

# 从微架构 AMD 看CPU 发展趋势

许宏旭 詹孟奇 袁聪思

# 目录

- AMD Zen简介
- AMD x86 (x64) 处理器历史
- AMD Bulldozer主要技术特性
- AMD Zen的改进
- CPU技术解释和发展趋势

# AMD Zen简介

AMD新型x86微架构Zen



(AMD, 2017)

# AMD Zen

- 2017 Q1
- x86微处理器架构
- 14nm
- AMD Ryzen



# AMD Ryzen™

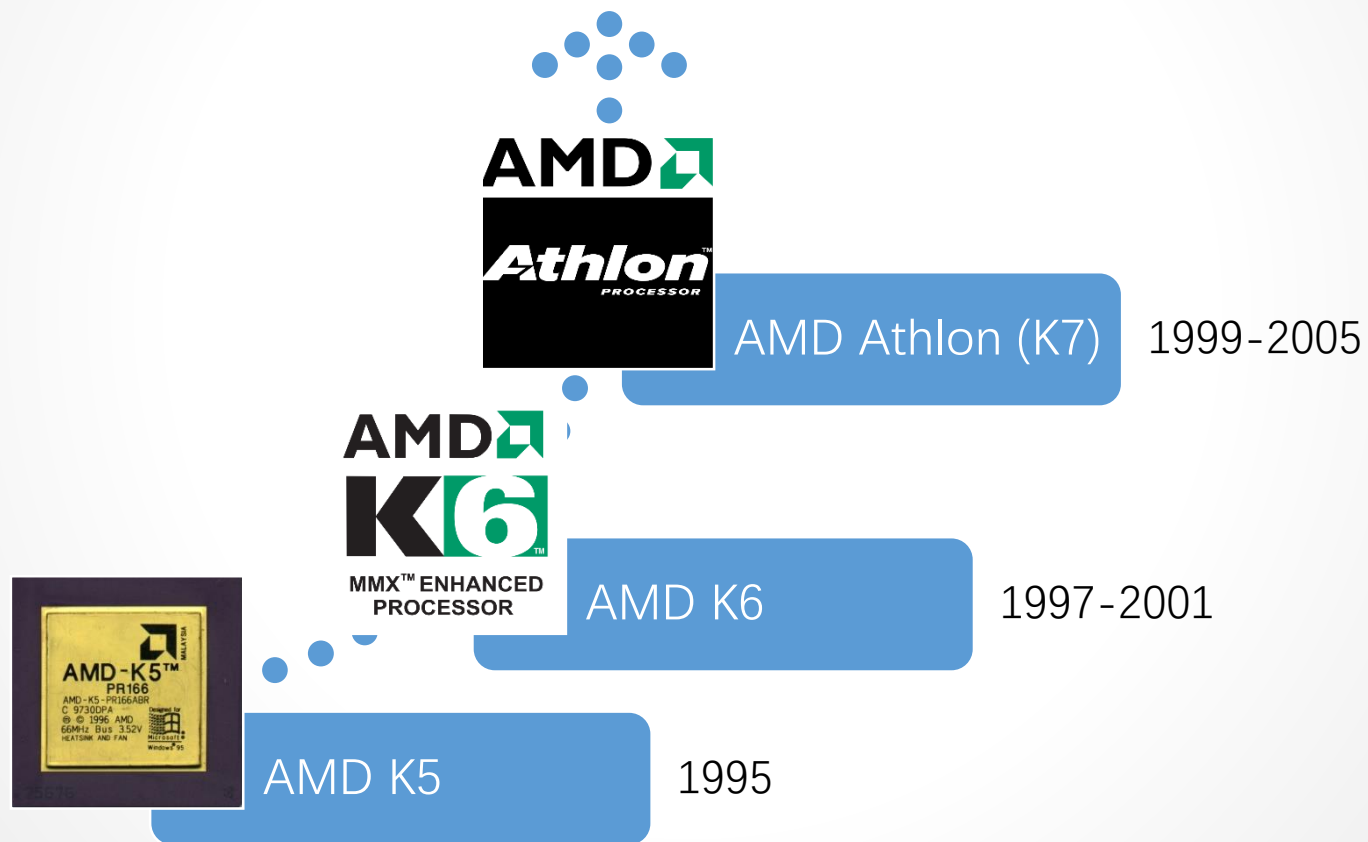


# AMD微架构历史

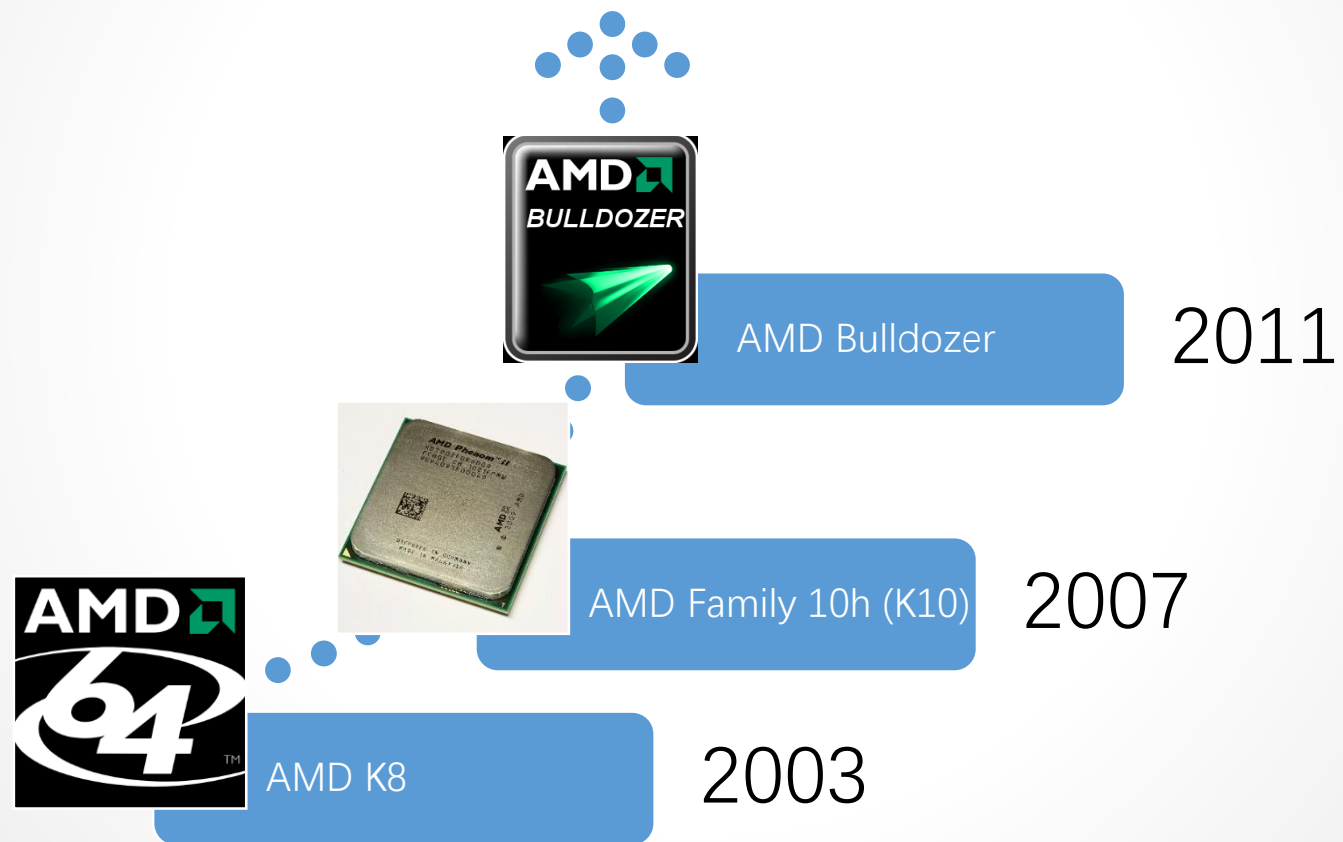
AMD x86 (x64) 微架构历史



# AMD x86微架构历史



# AMD x86-64微架构历史







# AMD Bulldozer

AMD推土机微架构主要技术特性



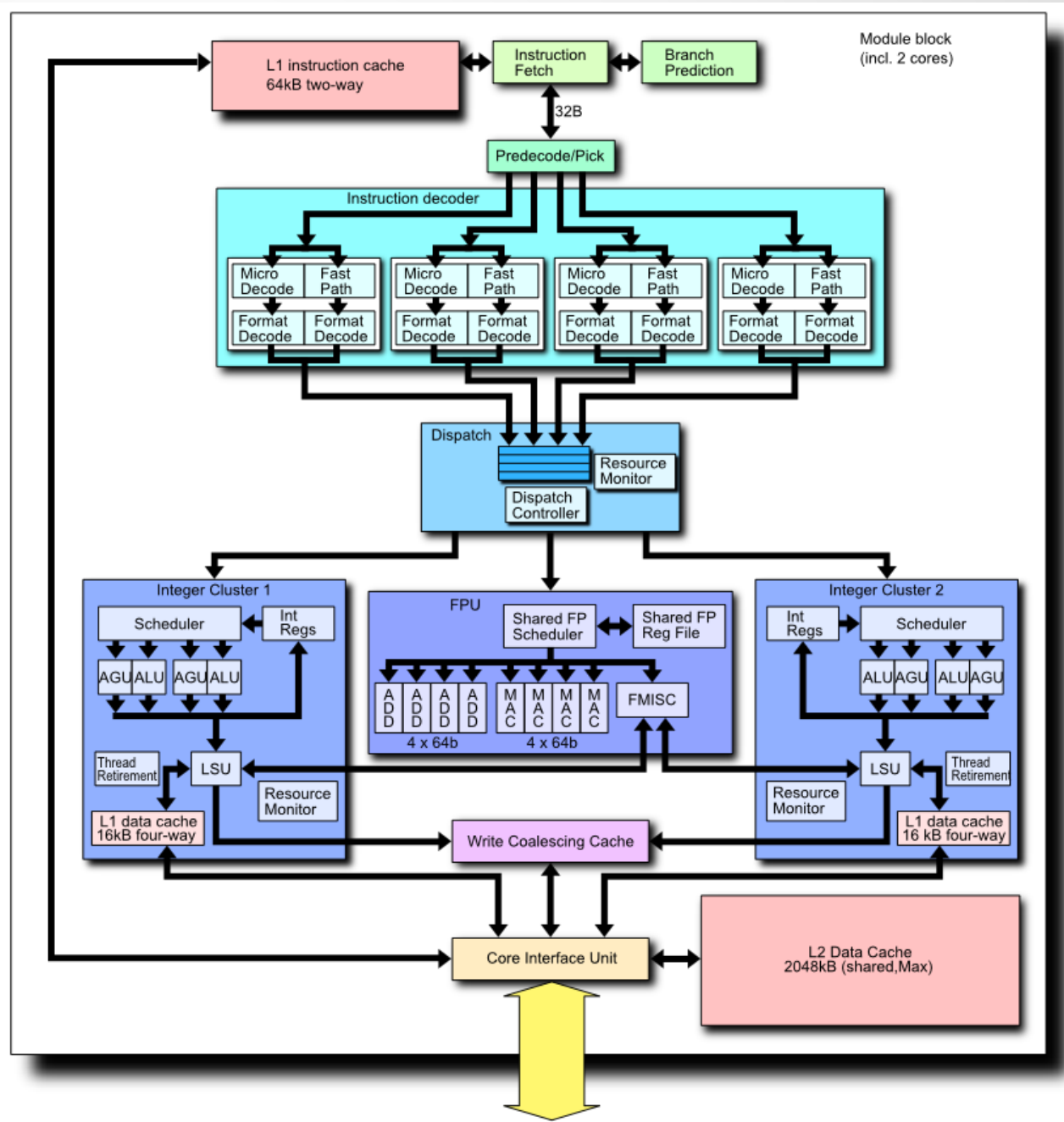
# AMD Bulldozer

- AMD Bulldozer 推土机微架构
  - 第二代：AMD Piledriver 打桩机
  - 第三代：AMD Steamroller 压路机
  - 第四代：AMD Excavator 挖掘机



# AMD Bulldozer 微架构模块图

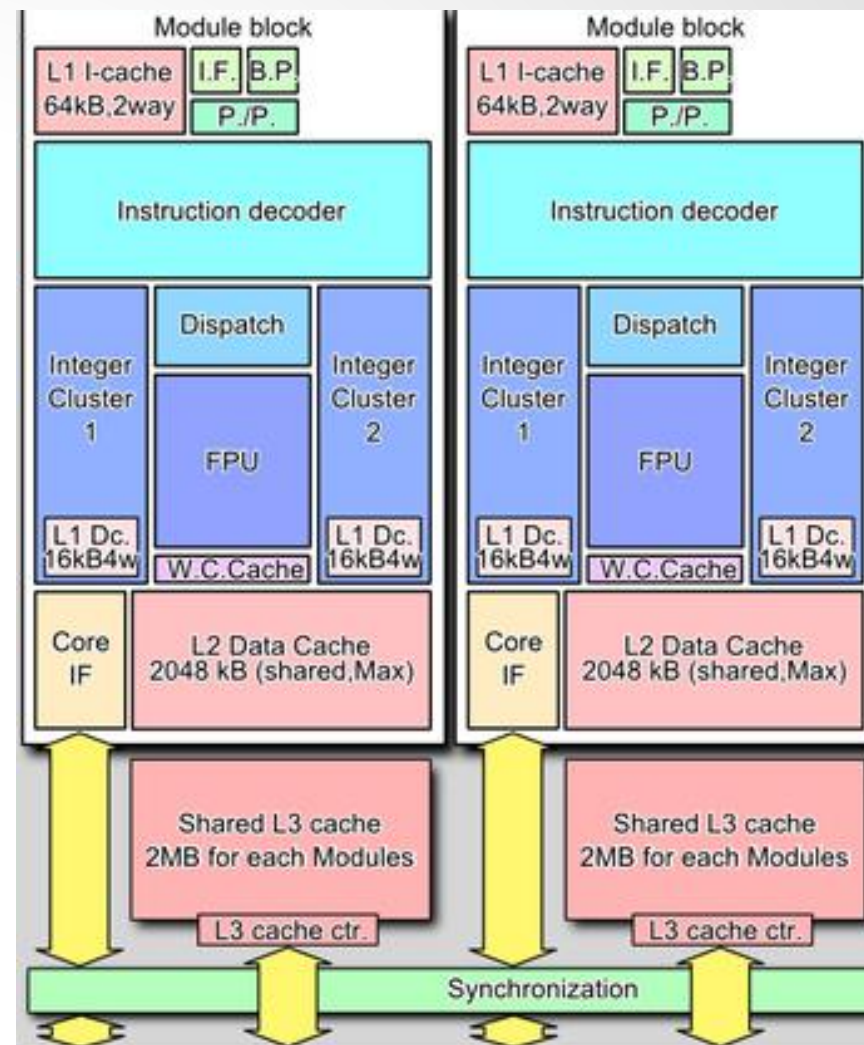
- 缓存
- 分支预测
- 指令抓取和译码
- 转译旁路缓存
- 整数单元
- 浮点单元
- 存取单元
- 集成内存控制器
- 其他单元





# 1. 缓存

- L1指令缓存
  - 64KB 2路-组相连映射
- L1数据缓存
  - 16KB 4路
  - 透写 (Write-Through)
- 计算单位共享的L2缓存
  - 回写 (Write-Back)
  - 包含式 (Inclusive)
- 片上共享的L3缓存
  - 非包含式受害者缓存 (Non-inclusive victim cache)



## 2. 分支预测

- 分支预测机构
  - 下一地址逻辑单元 (Next-Address Logic)
  - 2级分支目标缓冲 (BTB, Branch Target Buffer)
  - 混合分支预测 (Hybrid Branch Predictor)
  - 直接目标预测 (Direct Target Predictor)
  - 间接目标预测 (Indirect Target Predictor)
    - 512项
  - 返回地址栈 (RAS, Return Address Stack)
    - 24项
  - 抓取窗口跟踪结构 (Fetch Window Tracking Structure, BSR)

### 3. 集群多线程

#### CMT, Clustered Multi-Threading

推土机一个模块有

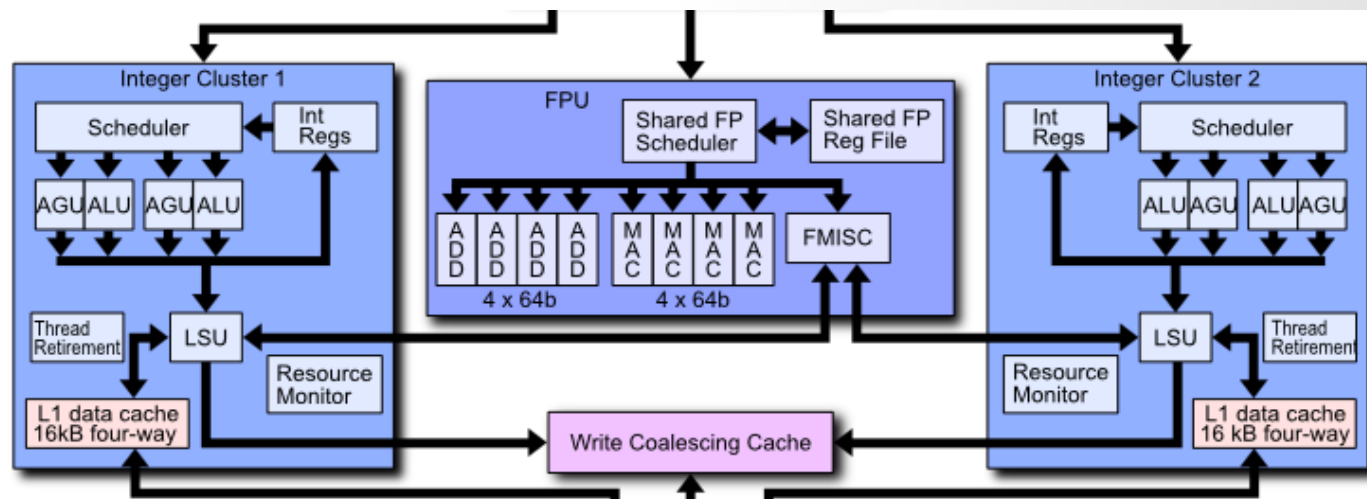
- 两个整数集群 (Integer Cluster)
- 每个整数集群有2个ALU和2个AGU
- 两个集群共享一个浮点单元 (FPU)

整数能力：相当于双核处理器

浮点能力：取决于

- 浮点指令是否在两个线程中同时出现
- FPU处理的是128位还是256位浮点数

缺点：单线程时，整数计算单元闲置



(Wikipedia, 2017)

## 4. 指令集

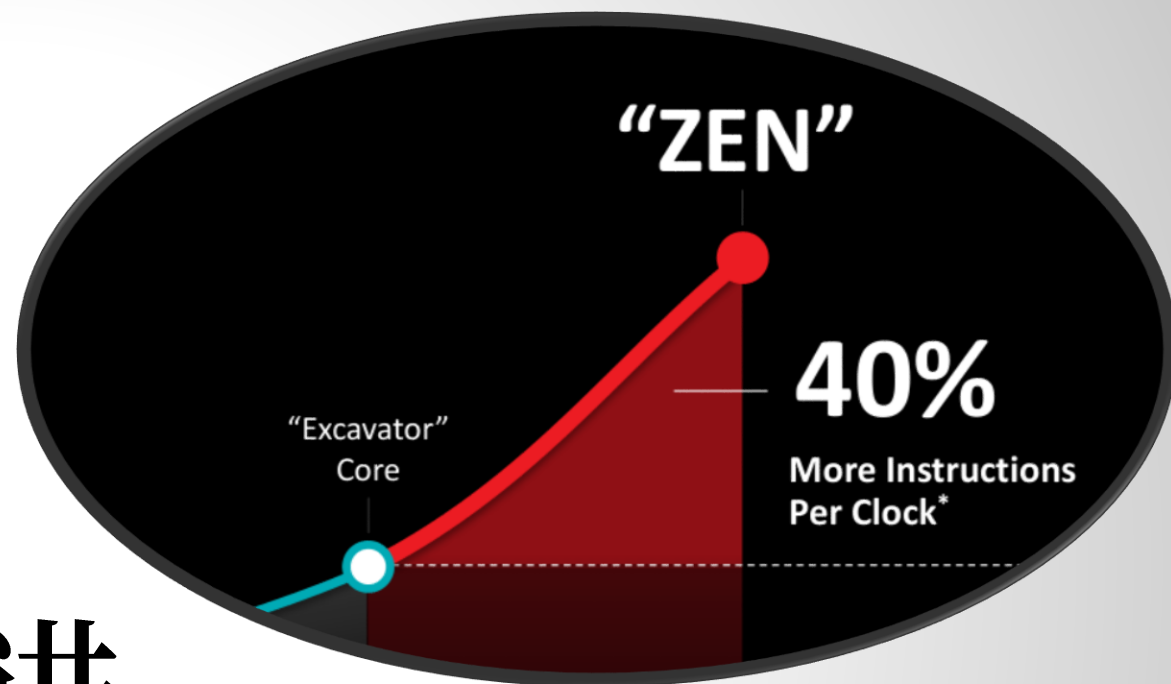
- 支持的x86扩展指令集
  - SSE4 : Streaming SIMD (Single Instruction, Multiple Data) Extension 4
  - AES-NI : Advanced Encryption Standard Instructions
  - AVX : Advanced Vector Extensions
  - XOP : AMD的SSE5修订
  - FMA4 : Fused Multiply-Add (FMAC) Instructions
  - TBM : Trailing Bit Manipulation
  - LWP : Lightweight Profiling

■ Intel的扩展指令集 ■ AMD的扩展指令集

(Hollingsworth, 2012)

# AMD Zen的改进

40%的IPC提升

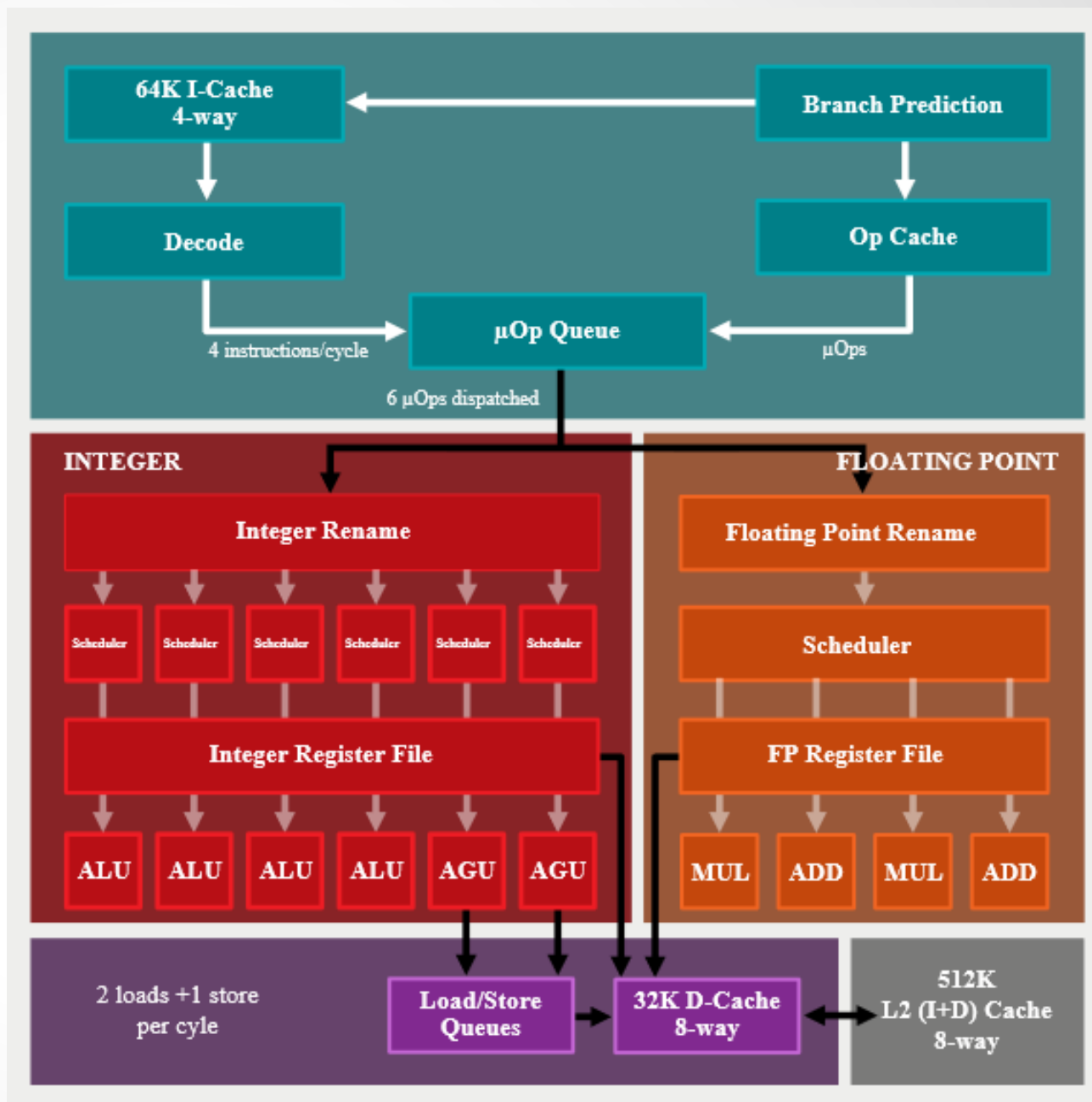


(AMD, 2016)



# AMD Zen微架构概览

- 64K 4路L1指令缓存
- 分支预测
- 微指令缓存
- 整数单元
- 浮点单元
- 存取单元
- 32K 8路L1数据缓存
- 512K 8路L2指令和数据缓存



# 1. 缓存

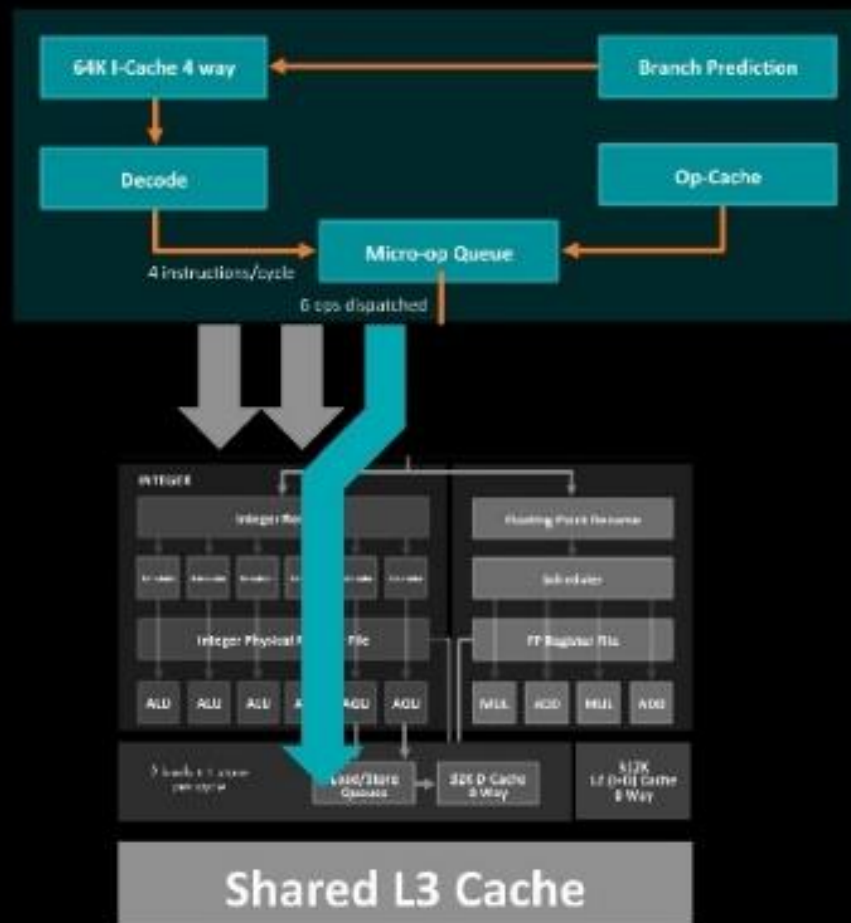
- L1缓存
  - ~~透写 (Write-Through)~~ 改为回写 (Write-Back)
  - 降低延迟和功耗, 提高带宽
- 更大的L2和L3缓存
- 微指令缓存 (uop Cache)
- 更好的L1和L2数据预取器

# Neural Net Prediction



## Scary Smart Prediction

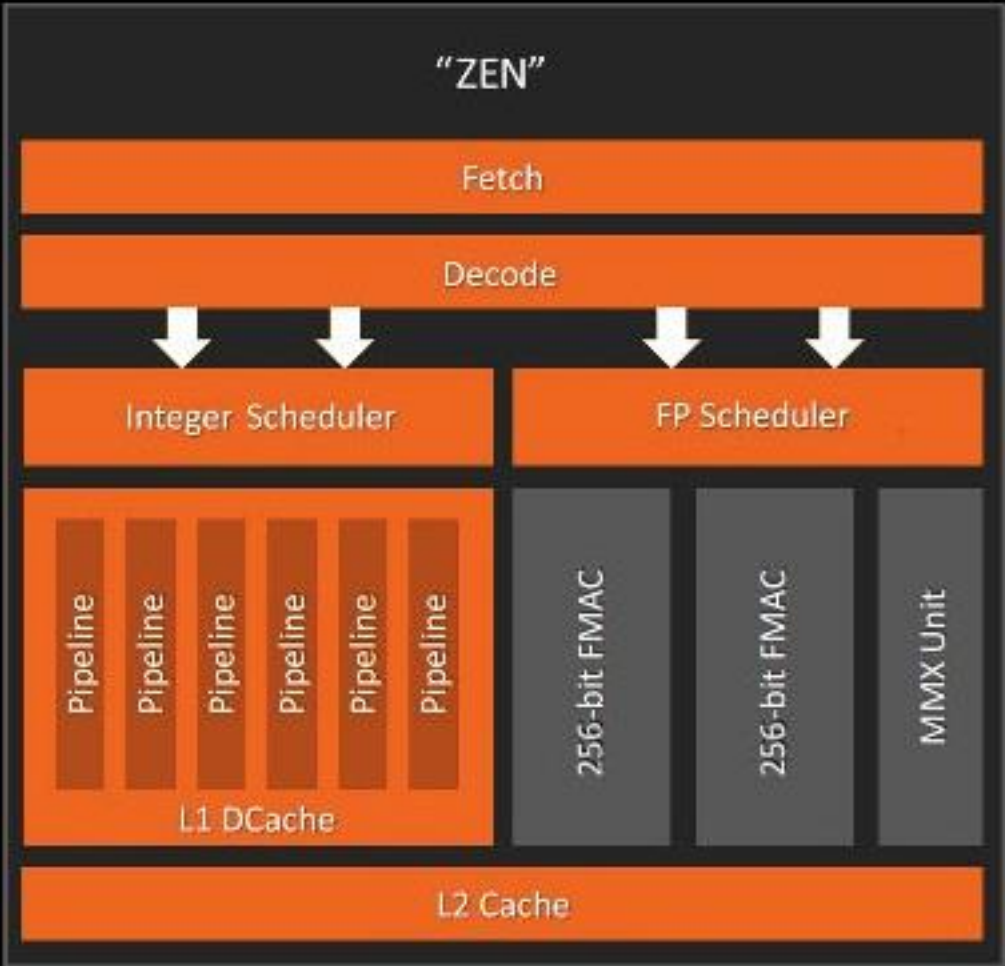
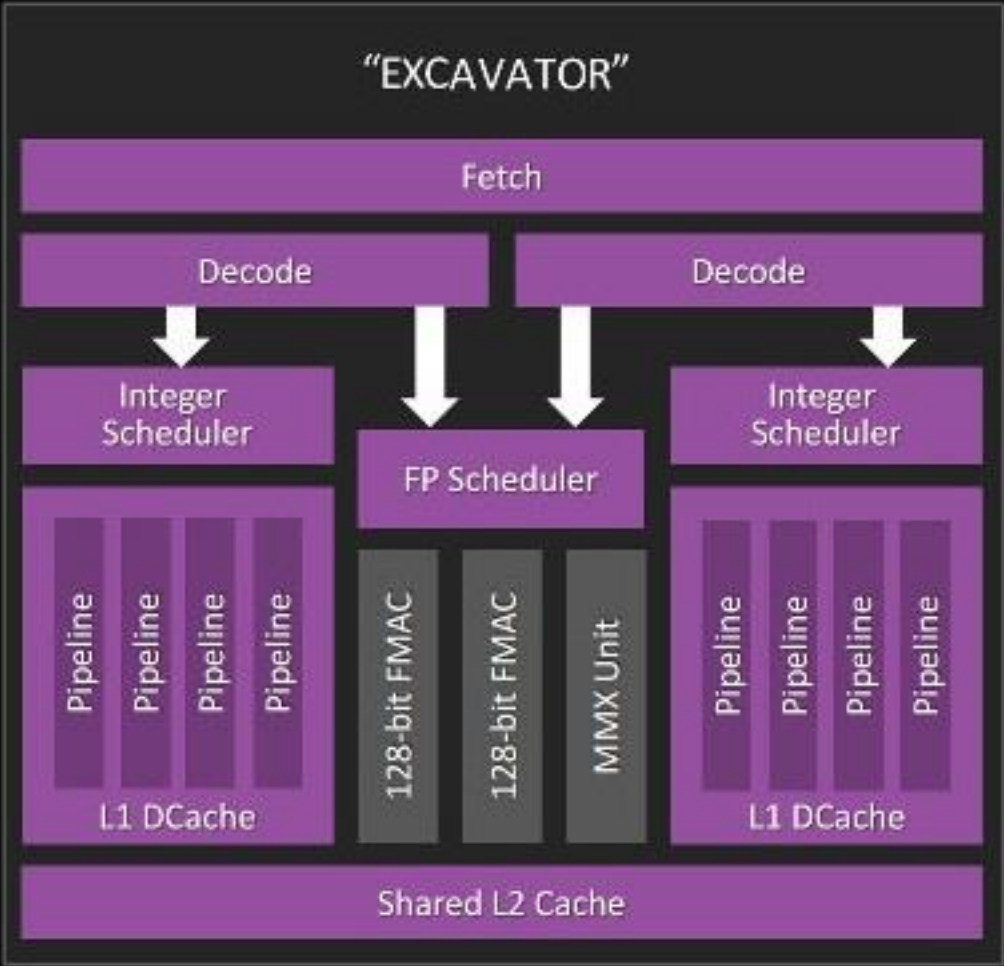
- ▲ A true artificial network inside every “Zen” processor
- ▲ Builds a model of the decisions driven by software code execution
- ▲ Anticipates future decisions, pre-load instructions, choose the best path through the CPU



## 2. 分支预测

- 更大的分支目标缓存 (BTB)
- 每个BTB项包含两个分支
- 返回地址栈从24项增加到32项
- “神经网络”分支预测器 (感知器分支预测器)

# INTRODUCING AMD'S "ZEN" CORE



### 3. 并发多线程

SMT, Simultaneous Multi-Threading

- 抛弃集群多线程 (CMT)
- 采用与Intel基本一致的并发多线程 (SMT)
  - 单线程时, 不存在整数单元闲置
    - 提高吞吐率
    - 提高单线程性能
  - 多线程共享整数单元里的多个ALU
    - 简化电路, 使空间紧凑
    - 扩展性强
    - 功耗降低



# MICROARCHITECTURE

## INSTRUCTION SET EVOLUTION



YEAR	FAMILY	PRODUCT FAMILY	ARCHITECTURE	EXAMPLE MODEL	ADX	CLFLUSHOPT	RDSEED	SHA	SMAP	XGETBV	XSAVEC	XSAVES	AVX2	BMI2	MOVBE	RDRND	SMEP	FSGSBASE	XSAVEOPT	BMI	FMA	F16C	AES	AVX	OSXSAVE	PCLMULQDQ	SSE4.1	SSE4.2	XSAVE	SSSE3	CLZERO	FMA4	TBM	XOP	
2017	17h	“Summit Ridge”	“Zen”	Ryzen 7 1800X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
2015	15h	“Carrizo”/”Bristol Ridge”	“Excavator”	A12-9800	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
2014	15h	“Kaveri”/”Godavari”	“Steamroller”	A10-7890K	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
2012	15h	“Vishera”	“Piledriver”	FX-8370	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
2011	15h	“Zambezi”	“Bulldozer”	FX-8150	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	1	0	1
2013	16h	“Kabini”	“Jaguar”	A6-1450	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0
2011	14h	“Ontario”	“Bobcat”	E-450	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
2011	12h	“Llano”	“Husky”	A8-3870	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2009	10h	“Greyhound”	“Greyhound”	Phenom II X4 955	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- + ADX multi precision support
- + CLFLUSHOPT Flush Cache Line Optimized SFENCE order
- + RDSEED Pseudorandom number generation Seed
- + SHA Secure Hash Algorithm (SHA-1, SHA-256)
- + SMAP Supervisor Mode Access Prevention
- + XGETBV Get extended control register
- + XSAVEC, XSAVES Compact and Supervisor Save/Restore

+ CLZero Zero Cache Line

- FMA4
- TBM
- XOP

## 4. 指令集

- 取消4个AMD特有的指令集
  - TBM、FMA4、XOP、LWP
- 新增大量主流指令的支持
  - SHA、ADX、RDSEED……
- 保持与Intel的良好兼容



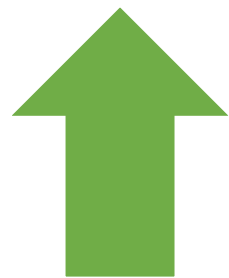
# CPU技术解释和发展趋势

缓存、分支预测、多线程技术和指令集

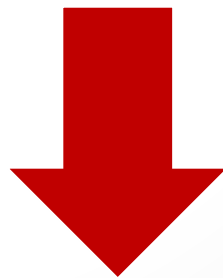
# 1. 缓存

## 1.1. 缓存容量与延迟

增加缓存容量



提高缓存命中率



增加访问延迟

# 1. 缓存

## 1.1. 缓存容量与延迟

- 最佳实践

- L1缓存：32KB-128KB
  - 容量过大：延迟带来的性能下降凸显、成本过高
- 提高CPU时钟频率

(Prybylski, Horowitz, & Hennessy, 1988)

# 1. 缓存

## 1.2. 写入策略

### 透写 (Write-Through)

- 未命中：访问内存
- 存储新数据：立即写入内存
- 更慢
- 结构清晰、构造简单
- 内存数据保持更新

### 回写 (Write-Back)

- 未命中：访问内存
- 存储新数据：缓存项被剔除时
- 更快
- 结构复杂、硬件设计难度大
- 多核共享内存时，尤为复杂

# 1. 缓存

## 1.2. 写入策略

- Zen的L1缓存：改为了回写策略
  - 提高了L1缓存的性能
- 硬件设计复杂度的部分提高是**值得和必要的**

# 1. 缓存

## 1.3. 专用缓存

### 1.3.1. 受害者缓存

- 受害者缓存 (Victim Cache)
  - 与上一级缓存配合使用
  - 较小、全相联
- 行为
  - 上级缓存未命中
    - 检查受害者缓存是否命中
  - 受害者缓存命中
    - 受害者缓存项换入上级缓存对应位置
  - 均未命中
    - 访问内存
    - 上级缓存剔除原缓存项
    - 原缓存项暂存入受害者缓存

# 1. 缓存

1.3. 专用缓存

1.3.1. 受害者缓存

## 访问20001

上级缓存

缓存项	地址	内容
	未命中	
1	10001	A
2	10002	
3	10003	
...	...	...

受害者缓存

缓存项	地址	内容
	命中	
1	20001	C
2		
3		
...	...	...



由受害者缓存换入上级缓存

# 1. 缓存

1.3. 专用缓存

1.3.1. 受害者缓存

## 访问10001

上级缓存

缓存项	地址	内容
	未命中	
1	20001	C
2	10002	
3	10003	
...	...	...

受害者缓存

缓存项	地址	内容
	命中	
1	10001	A
2		
3		
...	...	...



由受害者缓存换入上级缓存



# 1. 缓存

1.3. 专用缓存

1.3.1. 受害者缓存

## 访问30002

上级缓存

缓存项	地址	内容
1	10001	A
2	10002	B
3	10003	C
...	...	...

受害者缓存

缓存项	地址	内容
1	20001	C
2	20002	D
...	...	...

上级缓存剔除原缓存项，访存读入新项  
原缓存项暂存入受害者缓存

# 1. 缓存

## 1.3. 专用缓存

### 1.3.1. 受害者缓存

上级缓存

缓存项	地址	内容
1	10001	A
2	30002	E
3	10003	
...	...	...

受害者缓存

缓存项	地址	内容
1	20001	C
2	10002	B
3		
...	...	...

# 1. 缓存

## 1.3. 专用缓存

### 1.3.2. 微指令缓存

- 微指令缓存 (Uop Cache)
  - Micro-operation Cache
    - 又名UC : Uop Cache、 $\mu$ op Cache
  - 接收指令译码器和指令缓存的结果
  - 存储译码后指令的微操作
  - 当指令需要被译码时, 检查是否有可复用的微指令缓存
    - 避免重复译码

# 附：微操作

- Micro-Operations (Micro-ops、 $\mu$ ops)
- 功能单一的或原子的操作
  - 寄存器的数据交换
  - 寄存器的数值或逻辑计算（简单ALU运算）
- 单个时钟周期内完成

## 2. 分支预测

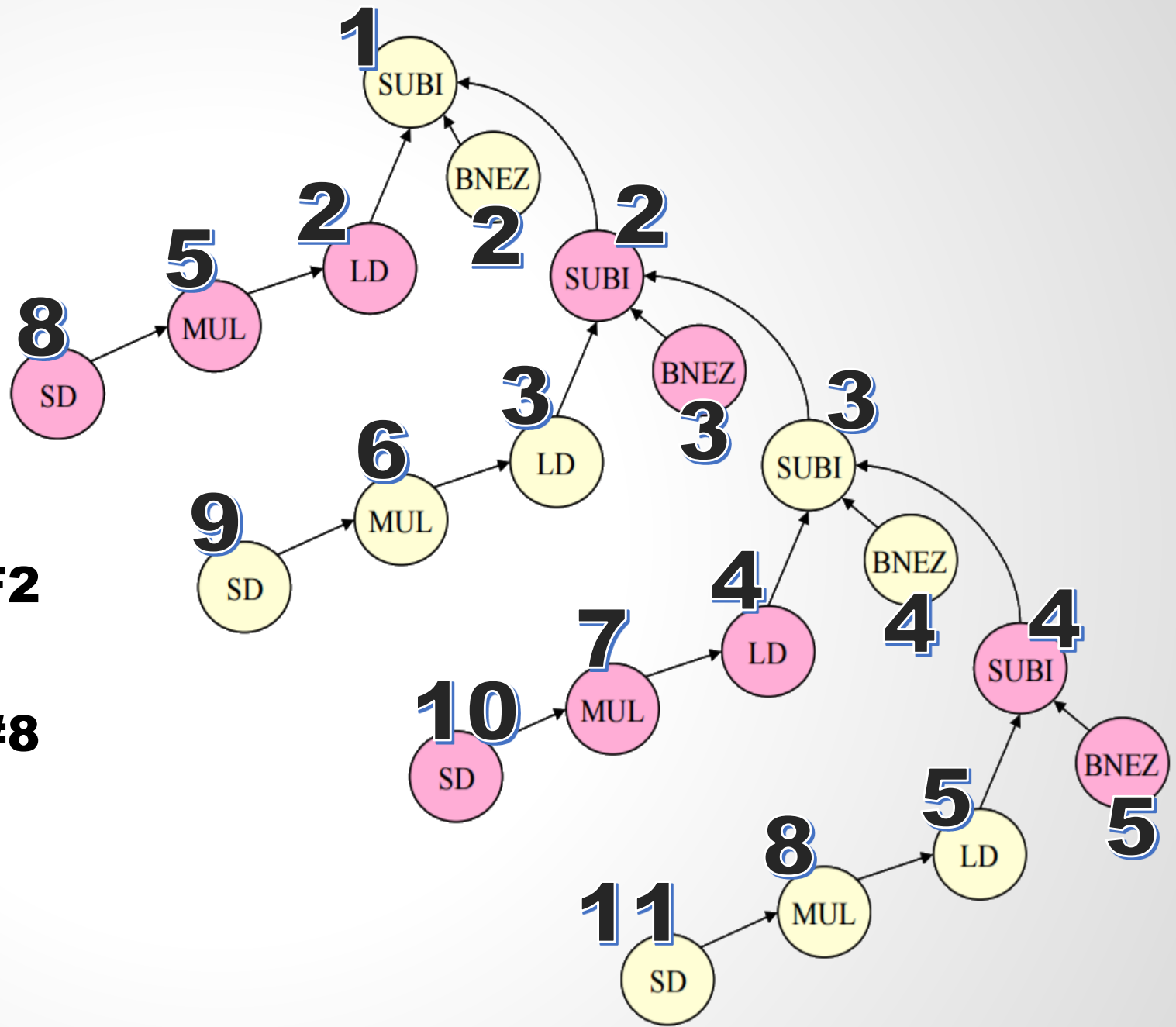
### 2.1. 简介

#### 2.1.1. 乱序执行 (OoO)

**OoO, Out-of-Order**

**Loop:**

<b>LD</b>	<b>F0,</b>	<b>0 (R1)</b>
<b>MULTD</b>	<b>F4,</b>	<b>F0, F2</b>
<b>SD</b>	<b>F4,</b>	<b>0 (R1)</b>
<b>SUBI</b>	<b>R1,</b>	<b>R1, #8</b>
<b>BNEZ</b>	<b>R1,</b>	<b>Loop</b>



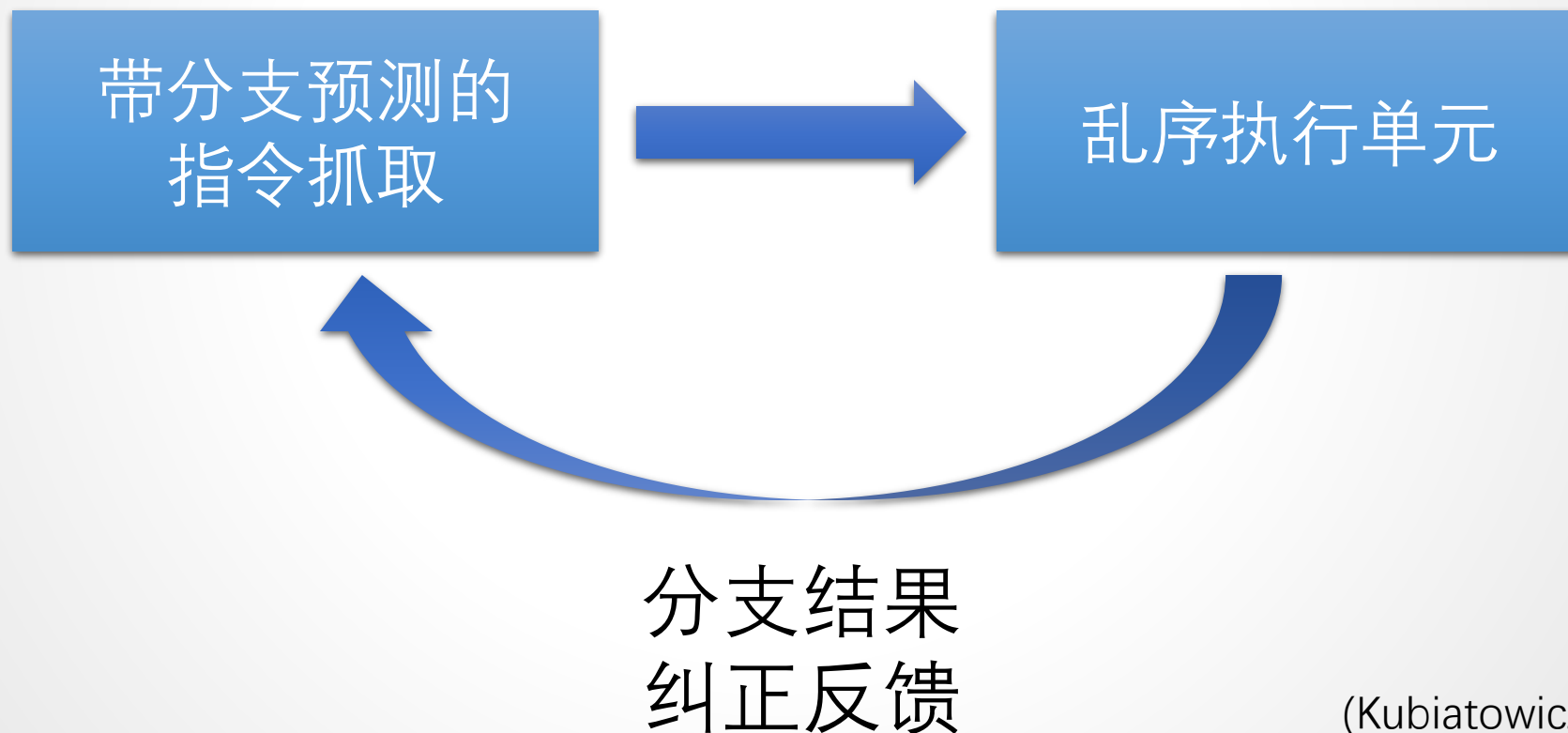
指令	周期1	周期2	周期3	周期4	周期5	周期6	周期7	周期8	...
LD	*	*	*						...
SUB	*								...
MUL				*	*	*			...
SD							*		...
BNE		*							...
LD		*	*	*					
SUB		*							
MUL					*	*	*		
SD								*	
BNE			*						

## 2. 分支预测

2.1. 简介

2.1.2. 目的

实现指令抓取与执行的解耦



(Kubiatowicz, 2011)

## 2. 分支预测

### 2.1. 简介

### 2.1.2. 目的

Loop:

```
LD          F0,    0,    R1
MULTD       F4,    F0,    F2
SD          F4,    0 (R1)
SUBI        R1,    R1,    #8
BNEZ        R1,    Loop
```

...

- 需等待**分支指令**完成才能继续**取指令**操作
- 分支预测
  - 跳转 (Taken)
  - 不跳转 (Not Taken)
- 避免流水线停顿

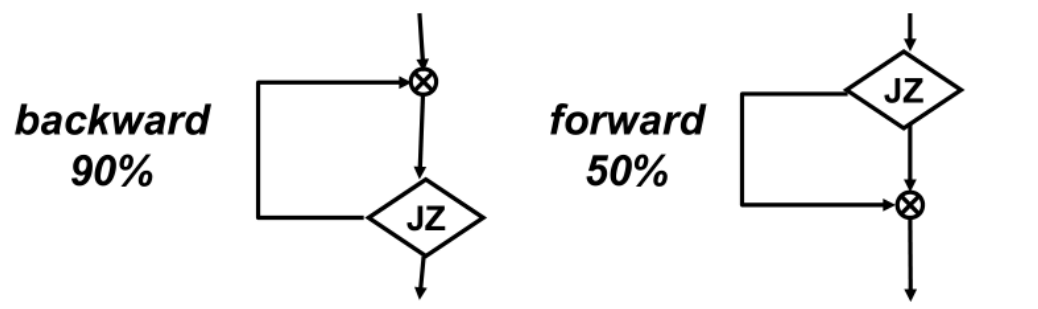


## 2. 分支预测

### 2.1. 简介

#### 2.1.3. 静态分支预测

- 最简单的分支预测
  - 分支跳转的概率大概有60%~70%
- 例如：Motorola MC88110
  - BNE：预测跳转
  - BEQ：预测不跳转



## 2. 分支预测

2.1. 简介

2.1.4. 动态分支预测

- 时间相关性
  - 一个分支当前执行的行为是下一次执行的很好预测
- 空间相关性
  - 几个不同分支的行为可能相互关联

## 2. 分支预测

2.1. 简介

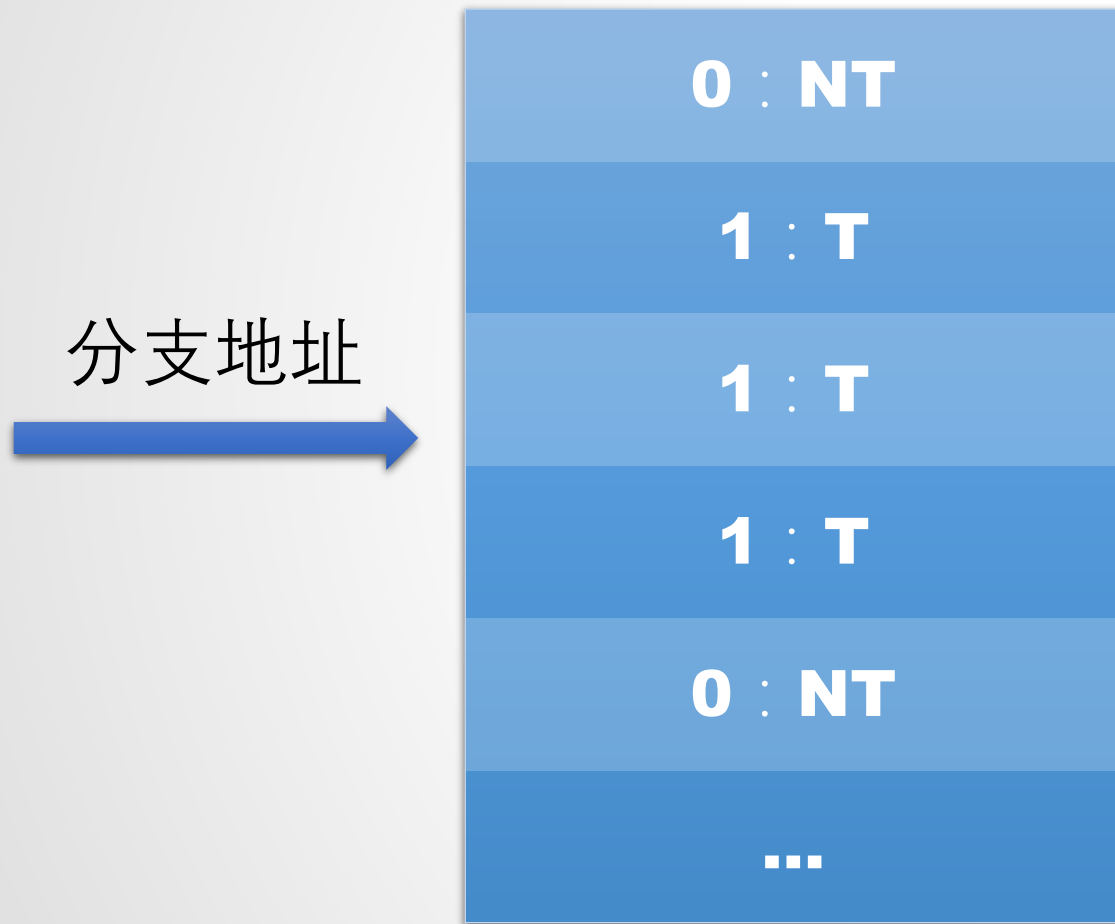
2.1.4. 动态分支预测



## 2. 分支预测

2.1. 简介

2.1.5. 分支历史表 (BHT, Branch History Table)



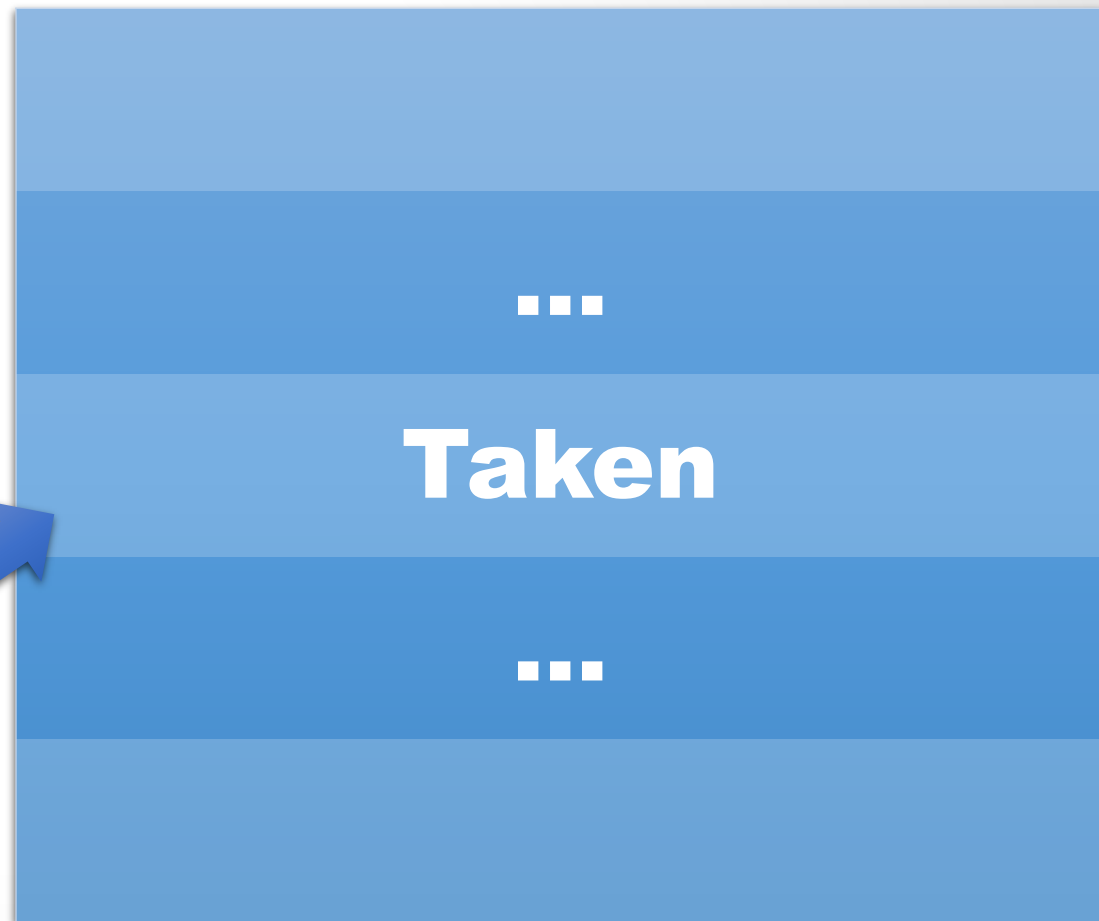
- 分支历史表是一个“预测器”表
- 每个分支有一个预测器状态机
  - 可以是**1 bit**的状态机
  - 跳转 (T, Taken)
  - 不跳转 (NT, Not Taken)

2.

2.1

根据BHT，第1次到达BNZ分支指令，预测Taken  
因为循环2次，所以成功预测

```
MOV    X,    2
LOOP:
...
SUB     X,    1
BNZ     X,    LOOP
```



2.  
2.1

第2次：仍预测Taken  
预测失败

```
MOV    X,    2
LOOP:
...
SUB     X,    1
BNZ     X,    LOOP
```

**Not Taken**



2.  
2.1

假设该循环再次被调用，又要循环2次  
第一次遇到BNZ：Not Taken  
预测失败

**LOOP:**

...

**BNZ**

**X, LOOP**

...

**MOV**

**X, 2**

**JMP**

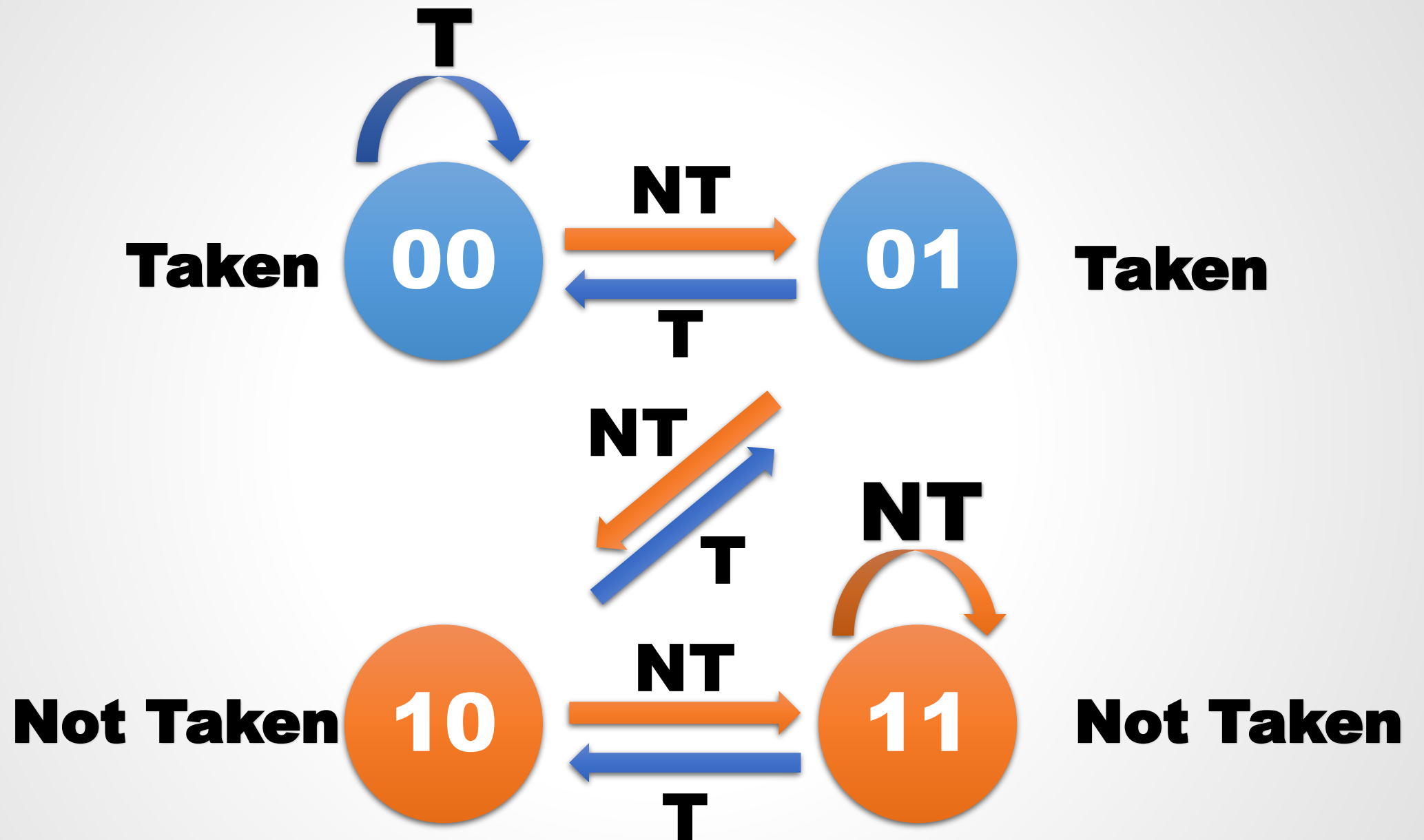
**LOOP**



...

**Taken**

...





## 2. 分支预测

2.1. 简介

2.1.5. 分支历史表 (BHT, Branch History Table)

- **2bit 状态机**
  - 相当于增加了一个延迟, 避免频繁改变

## 2. 分支预测

### 2.1. 简介

### 2.1.6. 分支目标缓冲 (BTB, Branch Target Buffer)

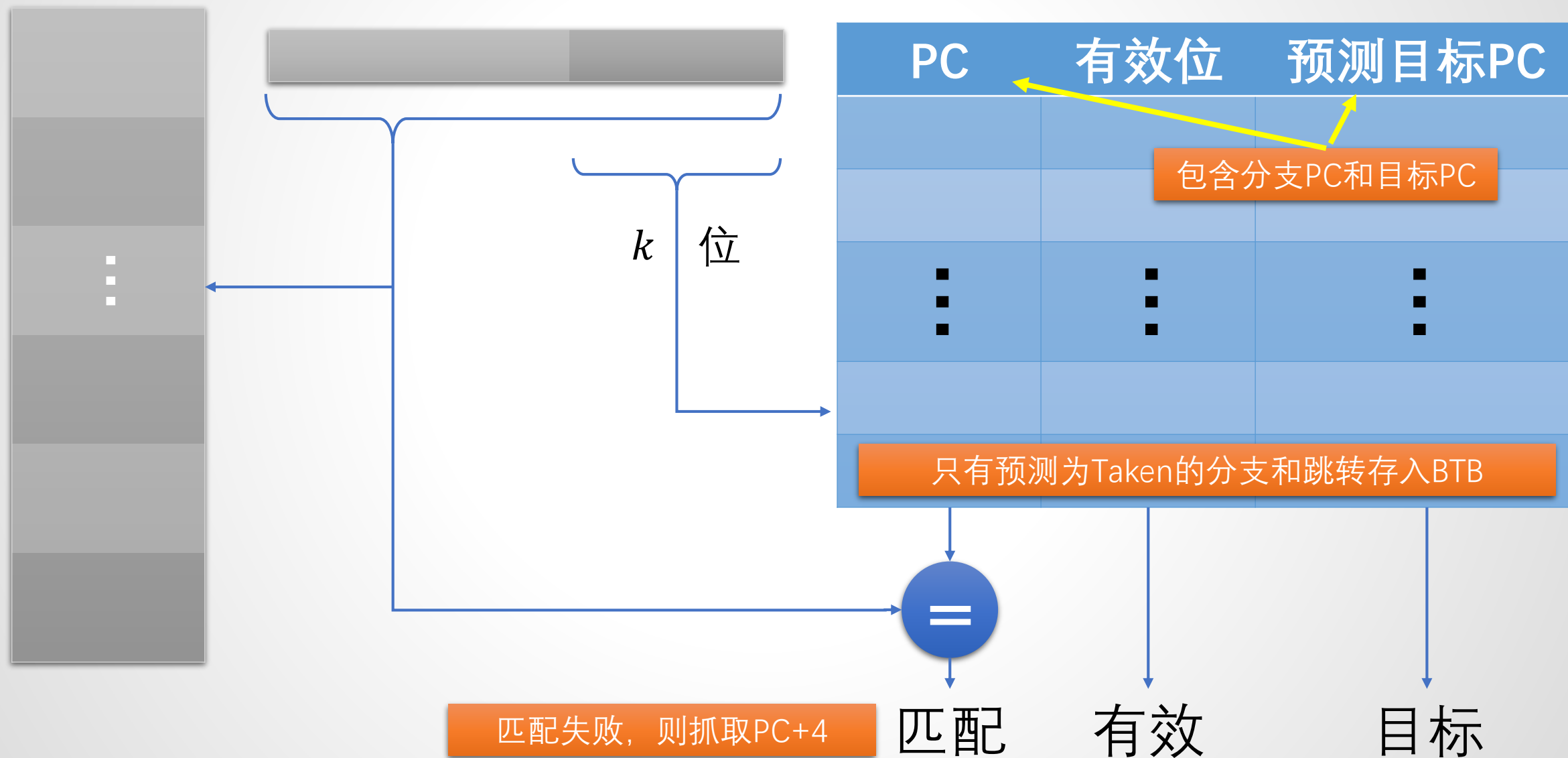
- 分支历史表仅作出预测
- 仍需计算分支目标地址
- 分支目标缓冲 (BTB, Branch Target Buffer)
  - 存储：分支跳转目标地址

# $2^k$ -项直接相联映射BTB

也可以组相联映射

指令缓存

PC



## 2. 分支预测

### 2.1. 简介

### 2.1.6. 分支目标缓冲 (BTB, Branch Target Buffer)

- BTB比BHT更复杂，成本更高
- 结合BTB和BHT
  - 指令抓取时，首先访问BTB
  - BTB未命中，使用BHT预测
  - 最后，更新二者



UltraSPARC-III指令抓取流水线阶段

## 2. 分支预测

2.1. 简介

2.1.7. 返回地址栈 (Return Address Stack)

- 返回地址栈

Return Address Stack

- 预测子程序返回的地址

- 调用子程序, 入栈返回地址
- 子程序返回, 出栈返回地址

## 2. 分支预测

2.1. 简介

2.1.8. 相关分支 (Correlating Branches)

**假定**

最近的分支互相关联

**即**

最近执行的分支**影响**对当前分支的预测

## 2. 分支预测

2.1. 简介

2.1.8. 相关分支 (Correlating Branches)

两种可能，当前分支依赖于：

- 全局：程序最后执行的m个分支
- 地址：当前分支最后m个目标地址

```
if (x < 7) a += 1;  
if (x < 5) b -= 1;
```

若第一个条件为False  
第二个条件也会为False

## 2. 分支预测

### 2.1. 简介

### 2.1.8. 相关分支 (Correlating Branches)

#### 全局分支历史寄存器 (GBHR)

Global Branch History Register

- 每得到一个分支的跳转结果时
  - GBHR原数据左移1位
  - 新跳转记录在GBHR中最低位
    - Taken & Not Taken

想一想：针对下列程序  
2-bit GBHR如何变化

```
a = b = c = 0;  
if (a == 0) a = 1;  
if (a == 1) b = 1;  
if (b == 1) c = 1;  
if (c == 0) c = 1;
```



## 2. 分支预测

2.1. 简介

2.1.8. 相关分支 (Correlating Branches)

2-bit 全局分支历史寄存器

**NT**

**NT**

初始情况

```
a = b = c = 0;  
if (a == 0) a = 1;  
if (a == 1) b = 1;  
if (b == 1) c = 1;  
if (c == 0) c = 1;
```

## 2. 分支预测

2.1. 简介

2.1.8. 相关分支 (Correlating Branches)

2-bit全局分支历史寄存器

**NT**

**NT**

条件成立, 不跳转  
NT

```
a = b = c = 0;  
if (a == 0) a = 1;  
if (a == 1) b = 1;  
if (b == 1) c = 1;  
if (c == 0) c = 1;
```

## 2. 分支预测

2.1. 简介

2.1.8. 相关分支 (Correlating Branches)

2-bit全局分支历史寄存器

**NT**

**NT**

条件成立, 不跳转  
NT

```
a = b = c = 0;  
if (a == 0) a = 1;  
if (a == 1) b = 1;  
if (b == 1) c = 1;  
if (c == 0) c = 1;
```

## 2. 分支预测

2.1. 简介

2.1.8. 相关分支 (Correlating Branches)

2-bit全局分支历史寄存器

**NT**

**T**

条件不成立, 跳转  
T

```
a = b = c = 0;  
if (a == 0) a = 1;  
if (a == 1) b = 1;  
if (b == 1) c = 1;  
if (c == 0) c = 1;
```

## 2. 分支预测

2.1. 简介

2.1.8. 相关分支 (Correlating Branches)

2-bit全局分支历史寄存器

T

T

条件不成立, 跳转  
T

```
a = b = c = 0;  
if (a == 0) a = 1;  
if (a == 1) b = 1;  
if (b == 1) c = 1;  
if (c == 0) c = 1;
```

## 2. 分支预测

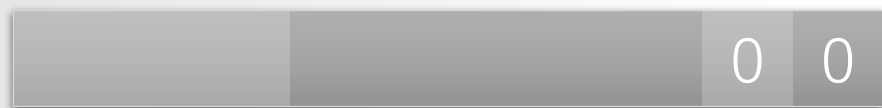
### 2.1. 简介

#### 2.1.8.1. 二级分支预测器 (Two-Level Branch Predictor)

- 利用全局分支历史寄存器
- 二维的分支历史表
  - 纵向PC索引
  - 横向历史索引

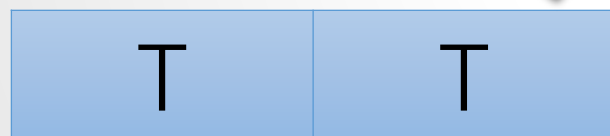
PC

1-bit 分支历史表



$k$  位

T,T	T,NT	NT,T	NT,NT
⋮	⋮	⋮	⋮
<u>T</u>	NT	NT	NT



2-bit全局分支历史寄存器

数据选择器

预测结果：**T**

## 2. 分支预测

### 2.1. 简介

#### 2.1.8.2.对比演示 (1-bit 分支历史表分支预测)

```
if (d == 0) d = 1; → 分支 b1  
if (d == 1) ...; → 分支 b2
```

假设这两个分支在一个循环中  
d的值2、0交替



d = ?	b1预测	b1实际	新b1预测	b2预测	b2实际	新b2预测
2						
0						
2						
0						



## 2. 分支预测

### 2.1. 简介

### 2.1.8.2. 对比演示 (1-bit 全局分支历史寄存器)

```
if (d == 0) d = 1;  分支 b1  
if (d == 1) ...;  分支 b2
```

假设这两个分支在一个循环中  
d的值2、0交替

1-bit 全局分支历史寄存器

**NT**



d = ?	b1预测	b1实际	新b1预测	b2预测	b2实际	新b2预测
2	<b>NT/NT</b>	<b>T</b>	<b>T/NT</b>			
0						
2						
0						

**错**

## 2. 分支预测

### 2.1. 简介

### 2.1.8.2.对比演示 (1-bit 全局分支历史寄存器)

```
if (d == 0) d = 1;  分支 b1  
if (d == 1) ...;  分支 b2
```

假设这两个分支在一个循环中  
d的值2、0交替

1-bit 全局分支历史寄存器

**T**



d = ?	b1预测	b1实际	新b1预测	b2预测	b2实际	新b2预测
2	<b>NT/NT</b>	<b>T</b>	<b>T/NT</b>	<b>NT/NT</b>	<b>T</b>	<b>NT/T</b>
0						
2						
0						

**错**

## 2. 分支预测

### 2.1. 简介

### 2.1.8.2.对比演示 (1-bit 全局分支历史寄存器)

```
if (d == 0) d = 1;  分支 b1  
if (d == 1) ...;  分支 b2
```

假设这两个分支在一个循环中  
d的值2、0交替

1-bit 全局分支历史寄存器

...

d = ?	b1预测	b1实际	新b1预测	b2预测	b2实际	新b2预测
2	NT/NT	T	T/NT	NT/NT	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T
2	T/NT	T	T/NT	NT/T	T	NT/T
0	T/NT	NT	T/NT	NT/T	NT	NT/T

## 2. 分支预测

### 2.2. 机构齐全

现代CPU综合利用多种分支预测技术

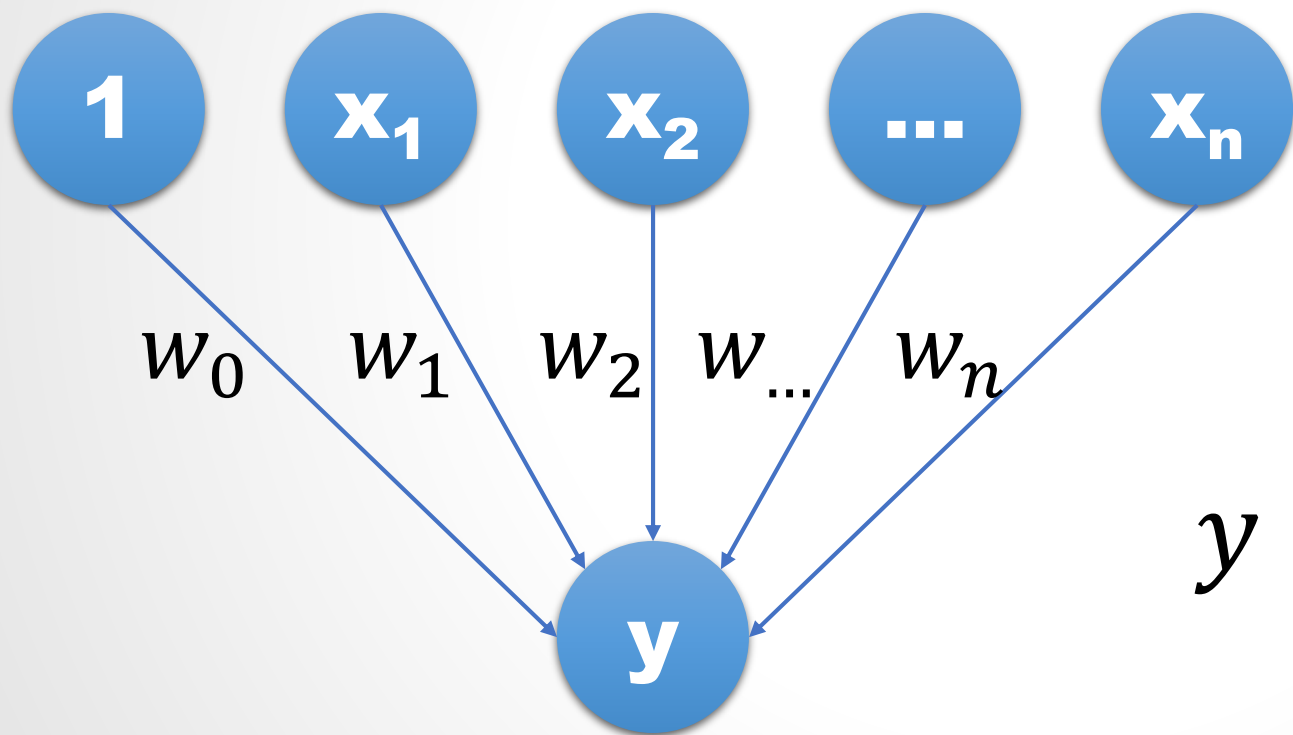
## 2. 分支预测

### 2.3. 机器学习的方法（感知器/神经网络分支预测器）

- AMD Zen引入神经网络分支预测技术
  - 实质上是一种感知器分支预测器
- 能够考虑更多的历史情况
  - 输入：分支跳转历史
    - $x_i$ 表示分支历史寄存器第 $i$ 位
    - 1为跳转，-1为不跳转
  - 输出： $< 0$ 预测不跳转， $\geq 0$ 预测跳转

## 2. 分支预测

### 2.3. 机器学习的方法（感知器/神经网络分支预测器）



$$y = w_0 + \sum_{i=1}^n x_i w_i$$

## 2. 分支预测

### 2.3. 机器学习的方法（感知器/神经网络分支预测器）

- $\theta$  为训练阈值
  - 防止训练过度
  - 更快适应新情况
- $|y| \leq \theta$  或  $y$  预测失败时，更新权重
- 更新权重：对每个分支历史  $x_i$ 
  - 若  $x_i$  与跳转结果一致， $w_i++$
  - 否则， $w_i--$

$x_{1..n}$  is the  $n$ -bit history register,  $x_0$  is 1.  
 $w_{0..n}$  is the weights vector.  
 $t$  is the Boolean branch outcome.  
 $\theta$  is the training threshold.

```
if  $|y| \leq \theta$  or  $((y \geq 0) \neq t)$  then
  for each  $0 \leq i \leq n$  in parallel
    if  $t = x_i$  then
       $w_i := w_i + 1$ 
    else
       $w_i := w_i - 1$ 
    end if
  end for
end if
```

## 3. 多线程技术

### 3.1. 多线程技术

- 命令依赖 导致 流水线冒险
  - 流水线互锁——效率低
  - 旁路直通——适用面窄
- 多线程技术
  - 单线程延迟增加
  - 改善多线程总体延迟
  - 提高CPU利用率

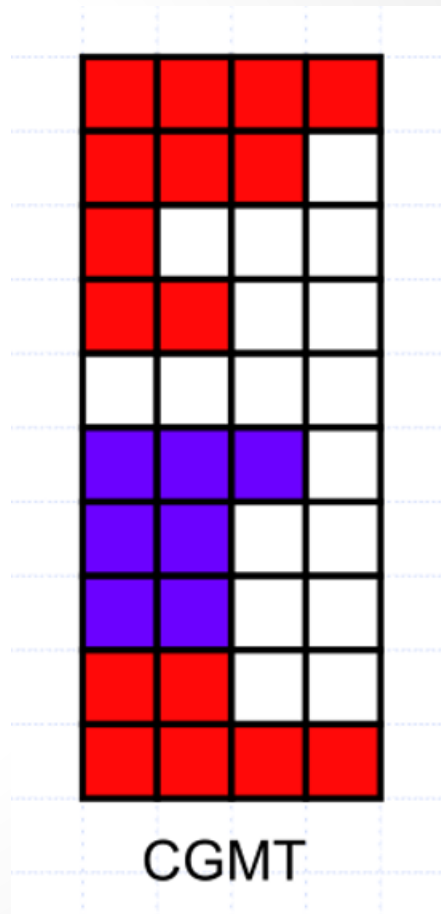


# 3. 多线程技术

## 3.1. 多线程技术

### 3.1.1. 实现

- 粗粒度多线程 (CGMT)  
Coarse-Grain Multi-Threading
- 指定一个优先线程A,  
持续执行
  - 当L2未命中时, 切换到线程B
  - L2未命中, 访存结果返回后,  
切换回线程A

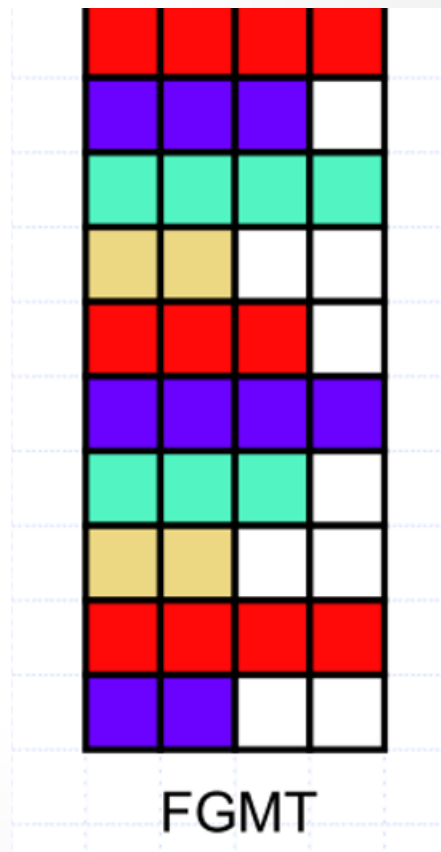


## 3. 多线程技术

### 3.1. 多线程技术

#### 3.1.1. 实现

- 细粒度多线程 (FGMT)  
Fine-Grain Multi-Threading
- 每个周期切换一次线程
  - 需要存在大量线程以供切换

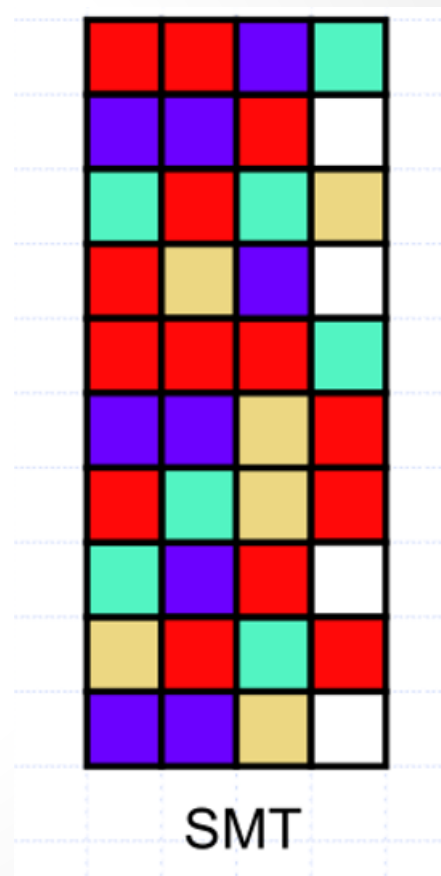


# 3. 多线程技术

## 3.1. 多线程技术

### 3.1.1. 实现

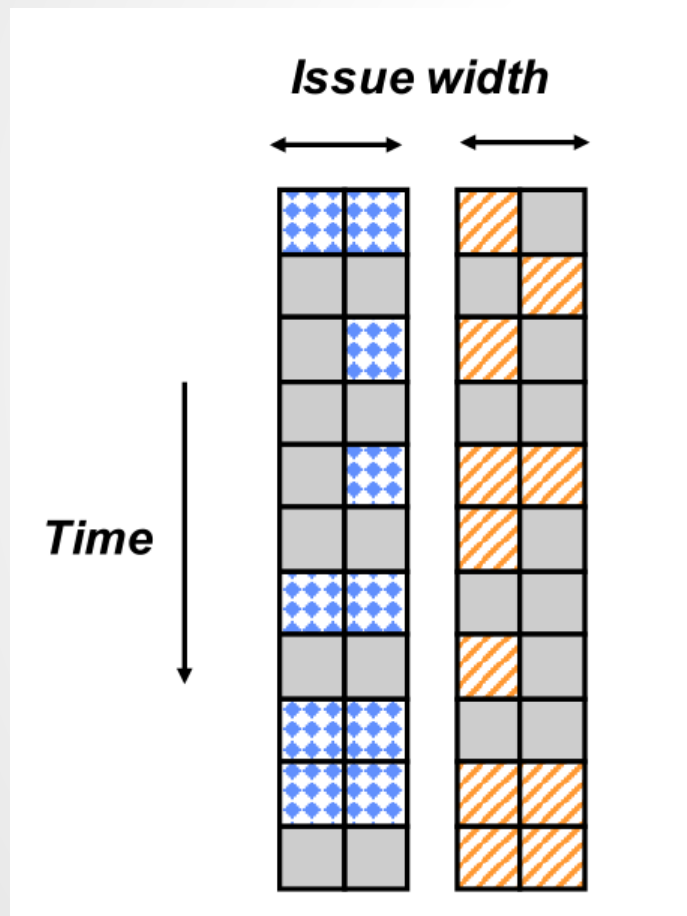
- 并发多线程 (SMT)  
Simultaneous Multi-Threading
- 每个周期切换一次线程
- 基于OoO处理器



# 3. 多线程技术

## 3.1. 多线程技术

### 3.1.2. 与多核处理器比较

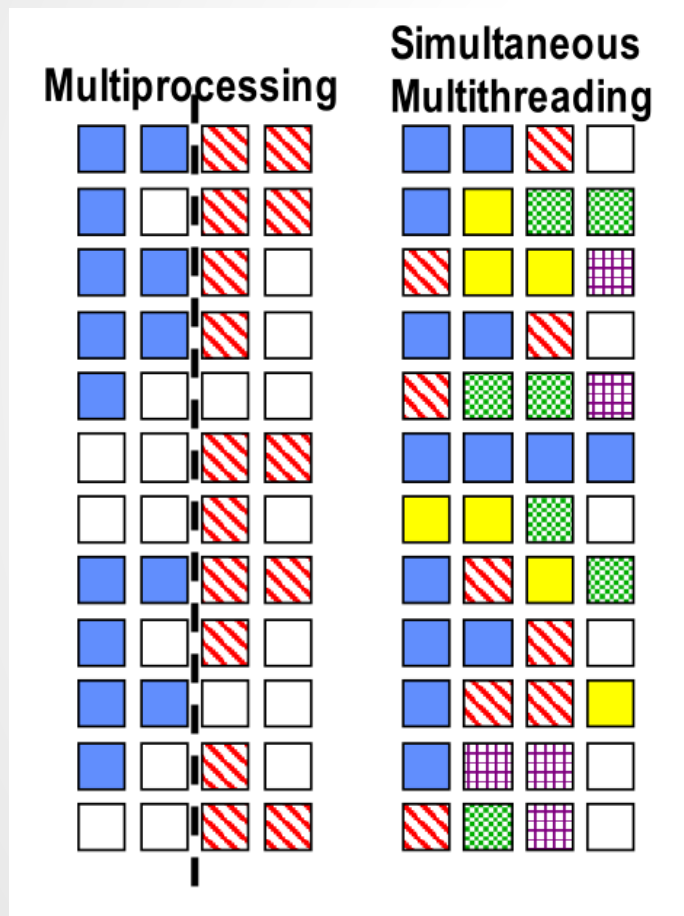


- 多核处理器
  - 线程级并发处理 (TLP)  
Thread-Level Parallelism
  - 保留部分纵向浪费

# 3. 多线程技术

## 3.1. 多线程技术

### 3.1.2. 与多核处理器比较



- 并发多线程（SMT）
  - 指令级并发处理（ILP）  
Instruction-Level Parallelism
  - 最大程度减少CPU浪费
- 现代CPU
  - SMT与多核处理器并用

# 3. 多线程技术

## 3.1. 多线程技术

### 3.1.3. 发展

- AMD Bulldozer的集群多线程（CMT）本质上也是一种SMT
  - 硬件设计复杂
  - 多线程性能强
  - 牺牲单线程性能
- AMD Zen抛弃CMT，拥抱SMT
  - Intel的Hyper Threading也是SMT
  - 降低功耗、提升单线程性能
- ILP热点问题持续研究和发展

## 4. 指令集

### 4.1. 兼容性

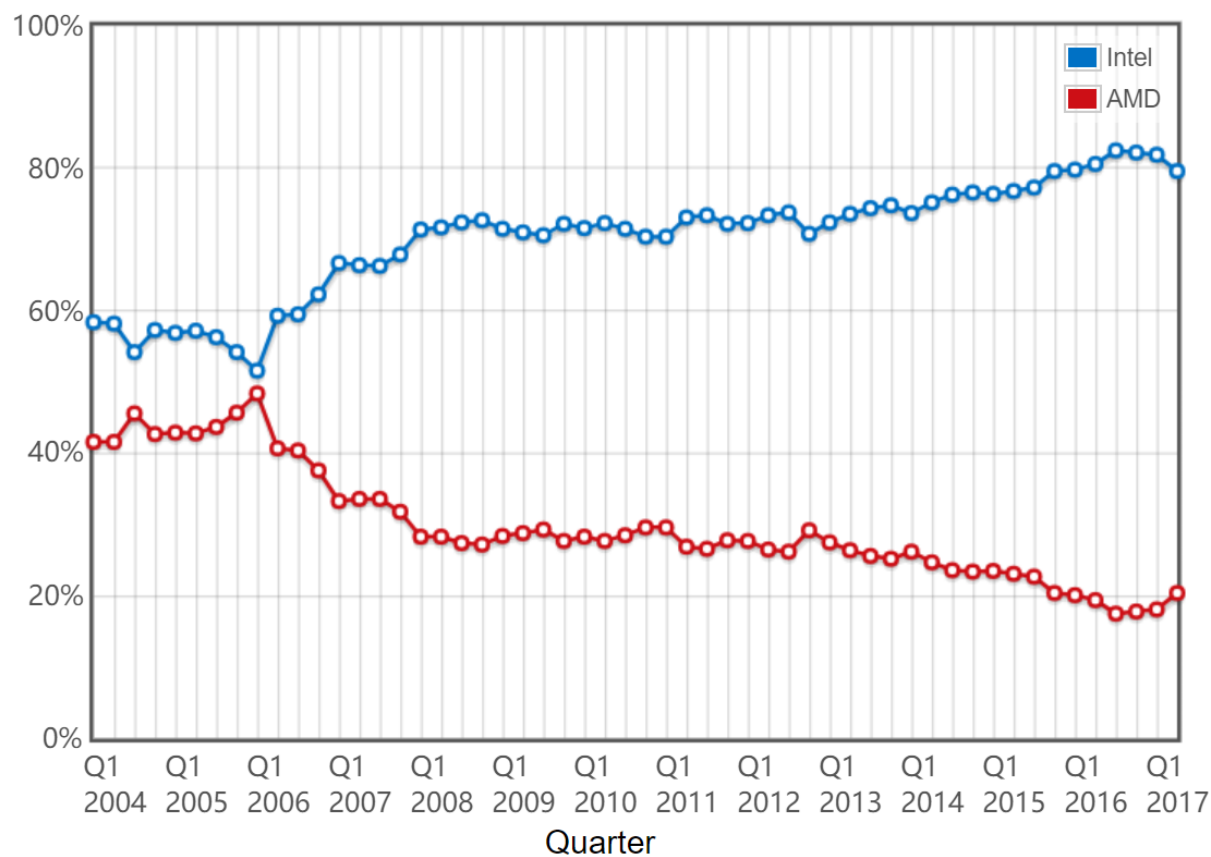
#### AMD Zen

- 取消AMD特有指令集
- 拥抱Intel主导指令集

Intel是x86/64处理器主流

AMD vs Intel Market Share

Updated 29th of May 2017



# 参考文献

- Amata. (2017年1月13日). Exclusive Interview With AMD's Robert Hallock on Ryzen | Architecture, Performance & Chip Details. 检索来源: Exclusive Interview With AMD's Robert Hallock on Ryzen | Architecture, Performance & Chip Details: <http://www.redgamingtech.com/exclusive-interview-with-amds-robert-hallock/>
- AMD. (2014年1月). Software Optimization Guide for AMD Family 15h Processor. United States. 检索来源: [https://support.amd.com/TechDocs/47414\\_15h\\_sw\\_opt\\_guide.pdf](https://support.amd.com/TechDocs/47414_15h_sw_opt_guide.pdf)
- AMD. (2016年8月23日). AMD and the new "Zen" High Performance x86 Core at Hot Chips 28. 检索来源: <https://www.slideshare.net/AMD/amd-and-the-new-zen-high-performance-x86-core-at-hot-chips-28/1>
- AMD. (2017年5月20日). "Zen" 核心架构. 检索来源: AMD: <https://www.amd.com/zh-hans/technologies/zen-core>
- AMD. (2017年3月2日). AMD Ryzen CPU Optimization. 检索来源: <http://32ipi028l5q82yhj72224m8j.wpengine.netdna-cdn.com/wp-content/uploads/2017/03/GDC2017-Optimizing-For-AMD-Ryzen.pdf>
- AMD. (2017年3月2日). 锐龙 AMD Ryzen7 台式处理器今日起全球上市！性能超越想象！. 检索来源: AMD: <http://www.amd.com/zh-cn/press-releases/Pages/press-releases-2017Mar02.aspx>
- Barragy Ted. (2012年1月23日). Bulldozer. 检索来源: [https://www.olcf.ornl.gov/wp-content/uploads/2012/01/TitanWorkshop2012\\_Day1\\_AMD.pdf](https://www.olcf.ornl.gov/wp-content/uploads/2012/01/TitanWorkshop2012_Day1_AMD.pdf)
- Kubiawicz John. (2011年2月14日). Prediction/Speculation (Branches, Return Addrs). 检索来源: <https://people.eecs.berkeley.edu/~kubitron/courses/cs252-S11/lectures/lec08-prediction.pdf>



# 参考文献 (续)

- Hollingsworth Brent. (2012年10月). New “Bulldozer” and “Piledriver” Instructions - A step forward for high performance software development . 检索来源: <https://developer.amd.com/wordpress/media/2012/10/New-Bulldozer-and-Piledriver-Instructions.pdf>
- University of California, San Diego. Out of Order Execution Continued. 检索来源: [https://cseweb.ucsd.edu/classes/wi14/cse141/pdf/09/17\\_ReorderBuffer.ppt.pdf](https://cseweb.ucsd.edu/classes/wi14/cse141/pdf/09/17_ReorderBuffer.ppt.pdf)
- Jimenez D.A., & Lin C. (2001). Dynamic branch prediction with perceptrons. High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on. Monterrey, Nuevo Leon, Mexico: IEEE.
- Martin, & Roth. (2005). Multithreading. UNIVERSITY OF PENNSYLVANIA, United States. 检索来源: [https://www.cis.upenn.edu/~milom/cis501-Fall05/lectures/12\\_smt.pdf](https://www.cis.upenn.edu/~milom/cis501-Fall05/lectures/12_smt.pdf)
- Mirkovic Jelena. (2005年10月27日). Correlating Branch Predictors. 检索来源: <https://www.eecis.udel.edu/~sunshine/courses/F05/CIS662/class14.pdf>
- PassMark. (2017年5月29日). AMD vs Intel Market Share. 检索来源: CPU Benchmarks: [https://www.cpubenchmark.net/market\\_share.html](https://www.cpubenchmark.net/market_share.html)
- Prybylski S., Horowitz M., & Hennessy J. (1988). Performance tradeoffs in cache design. Proceeding ISCA '88 Proceedings of the 15th Annual International Symposium on Computer architecture (页 290-298). Honolulu, Hawaii, USA: IEEE Computer Society Press Los Alamitos, CA, USA ©1988.

# 参考文献 (续2)

- Rotenberg E., Bennett S., & Smith J.E. (1996). Trace cache: a low latency approach to high bandwidth instruction fetching. Microarchitecture, 1996. MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on. IEEE.
- Scali. (2012年2月14日). The myth of CMT (Cluster-based Multithreading). 检索来源: Scali's OpenBlog: <https://scalibq.wordpress.com/2012/02/14/the-myth-of-cmt-cluster-based-multithreading/>
- Solomon Baruch, Mendelson Avi, Orenstein Doron, Almog Yoav, & Ronen Ronny. (2001). Micro-operation cache: a power aware frontend for the variable instruction length ISA. Proceeding ISLPED '01 Proceedings of the 2001 international symposium on Low power electronics and design (页 4-9). Huntington Beach, California, USA: ACM.
- Wawrzynek John. (2016年10月25日). Multithreading. University of California, Berkeley, United States. 检索来源: <http://www-inst.eecs.berkeley.edu/~cs152/fa16/lectures/L14-Multithread.pdf>
- Weatherspoon Hakim. (2013). Caches (Writing). United States: Cornell University. 检索来源: <http://www.cs.cornell.edu/courses/cs3410/2013sp/lecture/18-caches3-w.pdf>
- Wikipedia. (2017年5月9日). Bulldozer (microarchitecture). 检索来源: Bulldozer (microarchitecture): [https://en.wikipedia.org/wiki/Bulldozer\\_\(microarchitecture\)](https://en.wikipedia.org/wiki/Bulldozer_(microarchitecture))
- Wikipedia. (2017年5月21日). Zen (microarchitecture). 检索来源: Zen (microarchitecture): [https://en.wikipedia.org/wiki/Zen\\_\(microarchitecture\)](https://en.wikipedia.org/wiki/Zen_(microarchitecture))

# 谢谢！

许宏旭 詹孟奇 袁聪思