

计算机系统

处理器体系结构

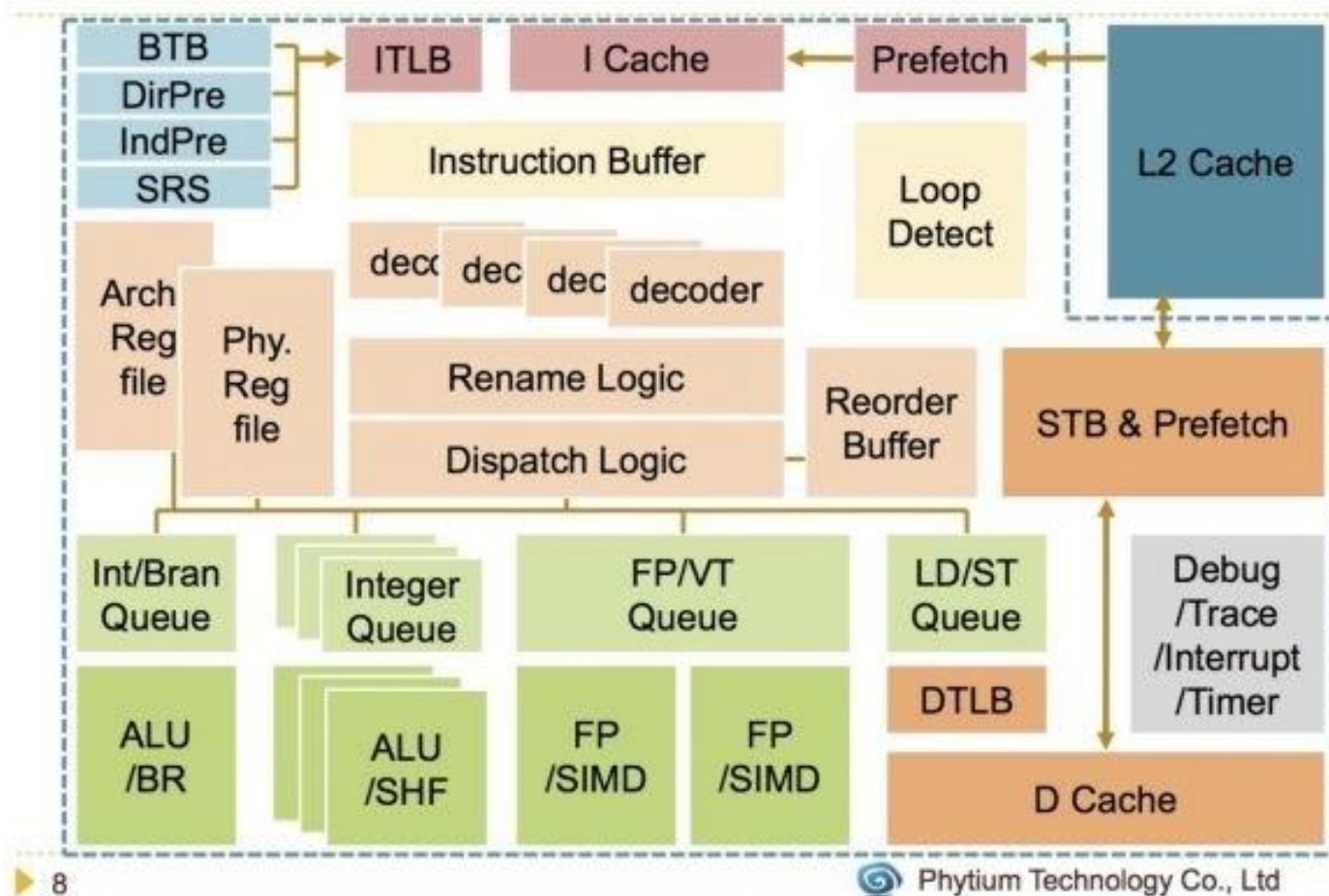


处理器体系结构

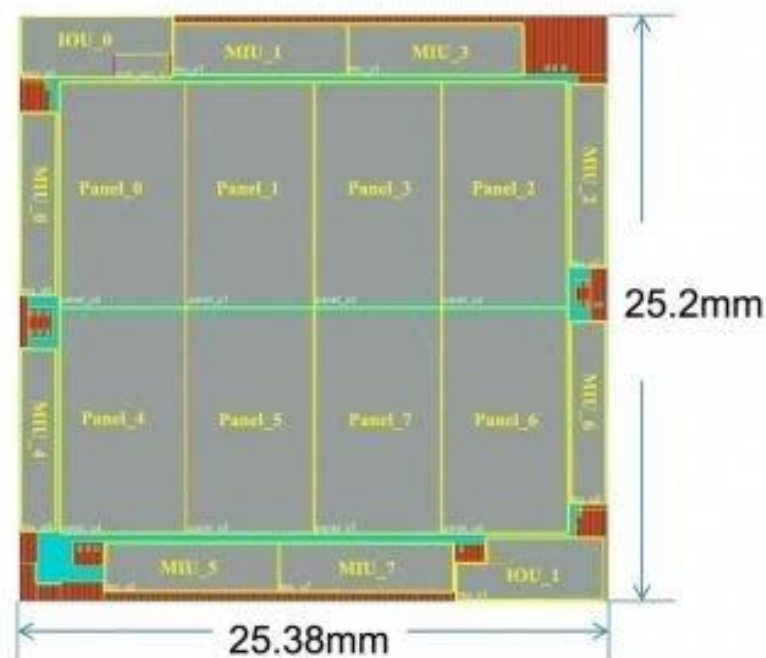
- **处理器必须执行一系列指令**
- **每条指令只执行简单的操作**
- **指令被编码为一个或者多个字节序列**
- **一个处理器的指令集体系结构 (ISA)**
- **使用者：仅看到指令**
- **处理器：执行这些指令**

计算机系统-处理器体系结构

Xiaomi Core



Physical Design

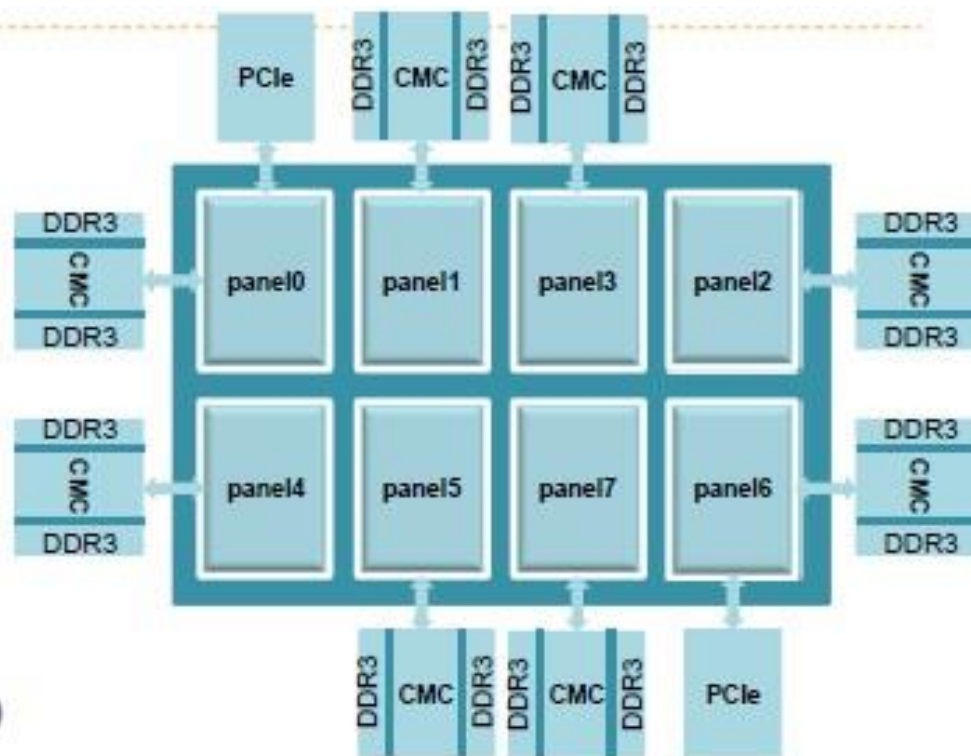


- ▶ 28nm process
- ▶ 0.9v core/1.8v IO
- ▶ 10 metal layers
- ▶ ~180M instances
- ▶ 2.0GHz
- ▶ 120W
- ▶ 640mm² die size
- ▶ FCBGA
- ▶ ~3000 pins

计算机系统-处理器体系结构

Architecture Features

- ▶ 64 Xiaomi cores, ARMv8 compatible
- ▶ Hardware-maintained global cache coherency
- ▶ Panel-based data affinity architecture
- ▶ Mesh topology on chip network
- ▶ 32MB L2 cache
- ▶ 8 Cache & Memory Chips (CMC)
 - ▶ 128MB L3 cache
 - ▶ 16 DDR3-1600 channels
- ▶ Two 16-lane PCIE3.0 i/f
- ▶ ECC and parity protection on all caches, tags and TLBs



Physical

- ~180M instances
- 2.0GHz@28nm
- 120W

Performance

- Peak: 512GFLOPS
- Mem BW: 204GB/s
- I/O BW: 32GB/s

4.1 Y86-64指令集体系结构

■指令集体系结构包括：

- 处理器状态定义
- 指令集和编码
- 编程规范
- 异常事件处理

4.1.1 程序员可见状态

- 每条指令都会读取或修改处理器状态的一部分，称为程序员可见状态

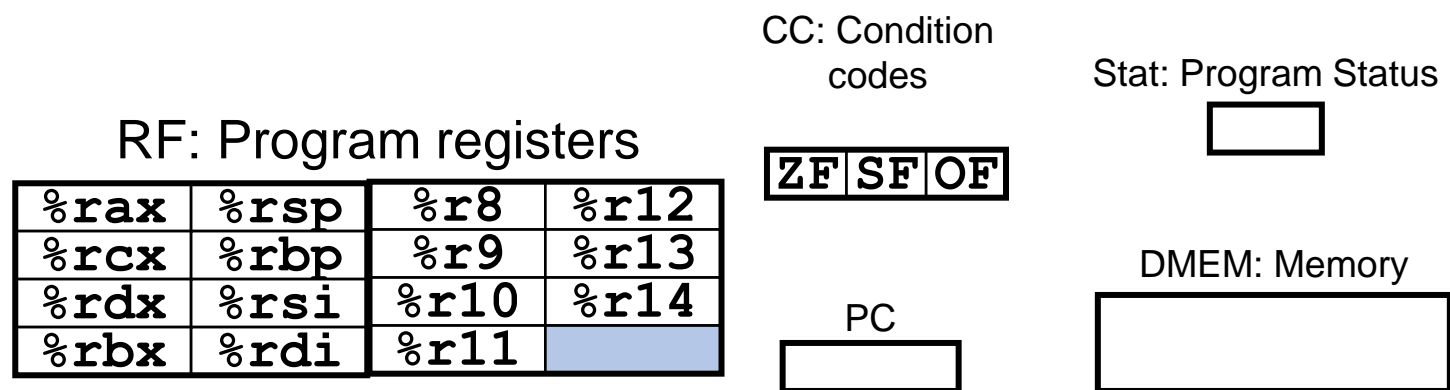


图4-1 Y86-64程序员可见状态

4.1.2 Y86-64指令

■是x86-64的子集

- 只包括8字节整数操作
- 较少的寻址方式

计算机系统-处理器体系结构

字节	0	1	2	3	4	5	6	7	8	9
halt	0	0								
nop	1	0								
rrmovq rA, rB	2	0	rA	rB						
irmovq V, rB	3	0	8	rB					V	
rmmovq rA, D(rB)	4	0	rA	rB					D	
mrmovq D(rB), rA	5	0	rA	rB					D	
OPq rA, rB	6	fn	rA	rB						
jXX Dest	7	fn							Dest	
cmovXX rA, rB	2	fn	rA	rB						
call Dest	8	0							Dest	
ret	9	0								
pushq rA	A	0	rA	F						
popq rA	B	0	rA	F						

图4-2 Y86-64指令集

4.1.3 指令编码

■ 指令=操作码+操作数

■ 一般第1个字节是操作码

➤ 高4位为代码部分

➤ 低4位为功能部分

运算指令		分支指令		传输指令															
addq	<table><tr><td>6</td><td>0</td></tr></table>	6	0	jmp	<table><tr><td>7</td><td>0</td></tr></table>	7	0	jne	<table><tr><td>7</td><td>4</td></tr></table>	7	4	rrmovq	<table><tr><td>2</td><td>0</td></tr></table>	2	0	cmovne	<table><tr><td>2</td><td>4</td></tr></table>	2	4
6	0																		
7	0																		
7	4																		
2	0																		
2	4																		
subq	<table><tr><td>6</td><td>1</td></tr></table>	6	1	jle	<table><tr><td>7</td><td>1</td></tr></table>	7	1	jge	<table><tr><td>7</td><td>5</td></tr></table>	7	5	cmovle	<table><tr><td>2</td><td>1</td></tr></table>	2	1	cmovge	<table><tr><td>2</td><td>5</td></tr></table>	2	5
6	1																		
7	1																		
7	5																		
2	1																		
2	5																		
andq	<table><tr><td>6</td><td>2</td></tr></table>	6	2	j1	<table><tr><td>7</td><td>2</td></tr></table>	7	2	jg	<table><tr><td>7</td><td>6</td></tr></table>	7	6	cmovl	<table><tr><td>2</td><td>2</td></tr></table>	2	2	cmovg	<table><tr><td>2</td><td>6</td></tr></table>	2	6
6	2																		
7	2																		
7	6																		
2	2																		
2	6																		
xorq	<table><tr><td>6</td><td>3</td></tr></table>	6	3	je	<table><tr><td>7</td><td>3</td></tr></table>	7	3			cmove	<table><tr><td>2</td><td>3</td></tr></table>	2	3						
6	3																		
7	3																		
2	3																		

图4-3 Y86-64指令集的功能码

4.1.3 指令编码

指令编码示例：

`rmmovq %rsp, 0x123456789abcd (%rdx)`

40 42 cd ab 89 67 45 23 01 00

数字	寄存器名字	数字	寄存器名字
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	无寄存器

`rmmovq rA, D(rB)`

4	0	rA	rB	D			
---	---	----	----	---	--	--	--

计算机系统-处理器体系结构

练习 4.1

rrmovq rA, rB

2	0	rA	rB
---	---	----	----

irmovq V, rB

3	0	F	rB	V
---	---	---	----	---

rmmovq rA, D(rB)

4	0	rA	rB	D
---	---	----	----	---

■ 写出指令字节编码

■ 方法提示:

- 首先确定每条指令的长度，写出地址
- 然后翻译字节码

数字	寄存器名字	数字	寄存器名字
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	无寄存器

```
.pos 0x100 # Start code at address 0x100
    irmovq $15,%rbx
    rrmovq %rbx,%rcx
loop:
    rmmovq %rcx,-3(%rbx)
    addq   %rbx,%rcx
    jmp   loop
```

练习 4.2

■从字节码，反汇编成汇编指令

- A. 0x100: 30f3fcffffffffffffffff40630008000000000000
- B. 0x200: a06f800c02000000000000000030f30a0000000000000090
- C. 0x300: 50540700000000000000000010f0b01f
- D. 0x400: 61137300040000000000000000
- E. 0x500: 6362a0f0

4.1.4 Y86-64异常

■程序员可见状态包括状态码Stat

值	名字	含义
1	AOK	正常操作
2	HLT	处理器执行halt指令
3	ADR	遇到非法地址
4	INS	遇到非法指令

图4-5 Y86-64状态码

■异常处理：简单停机或者进入异常处理程序

4.1.5 Y86-64程序

图4-6 Y86-64汇编与x86-64汇编比较

x86-64 code	Y86-64 code
<pre>long sum(long *start, long count) start in %rdi, count in %rsi 1 sum: 2 movl \$0, %eax sum = 0 3 jmp .L2 Goto test 4 .L3: loop: 5 addq (%rdi), %rax Add *start to sum 6 addq \$8, %rdi start++ 7 subq \$1, %rsi count-- 8 .L2: test: 9 testq %rsi, %rsi Test sum 10 jne .L3 If !=0, goto loop 11 rep; ret Return</pre>	<pre>long sum(long *start, long count) start in %rdi, count in %rsi 1 sum: 2 irmovq \$8,%r8 Constant 8 3 irmovq \$1,%r9 Constant 1 4 xorq %rax,%rax sum = 0 5 andq %rsi,%rsi Set CC 6 jmp test Goto test 7 loop: 8 mrmovq (%rdi),%r10 Get *start 9 addq %r10,%rax Add to sum 10 addq %r8,%rdi start++ 11 subq %r9,%rsi count--. Set CC 12 test: 13 jne loop Stop when 0 14 ret Return</pre>

4.3 Y86-64的顺序实现

- **每一条指令的执行，需要多个周期**
 - 可以理解为状态机的执行
- **一条指令执行完，下一条指令才能开始执行**
- **顺序实现（SEQ）很慢**

4.3.1 将处理组织成阶段

■ 通用的执行过程

- 取指 (Fetch) : 从存储器读取指令字节
- 译码 (Decode) : 从寄存器文件读入操作数
- 执行 (Execute) : ALU执行
- 访存 (Memory) : 读写存储器
- 写回 (WriteBack) : 结果写入寄存器
- 更新PC (Update PC) : 设置为下一条指令地址

4.3.1 将处理组织成阶段

- 所有指令需要“映射”到这个通用执行框架
- 可以共享硬件

阶段	OPq rA, rB	rrmovq rA, rB	irmovqV, rB
取指	icode: ifun \leftarrow M ₁ [PC] rA: rB \leftarrow M ₁ [PC+1] valP \leftarrow PC+2	icode: ifun \leftarrow M ₁ [PC] rA: rB \leftarrow M ₁ [PC+1] valP \leftarrow PC+2	icode: ifun \leftarrow M ₁ [PC] rA: rB \leftarrow M ₁ [PC+1] valC \leftarrow M ₈ [PC+2] valP \leftarrow PC+10
译码	valA \leftarrow R[rA] valB \leftarrow R[rB]	valA \leftarrow R[rA]	
执行	valE \leftarrow valB OP valA Set CC	valE \leftarrow 0 + valA	valE \leftarrow 0 + valC
访存			
写回	R[rB] \leftarrow valE	R[rB] \leftarrow valE	R[rB] \leftarrow valE
更新 PC	PC \leftarrow valP	PC \leftarrow valP	PC \leftarrow valP

图4-18 Y86-64指令的顺序实现

4.3.1 将处理组织成阶段

阶段	OPq rA, rB	subq %rdx, %rbx
取指	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC}+2$	$\text{icode:ifun} \leftarrow M_1[0x014]=6:1$ $\text{rA:rB} \leftarrow M_1[0x015]=2:3$ $\text{valP} \leftarrow 0x014+2=0x016$
译码	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	$\text{valA} \leftarrow R[\%rdx]=9$ $\text{valB} \leftarrow R[\%rbx]=21$
执行	$\text{valE} \leftarrow \text{valB OP valA}$ Set CC	$\text{valE} \leftarrow 21-9=12$ $\text{ZF} \leftarrow 0, \text{SF} \leftarrow 0, \text{OF} \leftarrow 0$
访存		
写回	$R[\text{rB}] \leftarrow \text{valE}$	$R[\%rbx] \leftarrow \text{valE}=12$
更新 PC	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}=0x016$

4.4 流水线的通用原理

- 把一个大的任务，切分为多个串行的小任务
- 每一部分硬件，仅负责一个小任务
- 大家可以一起执行
- 就像“接力棒”游戏一样

■ 流水线技术

- 提高了吞吐率（单位时间内执行指令的数目）
- 稍微增加了延迟（指令从入到出的时间）

计算机系统-处理器体系结构

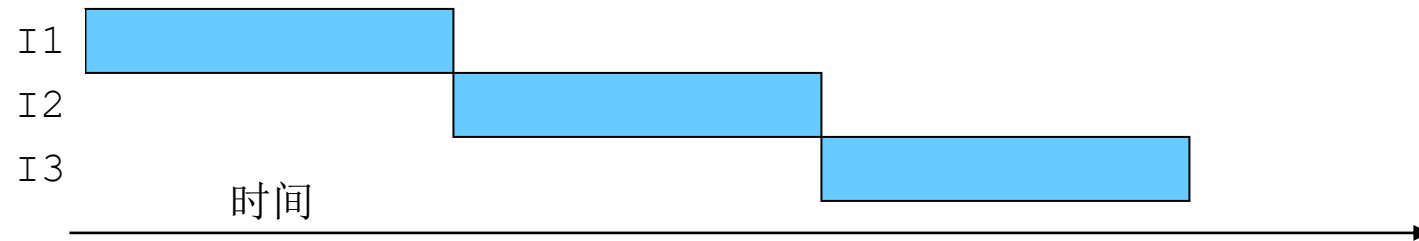
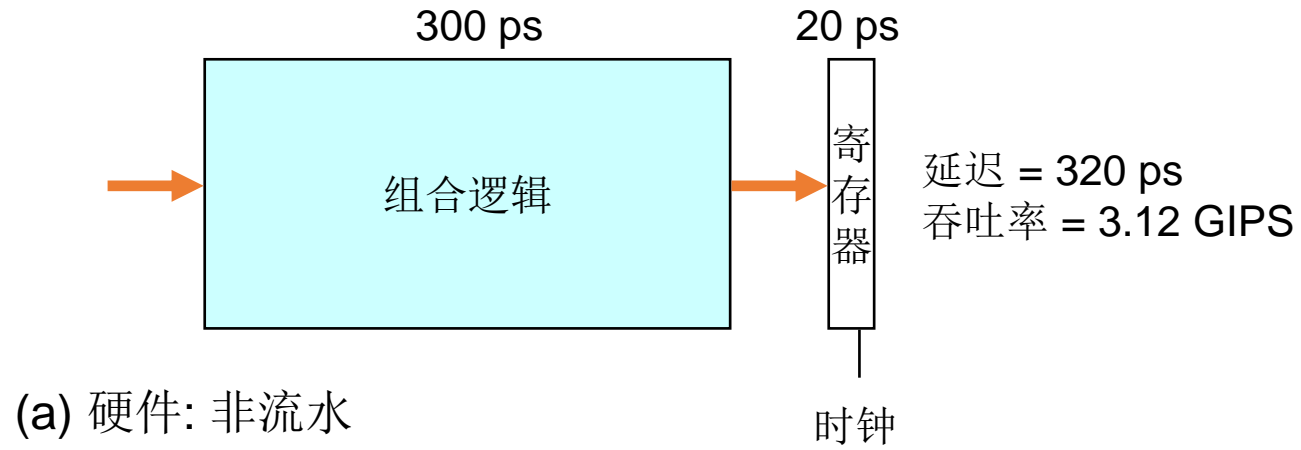


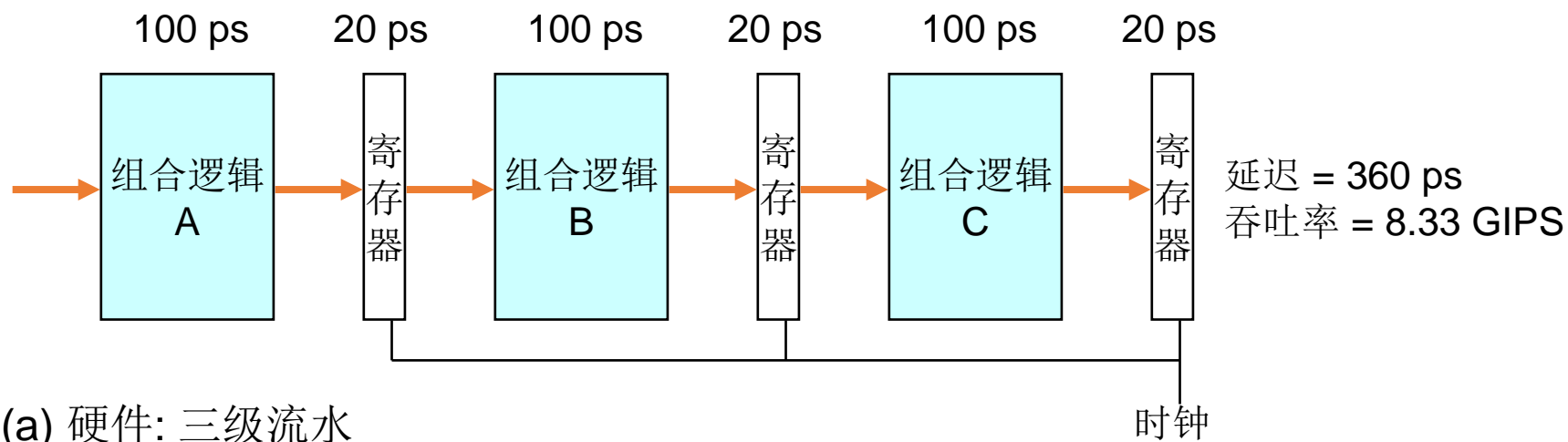
图4-32 非流水的计算硬件

4.4 流水线的通用原理

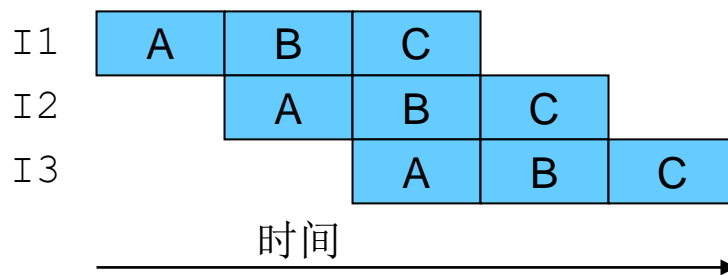
■吞吐率的计算（每秒x条指令）

$$\text{Throughput} = \frac{1 \text{ instruction}}{(20 + 300) \text{ picosecond}} \cdot \frac{1000 \text{ picosecond}}{1 \text{ nanosecond}} \approx 3.12 \text{ GIPS}$$

计算机系统-处理器体系结构



(a) 硬件: 三级流水



(b) 流水图

图4-33 三级流水的计算硬件

4.4.2 流水线操作的详细说明

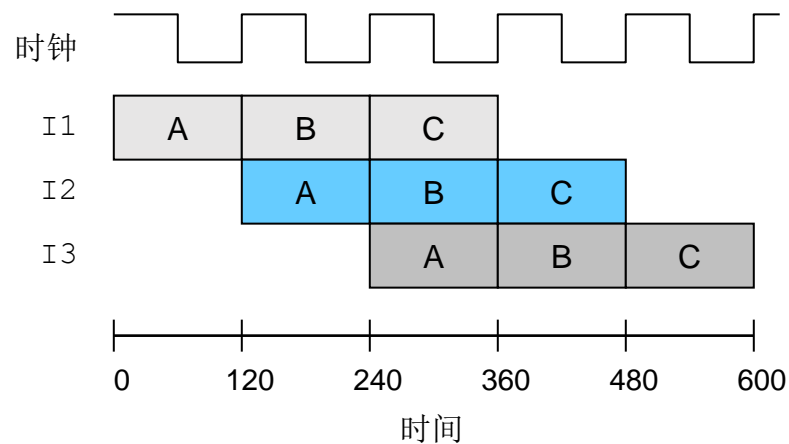
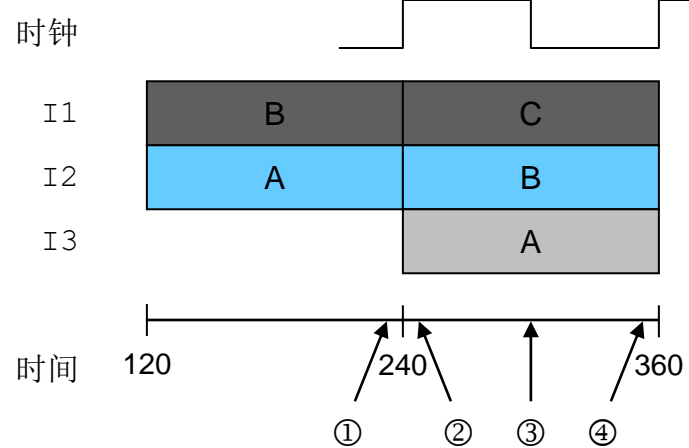
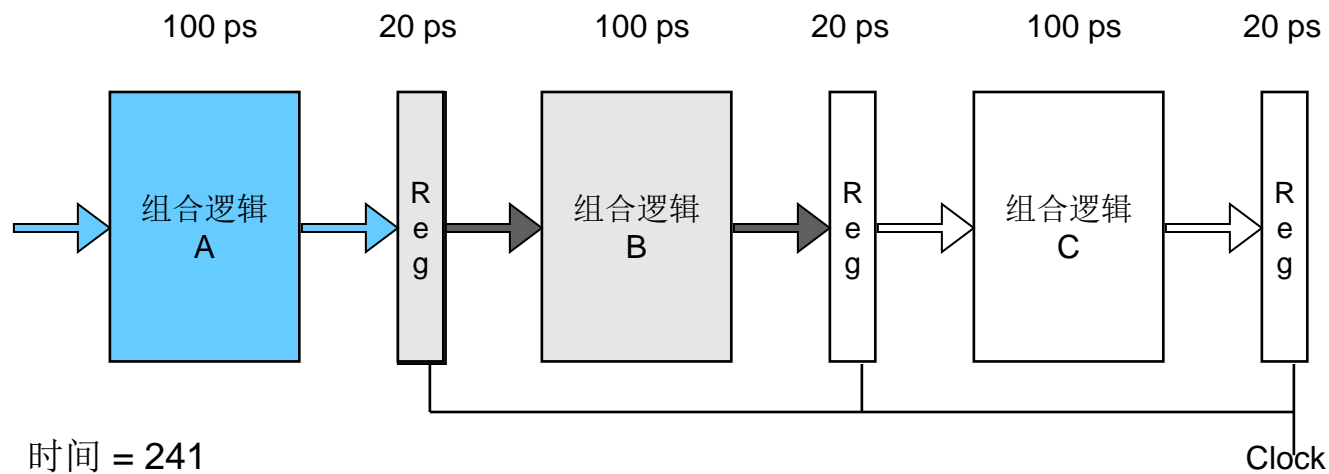


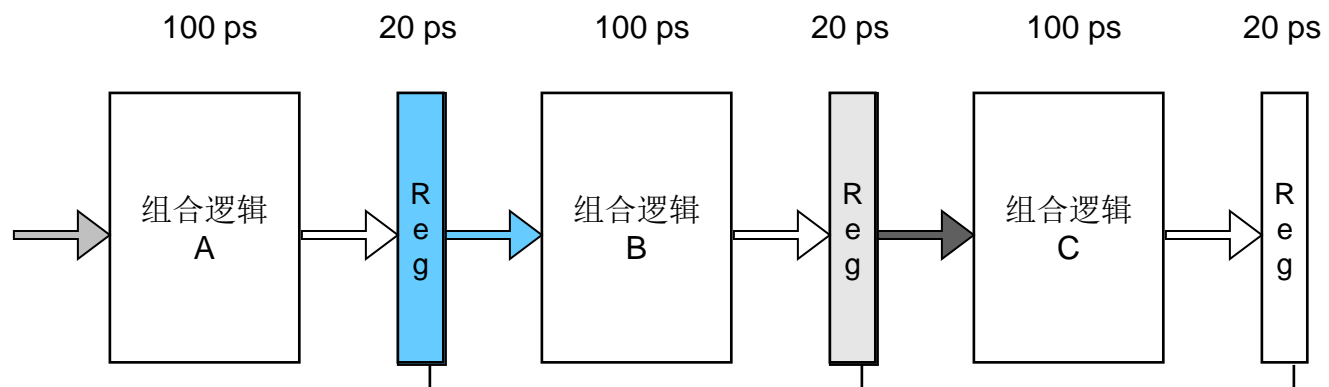
图4-34 三级流水线的时序



① 时间 = 239

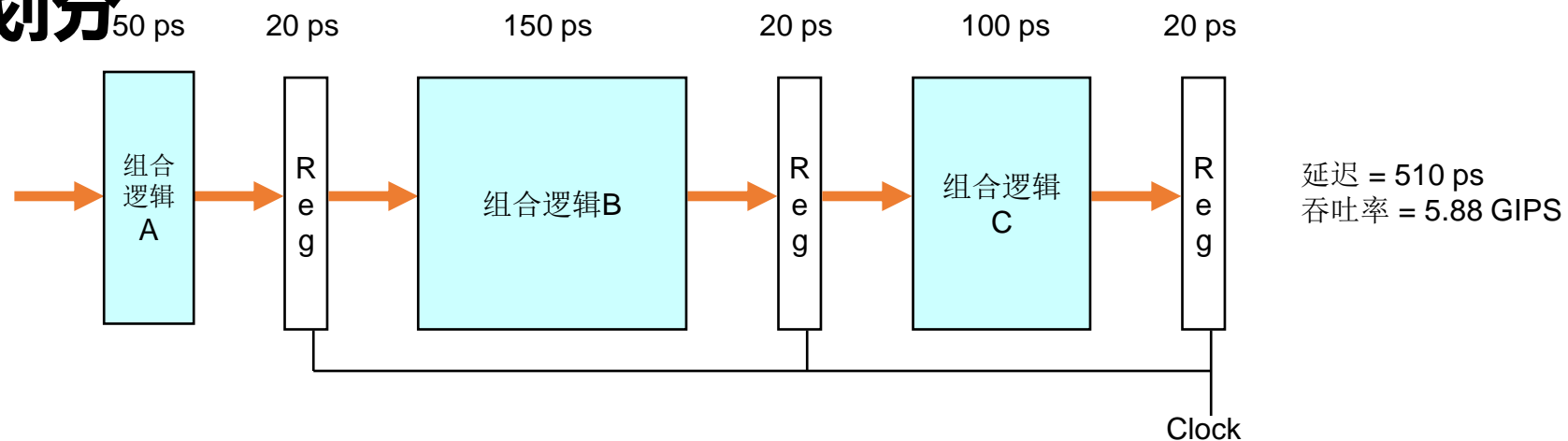


② 时间 = 241

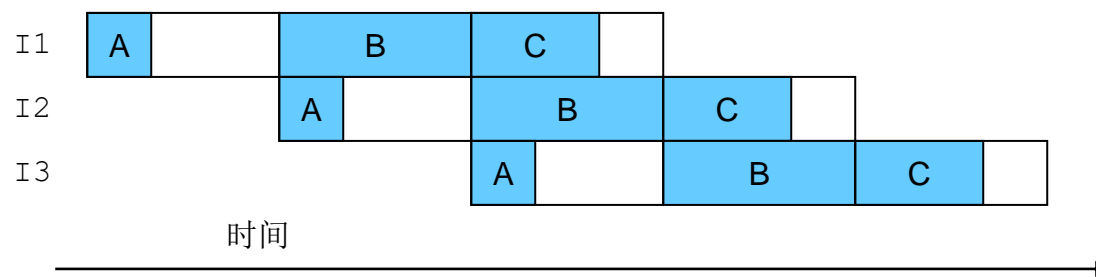


4.4.3 流水线的局限性

■ 不一致（平衡）的划分



(a) 硬件: 三级流水线, 不统一的流水站延迟



(b) 流水图

4.4.3 流水线的局限性

■流水线过深，收益反而下降

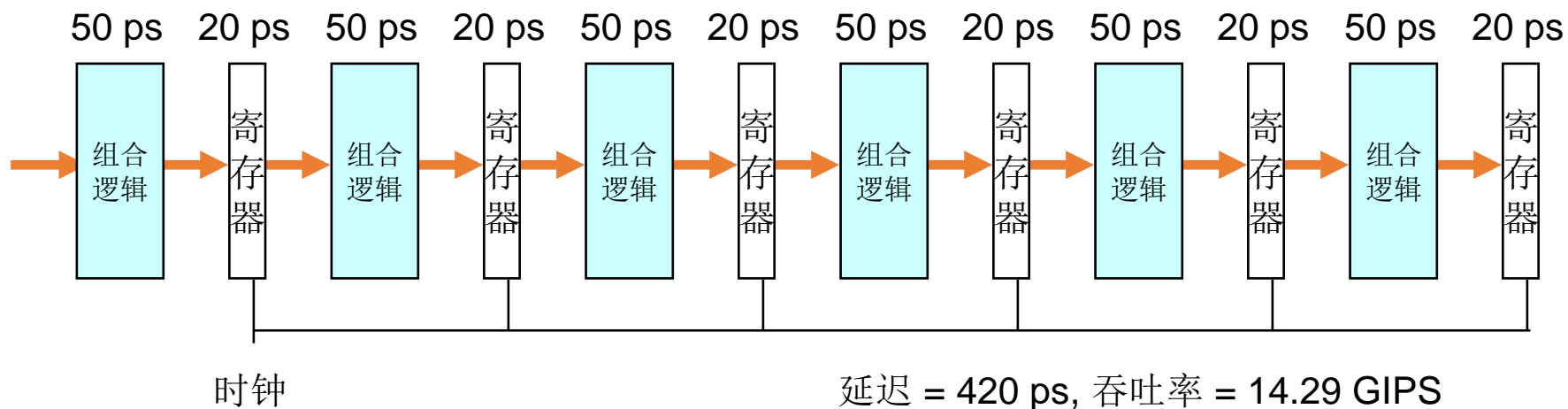
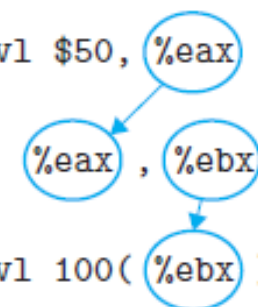


图4-37 过深的流水线

4.4.4 带反馈的流水线系统

■数据相关

```
1    irmovl $50, %eax
2    addl  %eax, %ebx
3    mrmovl 100(%ebx), %edx
```



```
1    irmovl $50,%eax
2    addl %eax,%ebx
3    mrmovl 100(%ebx),%edx
```

■控制相关

```
1    loop:
2        subl %edx,%ebx
3        jne targ
4        irmovl $10,%edx
5        jmp loop
6    targ:
7        halt
```

4.5 Y86-64处理器的流水线实现

■ 自学

4.6 小结

- **指令集体系结构ISA**，定义了一个抽象的接口层
- **Y86-64是简化的x86-64**，执行划分为5段，SEQ是最基本的实现
- **为提高性能，引入流水线**，提高系统吞吐率

实验

■使用Python/C/C++语言实现一个Y86-64模拟器

- 能够从文件中读取Y86-64机器码
- 能够解释执行Y86-64指令
- 能够将存储器/寄存器值输出
- 提示：可采用多周期、流水线实现

■测试：

- 使用Y86-64指令实现从1加到100，并将结果保存到存储器中

分支预测

- 现代处理器，多是超标量流水线处理器
- 有多条流水线同时运行
- 如果出现分支，则可能带来较大的性能损失
- 需要在取指阶段，就能准确“预测”到分支指令是否taken
- 如果猜测正确，则流水线不停顿
- 如果猜测错误，则流水线需要恢复
- 具有非常高的猜测准确度，是非常重要的

分支预测

■ 处理器结构--分支预测(Branch Prediction)

➤ <https://www.jianshu.com/p/be389eeba589>

■ 分支预测器

➤ <https://zh.wikipedia.org/wiki/%E5%88%86%E6%94%AF%E9%A0%90%E6%B8%AC%E5%99%A8>

■ 深入理解 CPU 的分支预测(Branch Prediction)模型

➤ <https://blog.csdn.net/hanzefeng/article/details/82893317>

分支预测实验

- 使用提供的分支预测框架、执行轨迹 (Trace) , 设计并实现分支预测算法
- 需要评测对给定Trace, 其分支预测准确率是多少
- 记住:
 - 你只有很短的时间 (1个时钟周期)
 - 你只有很少的资源 (数K存储)

基本描述

■现代CPU都是流水线处理器

■在取指段，CPU知道：

- 取回来的指令是否是分支指令、分支的目标地址是多少

`char GetPrediction(UINT64 PC); // 返回T或者N`

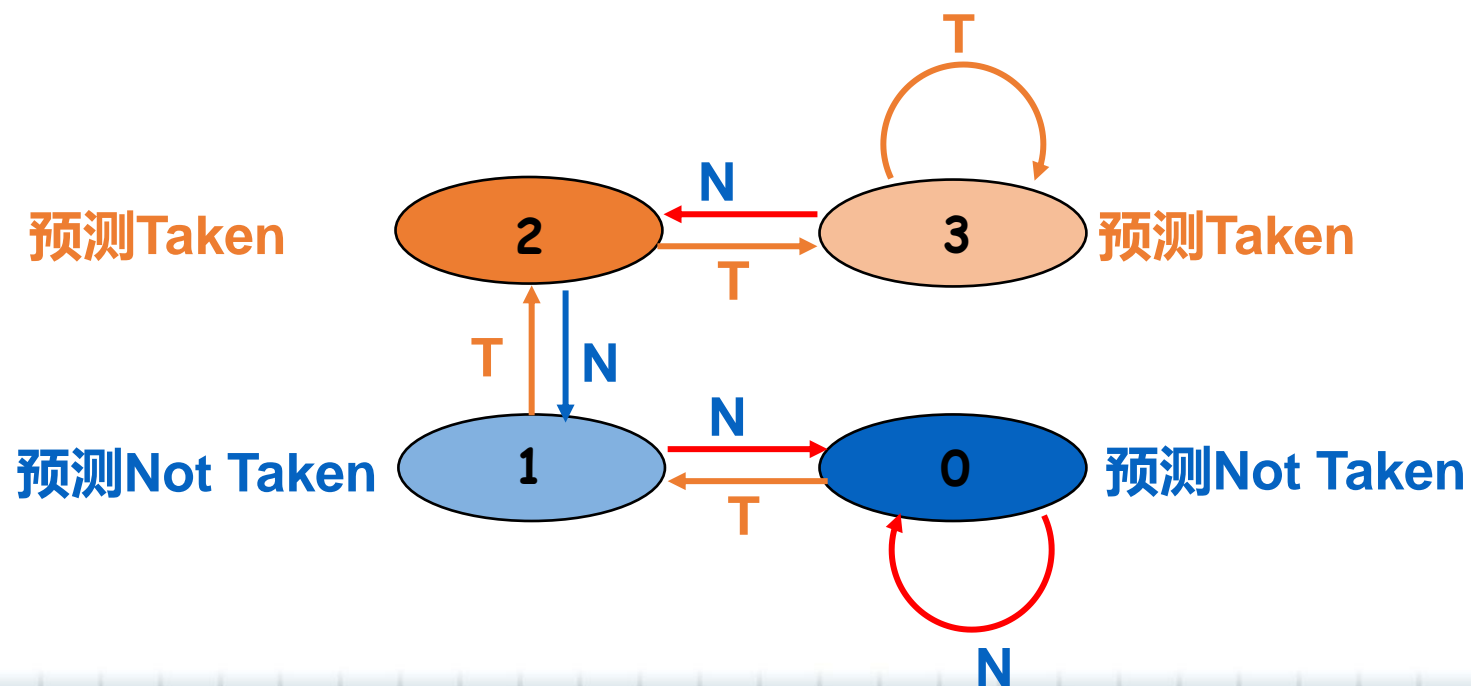
■在执行段，CPU知道：

- 该分支指令是否跳转（taken）
- //更新内部状态，以便下次更好的预测

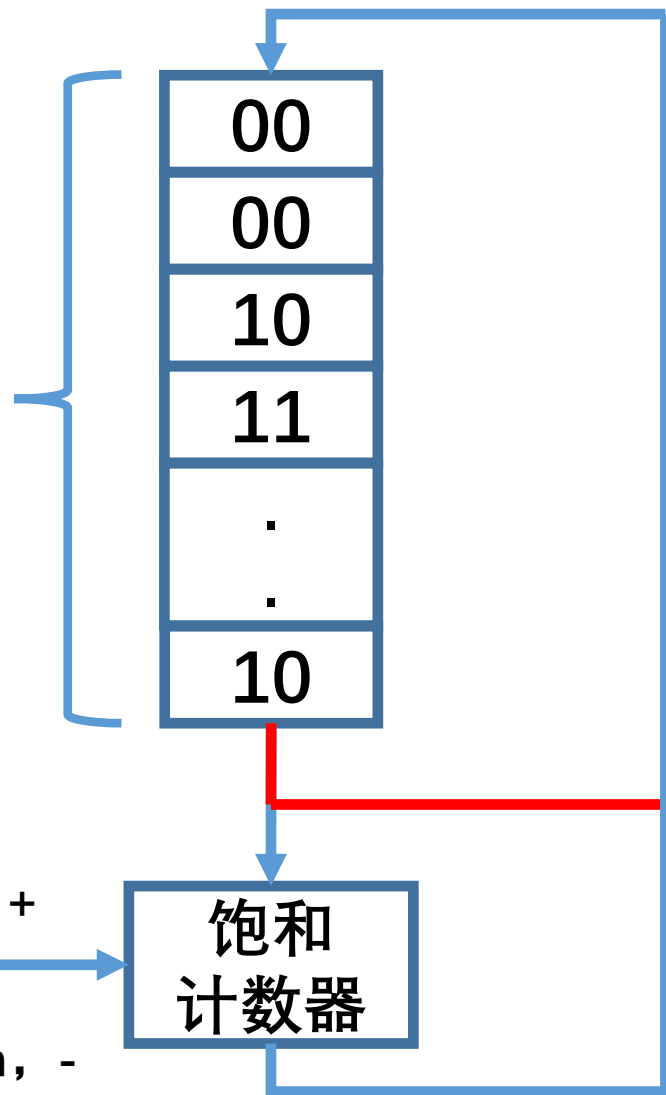
`void UpdatePredictor(UINT64 PC, OpType opType, char resolveDir, char predDir, UINT64 branchTarget)`

思路

- 本次预测结果，应当参考该指令以前是否跳转（历史信息）



条件分支指令的地址 (PC)



2b X 2¹³内存

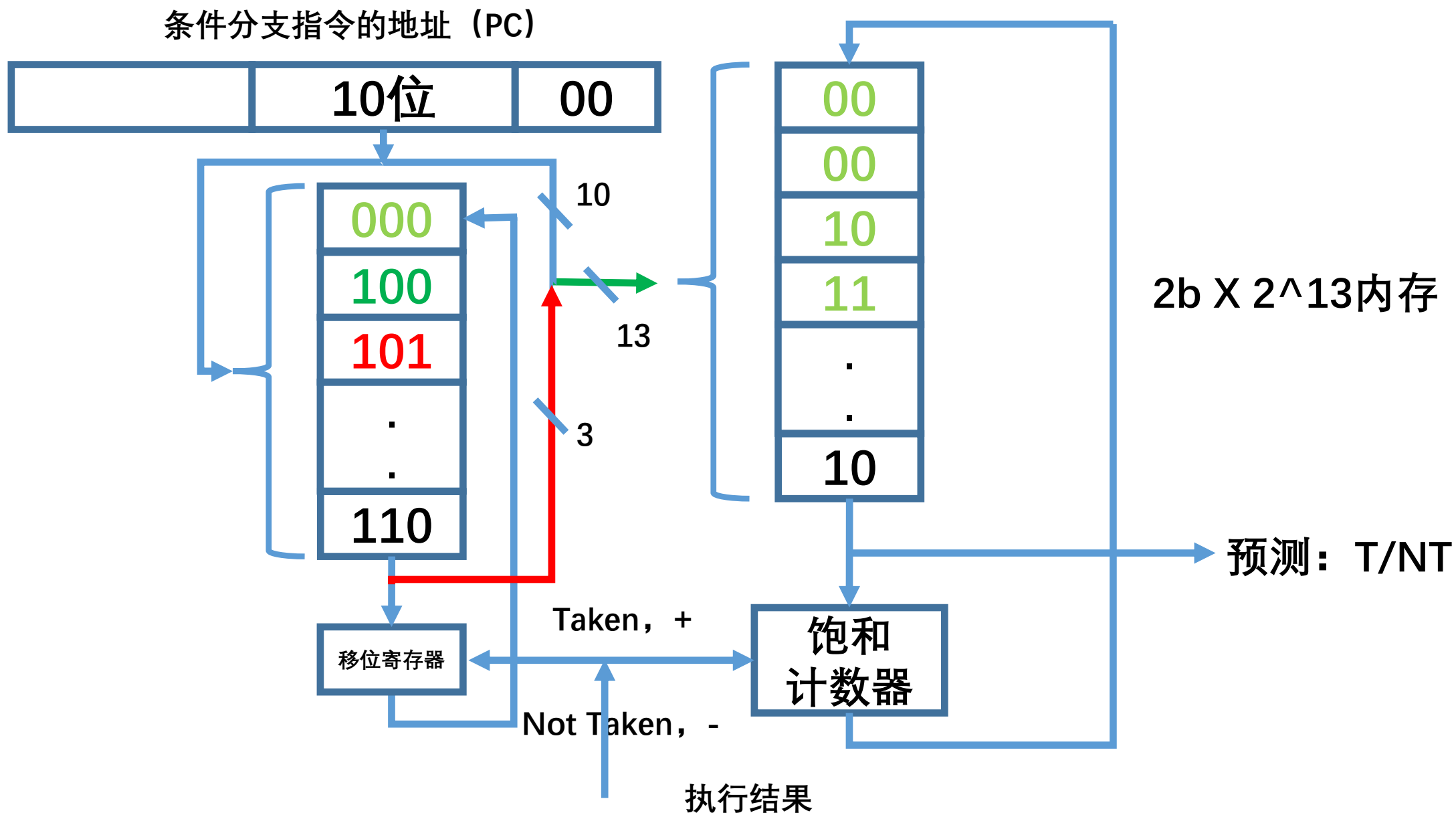
GetPrediction

预测: T/NT

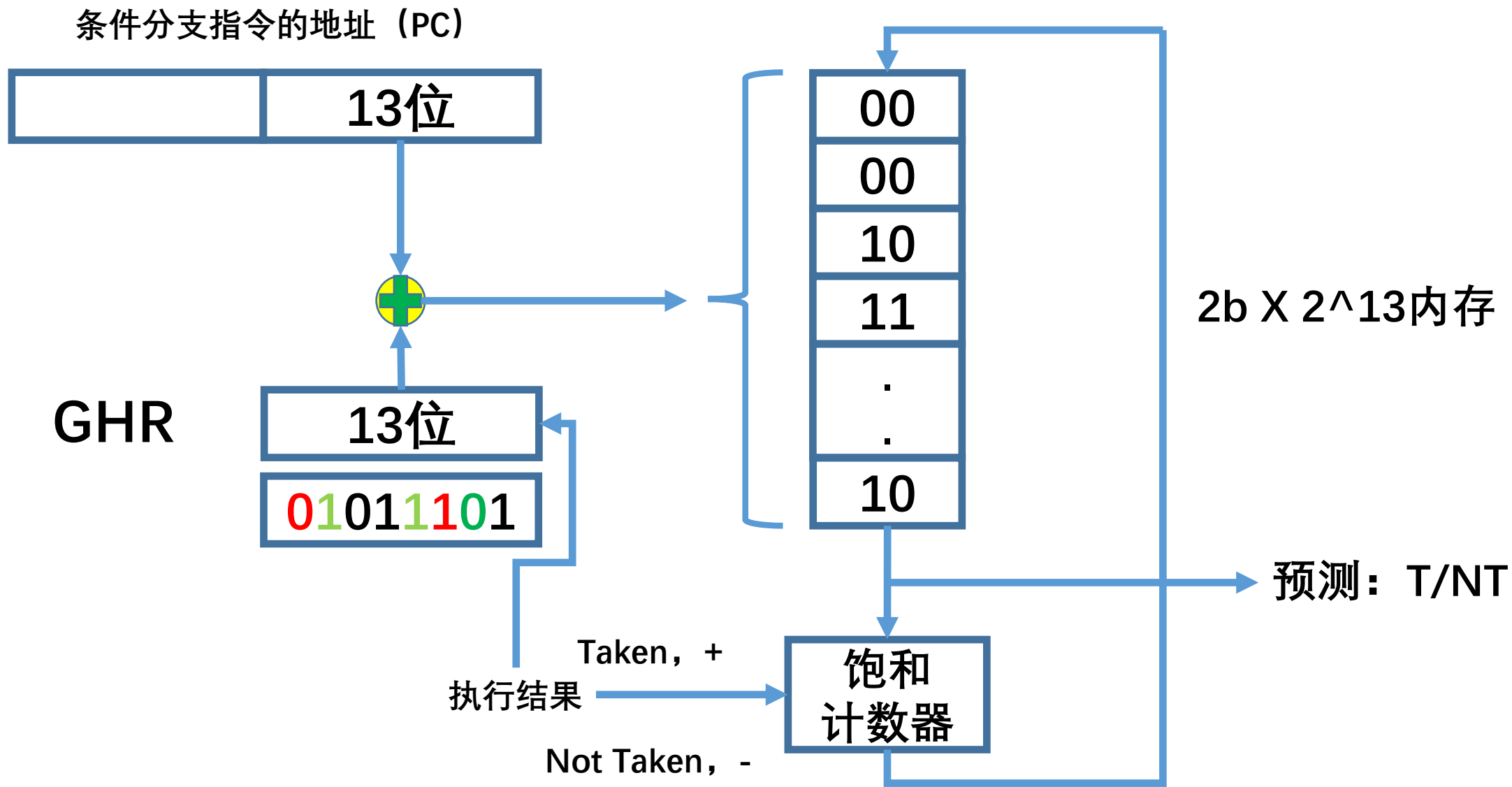
Taken, +
执行结果
Not Taken, -

UpdatePredictor

局部历史信息



Gshare



分支预测器实验

- 实验目标1：实现1位的分支预测器，并测试
- 实验目标2：扩展全局分支历史长度，并测试
- 实验目标3：为每个分支指令，分配完全独立的状态机，并测试
- 实验目标4：扩展为3位状态机，并测试
- 实验目标5：实现局部历史信息，并测试
- 实验目标6：实现局部+全局历史信息，并测试
- 实验目标7：考虑不用异或，使用其他hash函数（比如移位、加法）来实现，并测试
- *有人用神经网络把trace学习了一遍，达到了极其惊人的准确率！
- *请想办法优化框架的性能：每次读取trace、格式分析，太慢了~