



# 《计算机系统结构》课程直播

## 2020. 4.21

请将ZOOM名称改为“姓名”；

听不到声音请及时调试声音设备，可以下课后补签到

# 本节内容

---

- 1 静态多发射
- 2 动态多发射（超标量）
- 3 乱序超标量

- From : H&P Computer Architecture: A Quantitative Approach, Fifth Edition, (5th edition)

# 多发射:开发指令级并行 (ILP)

$$D = 3 * (a - b) + 7 * a * c ;$$

*ld a*

*ld b*

*sub a-b*

*mul 3\*(a-b)*

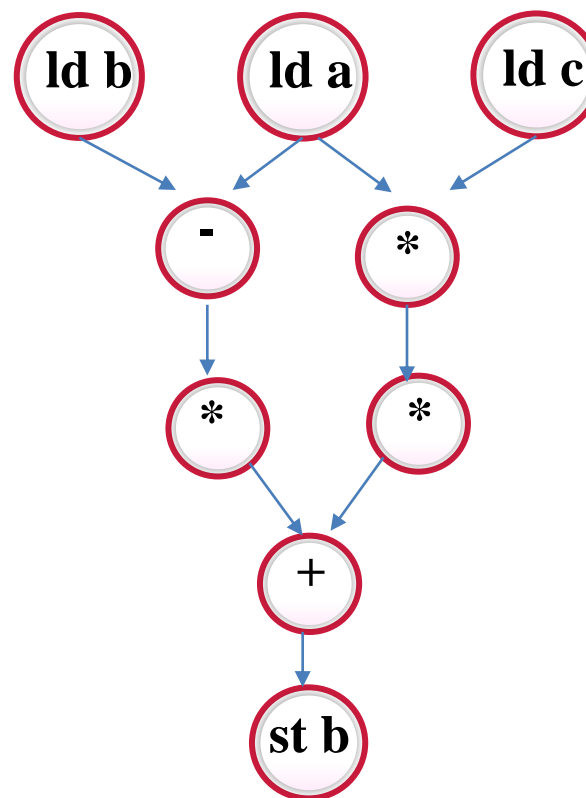
*ld c*

*mul a\*c*

*mul 7\*a\*c*

*add 3(a-b) + 7\*a\*c*

*st d*



# 现代处理器的微结构：超标量

| 处理器                           | 年份   | 时钟频率     | 流水级数 | 发射宽度 | 乱序执行? | 核数 | 功耗    |
|-------------------------------|------|----------|------|------|-------|----|-------|
| 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<br>Willamette | 2001 | 2000 MHz | 22   | 3    | Yes   | 1  | 75 W  |
| Intel Pentium 4<br>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  |

# 提问？

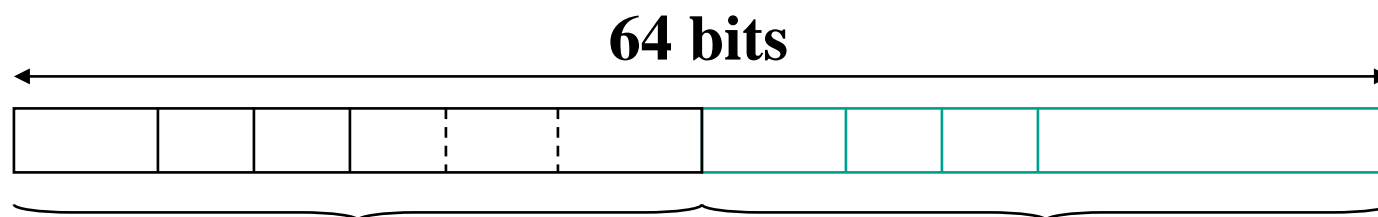
---

- 超流水线是通过细化流水、提高主频，使得可以在相同时间内完成更多个操作，
- 超标量结构 `superscalar` 的处理器每个时钟周期并行发射和执行多条指令
- 超长指令字 `VLIW` 的处理器每个时钟周期也并行发射和执行多条指令
- `Superscalar` 与 `VLIW` 区别在哪里？

# 多发射处理器的实现和主要特点

| 常用名                                     | 发射结构  | 冲突检测    | 调度方式   | 特点                 | 处理器举例                                 |
|---|-------|---------|--------|--------------------|---------------------------------------|
| 静态超标量<br>superscalar<br>(static)        | 动态    | 硬件      | 静态     | 按序执行               | 大部分嵌入式处理器，例如ARM cortex-A8             |
| 动态超标量<br>superscalar<br>(dynamic)       | 动态    | 硬件      | 动态     | 乱序执行               | 目前无                                   |
| 推测执行超标量<br>superscalar<br>(speculative) | 动态    | 硬件      | 带推测的动态 | 乱序、推测执行            | 大部分通用处理器，如Intel Core i3,i5,i7、arm A76 |
| <b>超长指令字</b><br>(VLIW)                  | 静态    | 主要由软件完成 | 静态     | 编译器（隐式）完成冲突检测、指令调度 | 某些特定领域,如信号处理器 <b>TI C6x</b>           |
| <b>显式并发指令运算</b> (EPIC)                  | 主要为静态 | 主要由软件完成 | 主要为静态  | 编译器（显式）完成冲突检测、指令调度 | <b>Intel 安腾 Itanium</b> 处理器           |

# 举例：一个 VLIW MIPS



ALU (R 型) 或 Branch (I 型)

Load 或 Store (I 型)

- ❑ 双发射的 MIPS 处理器，两条指令组成一个指令束
- ❑ 指令束中的指令成对取指、译码和发射
- ❑ 由编译器安排指令束，选取每次同时发射的两条指令
- ❑ 如果找不到合适的指令，就用空指令noop代替



# 代码调度举例

以下代码 lp:    lw     \$t0,0(\$s1)     # \$t0=array element  
                 addu   \$t0,\$t0,\$s2   # add scalar in \$s2  
                 sw     \$t0,0(\$s1)   # store result  
                 addi   \$s1,\$s1,-4   # decrement pointer  
                 bne    \$s1,\$0,lp    # branch if \$s1 != 0

- 假设： 总能正确预测转移指令的转移方向
- 编译器：
  - 将两条指令打包成为一束长指令
  - 在一束长指令上的两条指令必须无关
  - Load-use 指令必须间隔一周期



# 代码调度举例

以下代码 lp:   lw     \$t0,0(\$s1)     # \$t0=array element  
                  addu   \$t0,\$t0,\$s2   # add scalar in \$s2  
                  sw     \$t0,0(\$s1)     # store result  
                  addi   \$s1,\$s1,-4     # decrement pointer  
                  bne    \$s1,\$0,lp     # branch if \$s1 != 0

|     | ALU 或 branch                 | 数据传送                     | 时钟周期 |
|-----|------------------------------|--------------------------|------|
| lp: |                              | <u>lw   \$t0,0(\$s1)</u> | 1    |
|     | <u>addi   \$s1,\$s1,-4</u>   |                          | 2    |
|     | <u>addu   \$t0,\$t0,\$s2</u> |                          | 3    |
|     | <u>bne    \$s1,\$0,lp</u>    | <u>sw   \$t0,4(\$s1)</u> | 4    |

- 5 条指令花费4个周期, CPI= 0.8 (最好情况下是0.5)

# 代码调度举例

lp:

```
lw    $t0, 0($s1)
addu  $t0, $t0, $s2
sw     $t0, 0($s1)
addi   $s1, $s1, -4
bne    $s1, $0, lp
```

- 指令条数增加了: 增加指令级并行性 (ILP)
- 转移指令减少了: 降低转移指令引起的开销
- 寄存器换名: 消除由于寄存器名字引起的相关性

lp: lw

**\$t0**, 0(\$s1)

addu \$t0, \$t0, \$s2

sw \$t0, 0(\$s1)

lw **\$t1**, -4(\$s1)

addu \$t1, \$t1, \$s2

sw \$t1, -4(\$s1)

lw **\$t2**, -8(\$s1)

addu \$t2, \$t2, \$s2

sw \$t2, -8(\$s1)

lw **\$t3**, -12(\$s1)

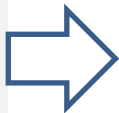
addu \$t3, \$t3, \$s2

sw \$t3, -12(\$s1)

addi **\$s1**, \$s1, **-16**

bne **\$s1**, \$0, lp

- 循环
- 展开





```
lp:      lw      $t0, 0($s1)
        addu    $t0, $t0, $s2
        sw      $t0, 0($s1)
        lw      $t1, -4($s1)
        addu    $t1, $t1, $s2
        sw      $t1, -4($s1)
        lw      $t2, -8($s1)
        addu    $t2, $t2, $s2
        sw      $t2, -8($s1)
        lw      $t3, -12($s1)
        addu    $t3, $t3, $s2
        sw      $t3, -12($s1)
        addi    $s1, $s1, -16
        bne    $s1, $0, lp
```



```
lp:      lw      $t0, 0($s1)
        lw      $t1, -4($s1)
        lw      $t2, -8($s1)
        lw      $t3, -12($s1)
        addu    $t0, $t0, $s2
        addu    $t1, $t1, $s2
        addu    $t2, $t2, $s2
        addu    $t3, $t3, $s2
        sw      $t0, 0($s1)
        sw      $t1, -4($s1)
        sw      $t2, -8($s1)
        sw      $t3, -12($s1)
        addi    $s1, $s1, -16
        bne    $s1, $0, lp
```



```
lp:  lw    $t0,0($s1)
      lw    $t1,-4($s1)
      lw    $t2,-8($s1)
      lw    $t3,-12($s1)
      addu  $t0,$t0,$s2
      addu  $t1,$t1,$s2
      addu  $t2,$t2,$s2
      addu  $t3,$t3,$s2
      sw    $t0,0($s1)
      sw    $t1,-4($s1)
      sw    $t2,-8($s1)
      sw    $t3,-12($s1)
      addi  $s1,$s1,-16
      bne   $s1,$0,lp
```



| ALU or branch       | Data transfer    | CC |
|---------------------|------------------|----|
| 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  |

- 14 条指令8个周期,
- CPI =0.57 (最佳情况是0.5)

# 提问

---

- VL IW的优缺点
  - 存在的问题？
  - 优点？



# VLIW的局限性

## 1 编译复杂、编译时间长

- 循环展开、冲突检测、指令调度
- 将if then else 结构转化为可预测得转移指令
- 存储器访问地址预测

## 2 代码膨胀

- 空指令浪费内存存储空间
- 循环展开后，也需要更多存储空间
- 需要更大内存带宽

## 3 锁步 (lock step) 机制

- 一条指令阻塞，其后所有指令都阻塞
- 相关性解除了，才允许发射有相关的指令
- 流水线段数越多，相关性越多

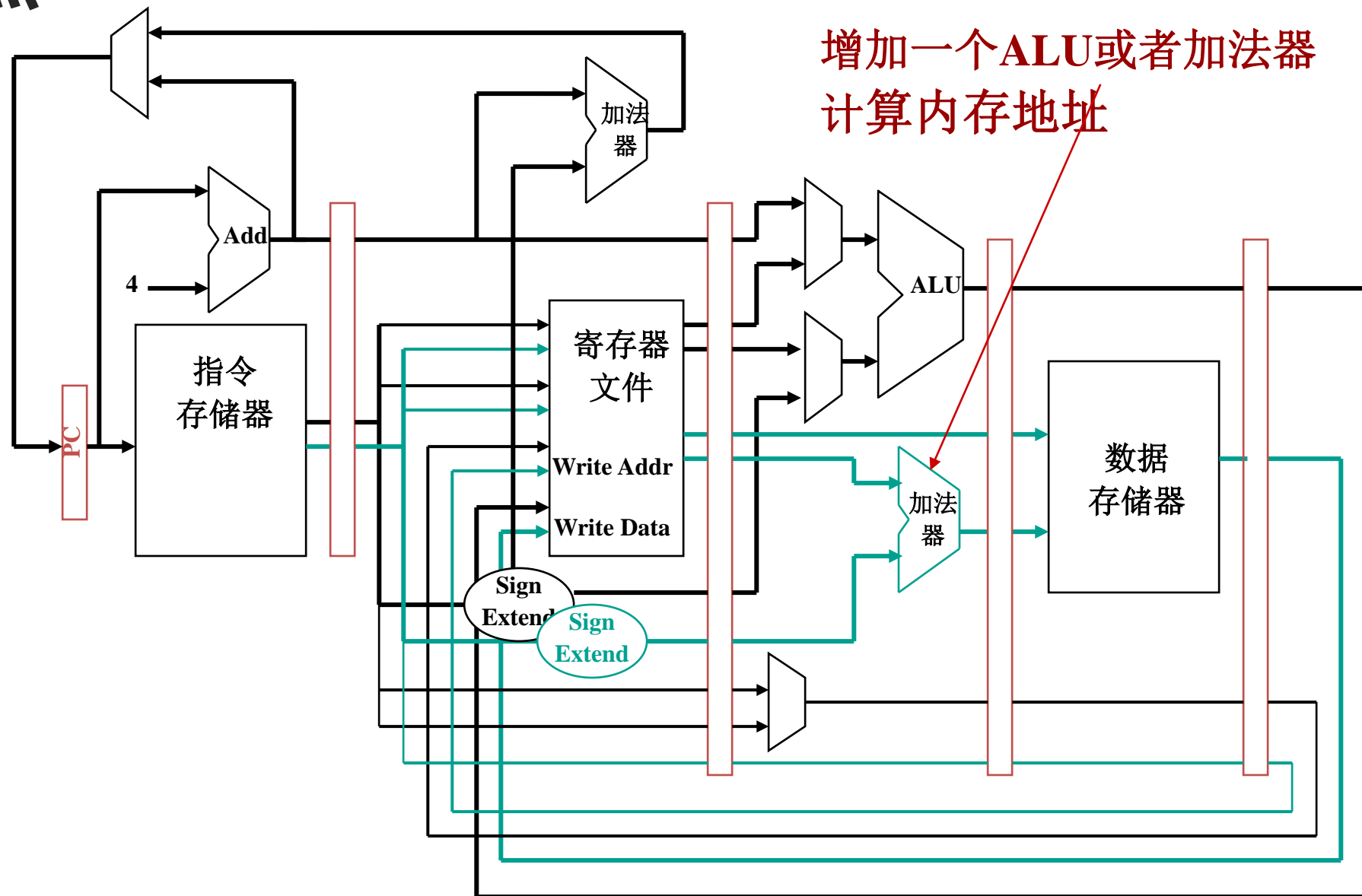
## 4 目标代码不兼容

- 市场接受意愿低



# VLIW的优点

- 硬件简单
- 成本低
- 能耗少





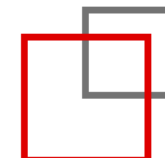


# 静态多发射处理器的现状

| 常用名                                     | 发射结构  | 冲突检测    | 调度方式   | 特点                 | 处理器举例                         |
|---|-------|---------|--------|--------------------|-------------------------------|
| 静态超标量<br>superscalar<br>(static)        | 动态    | 硬件      | 静态     | 按序执行               | 大部分嵌入式处理器，例如ARM cortex-A8     |
| 动态超标量<br>superscalar<br>(dynamic)       | 动态    | 硬件      | 动态     | 乱序执行               | 目前无                           |
| 推测执行超标量<br>superscalar<br>(speculative) | 动态    | 硬件      | 带推测的动态 | 乱序、推测执行            | 大部分通用处理器，如Intel Core i3,i5,i7 |
| <b>超长指令字</b><br>(VLIW)                  | 静态    | 主要由软件完成 | 静态     | 编译器（隐式）完成冲突检测、指令调度 | 某些特定领域,如信号处理器 <b>TI C6x</b>   |
| <b>显式并发指令运算</b> (EPIC)                  | 主要为静态 | 主要由软件完成 | 主要为静态  | 编译器（显式）完成冲突检测、指令调度 | <b>Intel 安腾 Itanium</b> 处理器   |



# 动态多发射超标量结构 (superscalar)



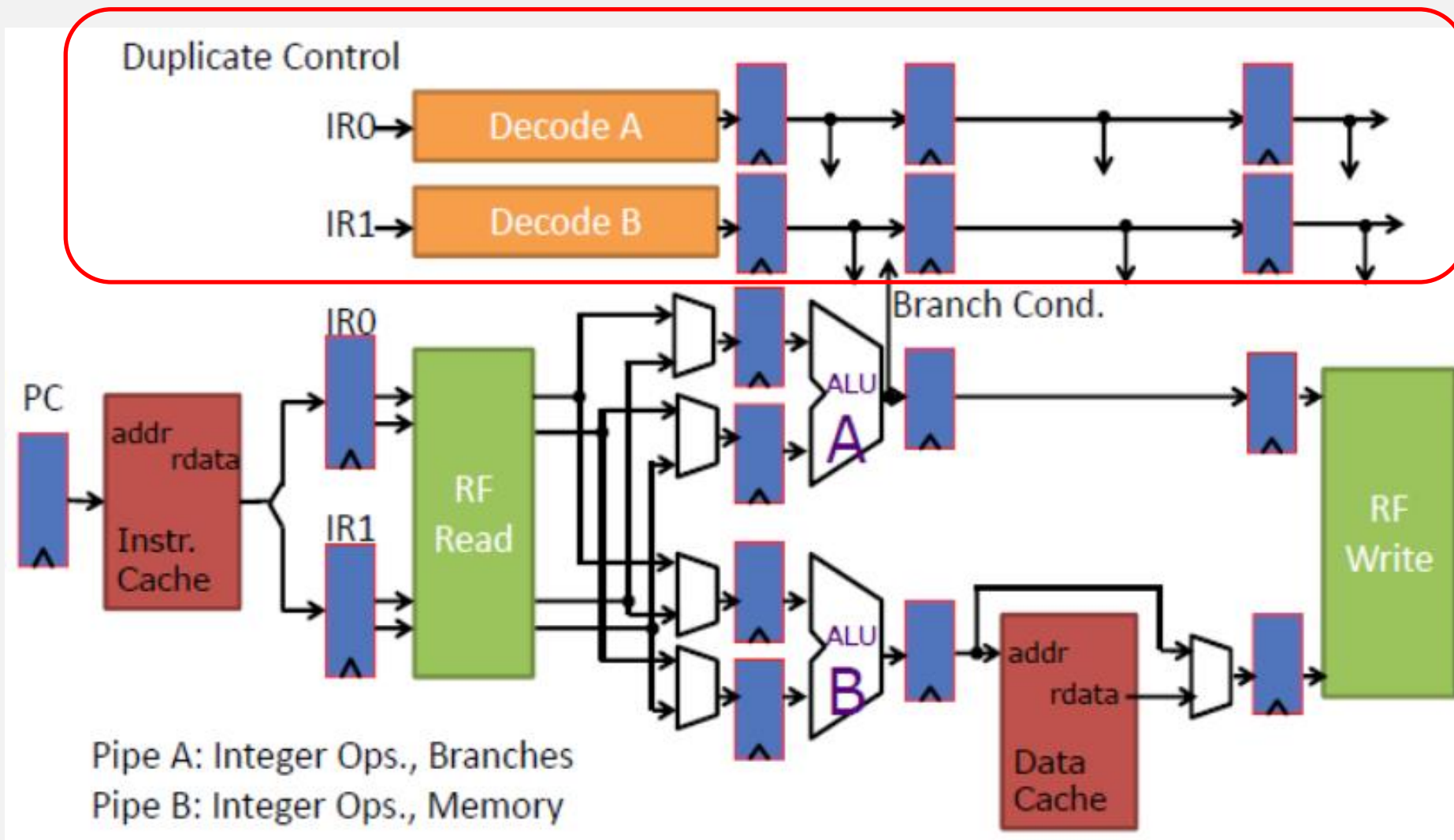


# 多发射处理器的分类

| 常用名                                     | 发射结构  | 冲突检测    | 调度方式   | 特点                   | 处理器举例                          |
|---|-------|---------|--------|----------------------|--------------------------------|
| 静态超标量<br>superscalar<br>(static)        | 动态    | 硬件      | 静态     | 按序执行                 | 大部分嵌入式处理器, 例如ARM cortex-A8     |
| 动态超标量<br>superscalar<br>(dynamic)       | 动态    | 硬件      | 动态     | 乱序执行                 | 目前无                            |
| 推测执行超标量<br>superscalar<br>(speculative) | 动态    | 硬件      | 带推测的动态 | 乱序、推测执行              | 大部分通用处理器, 如Intel Core i3,i5,i7 |
| 超长指令字<br>(VLIW)                         | 静态    | 主要由软件完成 | 静态     | 编译器 (隐式) 完成冲突检测、指令调度 | 某些特定领域,如信号处理器 TI C6x           |
| 显式并发指令运算(EPIC)                          | 主要为静态 | 主要由软件完成 | 主要为静态  | 编译器 (显式) 完成冲突检测、指令调度 | Intel 安腾 Itanium处理器            |



# 一个双发射超标量MIPS处理器



# 动态多发射：流水线时空图

|     |   |   |    |    |    |    |   |
|-----|---|---|----|----|----|----|---|
| 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 |

|       |   |   |    |    |    |    |    |   |
|-------|---|---|----|----|----|----|----|---|
| ADDIU | F | D | A0 | A1 | W  |    |    |   |
| LW    | F | D | B0 | B1 | W  |    |    |   |
| LW    |   | F | D  | B0 | B1 | W  |    |   |
| ADDIU |   | F | D  | A0 | A1 | W  |    |   |
| LW    |   |   | F  | D  | B0 | B1 | W  |   |
| LW    |   |   | F  | D  | D  | B0 | B1 | W |

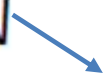
- 每次取指、译码、执行两条指令
- 理想情况：CPI=0.5

- 指令发射逻辑
  - 能调换指令所进入的功能部件
  - 能检测结构冒险、数据冒险等等

# 动态多发射：流水线时空图


No Bypassing: •无前向通路

|               |   |   |    |    |    |   |
|---------------|---|---|----|----|----|---|
| 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 |



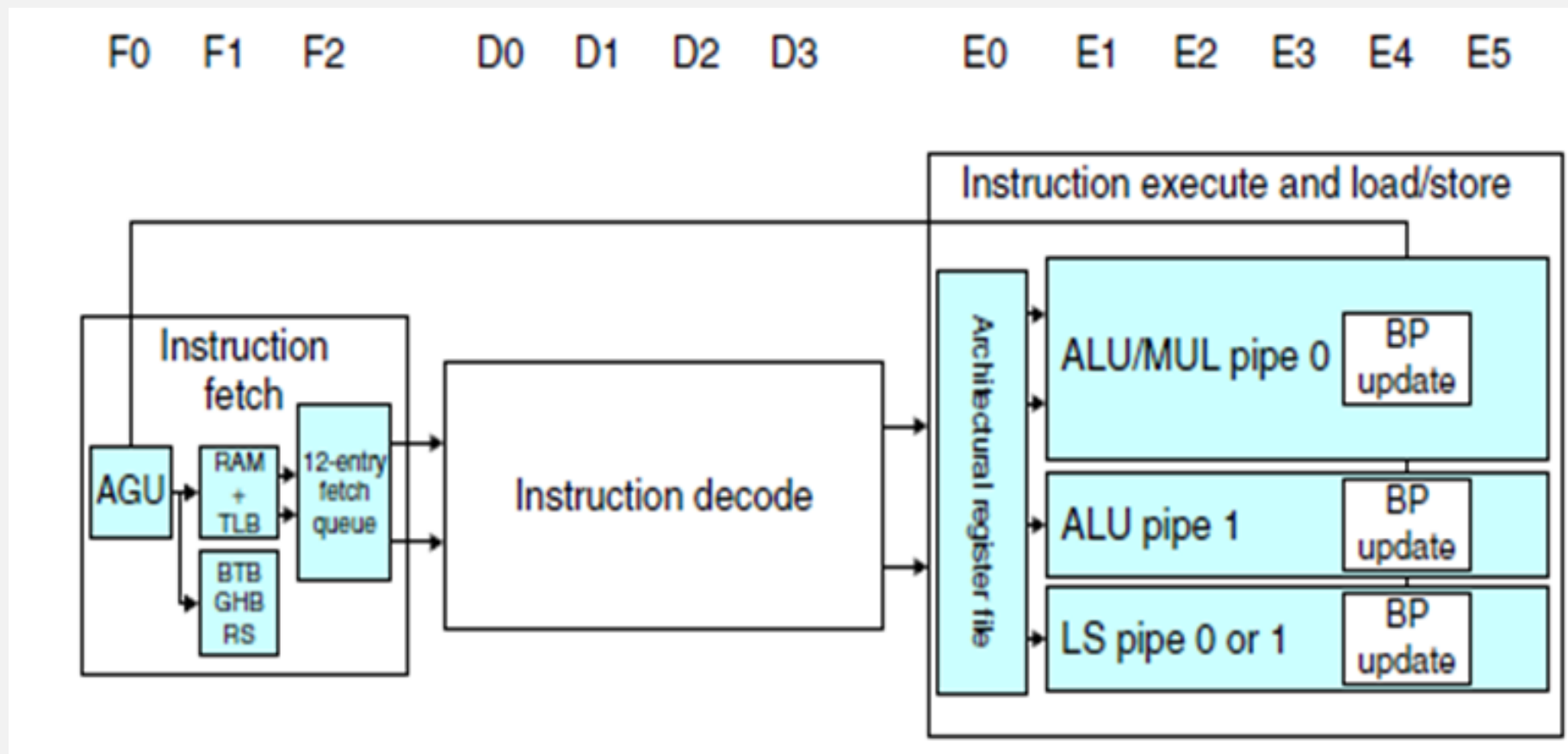
Full Bypassing: •有前向通路

|               |   |   |    |    |    |    |
|---------------|---|---|----|----|----|----|
| 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 |
|               |   |   |    |    | A0 | A1 |





# 按序超标量实例：ARM Cortex-A8





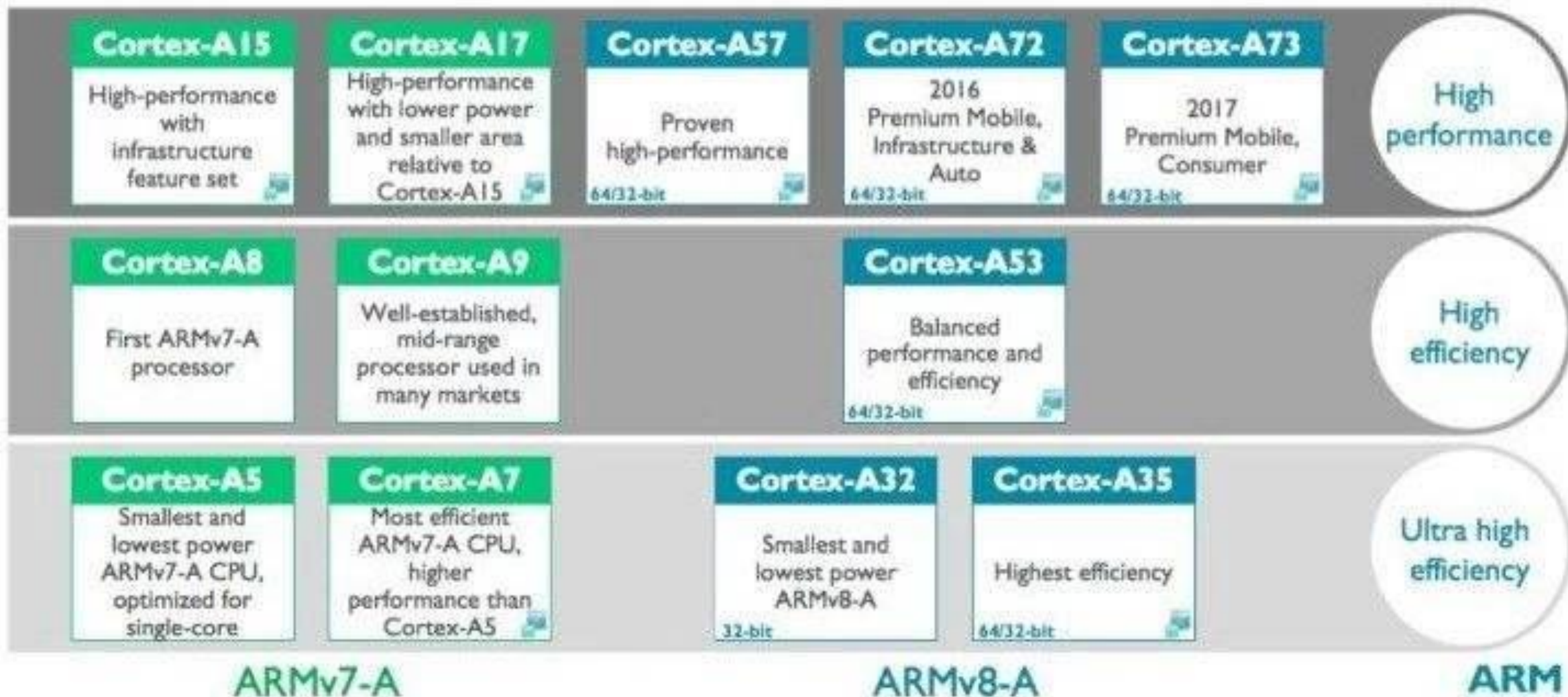


# A8流水线处理器的特点

- 静态调度、动态发射、按序超标量
- 静态调度，编译器尽力做：
  - 避免两条相邻指令使用同一部件（结构冒险）
  - 避免相邻的指令有数据依赖关系（数据相关）
- 动态发射结构：
  - 每个周期由控制逻辑判断是发射一条还两条指令，还是不发射指令；
  - **结构冒险**：编译器实在无法避免时，能检测出冒险，一次只发送一条；
  - **数据冒险**：编译器实在无法避免时，能检测出冒险，如果检测到冒险，要么把两条都停顿、要么停顿两条中的一条；
  - **控制冒险**：当转移预测错误时、清空流水线，从正确位置重新开始执行。



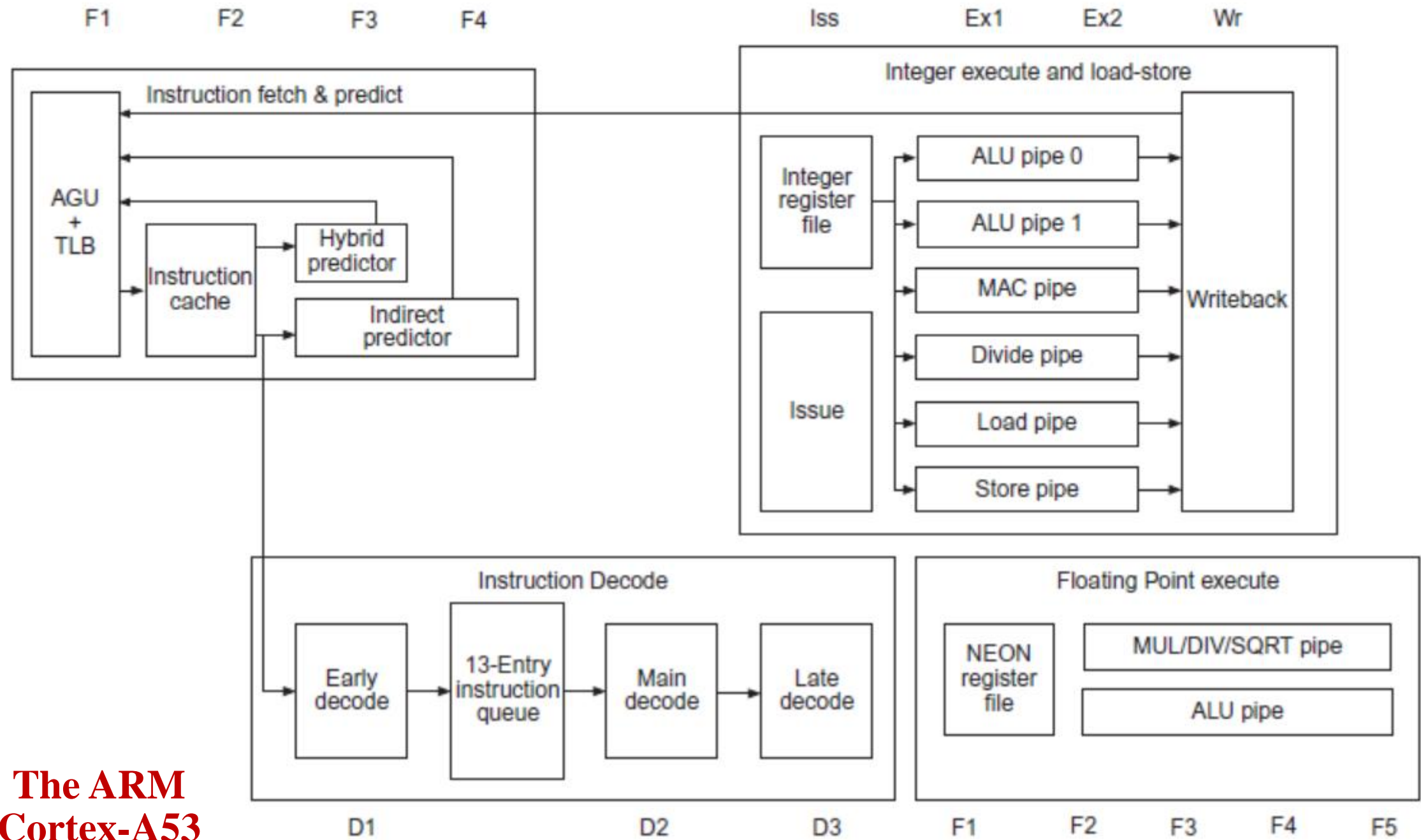
# ARM Cortex-A系列



# Arm 处理器微架构

---

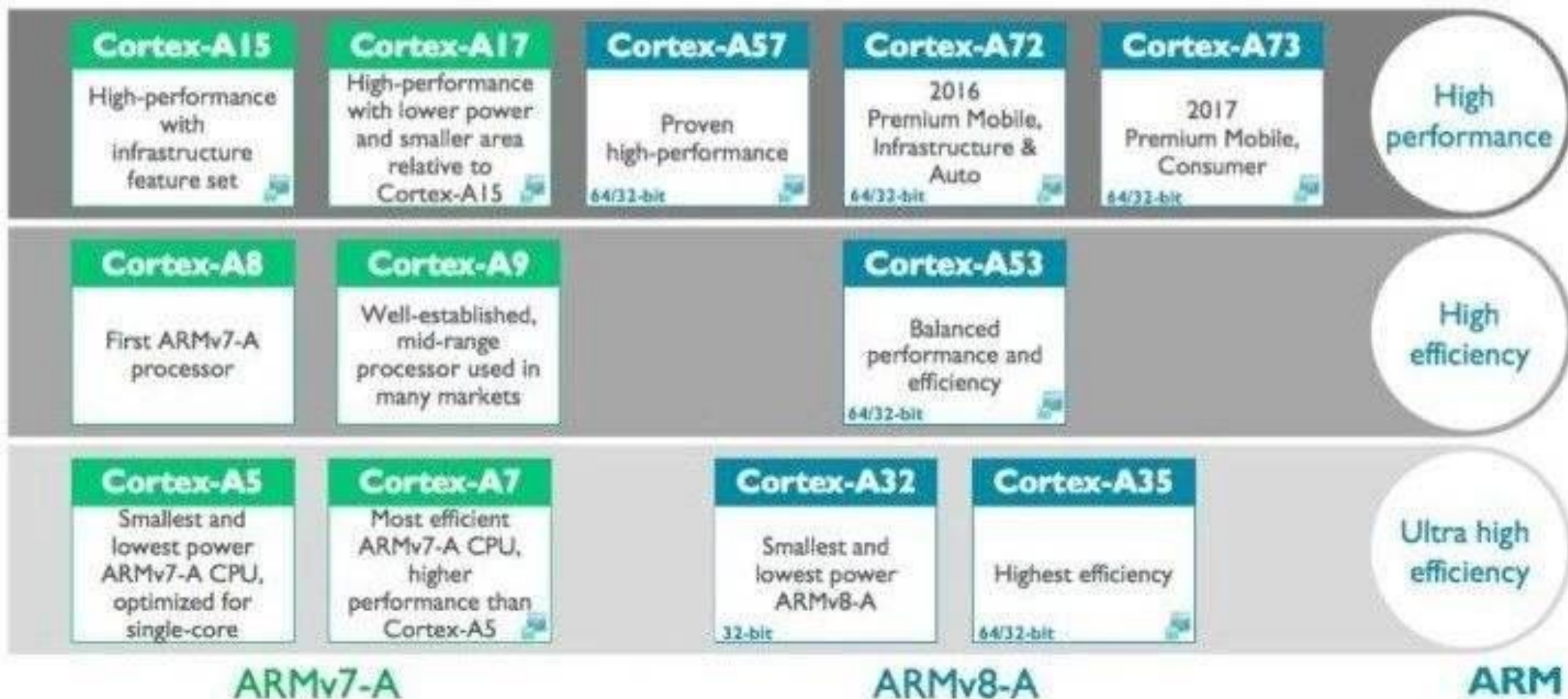
| Announced<br>(64-bit) |            | Announced<br>(32-bit) |            |
|-----------------------|------------|-----------------------|------------|
| Year                  | Core       | Year                  | Core       |
| 2012                  | Cortex-A53 | 2005                  | Cortex-A8  |
| 2012                  | Cortex-A57 | 2007                  | Cortex-A9  |
| 2015                  | Cortex-A72 | 2009                  | Cortex-A5  |
| 2015                  | Cortex-A35 | 2010                  | Cortex-A15 |
| 2016                  | Cortex-A73 | 2011                  | Cortex-A7  |
| 2017                  | Cortex-A55 | 2013                  | Cortex-A12 |
| 2017                  | Cortex-A75 | 2014                  | Cortex-A17 |
| 2018                  | Cortex-A76 | 2016                  | Cortex-A32 |



# The ARM Cortex-A53



# ARM Cortex-A系列





|       | 麒麟990 5G  | 麒麟990   |
|-------|---|---|
|       |    |  |
| 工艺    | 7nm+EUV   | 7nm   |
| CPU   | 2X Cortex-A76 Based@2.86GHz<br>2X Cortex-A76 Based@2.36GHz<br>4X Cortex-A55@1.95GHz | 2X Cortex-A76 Based@2.86GHz<br>2X Cortex-A76 Based@2.09GHz<br>4X Cortex-A55@1.86GHz |
| GPU   | 16 Core Mali-G76  | 16 Core Mali-G76  |
| NPU   | 2 Big Core +1 Tiny Core   | 1 Big Core +1 Tiny Core   |
| 存储    | UFS 3.0, UFS2.1   | UFS 3.0, UFS2.1   |
| Modem | 2G/3G/4G/5G   | 2G/3G/4G  |

## 华为MATE30手机中的处理器

### CPU:

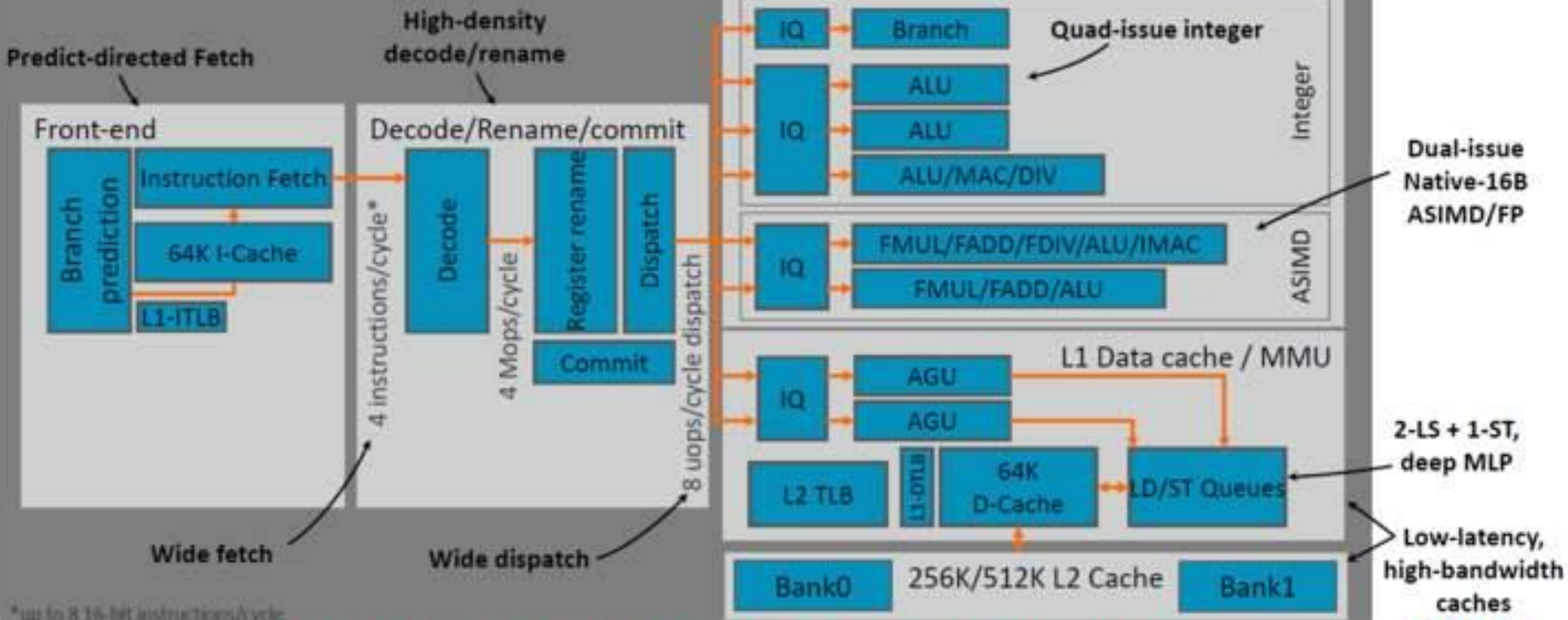
- 2大核+2中核+4小核能效架构
- 2×A76(2.86Ghz)、  
2×A76(2.36Ghz)、  
4×A55(1.95Ghz)
- 麒麟990对标骁龙855+的性能



# Cortex-A76: Microarchitecture overview

The foundation of a new family of high-performance products

## Cortex-A76



\*up to 8 16-bit instructions/cycle

© 2018 Arm Limited

The embargo for this content presented at Arm Tech Day will lift on Thursday, May 31<sup>st</sup> at 12pm Pacific Standard Time. Corresponding UK and Japan times are: Thursday, May 31<sup>st</sup> 8pm BST/Friday, June 1<sup>st</sup> at 4am JST

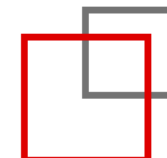
arm





# 乱序超标量、及其主要问题

(OOO superscalar)  
Out of Order



# 多发射处理器的实现和主要特点

| 常用名                                     | 发射结构  | 冲突检测    | 调度方式   | 特点                   | 处理器举例                          |
|---|-------|---------|--------|----------------------|--------------------------------|
| 静态超标量<br>superscalar<br>(static)        | 动态    | 硬件      | 静态     | 按序执行                 | 大部分嵌入式处理器, 例如ARM cortex-A8     |
| 动态超标量<br>superscalar<br>(dynamic)       | 动态    | 硬件      | 动态     | 乱序执行                 | 目前无                            |
| 推测执行超标量<br>superscalar<br>(speculative) | 动态    | 硬件      | 带推测的动态 | 乱序、推测执行              | 大部分通用处理器, 如Intel Core i3,i5,i7 |
| 超长指令字<br>(VLIW)                         | 静态    | 主要由软件完成 | 静态     | 编译器 (隐式) 完成冲突检测、指令调度 | 某些特定领域,如信号处理器 TI C6x           |
| 显式并发指令运算(EPIC)                          | 主要为静态 | 主要由软件完成 | 主要为静态  | 编译器 (显式) 完成冲突检测、指令调度 | Intel 安腾 Itanium处理器            |

# 双发射按序超标量

| •2                  | 0 | 1 | 2  | 3  | 4  | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---------------------|---|---|----|----|----|---|---|---|---|---|----|----|----|
| Ld [r1] → r2        | F | D | X  | M  | W  |   |   |   |   |   |    |    |    |
| add r2 + r3 → r4    | F | D | d* | d* | X  | M | W |   |   |   |    |    |    |
| xor r4 ^ r5 → r6    |   | F | D  | d* | d* | X | M | W |   |   |    |    |    |
| <u>ld [r7] → r8</u> |   | F | D  | p* | p* | X | M | W |   |   |    |    |    |

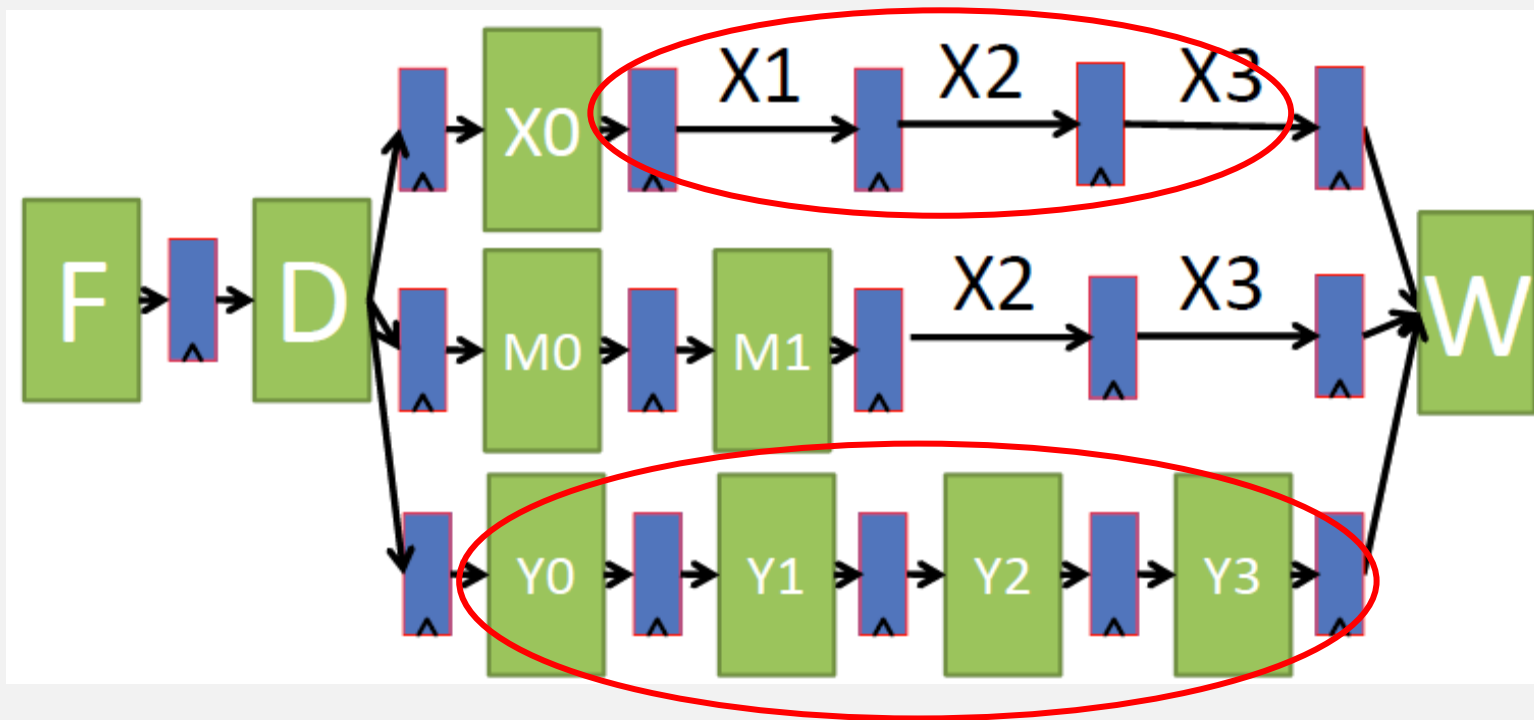
- 假设：load-use 相关性的开销为 (penalty) 2 个周期

# 双发射乱序超标量

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

- 假设：load-use 相关性的间隔1个周期

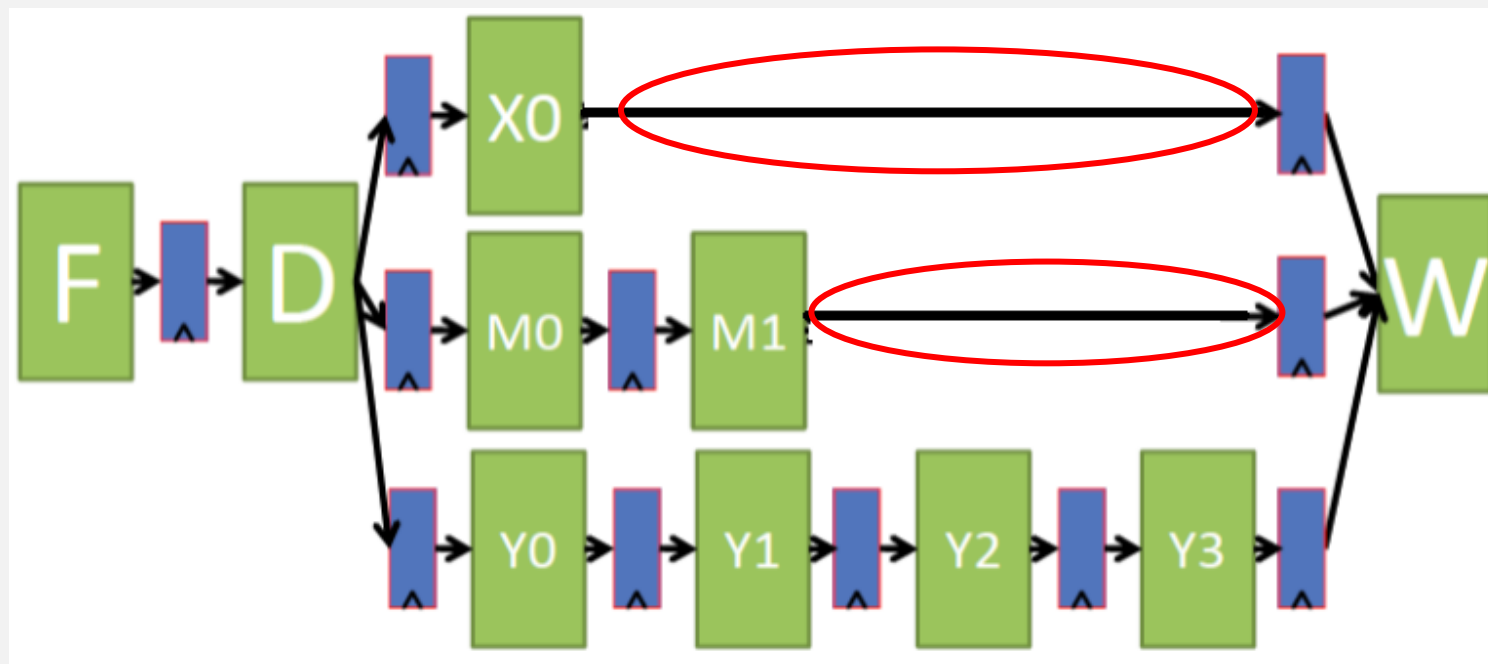
# 三发射按序执行



- 增加了流水段长度为4的乘法功能部件



# 三发射乱序执行



•快的部件可以先写结果

---

**问题1： 名字引起的假相关**



# 乱序执行导致的数据相关性

## •输出相关

```
LW      $t0, 0($s1)
ADDU    $t0, $t1, $s2
SUB      $t2, $t0, $s2
```

## •反相关

```
•DIVD   F0, F2, F4
ADDD     F10, F0, F8
SUBD     F8, F8, F14
```

## • 输出相关

- 如果 lw 晚于 addu 写 \$t0
- sub 会获得错误的 \$t0
- 又称为(Write after Write)WAW 相关

## • 反相关

- 如果 sub 写结果 早于 add 读 F8
- add 会获得错误的 F8
- 又称为(Write after Read) WAR 相关

# 消除“假”数据相关性

## •输出相关

LW      \$t0, 0(\$s1)

ADDU    \$t0, \$t1, \$s2

SUB      \$t2, \$t0, \$s2

LW      \$t0a, 0(\$s1)

ADDU    **\$t0b**, \$t1, \$s2

SUB      \$t2, **\$t0b**, \$s2

## •反相关

•DIVD   F0, F2, F4

ADDD    F10, F0, **F8**

SUBD    **F8**, F8, F14

•DIVD   F0, F2, F4

ADDD    F10, F0, **F8a**

SUBD    **F8b**, F8a, F14

# 检测“真”数据相关性

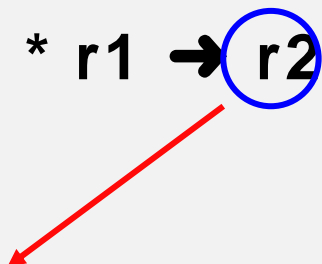
- 真相关:

- RAW (Read After Write)

mul r0 \* r1 → r2

...

add r2 + r3 → r4



# 课堂练习

---

- 1) LD R1, 0(R2) ; load R1 from address 0+R2
- 2) ADDI R1, R1, #1 ; R1=R1+1
- 3) SD R1, 0(R2) ; store R1 at address 0+R2
- 4) ADDI R2, R2, #4 ; R2= R2+4
- 5) SUB R4, R3, R2 ; R4=R3-R2
- 6) BNEZ R4, Loop ; branch to Loop if R4 !=0

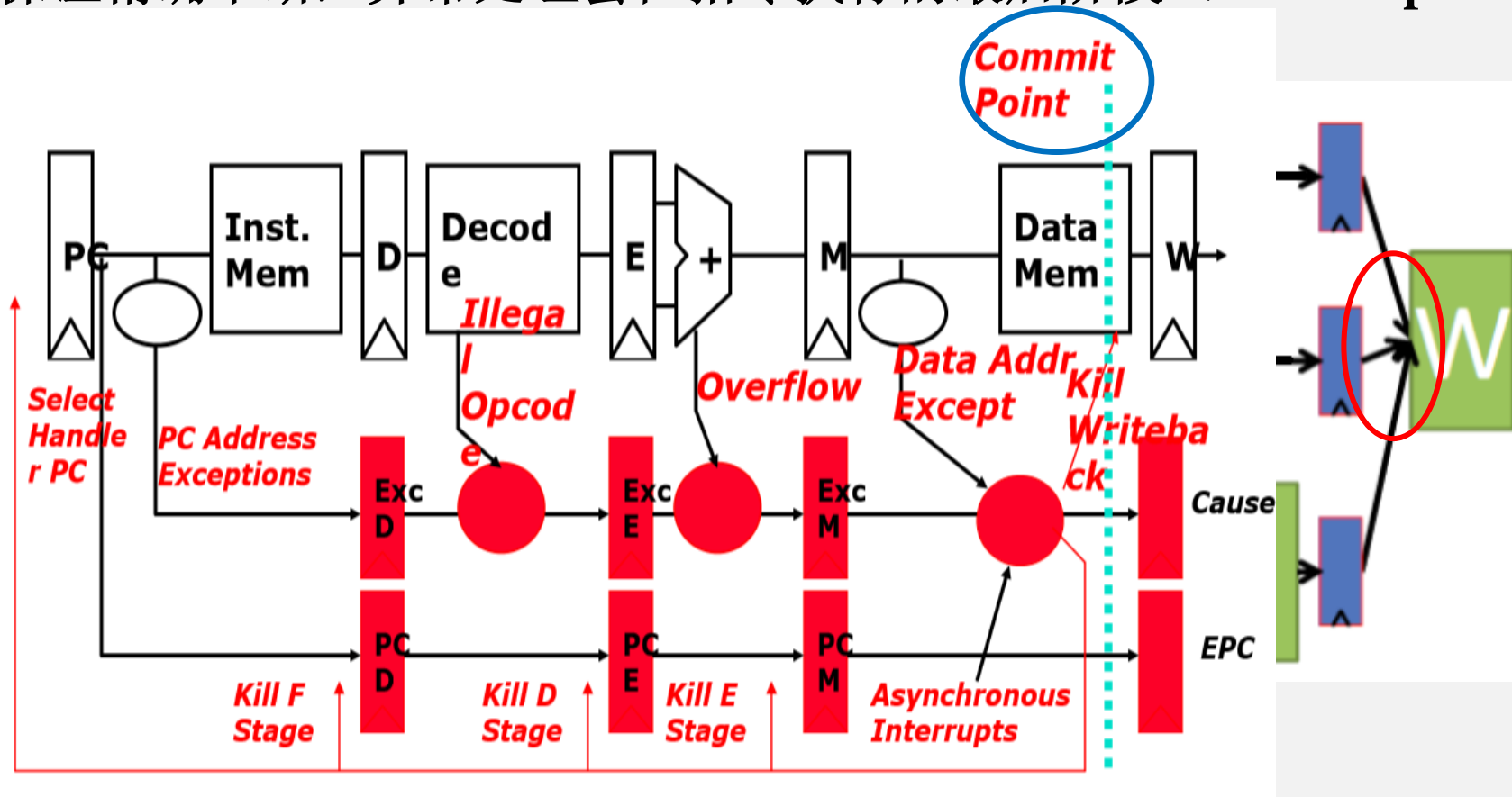
---

## 问题2： 精确中断、预测错误

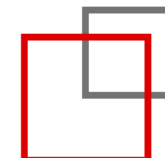
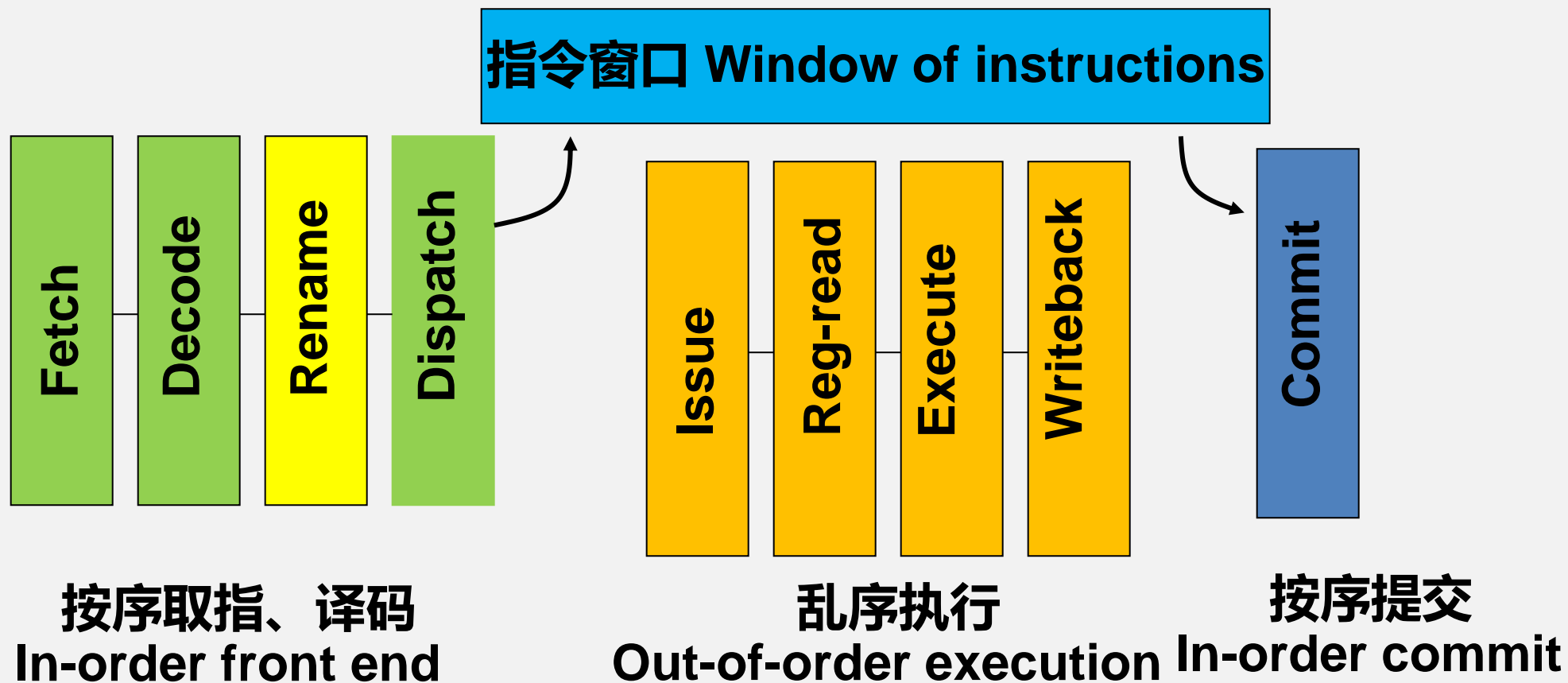
# 如何保证“精确异常”？

- MIPS:

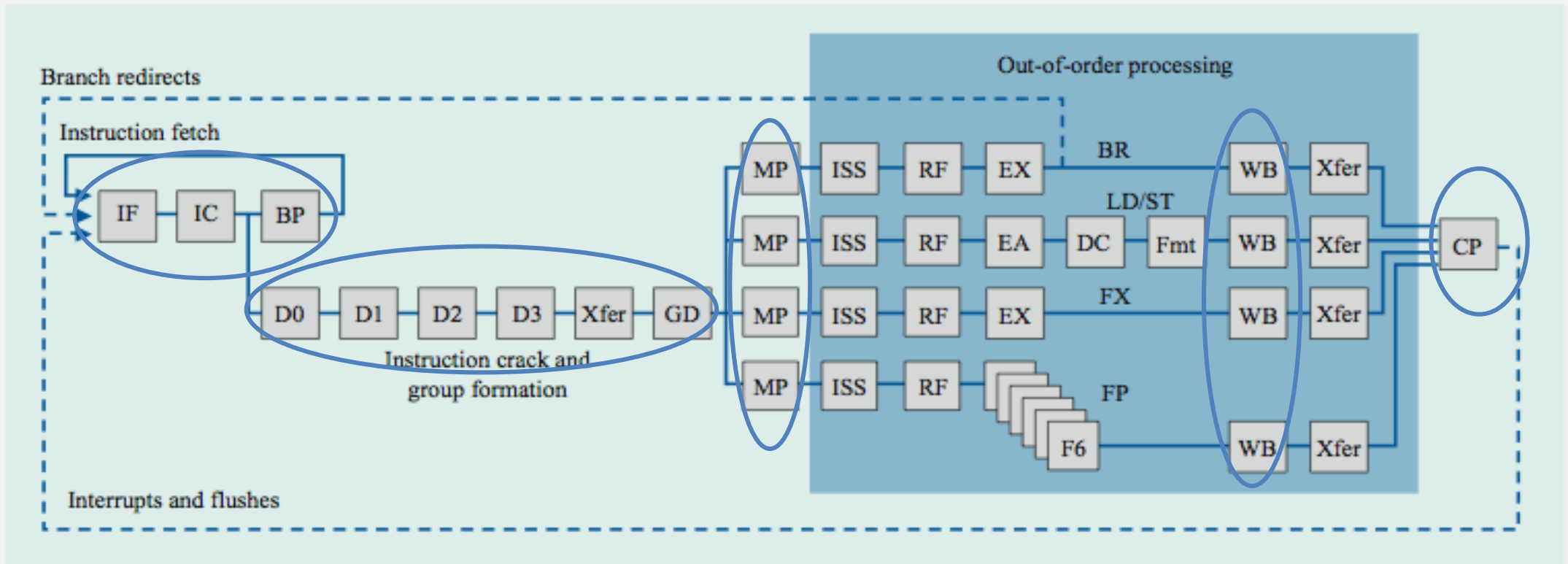
- 为保证精确中断，异常处理会在指令执行的最后阶段（commit point, 提交点）进行



# 乱序流水线



# 实例：IBM Power4



**Instruction pipeline (IF: instruction fetch, IC: instruction cache, BP: branch predict, D0: decode stage 0, Xfer: transfer, GD: group dispatch, MP: mapping, ISS: instruction issue, RF: register file read, EX: execute, EA: compute address, DC: data caches, F6: six-cycle floating-point execution pipe, Fmt: data format, WB: write back, and CP: group commit)**



## 下一节

---

- Branch Prediction
- Limitation of ILP

再见

