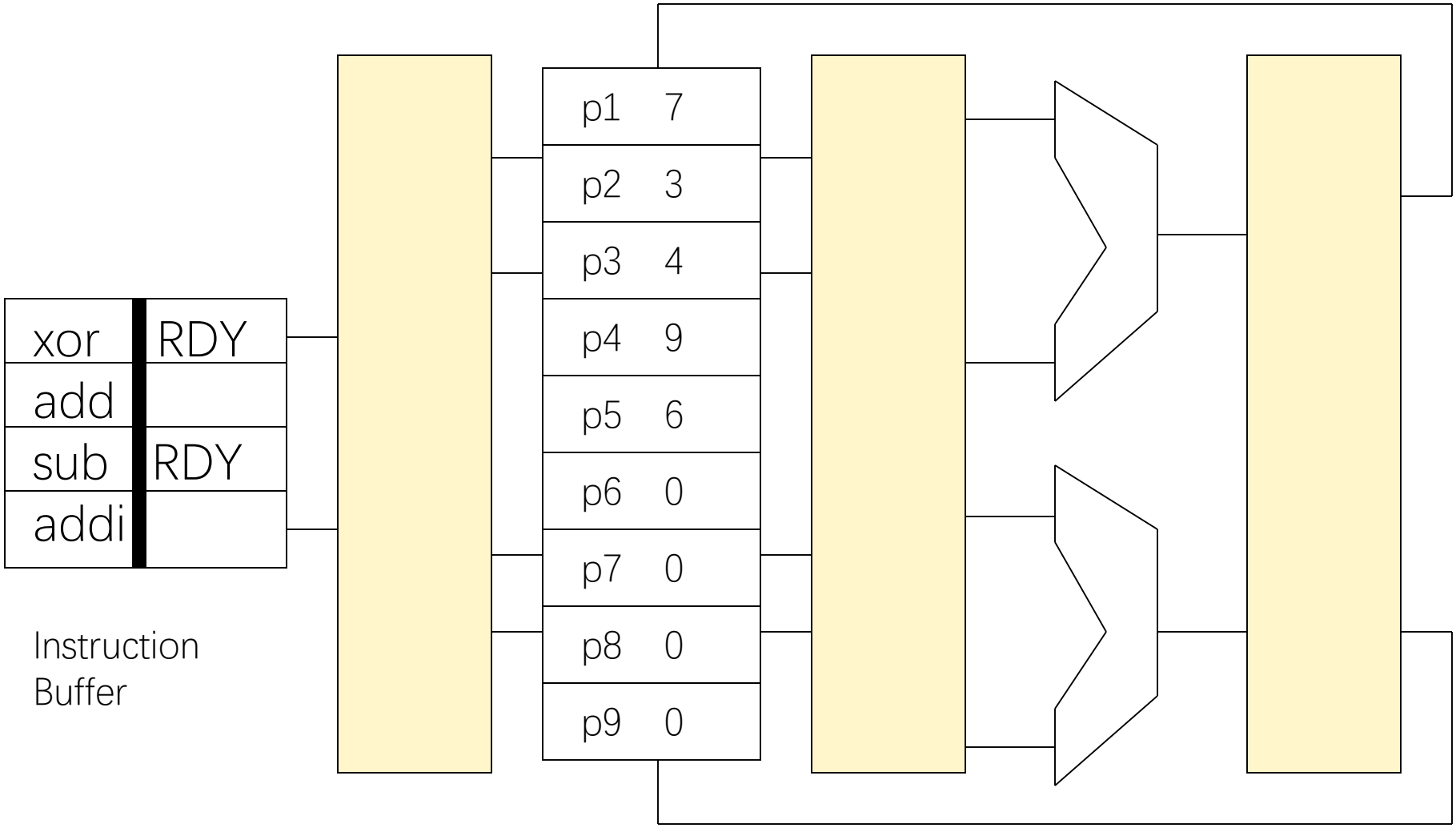
Time

P2	P3	P4	P5	P6	P7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

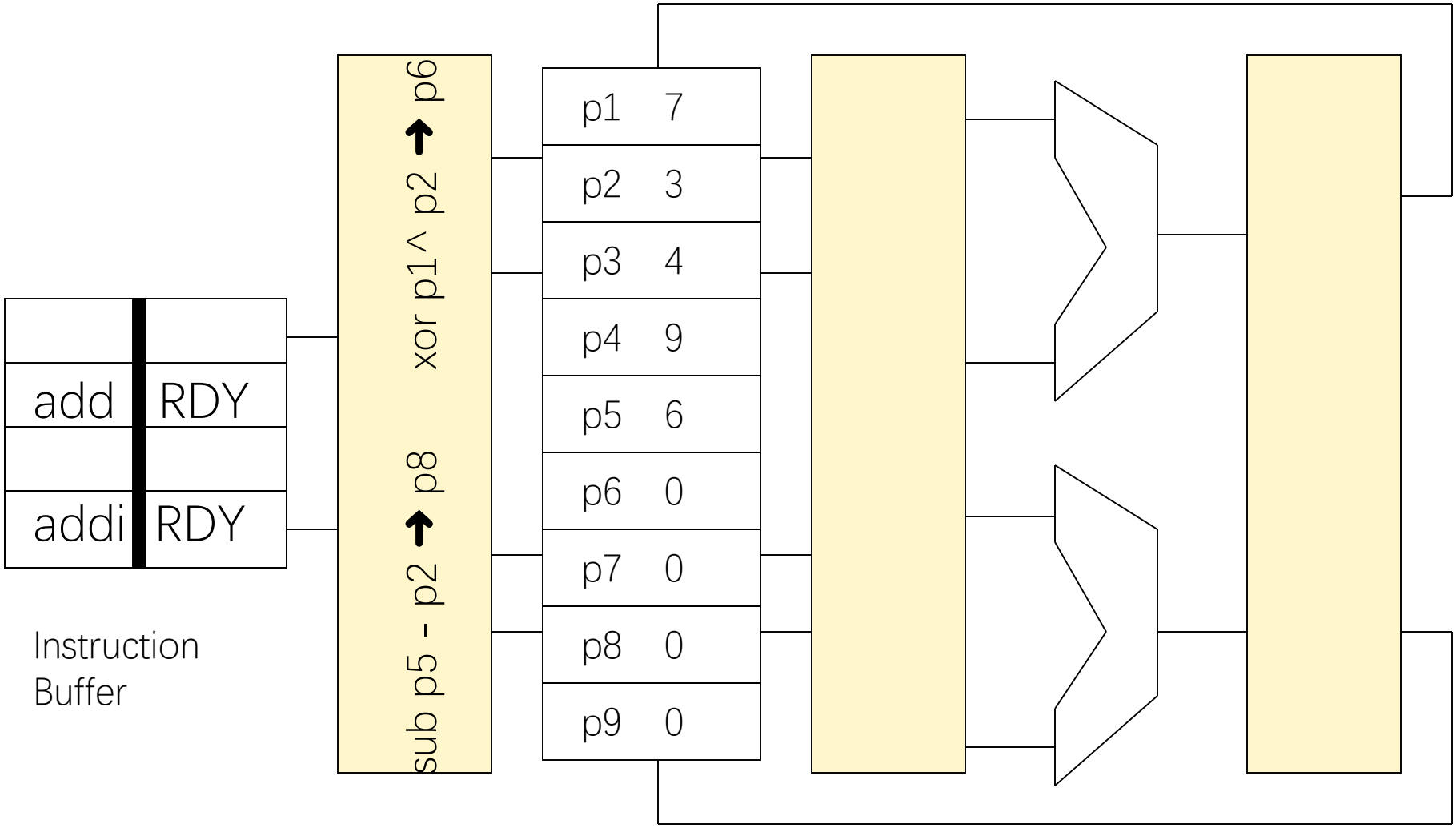
add p2,p3➜p4
sub p2,p4➜p5 and div p4,4➜p7
mul p2,p5➜p6

- 指令在 fetch/decoded/renamed 后进入 *Instruction Buffer*
  - 也叫“指令窗口”（“instruction window”）或“指令调度器”
- 每一个周期，都会重新检查指令的 ready bits
  - Execute oldest “ready” instruction, set output as “ready”

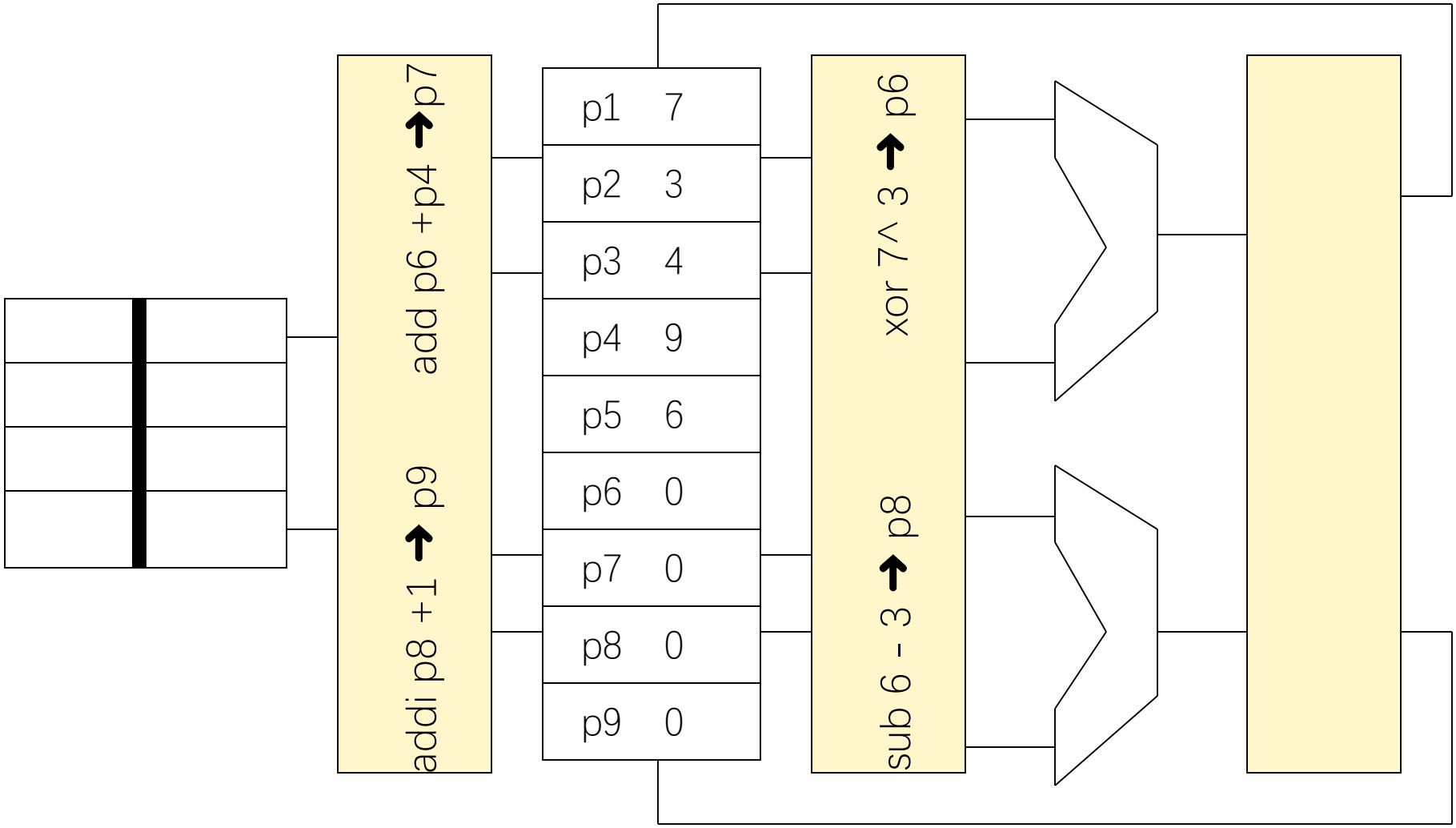
Step#3 OOO execution (2-wide)



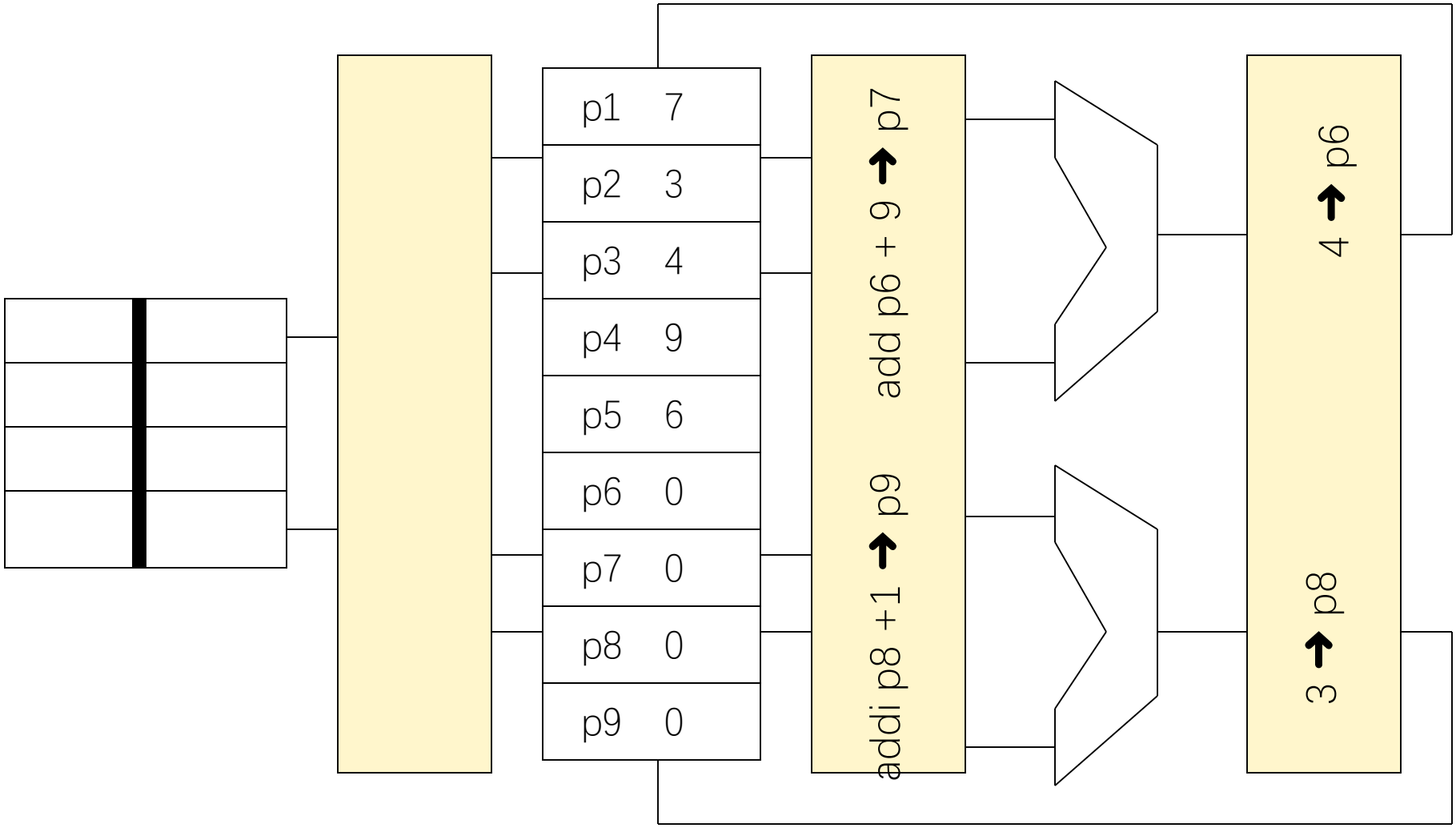
OOO execution (2-wide)



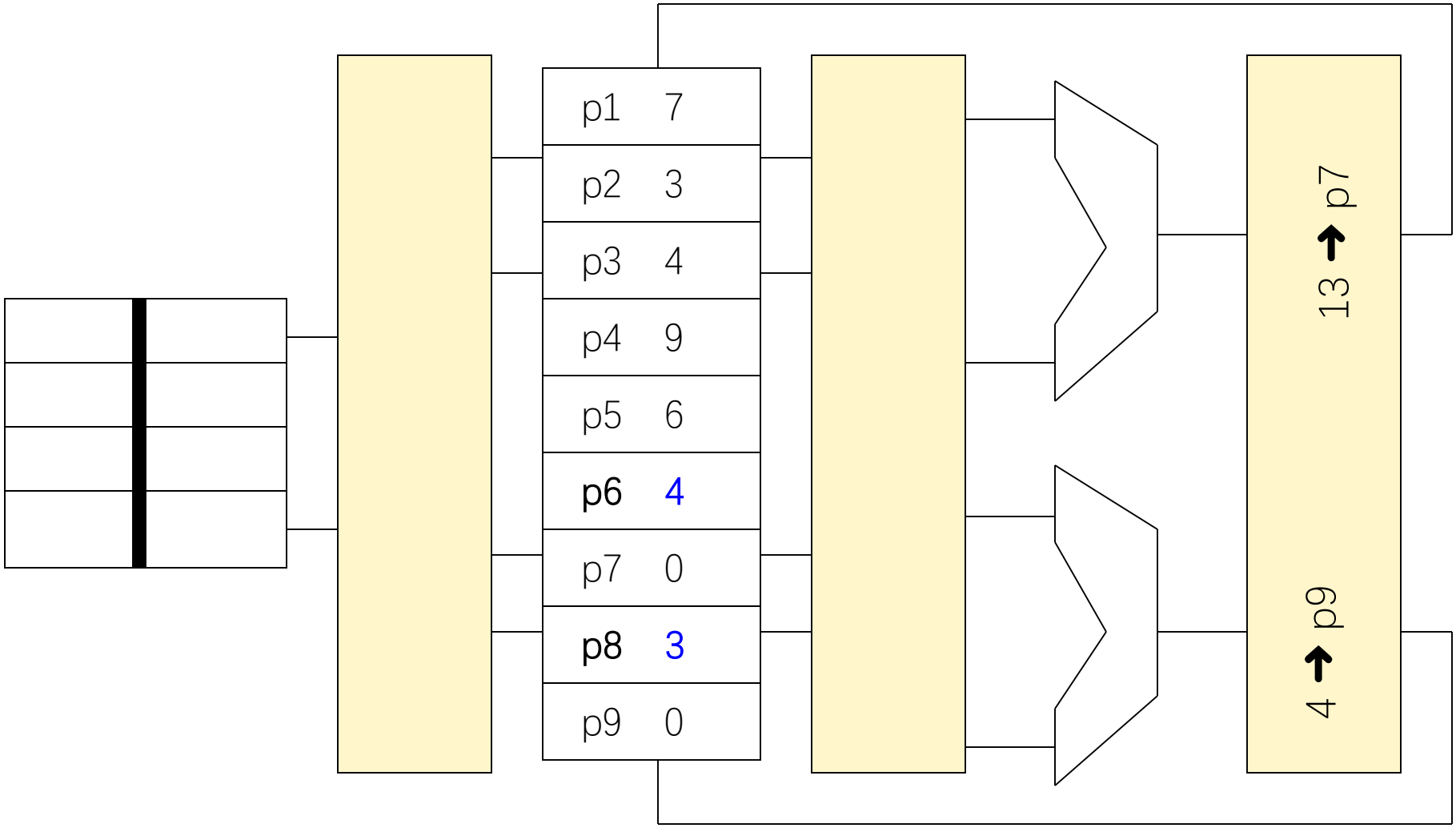
OOO execution (2-wide)



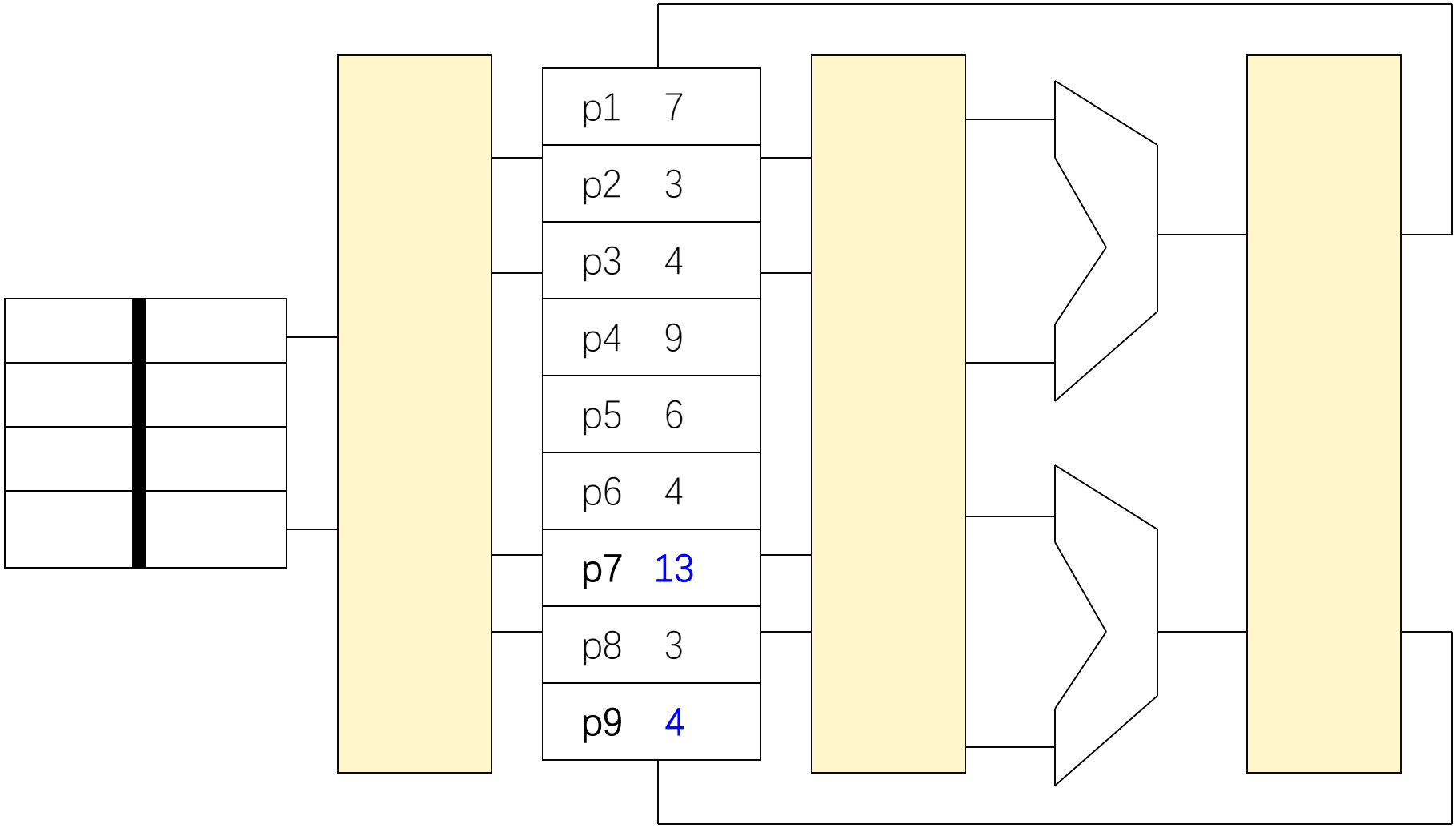
OOO execution (2-wide)



OOO execution (2-wide)



OOO execution (2-wide)



Out-of-Order “dynamic scheduling”

	0	1	2	3	4	5	6	7	8	9	10	11	12
Ld [p1] ➜ p2	F	Di	I	RR	X	M_1	M_2	W	C				
add p2 + p3 ➜ p4	F	Di				I	R R	X	W	C			
xor p4 ^ p5 ➜ p6		F	Di				I	RR	X	W	C		
ld [p7] ➜ p8		F	Di	I	RR	X	M_1	M_2	W		C		

- 硬件实现“动态调度”
- 双发射、乱序
- 一旦相关性消除，就可以发射
- 流水线变长
  - In-order front end: Fetch（F），“Dispatch”（D）
  - Out-of-order execution core:
    - “**Issue**”（I），“**RegisterRead**”（RR），Execute（X），Memory（M），Writeback（W）
  - In-order retirement: “**Commit**”（C）

OoO Execution is all around us

- 手机处理器: Qualcomm （高通）Krait 400
- based on ARM Cortex A15 processor
  - out-of-order 2.5GHz quad-core
  - 3-wide fetch/decode
  - 4-wide issue
  - 11-stage integer pipeline
  - 28nm process technology
  - 4KB DM L1$, 16KB 4-way SA L2$, 2MB 8-way SA L3$

Out of Order: 优势

- 允许猜测执行、乱序执行
  - Loads / stores
  - Branch prediction to look past branches
- 由硬件完成指令调度，更灵活
  - Compiler may want different schedule for different hw configs
  - Hardware has only its own configuration to deal with
  - Schedule can change due to cache misses
- Memory-level parallelism
  - 发生 cache misses时，可以寻找不相关的指令继续执行
  - reducing memory latency
    - Especially good at hiding L2 hits (~12 cycles in Core i7)

Challenges for Out-of-Order Cores

- 设计的复杂性
  - More complicated than in-order? Certainly!
  - But, we have managed to overcome the design complexity
- 时钟频率 Clock frequency
  - Can we build a “high ILP” machine at high clock frequency?
  - Yep, with some additional pipe stages, clever design
- 受限于指令窗口和ILP
  - Large physical register file （大的物理寄存器文件）
  - Fast register renaming/wakeup/select/load queue/store queue
    - Active areas of micro-architectural research
  - Branch & memory depend. prediction (limits effective window size)
    - 95% branch mis-prediction: 1 in 20 branches, or 1 in 100 insn.
  - Plus all the issues of building “wide” in-order superscalar
- 维持低功耗 （ Power efficiency ）
  - Today, even mobile phone chips are out-of-order cores

CISC vs RISC vs SS vs VLIW

	CISC	RISC	Superscalar	VLIW
Instr size	variable size	fixed size	fixed size	fixed size (but large)
Instr format	variable format	fixed format	fixed format	fixed format
Registers	few, some special Limited # of ports	Many GP Limited # of ports	GP and rename (RUU) Many ports	many, many GP Many ports
Memory reference	embedded in many instr's	load/store	load/store	load/store
Key Issues	decode complexity	data forwarding, hazards	hardware dependency resolution	(compiler) code scheduling

超标量处理器流水线的变化

	Year	Clock Rate	# Pipe Stages	Issue Width	OOO?	Cores/ Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	2930 MHz	14	4	Yes	2	75 W
Sun USPARC III	2003	1950 MHz	14	4	No	1	90 W
Sun T1 (Niagara)	2005	1200 MHz	6	1	No	8	70 W

Summary:Contemporary Processor Micro-Architectures

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Cortex-A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 3.15 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

总结回顾

- Instruction Level Parallism（指令级并行性: ILP ）
- Multiple-Issue (多发射)
- 静态多发射 = Very long instruction word (超长指令字)
- 动态多发射 = Superscalar (超标量)
  - 按序执行
  - Out of order execution (乱序执行)
    - 按序取指
    - Branch prediction (分支预测)
    - Speculation（猜测执行）
    - Register renaming (寄存器换名)
    - 发射（issue）
    - 指令窗口（instruction window, instruction buffer）
    - In order commit （按序提交）

谢谢！