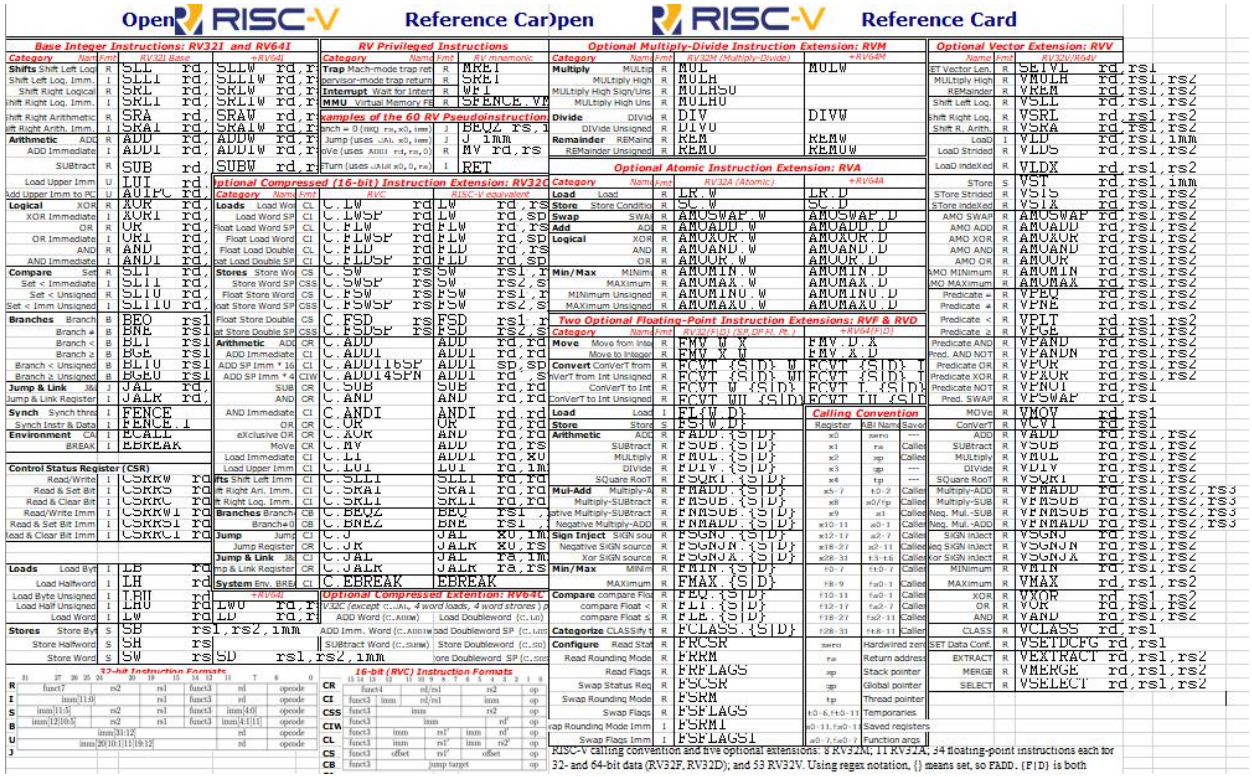
RISCV Core
基础知识简介

Part 1
RISCV ISA导览

RISCV架构设计哲学

大道至简



RISCV ISA Reference Card

- 架构文档的篇幅短小精悍

  指令集文档：200多页

  特权架构文档：90多页

- 模块化指令集

  用户可灵活选择不同模块进行组合，通过一套统一的架构满足不同场景的应用

- 指令的数目简洁

  基本指令数目仅有40多条，加上其他模块化扩展指令总共几十条

X86, ARM 随着时间推移指令集越发臃肿，文档厚度堪比字典

什么是 ISA（Instruction Set Architecture）

◆数据类型

◆存储模型

◆软件可见的处理器状态
- 通用寄存器（General registers）
- PC（Program Counter）
- 处理器状态（Processor status）

◆指令集
- 指令类型与编码（Instructions and formats）
- 寻址模式（Addressing modes）
- 数据结构（Data structures）

◆系统模型
- 状态（States）
- 特权级别（ Privilege Level）
- 中断和异常（ Interrupts and Exceptions）

◆外部接口
- 输入输出接口（ IO ）
- 管理（ Management ）

RISCV ISA架构特点

模块化的指令子集

- RISC-V的指令集使用模块化的方式进行组织，每一个模块使用一个英文字母来表示
- RISC-V最基本也是唯一强制要求实现的指令集部分是由I字母表示的基本整数指令子集

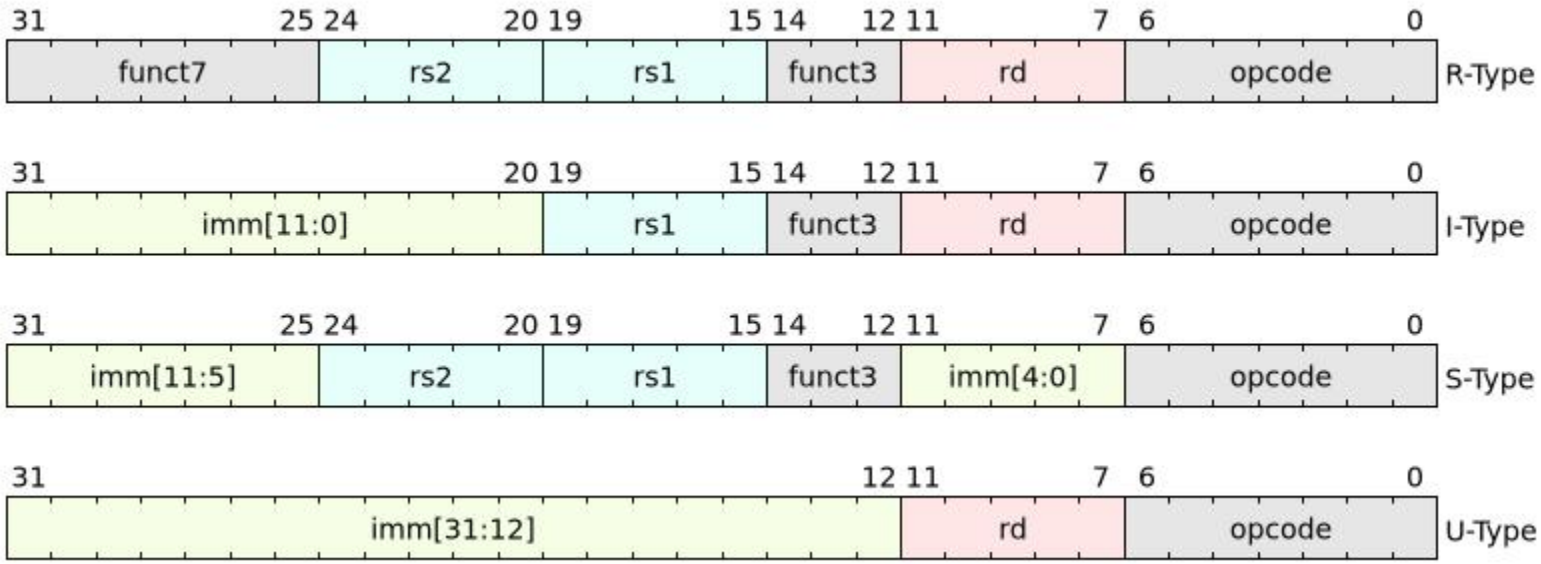
基本指令集	指令数	描　述
RV32I	47	32位地址空间与整数指令，支持32个通用整数寄存器
RV32E	47	RV32I的子集，仅支持16个通用整数寄存器
RV64I	59	64位地址空间与整数指令及一部分32位的整数指令
RV128I	71	128位地址空间与整数指令及一部分64位和32位的指令

扩展指令集	指令数	描　述
M	8	整数乘法与除法指令
A	11	存储器原子（Atomic）操作指令和Load-Reserved/Store-Conditional指令
F	26	单精度（32比特）浮点指令
D	26	双精度（64比特）浮点指令，必须支持F扩展指令

RISCV ISA 指令集官方扩展

Identifier	Standard Extension	Status	This Design
A	Atomic Instructions	Ratified	Partial
B	Bit Manipulation	Draft	Partial
C	Compressed Instructions	Ratified	Complete
Counters	Counters	Draft	Partial
D	Double-Precision Floating Point	Ratified	
F	Single-Precision Floating Point	Ratified	
H	Hypervisor Extension	Draft	
I	Base Instruction Set	Ratified	Complete
J	Dynamically Translated Languages	Draft	
K	Scalar Cryptography	Draft	
L	Decimal Floating Point	Draft	
M	Integer Multiplication and Division	Ratified	
N	User-level Interrupts	Draft	
P	Packed-SIMD	Draft	
Q	Quad-Precision Floating Point	Ratified	
T	Transactional Memory	Draft	
V	Vector Operations	Draft	
Zam	Misaligned Atomics	Draft	Complete
Zicsr	Control and Status Register Instructions	Ratified	Complete
Zifencei	Instruction-Fetch Fence	Ratified	Complete
Zihintpause	Pause Hint	Ratified	
Ztso	Total Store Ordering	Frozen	

RISCV ISA 指令格式



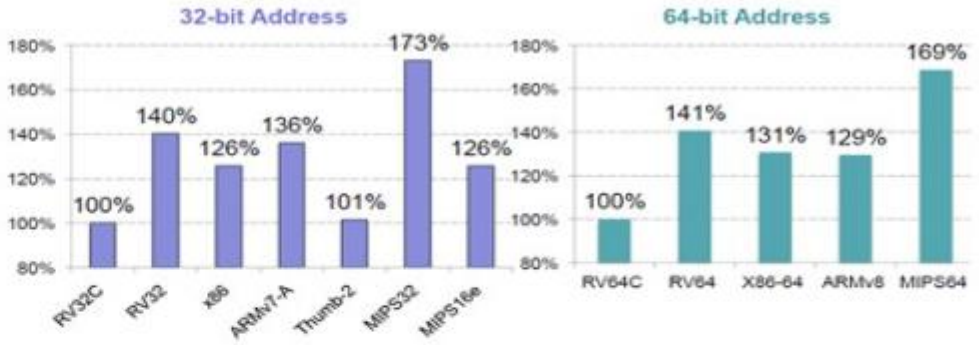
RISCV ISA 寄存器 ABI

rd, rs1, rs2 encoding	Register Name	Register ABI Name	Register ABI Description
00000	x0	zero	Hard-wired zero
00001	x1	ra	Return address
00010	x2	sp	Stack pointer
00011	x3	gp	Global pointer
00100	x4	tp	Thread pointer
00101	x5	t0	Temporary register 0
00110	x6	t1	Temporary register 1
00111	x7	t2	Temporary register 2
01000	x8	s0/fp	Saved reg 0/Frame pointer
01001	x9	s1	Saved register 1
01010	x10	a0	Function return value 0
01011	x11	a1	Function return value 1
01100	x12	a2	Function argument 2
01101	x13	a3	Function argument 3
01110	x14	a4	Function argument 4
01111	x15	a5	Function argument 5
10000	x16	a6	Function argument 6
10001	x17	a7	Function argument 7
10010	x18	s2	Saved register 2
10011	x19	s3	Saved register 3
10100	x20	s4	Saved register 4
10101	x21	s5	Saved register 5
10110	x22	s6	Saved register 6
10111	x23	s7	Saved register 7
11000	x24	s8	Saved register 8
11001	x25	s9	Saved register 9
11010	x26	s10	Saved register 10
11011	x27	s11	Saved register 11
11100	x28	t3	Temporary register 3
11101	x29	t4	Temporary register 4
11110	x30	t5	Temporary register 5
11111	x31	t6	Temporary register 6

RISCV ISA架构特点

优雅的压缩指令子集

- 16位的压缩指令有其对应的普通32位指令



自定义指令扩展

- 用户可扩展自己的指令子集

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 ($>32b$)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	$48b$
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	$64b$
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	$48b$
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq 80b$

RISCV软件工具链

RISC-V软件工具链

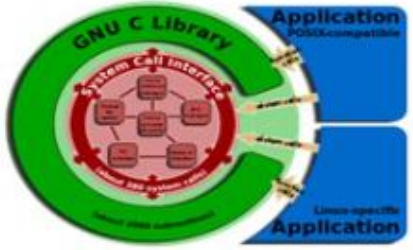
RISC-V软件工具链由开源社区维护，可通过RISC-V基金会网站找到相关信息，并下载。

riscv-tools （ISA Simulator and Tests）

- riscv-isa-sim：基于C/C++开发的指令集模拟器“Spike”
- riscv-openocd：基于OpenOCD的RISC-V调试器软件
- riscv-opcodes：RISC-V操作码信息转换脚本
- riscv-tests：一组RISC-V指令集测试用例
- riscv-pk：RISC-V可执行程序运行环境软件，同时提供最简单的bootloader

riscv-gnu-toolchain

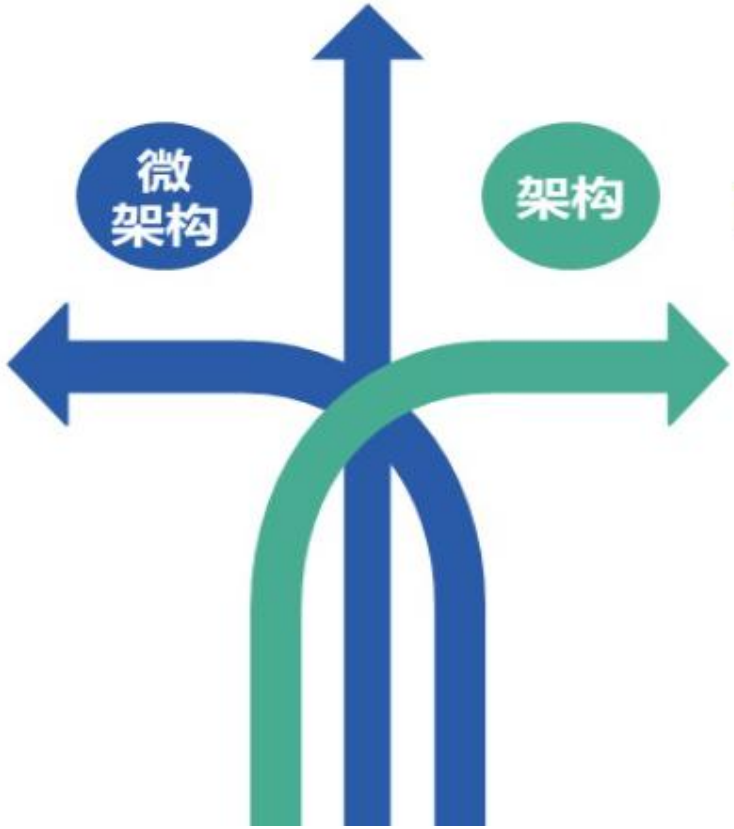
- riscv-gcc：GCC编译器。
- riscv-binutils：二进制工具（链接器，汇编器等）
- riscv-gdb：GDB调试工具
- riscv-glibc：GNU C标准库实现
- riscv-newlib：开源C标准库，主要用于嵌入式系统
- riscv-qemu：支持RISC-V的QEMU模拟器



RISCV 实现简介

RISC-V处理器内核

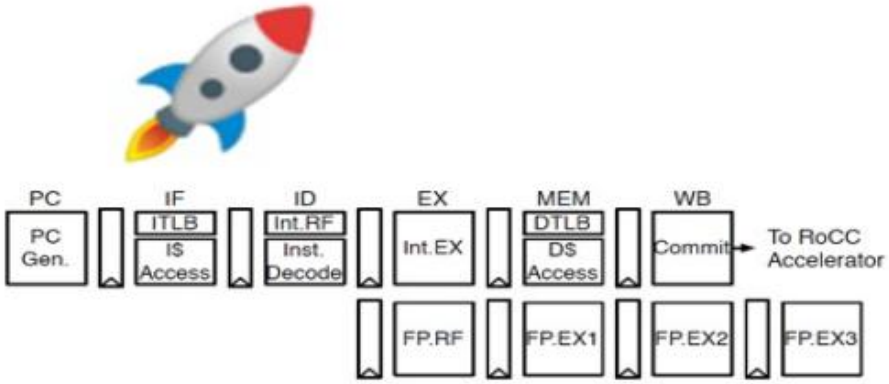
自RISC-V架构诞生以来，已经出现了数十个版本的RISC-V架构处理器，有开源免费的，也有商业公司开发用于内部项目的，还有商业IP公司开发的RISC-V处理器IP。



RISC-V指令集构架

RISC-V是一种开放的指令集架构，而不是一款具体的处理器。任何组织与个人均可以依据RISC-V架构设计实现自己的处理器，只要是依据RISC-V标准设计的处理器，都可称为RISC-V处理器。

RISCV 实现简介



Category	ARM Cortex-A5	RISC-V Rocket
ISA	32-bit ARM v7	64-bit RISC-V v2
Architecture	Single-Issue In-Order	Single-Issue In-Order 5-stage
Performance	1.57 DMIPS/MHz	1.72 DMIPS/MHz
Process	TSMC 40GPLUS	TSMC 40GPLUS
Area w/o Caches	0.27 mm^2	0.14 mm^2
Area with 16K Caches	0.53 mm^2	0.39 mm^2
Area Efficiency	2.96 DMIPS/MHz/mm^2	4.41 DMIPS/MHz/mm^2
Frequency	>1GHz	>1GHz
Dynamic Power	<0.08 mW/MHz	0.034 mW/MHz

Rocket Core（开源）

Rocket Core是Berkeley 开发的一款开源64位 RISC-V 处理器核，可以由伯克利开发的**SoC生成器Rocket-Chip**生成。

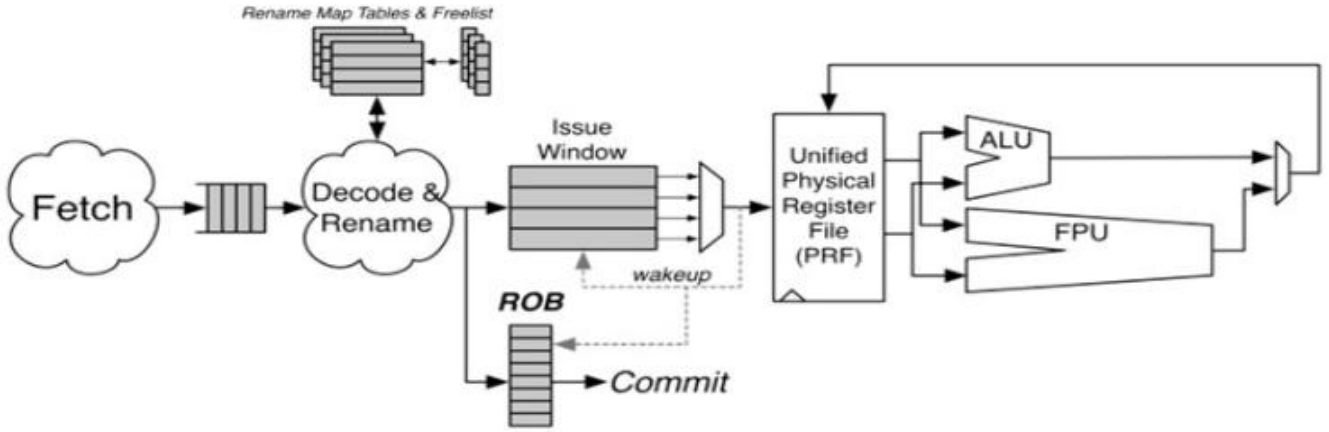
- 具备可配置性，支持多种RISC-V的指令集扩展组合
- 配备可扩展指令接口（Rocket Custom Coprocessor，RoCC）
- 使用Chisel（Constructing Hardware in an ScalaEmbedded Language）语言进行开发
- 成功进行了多次投片，且在芯片原型上成功运行了Linux操作系统

RISCV 实现简介

BOOM Core （开源）

BOOM（Berkeley Out-of-Order Machine）Core也是伯克利开发的一款开源RISC-V处理器核，也用Chisel语言开发，同样Rocket-Chip生成。
BOOM Core面向更高的性能目标，是一款**超标量乱序发射、乱序执行的处理器核，**配备了高性能的分支预测器，I-Cache、D-Cache和FPU，且支持多核结构，L2 Cache和多核Cache一致性（Coherency）。

Category	ARM Cortex-A9	RISC-V BOOM-2w
ISA	32-bit ARM v7	64-bit RISC-V v2 (RV64G)
Architecture	2 wide, 3+1 issue Out-of-Order 8-stage	2 wide, 3 issue Out-of-Order 6-stage
Performance	**3.59** CoreMarks/MHz	**4.61** CoreMarks/MHz
Process	TSMC 40GPLUS	TSMC 40GPLUS
Area with 32K caches	2.5 mm²	1.00 mm²
Area efficiency	1.4 CoreMarks/MHz/mm²	4.6 CoreMarks/MHz/mm²
Frequency	1.4 GHz	1.5 GHz



RISCV 实现简介

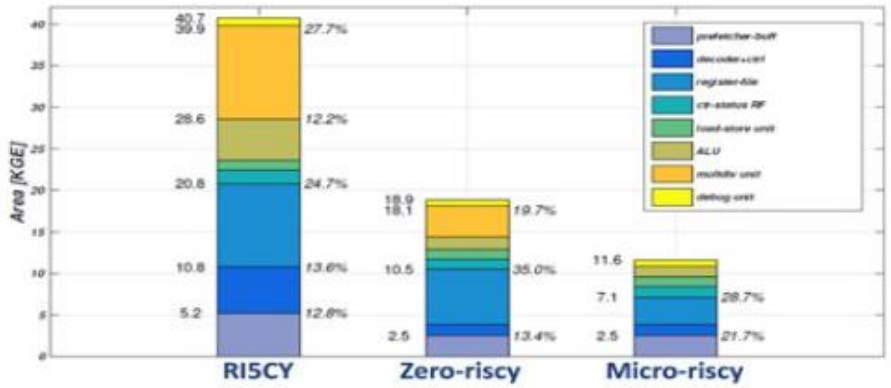
Freedom SoC（开源）

SiFive公司是由伯克利几个主要的RISC-V发起人所创办，旨在进行RISC-V架构的处理器开发与服务的商业公司。Freedom Everywhere 310 SoC是由SiFive公司推出的一款开源SoC。

LowRISC SoC（开源）

LowRISC是一个非营利组织，同时也是由剑桥大学的开发者基于Rocket Core而开发的一款开源SoC平台名称。LowRISC组织的愿景是成为“硬件世界的Linux”，目标是提供高质量、安全、开放的平台，计划实现量产芯片并提供低成本的开发板。

RISCV 实现简介



PULPino Core and SoC（开源）

PULPino是由苏黎世瑞士联邦理工学院（ETH Zurich）开发的一款开源的单核MCU SoC平台。

ETH Zurich还开发了多款32位RISC-V处理器核：

- RI5CY：四级流水线，按序单发射处理器，支持RV32I，还可配置C、M、F指令子集，还增加了很多自定义指令用于低功耗DSP应用。
- Zero-riscy：二级流水线，按序单发射处理器，支持RV32I，还可配置C、M，甚至可以被配置成RV32E。主要面向超低功耗、超小面积的场景。
- Micro-riscy：面积更小的处理器核，仅支持RV32EC，并且无硬件乘除法单元，面积小于12K个逻辑门。

RISCV 实现简介

PicoRV32 Core（开源）

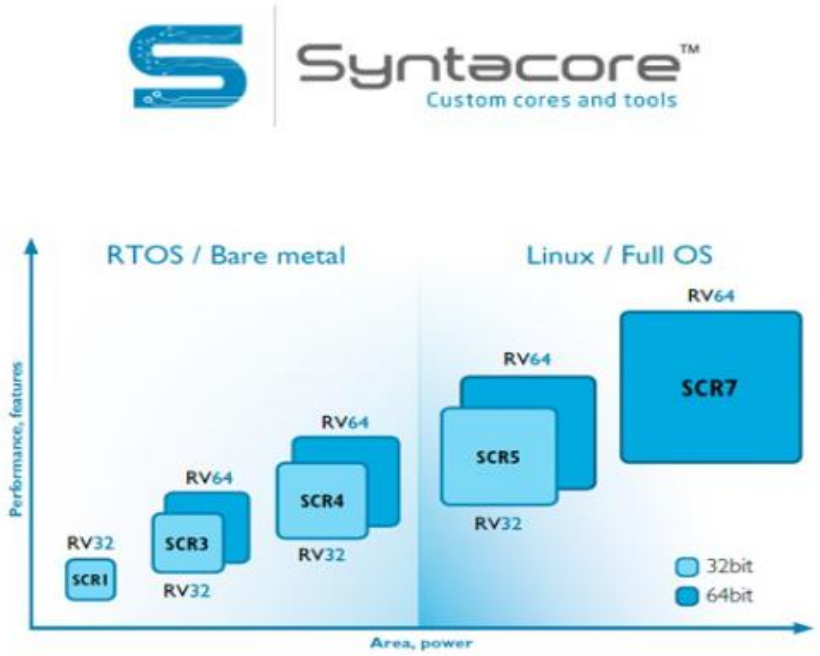
PicoRV32是一款由著名的IC设计师Clifford Wolf开发并开源的一款RISC-V处理器核，其重点在于追求面积和频率的优化，此处理器核明确说明它是为面积做优化。其公布的数据在Xilinx7-Series FPGA上的开销为750-2000 LUTs，并且能够综合到250～450MHz的主频。

VectorBlox
embedded supercomputing

ORCA Core（开源）

ORCA是一款由Vectorblox公司用VHDL语言设计编写的面向FPGA的开源RISC-V处理器核，可以配置成为RV32I或RV32IM，其诞生初衷是为了能够作为主控制处理器和Vectorblox公司的商用协处理器适配使用。

RISCV 实现简介



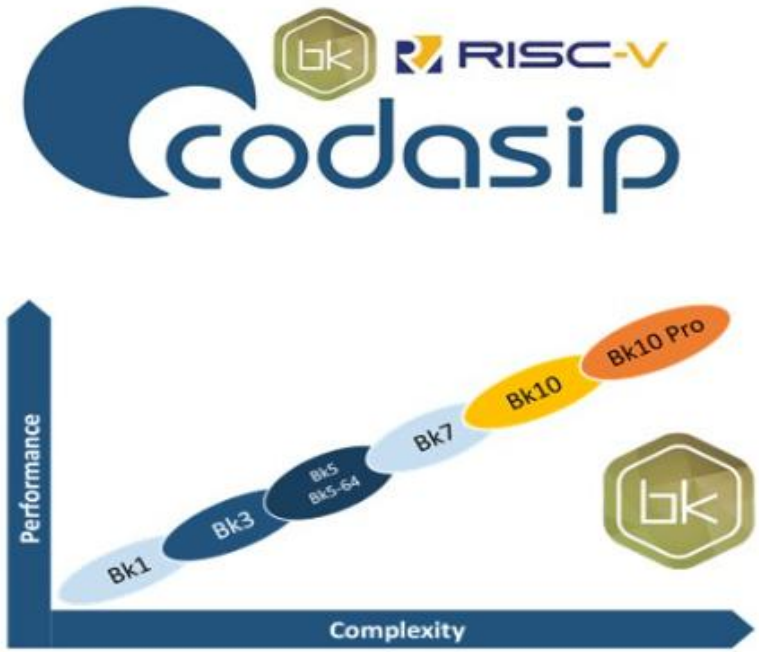
SCR1（开源）

SCR1是一款由俄罗斯Syntacore公司用System Verilog语言设计编写的一款极低功耗开源RISC-V处理器核。

- 具有可配置性，可配置为RV32I/EMC指令子集组合
  - ✓ 最小配置RV32EC的面积开销为12K个逻辑门
  - ✓ 最高的RV32IMC配置面积开销约为28K个逻辑门
- SCR1仅支持机器模式，还配备了可选的中断控制器与调试（Debugger）模块。

Syntacore专注于为客户定制开发和授权具有高能效比的可综合可编程处理器核，其基于RISC-V架构开发了多款处理器核**SCRx系列**，并将SCR1开源。

RISCV 实现简介



Codasip Core（商业）

Codasip是一家拥有多年经验专注于为嵌入式IoT领域定制处理器核提供处理器IP和服务的公司，是最早正式设计并提供商用RISC-V处理器IP的公司之一。

目前该公司提供Codix-BK Processor IP，支持多种的指令子集配置可定制指令接口。

- Codix-BK3是一款三级流水线的32位处理器
- Codix-BK5是一款五级流水线的处理器，可以配置为32位或者64位，同时还支持硬件单精度浮点运算器
- 此外还有Bk1、Bk7，并将有Bk10、Bk10 Pro等更高性能的核

RISCV 实现简介

Andes Core（商业）

晶心科技（Andes）是专注于提供处理器IP的一家公司，其商业模式与ARM相同。Andes有其自有的处理器指令集架构，是商用主流CPU IP公司之一。

- Andes于2017年初发布最新一代的AndeStar处理器架构，开始使用RISC-V指令集。
- AndeStarV5架构不但将RISC-V兼容性完全纳入，同时也包含多项Andes独创的通用便利功能及应用强化单元。

RISCV 实现简介

Microsemi Core（商业）

Microsemi公司是最早支持并使用RISC-V处理器的公司之一，推出了业界首个基于RISC-V内核的FPGA系列产品，即IGLOO2、SmartFusion2 SoC 和 RTG4。其FPGA产品因一向出色的功耗与可靠性指标，被广泛应用于对可靠性要求苛刻的场景。

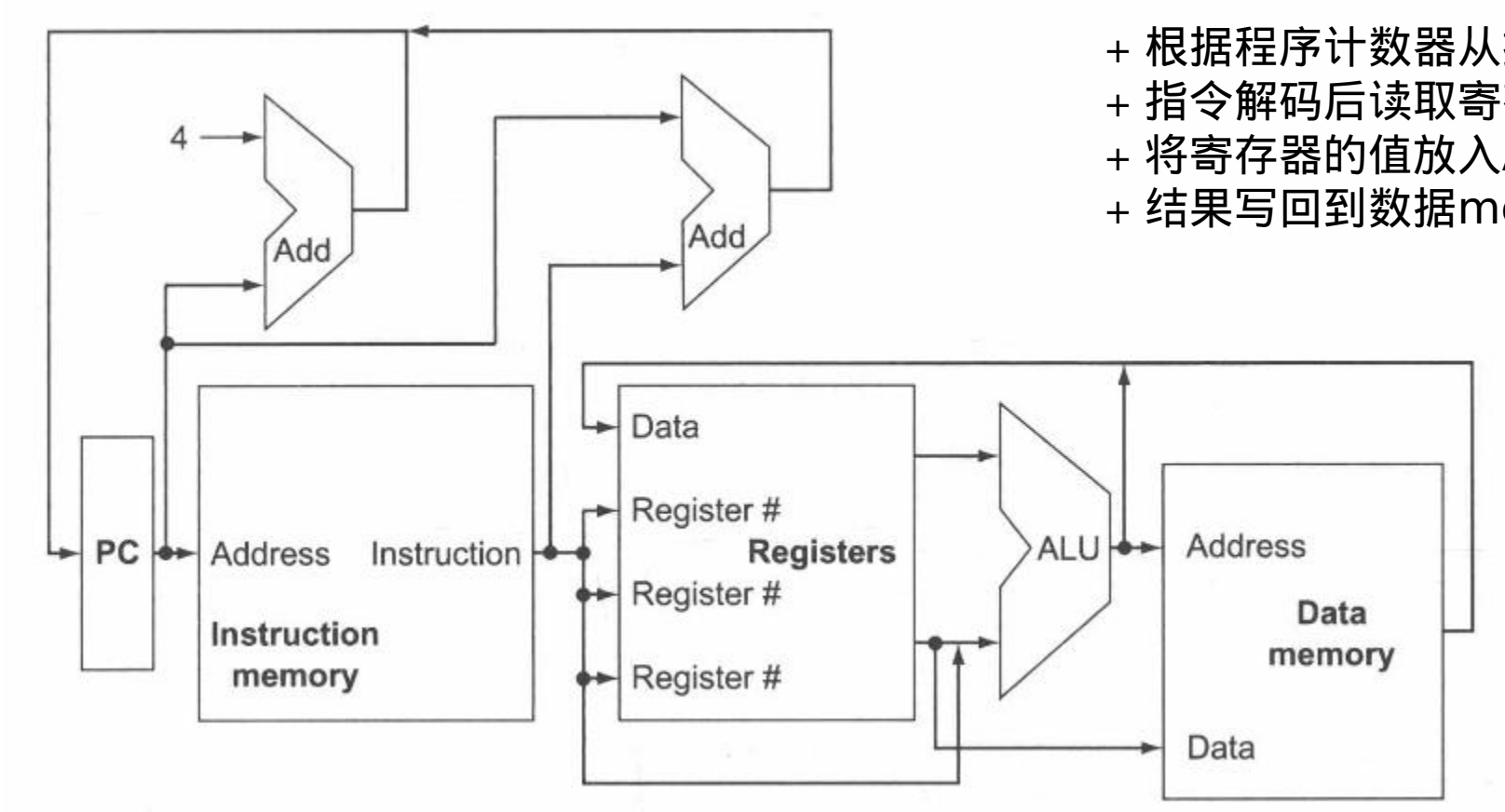
集成RISC-V内核的FPGA的特色主要是在开放性、可移植性和设计灵活性方面表现得更好。

Part 2

顺序单发射处理器基础简介

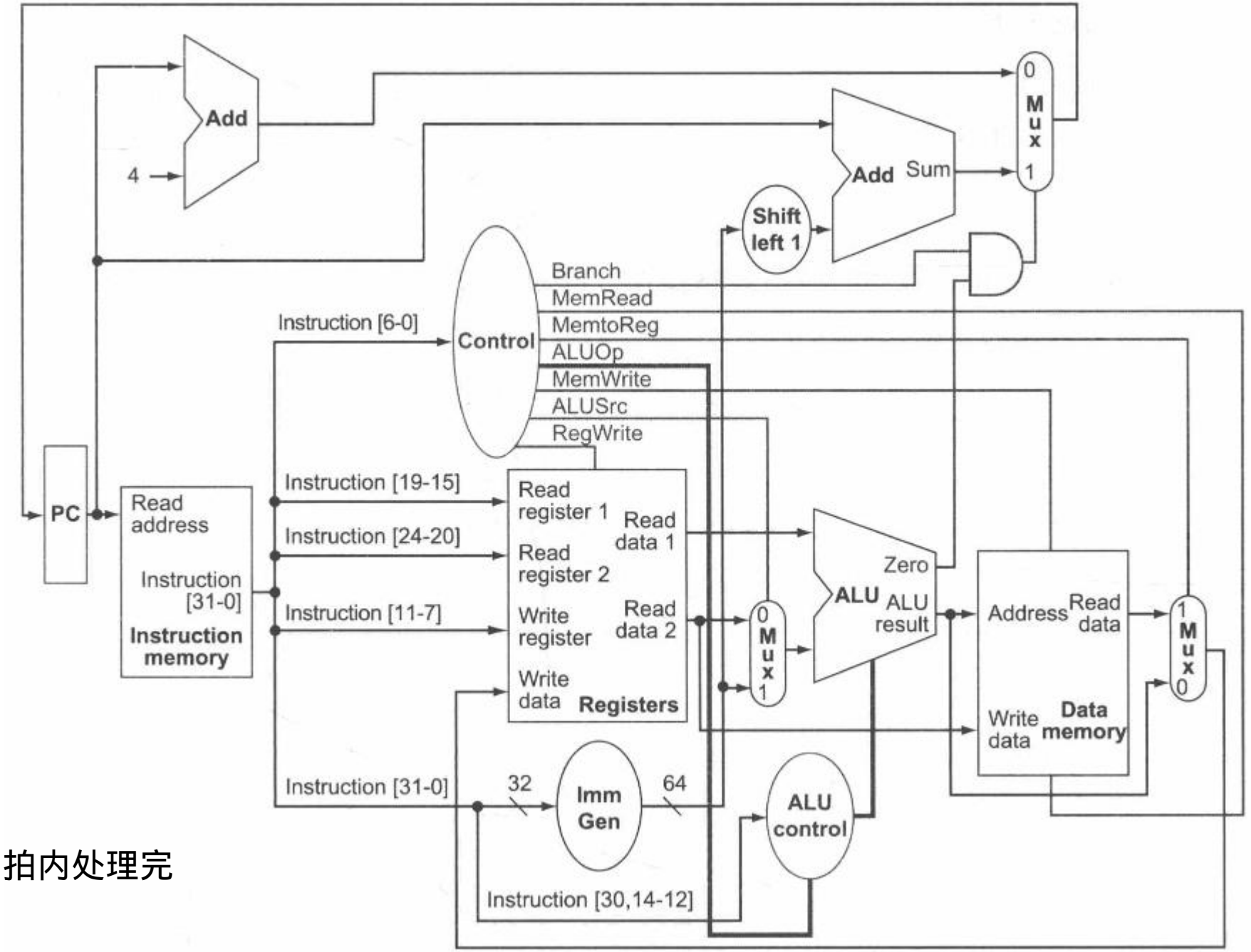
何为CPU？

CPU的抽象功能图:



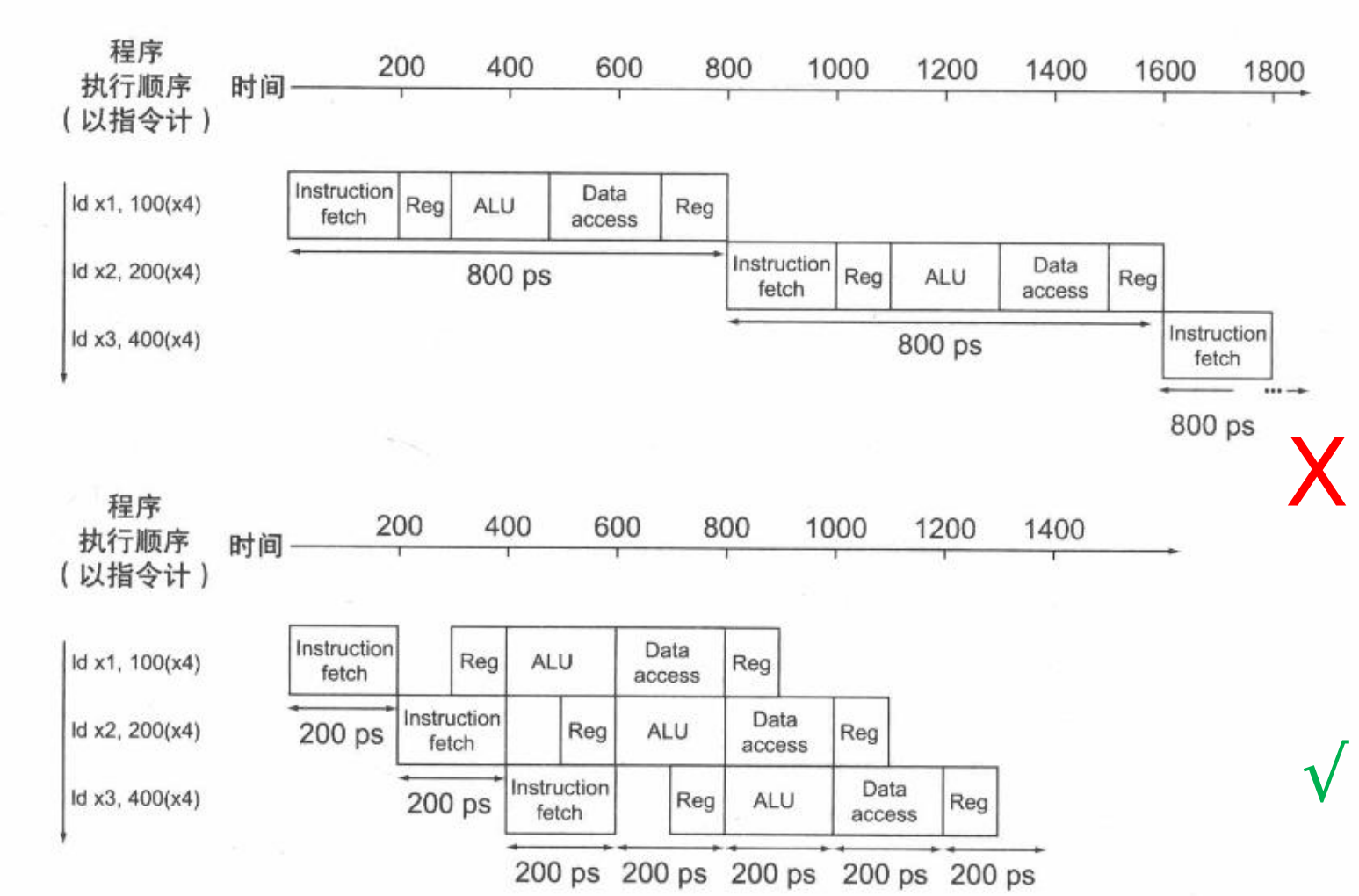
+ 根据程序计数器从指令mem中一条一条读出指令
+ 指令解码后读取寄存器堆
+ 将寄存器的值放入ALU进行计算
+ 结果写回到数据mem或者寄存器堆

单周期的RV CPU



+ 所有逻辑一拍内处理完

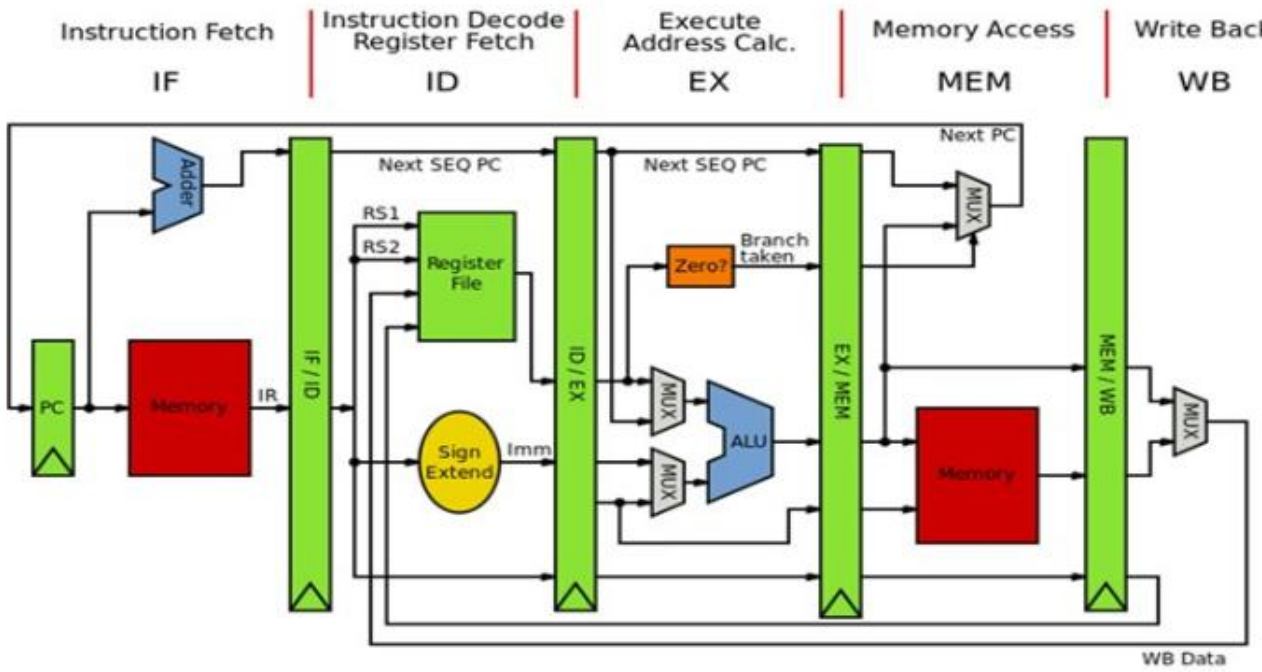
单周期CPU的局限



单周期处理器主频上不去，因为很长组合逻辑路径的指令（load, div等）限制了整体频率

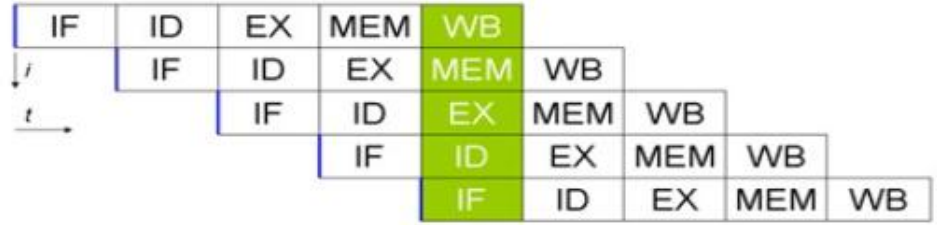
CPU 流水线概览

流水线（Pipeline）结构是处理器微架构最基本的一个要素，它承载并决定了处理器其他微架构的细节。



经典的五级流水线设计,一条指令生命周期分为5个步骤

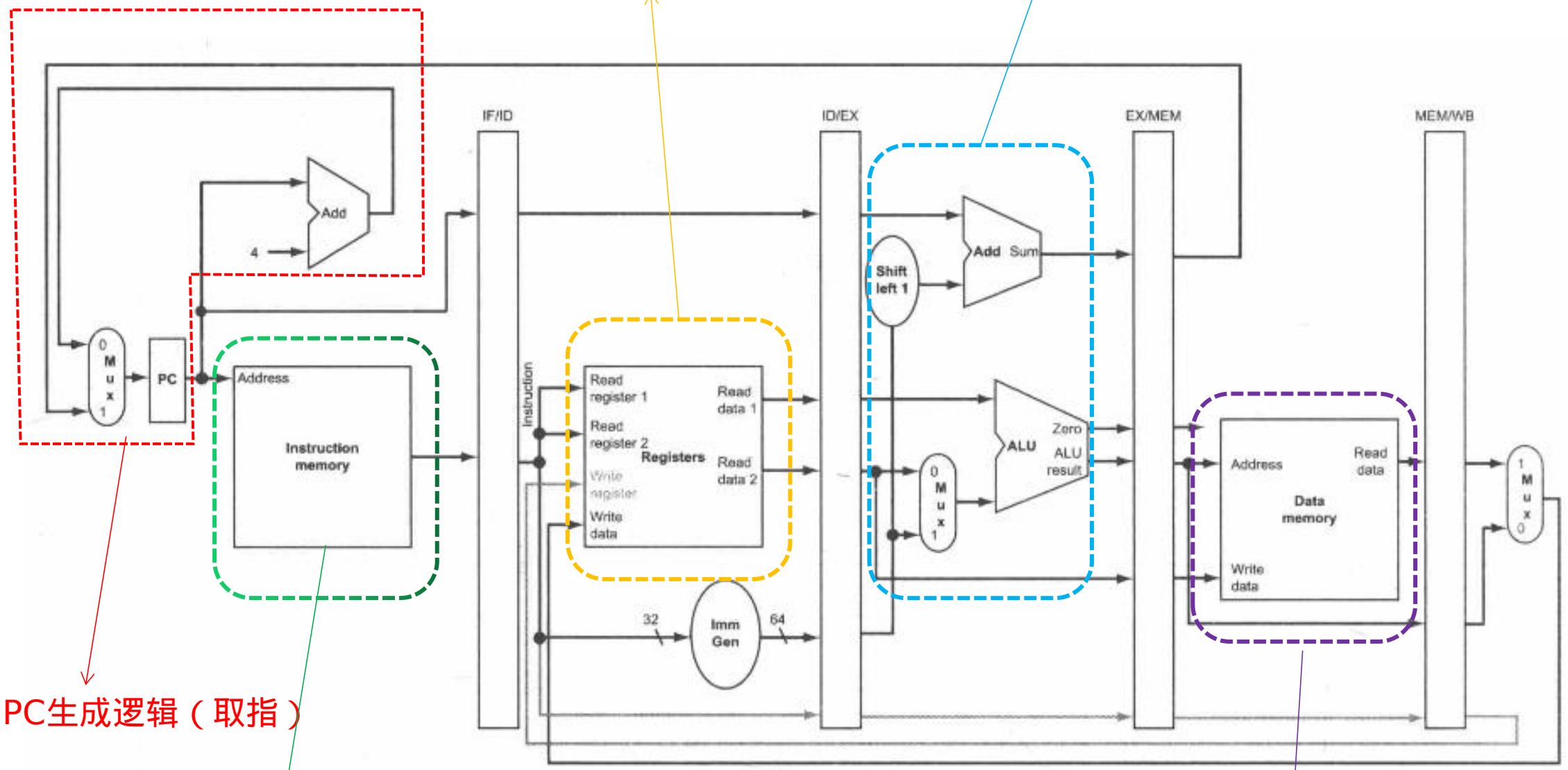
- **取指(IF)**：从存储器中取出指令
- **译码(ID)**：对指令进行解析获取操作类型和操作数
- **执行(EX)**：对译码后的指令进行运算
- **访存(MEM)**：从存储器中读出或写入数据
- **写回(WB)**：将执行结果写回通用寄存器组



CPU 流水线组件

寄存器堆（x0~x31）

计算单元（加法器，逻辑单元，分支单元...）



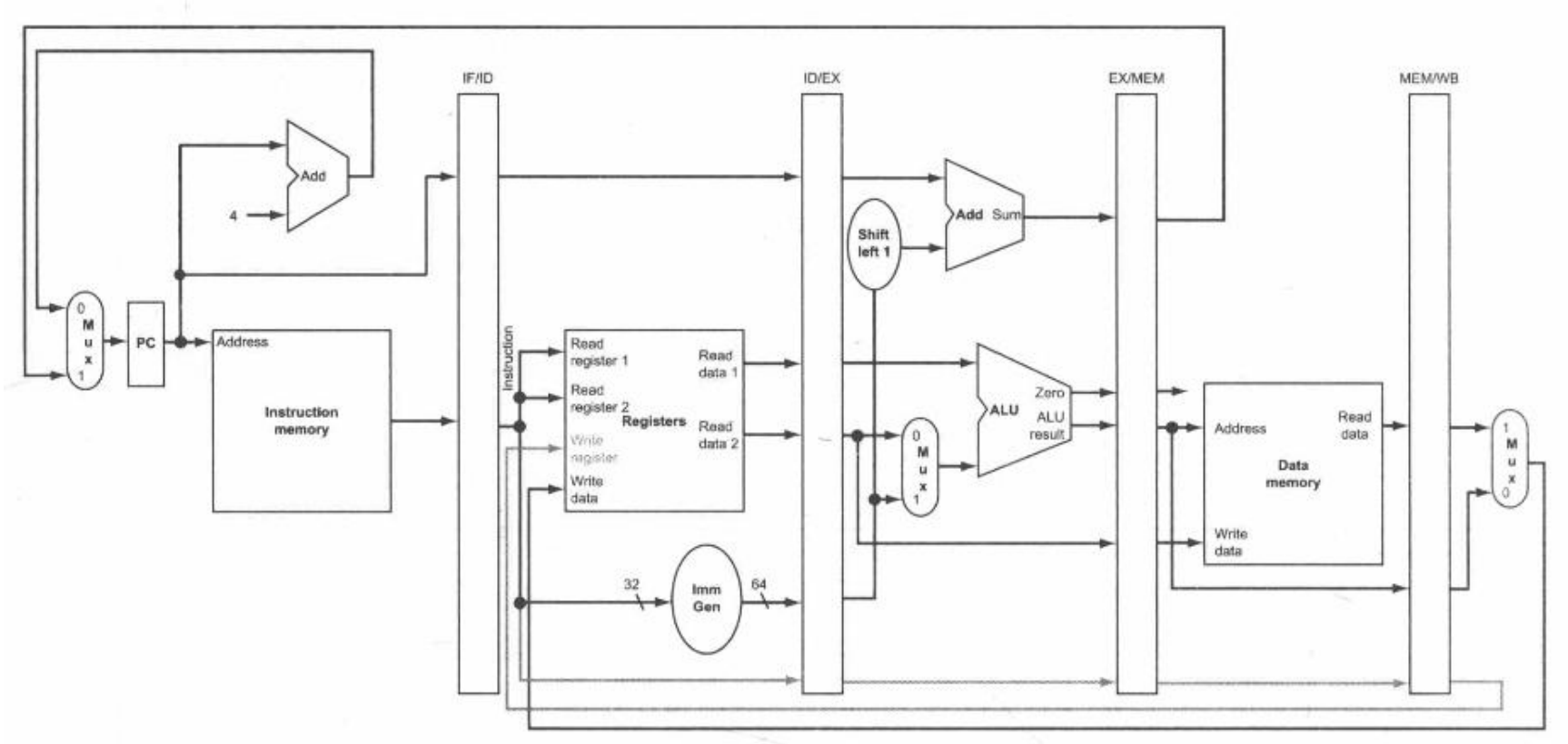
PC生成逻辑（取指）

指令Mem（取指）

数据Mem访存（load/store）

CPU流水线设计引入的问题

1. **结构冒险**：硬件不支持多条指令在同一周期执行。例如指令和数据存储器是同一个，那么同一周期会遇到同时取指令和访问数据的操作，就造成了结构冒险
2. **数据冒险**：流水线的某级需要另一级的结果而造成流水线停顿
3. **控制冒险**：一条分支指令需要决定是否跳转，而之后的指令依赖这个结果选择是否执行



有哪些数据冒险？

1. Read After Write (RAW)
2. Write After Write (WAW)
3. Write After Read(WAR)

RAW

```
add  x1, x2, x3
sub  x4, x1, x5
and  x6, x1, x7
```

WAR

```
add  x1, x2, x3
sub  x2, x4, x5
and  x6, x1, x7
```

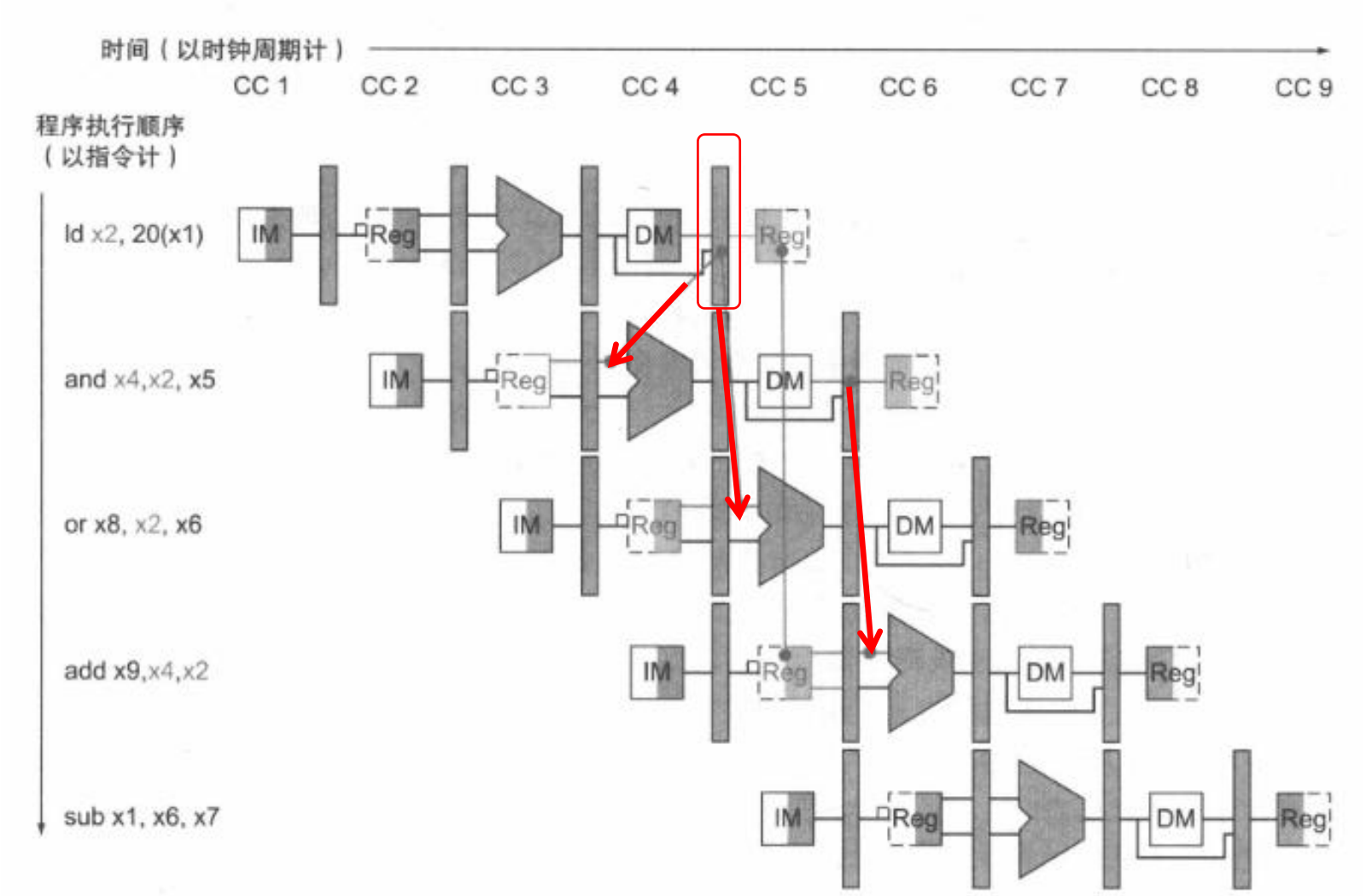
WAW

```
add  x2, x1, x3
sub  x2, x4, x5
and  x6, x1, x7
```

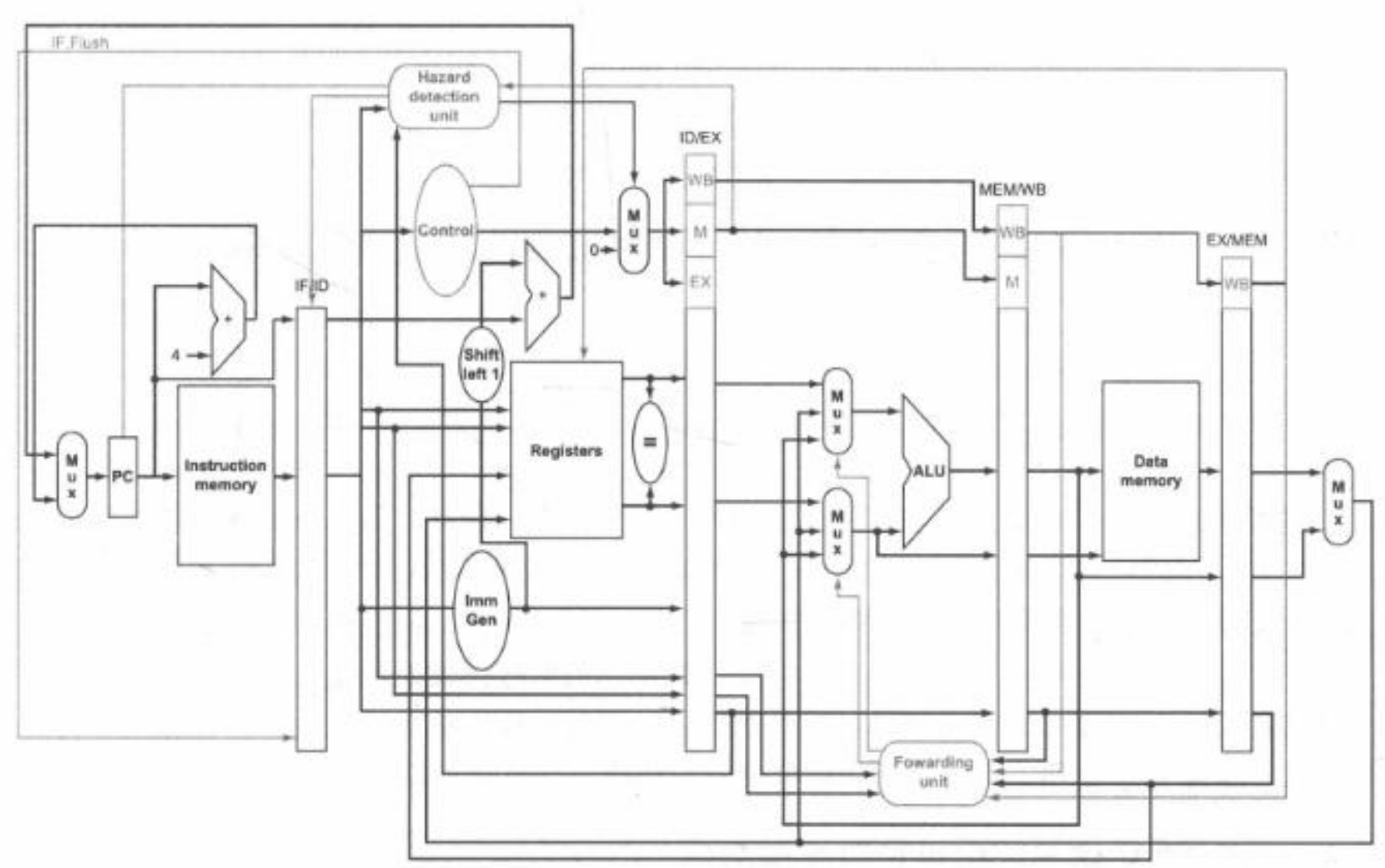
+ 对于顺序单发射处理器，指令都是挨个执行下来的，所以不存在WAR, WAW相关性

如何解决流水线中的数据冒险？

——流水线前递
（解决RAW冒险）



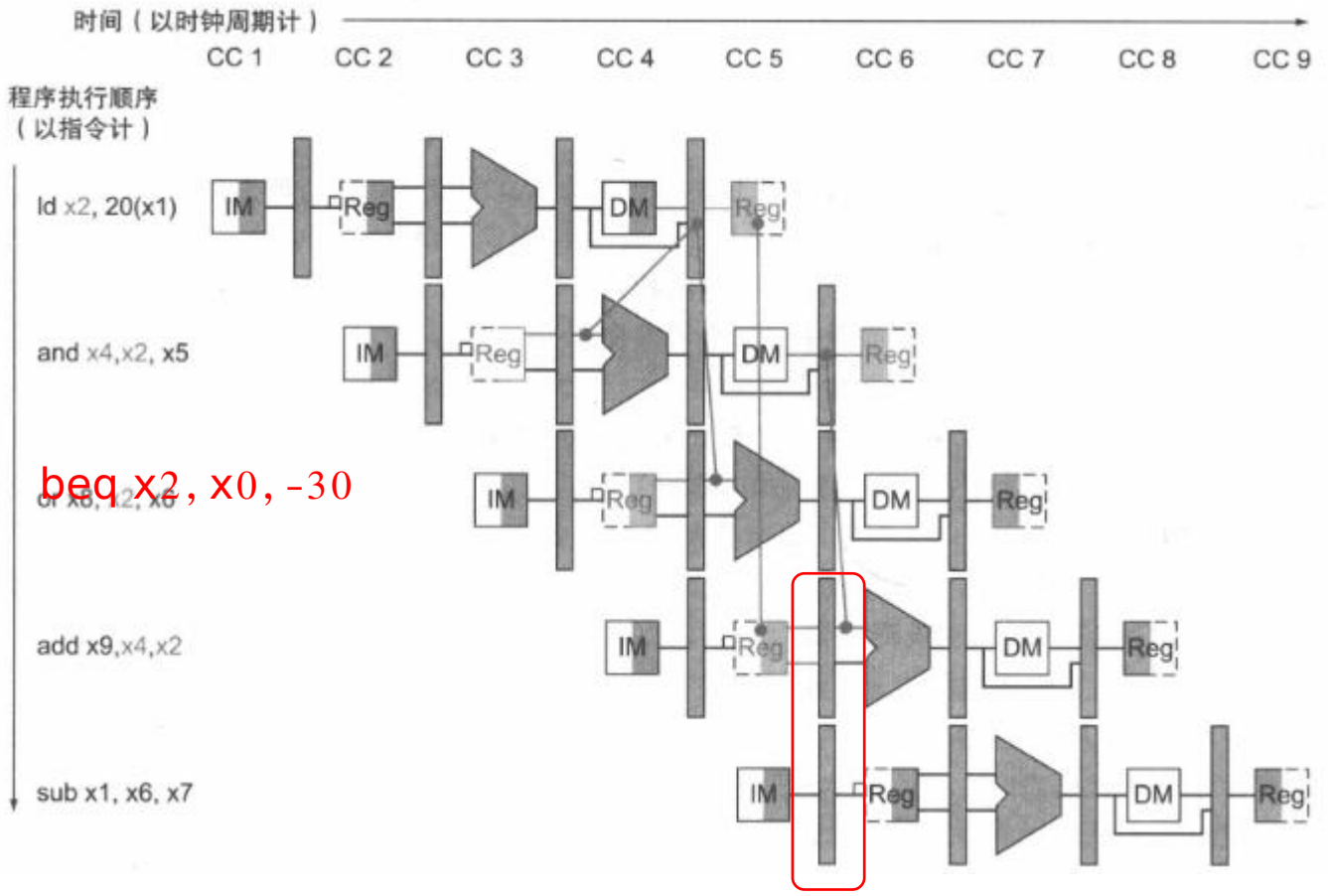
加入Forwarding后的流水线示意图：



控制冒险？

考虑下面的指令流：

1. ld x2, 20(x1)
2. and x4,x2, x5
3. beq x2, x0, -30
4. add x3, x2,x2
5. sub x2,x5, x4

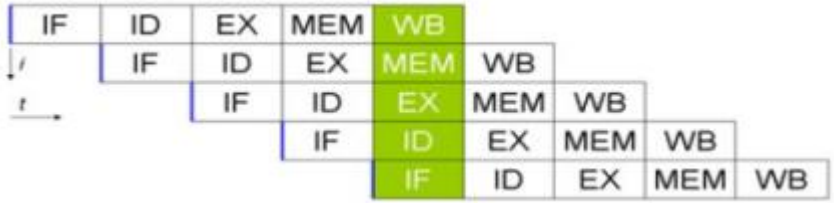


这两级pipeline 寄存器需要清除

+ 分支指令未发生跳转时，万事大吉，流水线不会停顿

+ 当分支指令发生跳转时，流水线中**EX阶段之前的指令需要被抹除**。之后需要重填流水线，造成性能损失。

如何解决流水线中的控制冒险？



Stupid, but efficient.

1. 假设分支不发生：不管怎么样，让流水线流下去，当发现分支确实会发生（通常在Exe阶段就能知道），再暂停流水线，Flush掉EXE阶段之前的流水线寄存器，设置PC寄存器从正确的分支地址开始取指，重新装填流水线

Naive, and so naive.

2. 尽早得到分支结果：如果能在Fetch取指令阶段就立即知道当前这条分支指令的结果，那就能立马设置PC寄存器，在下一个周期从正确的地址取指，就没有控制冒险了。但是理想是美好的，现实是哪怕只是将分支结果的判断移到ID,也就是从EXE阶段移到解码阶段，代价也是很大的，因为判断分支指令的结果通常还需要读寄存器以及逻辑判断，而这些都在EXE阶段才发生。强行提前需要增加额外的寄存器端口、ALU组件，还需要额外的前递逻辑，引入额外的停顿等

Smart, but expensive.

3. 分支预测：CPU中的分支一般就那么几种，想象一下你在写C代码的时候，最常用的就是 if else, for, while, switch... 或者是函数调用。这些代码都会编译出分支指令，而实际上这些代码很多时候的行为是可预测的，比如for(i=0;i<10;i++), 运行后你会看到应该有连续的9次会跳转到这段for的开头部分（最后一次跳出循环）。所以可以用一些简单的预测方式来假定一个分支结果，后面发现错了再改回去就行，总比上面总是假设分支发生或不发生好。

Part 3
超标量处理器基础简介

多发射处理器分类

静态多发射 —— 超长指令字（VLIW）

+ 在一个周期内可以取出多条指令，并并行执行
+ 并行执行的指令不能有依赖关系

指令类型	流水线阶段							
ALU或分支指令	IF	ID	EX	MEM	WB			
load或store指令	IF	ID	EX	MEM	WB			
ALU或分支指令		IF	ID	EX	MEM	WB		
load或store指令		IF	ID	EX	MEM	WB		
ALU或分支指令			IF	ID	EX	MEM	WB	
load或store指令			IF	ID	EX	MEM	WB	
ALU或分支指令				IF	ID	EX	MEM	WB
load或store指令				IF	ID	EX	MEM	WB

图 4-65　静态双发射流水线操作。ALU 和数据传输类指令同时被发射。假设使用和单发射流水线相同的五级流水结构。虽然这样的流水级划分并不是严格必需的，但确实能带来一些好处。尤其是，所有指令统一在最后一个流水级进行寄存器更新，这样有助于实现精确例外模型，简化例外处理的实现。如上所述，例外处理在多发射处理器实现中是一个难点

+ 依赖编译器将指令流编译成硬件最高效的方式。例如将循环展开，将两条没有相关性的算术指令和访存指令安排在一起

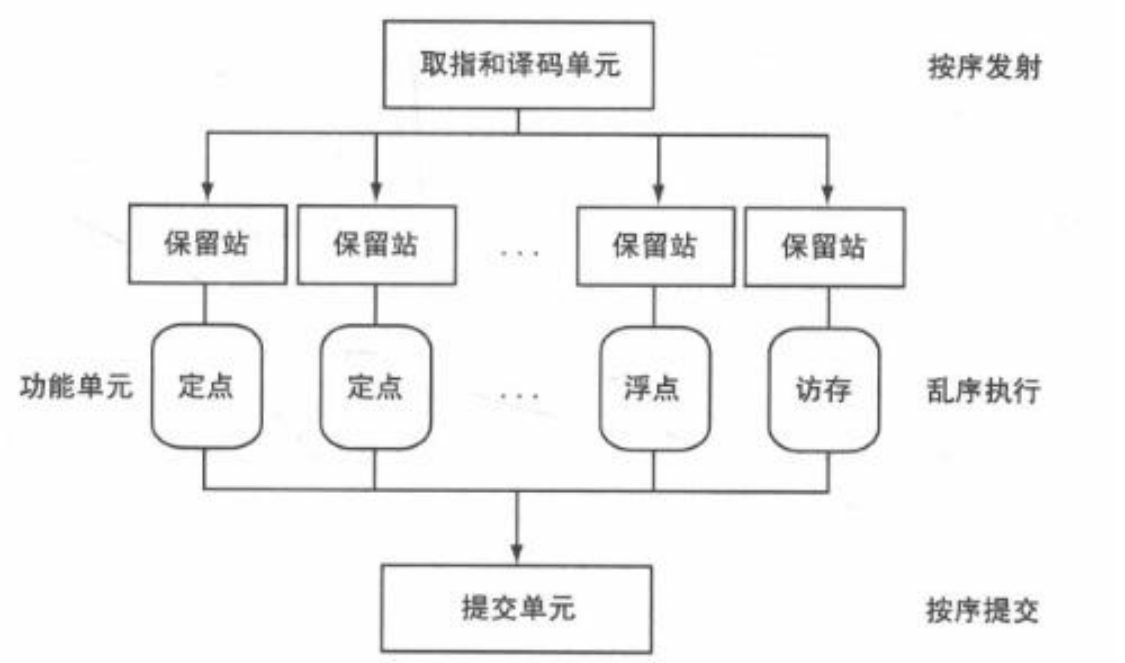
多发射处理器分类

动态多发射 —— 乱序多发射（Out Of Order）

+ 在一个周期内可以取出多条指令，并并行执行

+ 并行执行的指令可以有依赖关系，会有专门的组件处理WAW, WAR相关性

+ 指令取出时是顺序取出，但是执行时是乱序执行（一条指令后面的指令可能会先执行），最后提交结果时按照原来顺序提交

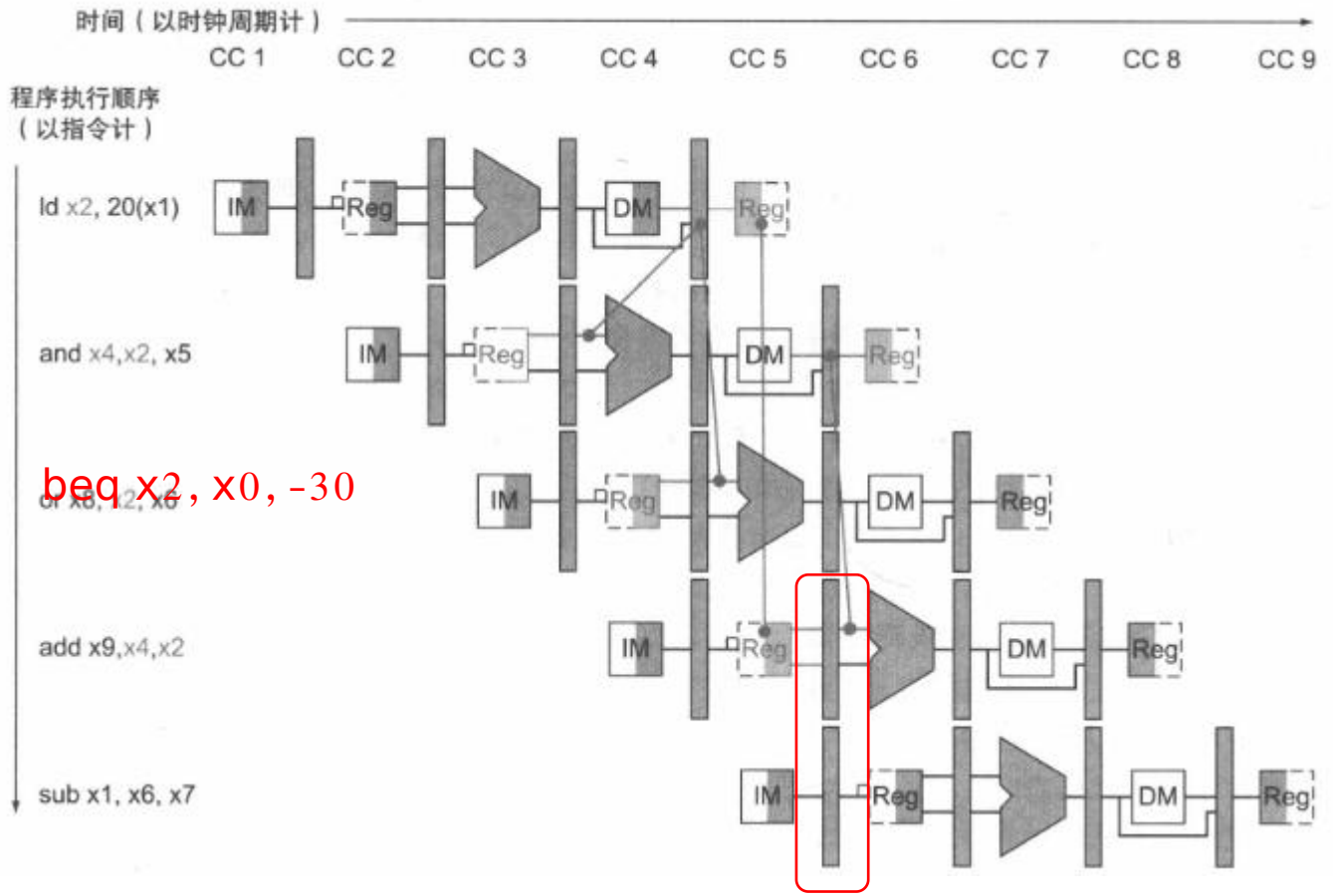


超标量处理器的标配——分支预测器件

分支预测再探：

考虑下面的指令流：

1. ld x2, 20(x1)
2. and x4,x2, x5
3. beq x2, x0, -30
4. add x3, x2,x2
5. sub x2,x5, x4



这两级pipeline 寄存器需要清除

+ 分支指令未发生跳转时，万事大吉，流水线不会停顿

+ 当分支指令发生跳转时，流水线中**EX阶段之前的指令需要被抹除**。之后需要重填流水线，造成性能损失。

超标量处理器的标配——分支预测器件

两位饱和预测器：

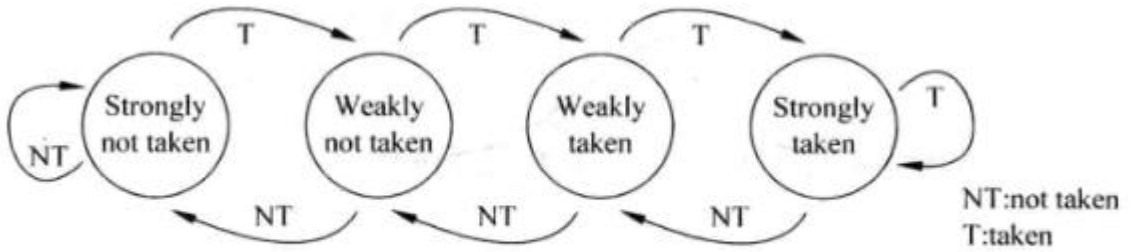


图 4.8　基于两位饱和计数器的分支预测

```
for(i=0;i <8;i++){
  ...
}
```

+ 对于一些循环，两位饱和预测器的工作方式：

  + 假设Branch的执行情况：TTTTTTTN, 则多次跳转后预测其就进入饱和状态，之后也能预测为T，只有最后一次预测失败。

+ 状态机在饱和时需要连续两次预测失败才会改变预测结果，相当于一个**去抖**电路

超标量处理器的标配——分支预测器件

每一个PC都会对应一个两位饱和计数器，但是这样对于32位的PC长度来说，全部分配一个显然是不现实的，因此通常用PHT(Page History Table)，如下图：

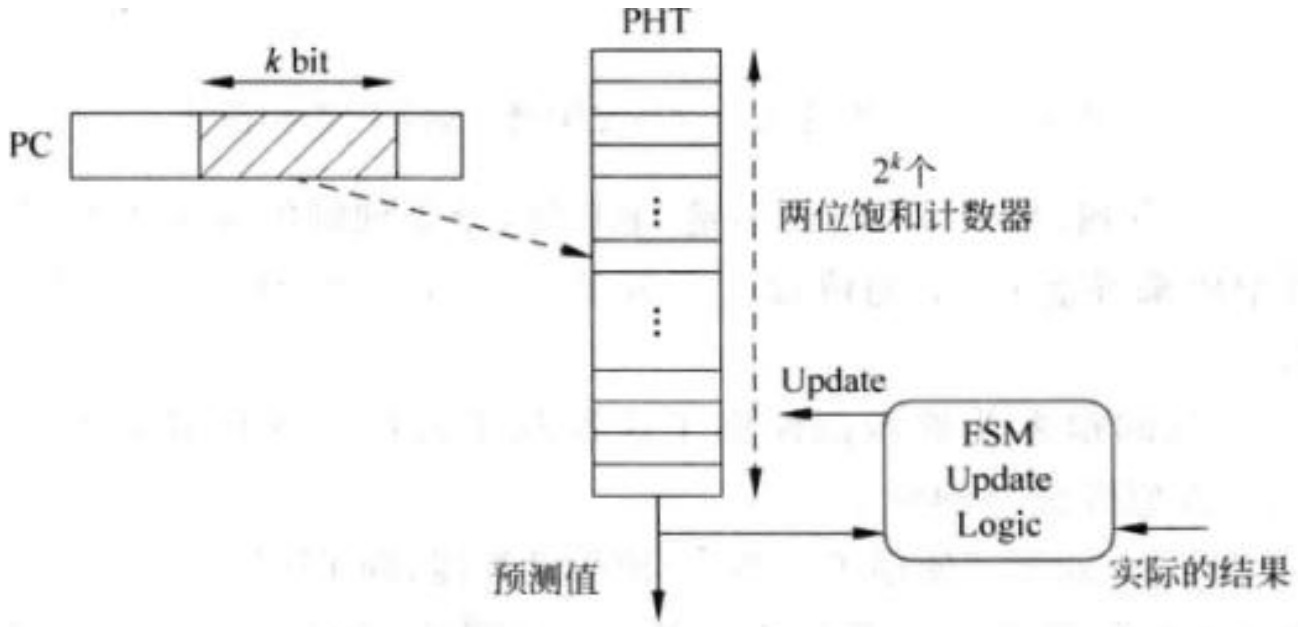


图 4.10　使用 PC 值的一部分来寻址饱和计数器

超标量处理器的标配——分支预测器件

上面的预测器无法处理有些频繁跳转的Branch情况：

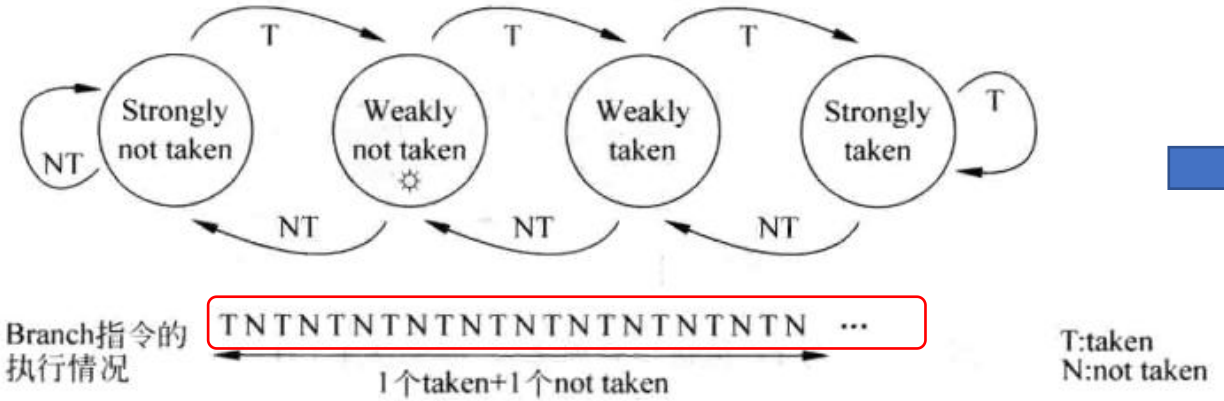


图 4.13 对于一些很有规律的分支指令，使用两位饱和计数器却不会有很高的准确

自适应的两级分支预测器

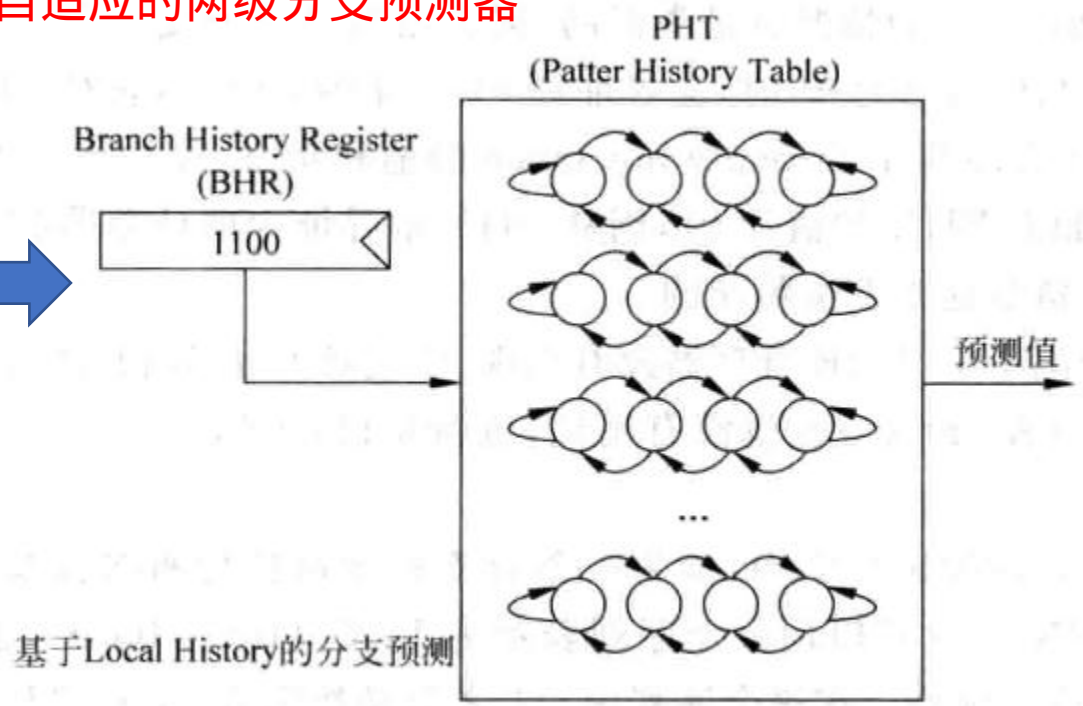


图 4.14 基于局部历史的分支预测

假设BHR位宽为2，对应一个4个表项的PHT，假设分支序列是T - N - T - N - T ...

+ BHR是一个移位寄存器，所以相当于2位的寄存器在一个101010...的序列上滑动， 寄存器的值要么是10，要么是01

+ 当BHR是10时，下一次进来的一定是1，也就是一定是跳转，10寻址PHT的第三个表项，当分支结果计算出来之后会更新这个表项，则之后再遇到BHR是10的情况，去PHT中获取历史预测结果就可以得到预测为跳转的结果了。 同理BHR是01是处理过程也是类似的。

超标量处理器组件——寄存器重命名

+ **只有RAW是真相关性**。WAW是先写A再写A，可以通过重命名让两条指令先写到不同的寄存器；WAR是先读A后写A，可以通过重命名让写指令写到另一个寄存器，就不会对读产生干扰。而先写A后读A，读指令必须读到新写进去的值，所以只有RAW才是真的数据相关。

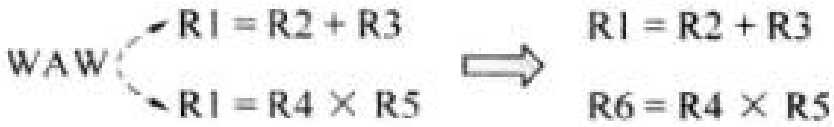


图 7.1　通过更换寄存器来解决 WAW 相关性

图 7.2　通过更换寄存器名字来解决 WAR 相关性

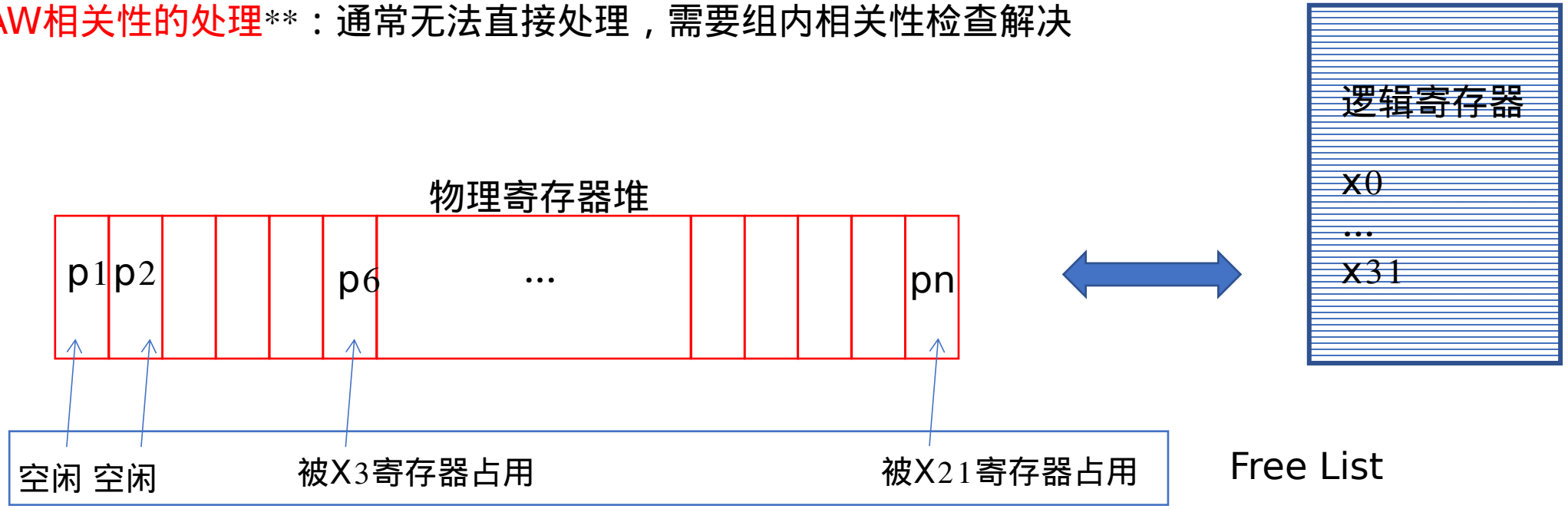
+ 指令集定义的寄存器为**逻辑寄存器**（Logical Register 或 Architecture Register）**。
+ 处理器内部实际存在的寄存器为**物理寄存器**（Physical Register）**。
+ 将逻辑寄存器映射到物理寄存器：**重命名映射表**（Register Renaming Table），Intel也将其称为Register Alias Table, **RAT**

超标量处理器组件——寄存器重命名

WAW相关性的处理：
1. 写入RAT时，只需要将逻辑寄存器的最新映射关系写道RAT中。如果一个周期内多条指令的有同一个目的寄存器，执行需要写入最新的那条指令的映射关系。

WAR相关性的处理：寄存器重命名可直接消除。

RAW相关性的处理：通常无法直接处理，需要组内相关性检查解决



超标量处理器组件——寄存器重命名

寄存器重命名过程:

对于Dest= (Srcl) OP (Src2)这样的指令来说，寄存器重命名的过程如下:

1. 从RAT中找到Src1和Src2对应的物理寄存器Psrc1和Psrc2。

2. 从Free List中找到一个空闲的物理寄存器Pdest，作为目的寄存器Dest对应的物理寄存器。

3. 将映射关系写到RAT中：

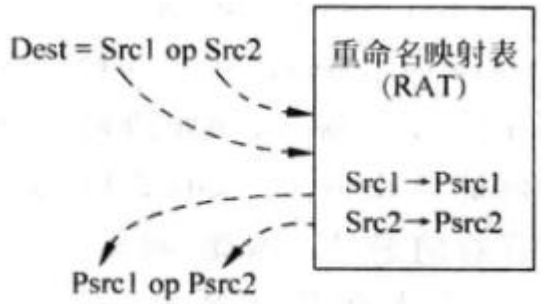


图 7.13　找到两个源寄存器对应的物理寄存器

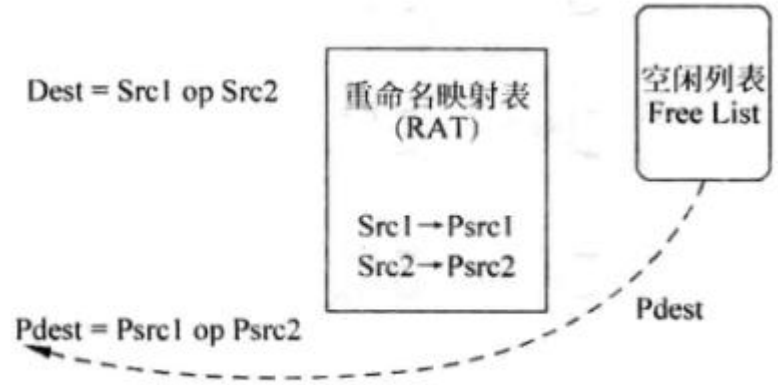


图 7.14　为指令的目的寄存器指定一个物理寄存器

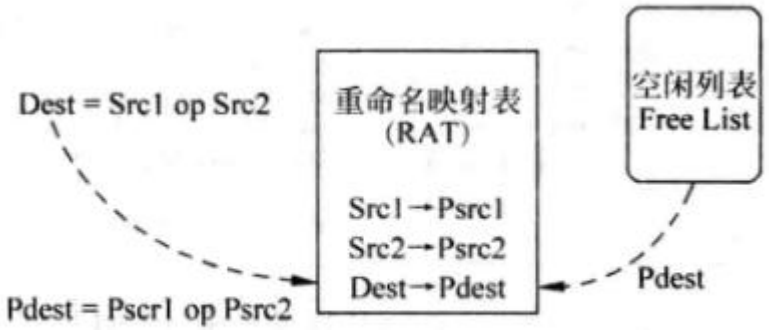


图 7.15　将新的映射关系写到 RAT 中

超标量处理器组件——发射队列（保留站）

什么是保留站？

——在执行单元的入口处放置的一个队列用于存放发射过来的指令，及其基本信息(源寄存器，目的寄存器等）

为什么需要保留站？

——超标量处理器一次可以处理N条指令，如果这些指令有RAW相关性，或者对应的功能单元还在处理之前的指令，就需要暂存一下，等到可以执行该指令时再将其送到计算单元。暂存的部件就是保留站，也叫发射队列。

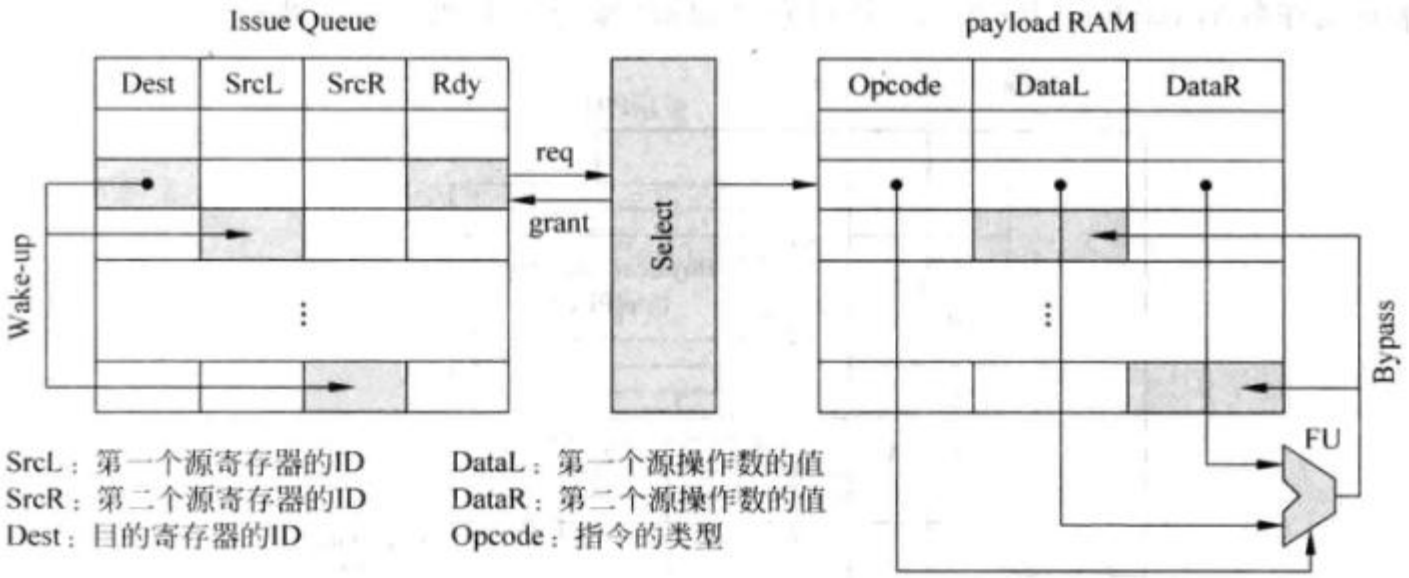


图 8.2　使用 payload RAM 来存储发射队列中的源操作数

+ 寄存器重命名后的指令先读PRF,将读出值与指令一起写入IQ。
+ 发射队列中存储指令操作数的地方是payload RAM
+ 当目的寄存器的值经过FU计算出来，就会经过**bypass网络**来更新payload RAM中做了标记的位置。

超标量处理器组件——发射队列（保留站）

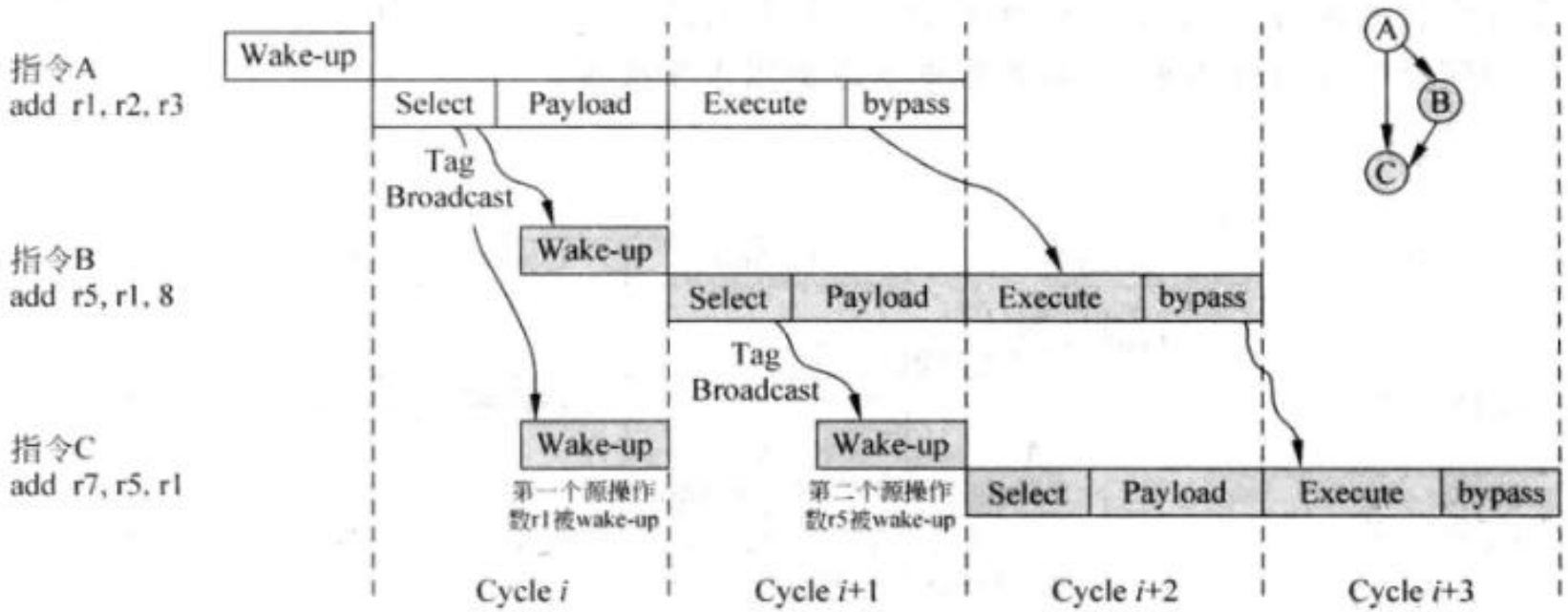


图 8.19　数据捕捉结构的流水线

+ **仲裁**（Select）负责从发射队列中找出一条合适的指令。
+ **唤醒**（Wake-up）负责将发射队列中相关寄存器置为准备好的状态。

B指令和C指令的源操作数依赖A指令的结果，A指令有效后的下一个周期，会唤醒B指令和C指令中依赖A的结果的源寄存器。当B/C指令的两个源操作数都被唤醒，该指令才会有效，在下个周期真正执行

超标量处理器组件——Reorder Buffer (ROB)

+ **Reorder Buffer, ROB**, 本质上是一个FIFO

+ 在执行阶段乱序执行的指令，需要按照原始的指令顺序登记在ROB中——ROB是保证乱序处理器最后能够按照程序顺序执行的关键

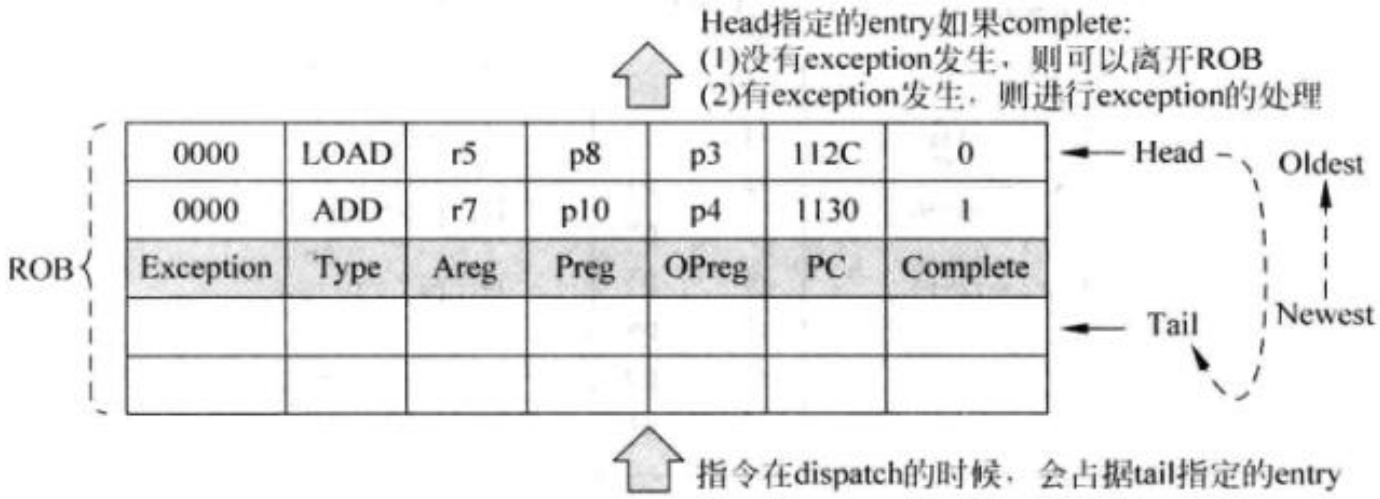


图 10.2 重排序缓存(Reorder Buffer)

+ 当指令变为ROB中最旧的指令, 可以用header pointer来表示, 如果他的complete状态位=1, 则这条指令可以退休了. 如果其exception部分=0, 则他可以顺利离开流水线。

超标量处理器——案例

1. 指令队列中的指令根据RS是否有空表项决定指令去留。并将指令顺序地在ROB中进行登记

2. RS中记录每条指令的源操作数的来源，可以是寄存器堆，也可以是ROB中的表项。同时也会记录指令结果保存到ROB的表项的index

3. 当RS中指令的两个源操作数都准备好，就可以发射到功能单元进行执行，结果根据RS中保存的ROB index放到ROB对应表项

4. ROB每次都将最顶上的一条（或几条）已经完成的指令pop出去，表示指令退休成功，如果该指令修改了逻辑寄存器的内容，需要将对应修改同步到寄存器堆

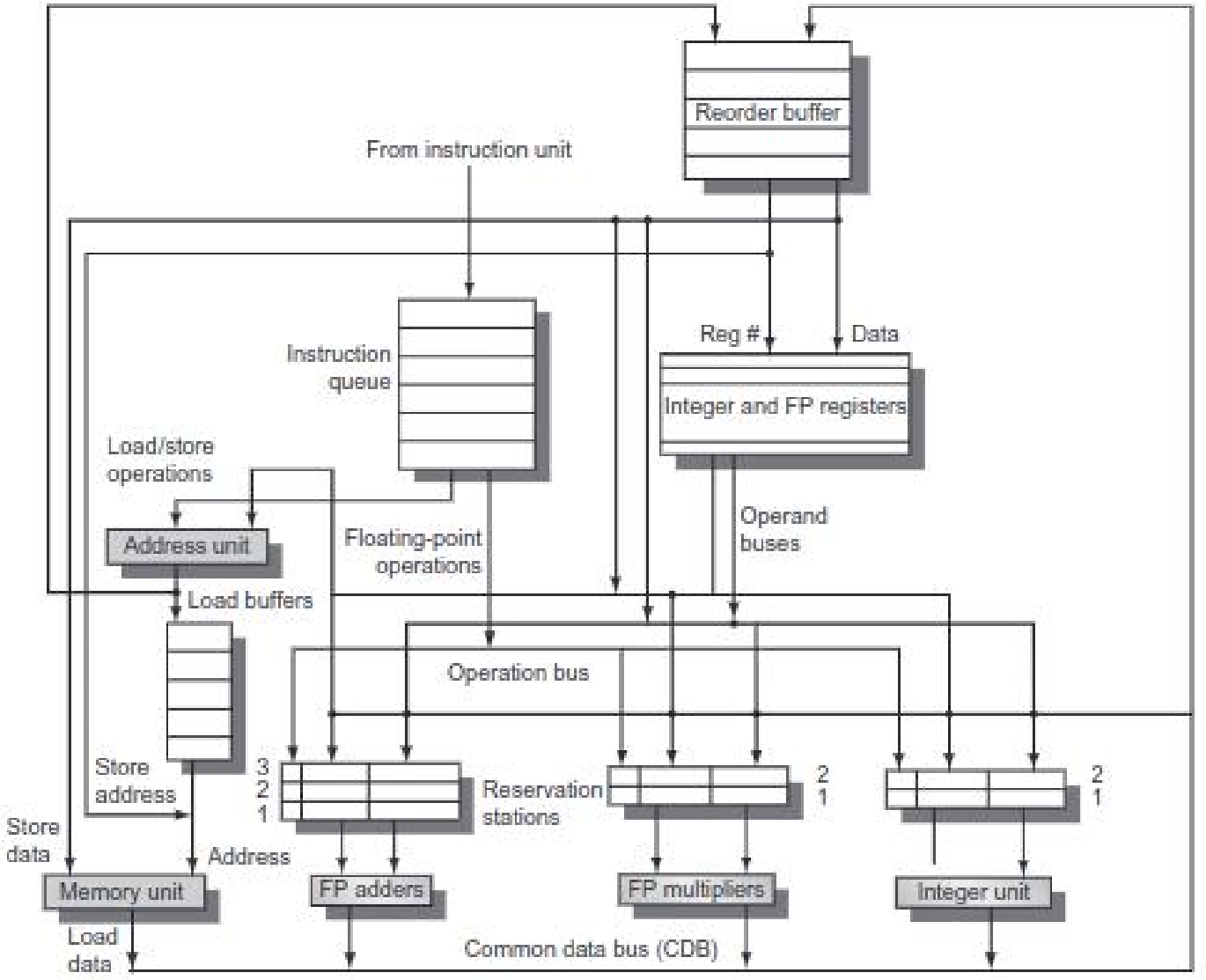
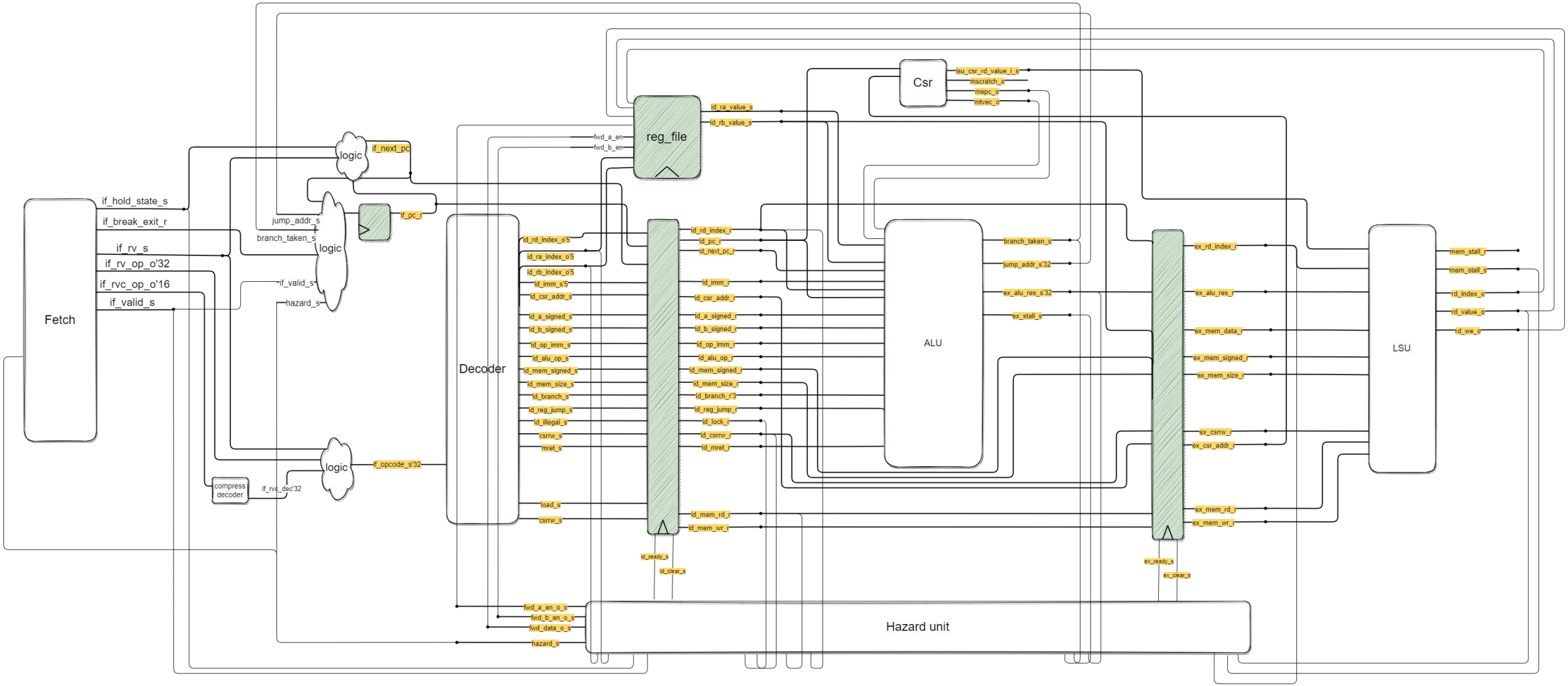


Figure 3.21 The basic organization of a multiple issue processor with speculation. In this case, the organization could allow a FP multiply, FP add, integer, and load/store to all issues simultaneously (assuming one issue per clock per functional unit). Note that several datapaths must be widened to support multiple issues: the CDB, the operand buses, and, critically, the instruction issue logic, which is not shown in this figure. The last is a difficult problem, as we discuss in the text.

Thanks

In house RISCV core 简介



Fetch

ID	Instruction[15:0]	Instruction[31:16]
1	32low	32high
2	16c	16c
3	16c	32low
4	32high	16c
5	32high	32low

Load/Store Unit

RISC-V架构对于存储器访问指令的简化

RISC-V的简化硬件哲学，对于存储器访问指令而言，通过如下特点大幅简化硬件实现：

- 仅支持小端模式
- 无地址自增自减模式
- 无“一次读或写多个数据”指令

RISC-V架构的存储器相关指令

Load & Store指令

LH、LHU、LB、LBU、LW

SB、SH、SW

“A”扩展指令

Atomic Memory Operation（AMO）指令

Load-Reserved和Store-Conditional指令

Fence指令

Fence

Fence.I

CSR, 中断

RISC-V架构异常处理机制 (Machine Mode)

进入异常

1. 停止执行当前程序流，转而从CSR寄存器**mtvec**定义的PC地址开始执行。
2. 硬件同时更新以下4个CSR寄存器:
   a. 机器模式异常原因寄存器**mcause**
   b. 机器模式异常PC寄存器**mepc**
   c. 机器模式异常值寄存器**mtval**
   d. 机器模式状态寄存器**mstatus**

退出异常

在机器模式下退出异常时，软件必须使用MRET指令:

1. 停止执行当前程序流，转而从CSR寄存器**mepc**定义的PC地址开始执行
2. 硬件同时更新CSR寄存器机器模式状态寄存器**mstatus**（Machine Status Register）

注意：由于 RISC-V 架构规定的进入异常和退出异常机制中没有硬件自动保存和恢复上下文的操作，因此需要软件明确地使用指令进行上下文的保存和恢复。

Cache 子系统

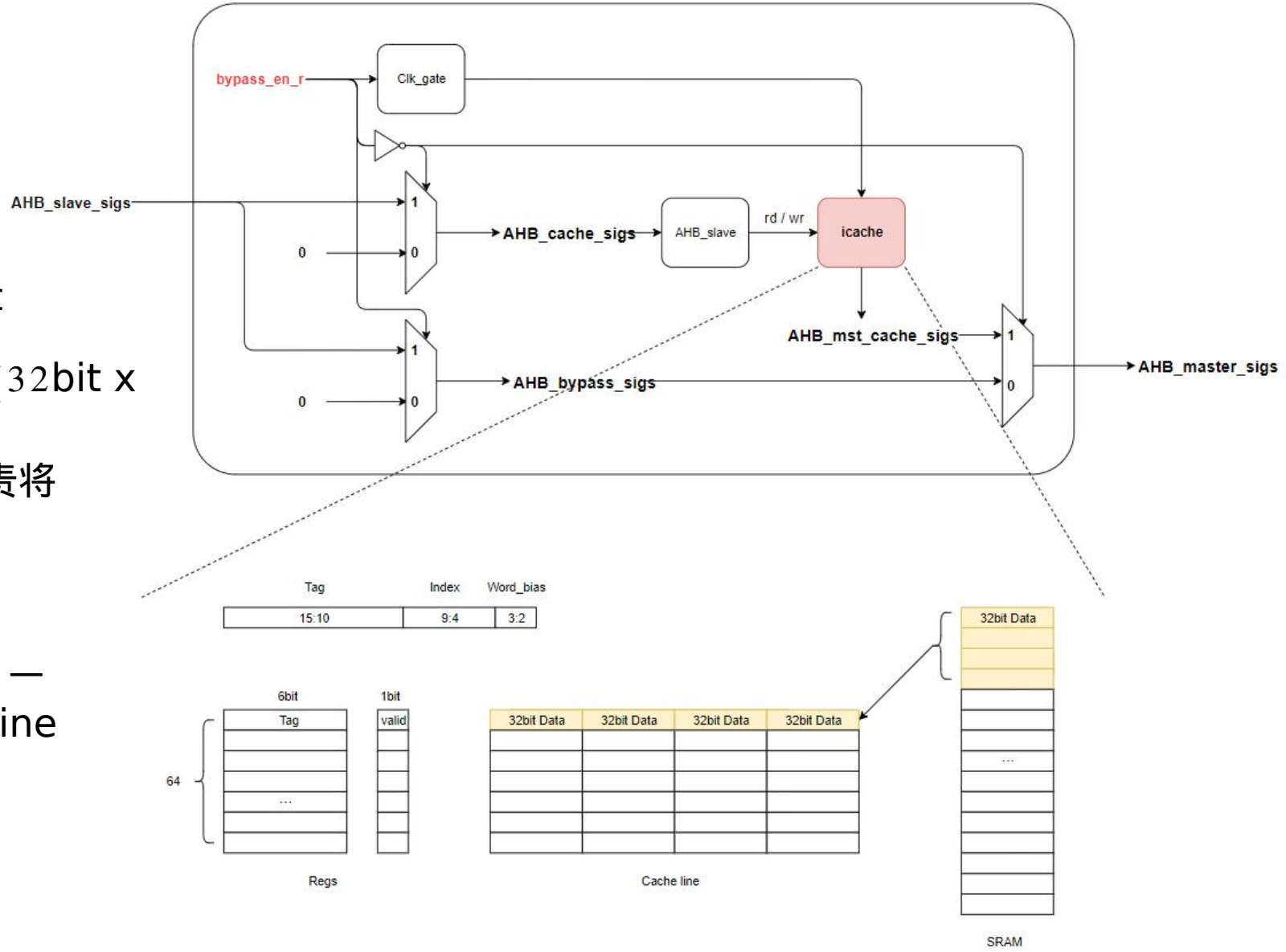
◎ 直接相连

◎ 6bit Tag, 6bit Index, 2bit offset

◎ 64条line， line width = 128bit (32bit x 4)

◎ 对外有一个AHB master接口, 负责将缺失数据从Flash/DRAM导入

◎ 该AHB mst接口只能进行读操作, hsize为32bit, hburst类型为wrap4, 一次读出4个连续的32bit数据组成一条line



In house RISCV SOC