计算机系统（3）

指令

寄存器编号

编号	寄存器名	作用
8-15	t0-t7	临时寄存器（可以被调用者改写）
16-23	s0-s7	保存参数（必须被调用者保存和恢复，当作临时寄存器使用也可以）
24-25	t8-t9	临时寄存器（可以被调用者改写）
0	zero	值始终为0
1	at	保留寄存器
2-3	v0-v1	函数返回值
4-7	a0-a3	传递函数参数
26-27	k0-k1	保留寄存器，中断处理函数使用
28	gp	静态数据的全局指针寄存器
29	sp	堆栈指针寄存器
30	fp	帧指针寄存器-保存过程帧的第一个字
31	ra	返回地址寄存器

R型指令

op[31,26]	rs[25,21]	rt[20,16]	rd[15,11]	shamt[10,6]	funct[5,0]
操作码	第一个源寄存器	第二个源寄存器	目的寄存器	移位位数	功能码
6	5	5	5	5	6

I型指令

op[31,26]	rs[25,21]	rt[20,16]	constant or address[15,0]
操作码	第一个源寄存器	目的寄存器	第二个源操作数
6	5	5	16

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
addi	I	8	18	17	100			addi $s1,$s2,100
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer format

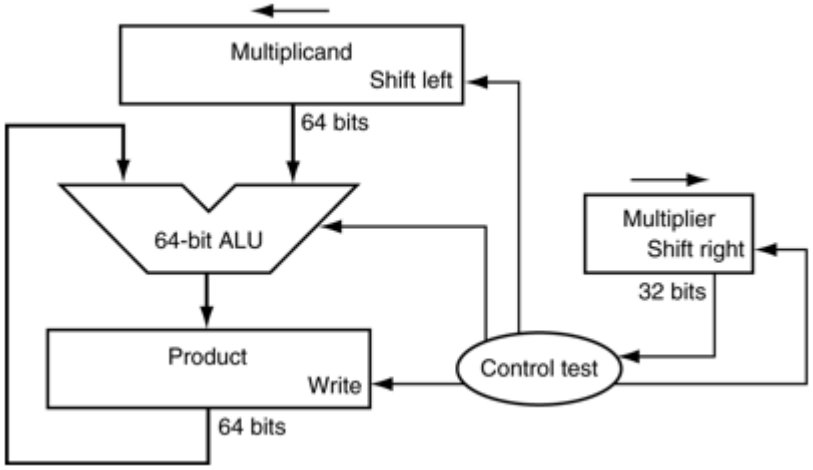
J型指令

op[31,26]	address[25,0]
操作码	地址
6	26

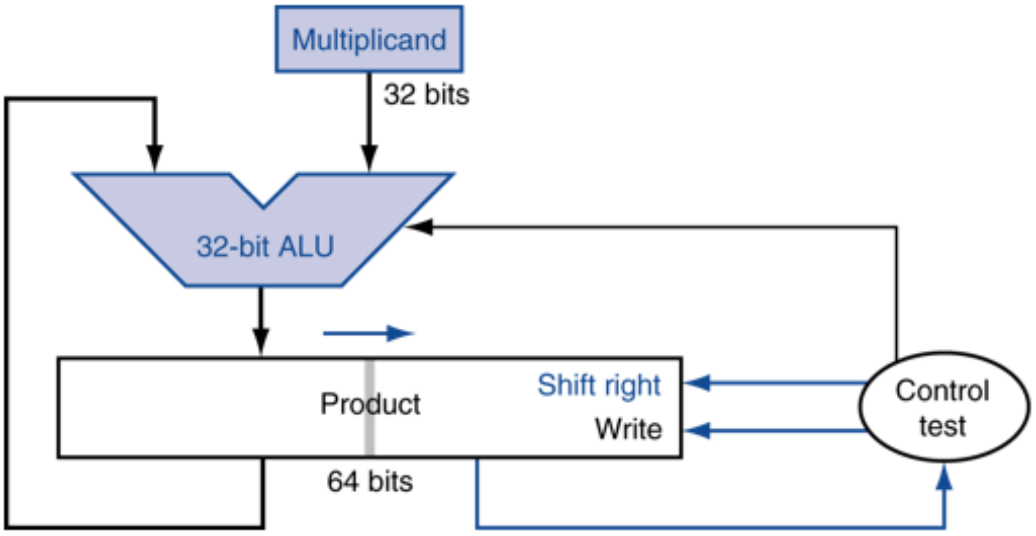
跳转位置$Target = PC_{31...28} : (address \times 4)$

运算

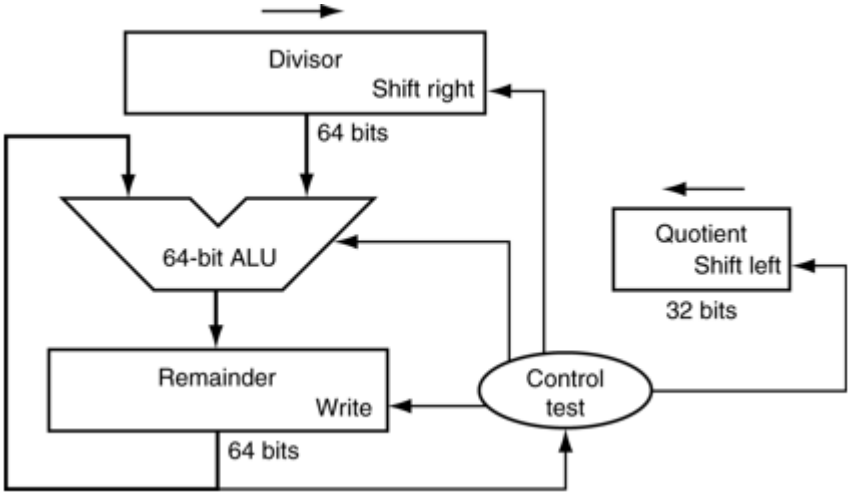
乘法器



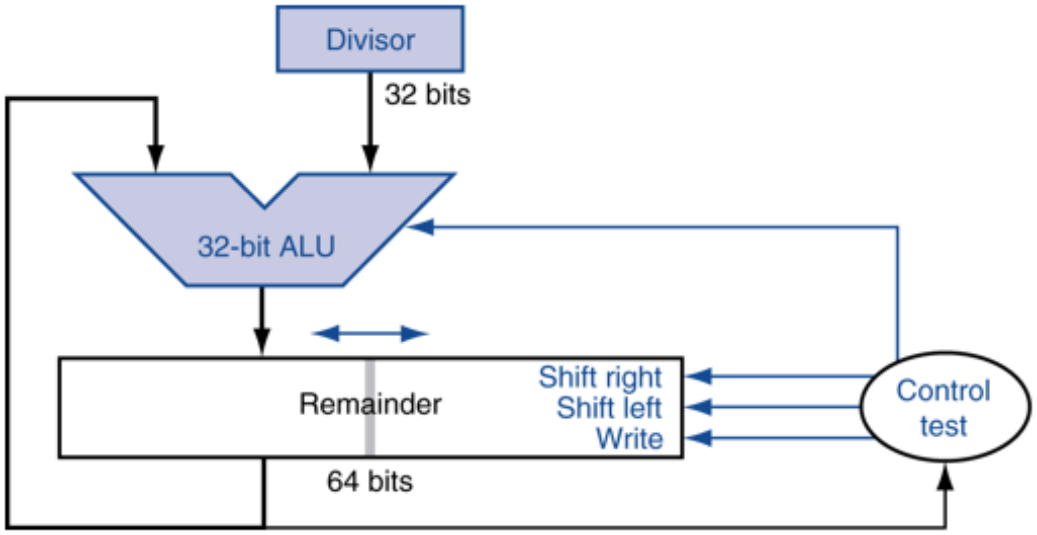
优化后的乘法器



除法器



优化后的除法器



浮点数

IEEE754详解（最详细简单有趣味的介绍）-CSDN博客
文章浏览阅读9.2w次，点赞672次，收藏2.5k次。本文详细解析了IEEE754浮点数在内存中的存储方式，包括符号位、指数位和尾数位的作用。通过对32位单精度浮点数的分析，展示了其取值范围和精度计算，揭示了精度大约为7位有效数的原因。此外，还讨论了非规格数、特殊数（±0、±infinity、NaN）的状态和作用，以及浮点数精度计算的两种理解角度。最后，澄清了一些关于32位浮点数的误解，如其能存储的整数和小数范围。

	符号位	阶码	尾数
单精度	1	8	23
双精度	1	11	52

阶码表示

	全0	全1，且尾数为0	全1，但尾数不为0	规格数
单精度	2^{1-127}，尾数隐藏位是0	无穷大	NaN	$[2^{1-127}, 2^{254-127}]$
双精度	2^{1-1023}，尾数隐藏位是0	无穷大	NaN	$[2^{1-1023}, 2^{2046-1023}]$

浮点数加法运算

对阶

阶码小的向阶码大的看齐：如果移位发生数据丢失，丢的也只是尾数最右边的数据位，只会影响精度不会影响数据的整体大小。如果向小阶码对其，大数的最高位可能被丢失，导致整体数据出现错误。

尾数求和

使用补码加法进行相加

规格化

需要表示成$1.xxx \times 2^y$

舍入

IEEE754标准中的4种舍入模式_就近舍入-CSDN博客
文章浏览阅读1.3w次，点赞39次，收藏105次。一、前言最近在写一个基于IEEE754标准的浮点加法器，其中有一项要求就是要满足IEEE754标准的四种舍入模式。我们在进行对阶或者右规格化的时候，阶数较小的操作数在进行右移的时候，会造成尾数部分的低位丢失，从而会造成误差。因此我们才需要根据需求，采取四种舍入模式中的一种对尾数进行舍入操作以减少误差。二、IEEE754标准中的4种舍入模式1、就近舍入：即十进制下的四舍五入。但是也会出现以下几种情况：多余数字是1001，它大于0.5，故最低位进
数据的长度超过了计算机当中存储数据的物理器件所保存的数据长度。低位部分就要进行处理
https://blog.csdn.net/qq_39507748/article/details/110219526

就近舍入

多余数字	舍入情况
x_1001	尾数大于0.5，加1
x_0111	尾数小于0.5，直接丢弃
0_1000	尾数等于0.5，判断除去要舍弃的数的奇偶，为偶数直接舍弃
1_1000	尾数等于0.5，判断除去要舍弃的数的奇偶，为奇数加1

向0舍入

多余数字	舍入情况
正数	直接舍去
负数	直接舍去

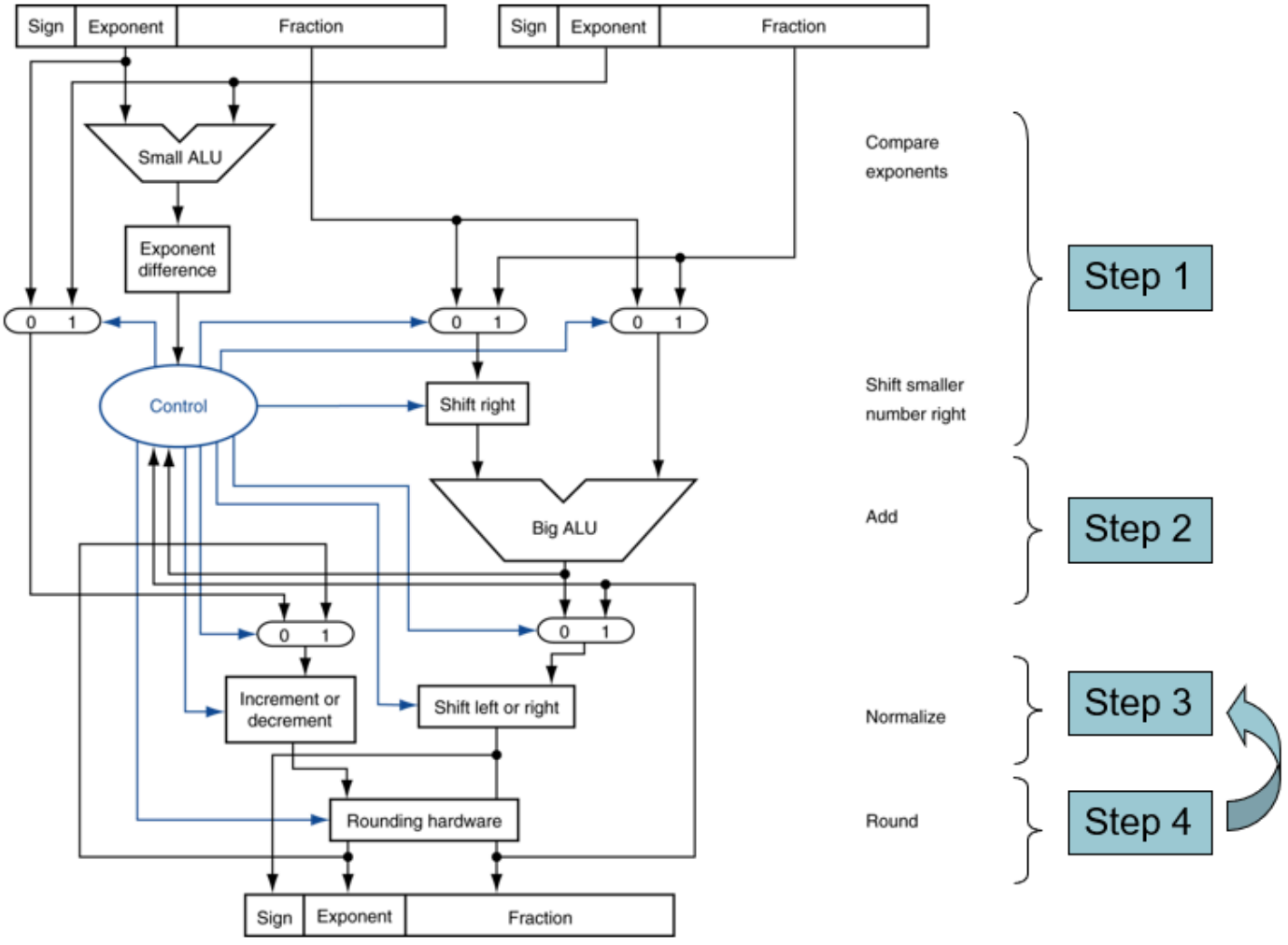
向正无穷舍入

多余数字	舍入情况
正数_0010（不全为0）	加1
正数_0000	直接丢弃
负数	直接丢弃

向负无穷舍入

多余数字	舍入情况
正数	直接丢弃
负数_0000	直接丢弃
负数_0010（不全为0）	加1

浮点数加法硬件



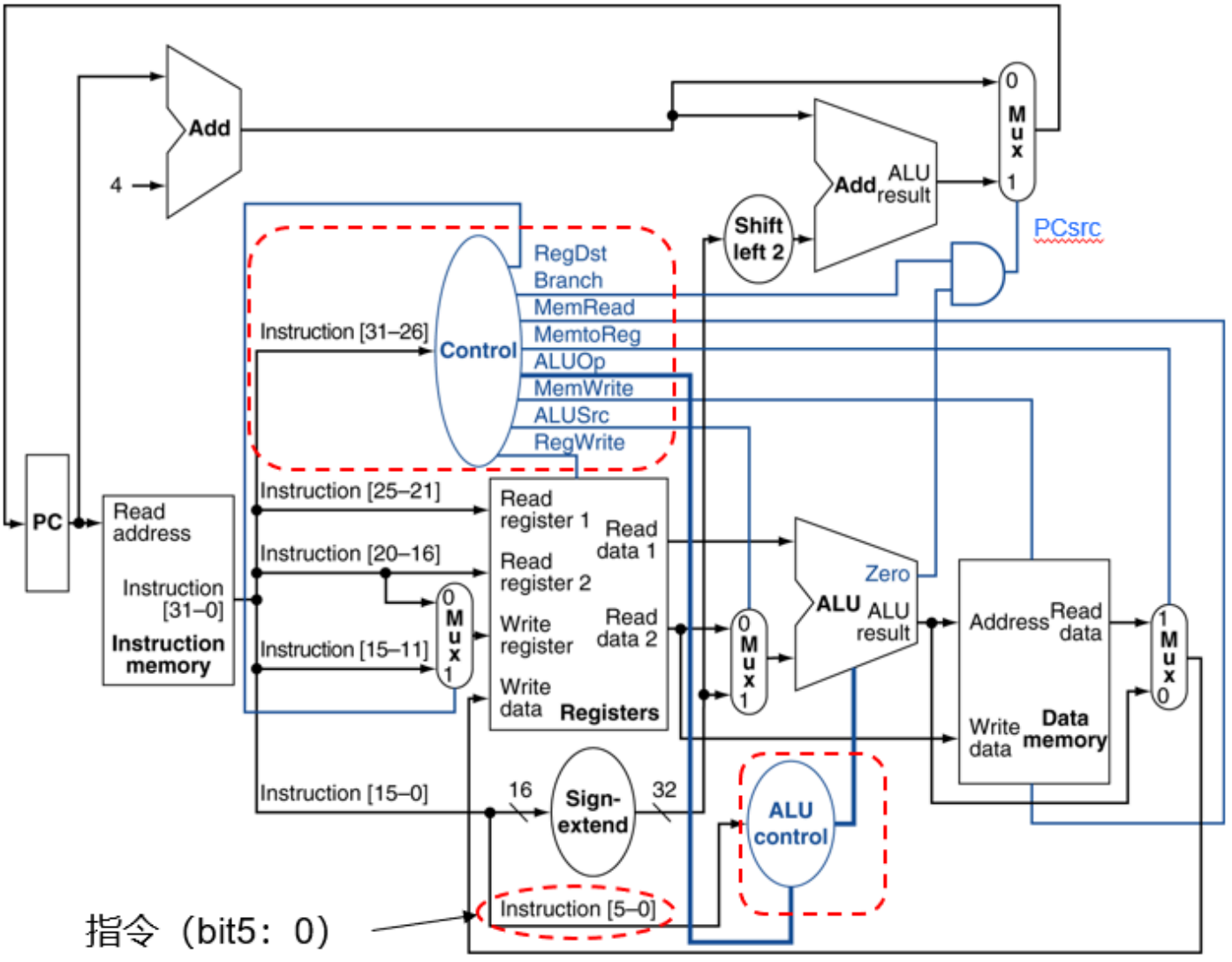
处理器

ALU控制信号

opcode	ALUOp	Operation	funct	ALU function	AL

lw	00	load word	XXXXXX	add	00
sw	00	store word	XXXXXX	add	00
beq	01	branch equal	XXXXXX	subtract	01
R-type	10	add	100000	add	00
		subtract	100010	subtract	01
		AND	100100	AND	00
		OR	100101	OR	00
		set-on-less-than	101010	set-on-less-than	01

控制单元



信号名	无效时的含义	有效时的含义
RegDst	写寄存器操作的目标寄存器号来源于指令rt字段(bits 20:16).	写寄存器操作的目标寄存器号来源于指令的rd字
RegWrite	无	寄存器堆写使能
ALUSrc	ALU第二个输入来源于寄存器堆的第二个输出	ALU第二个输入来源于指令的低16位（目标地址
PCSrc	顺序执行，取 PC + 4.	跳转，使用目标地址替代PC+4
MemRead	无	数据存储器读使能
MemWrite	无	数据存储器写使能
MemtoReg	写入寄存器的值来源于ALU.	写入寄存器的值来源于数据存储器
Branch	不跳转	分支跳转
ALUop(2bit)	基本都有效	控制ALU使用什么类型运算（结合func，然后选

控制信号真值表

Input or output	Signal name	R-format	lw	sw	beq
Inputs	op5	0	1	1	0
	op4	0	0	0	0
	op3	0	0	1	0
	op2	0	0	0	1
	op1	0	1	1	0
	op0	0	1	1	0
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0

	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	AluOp0	0	0	0	1

指令流水线

流水线的吞吐量

- 最大吞吐量：流水线在达到稳定状态后所得到的吞吐率

> $$T_{pmax} = \frac{1}{\Delta t}$$

- 实际吞吐量：m段流水线完成n个任务所达到的吞吐率

> $$T_p = \frac{n}{((m-1)+n) * \Delta t}$$

流水线的加速比

- 加速比：非流水线与m段流水线的速度比

> $$S_p = \frac{n * m * \Delta t}{((m-1)+n) * \Delta t} = \frac{n * m}{m-1+n}$$

流水线冒险

结构冒险

- 某个资源使用的上的冲突
  - 存取操作都需要访问内存，取指令将因此而停顿，引起流水线中的气泡“bubble”

数据冒险

指令依赖于前面某条执行的计算结果

前推/旁路

- 计算结果出来就加以利用

- 不要等待计算结果保存到目的寄存器
- 需要从计算结果到“使用处”的额外数据通路

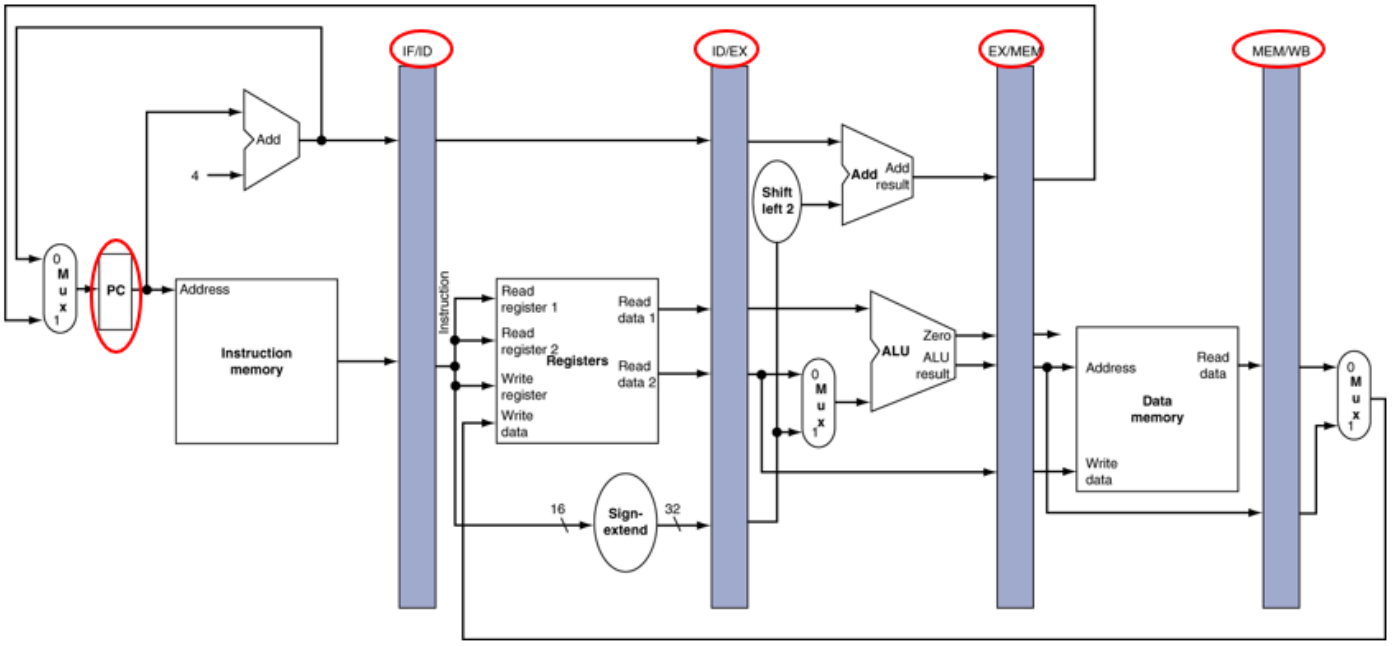
- 并非所有冒险都能通过前推来避免气泡停顿
  - 在需要使用数据时，输入数据并未计算出来，即使用前推也没有用

控制冒险

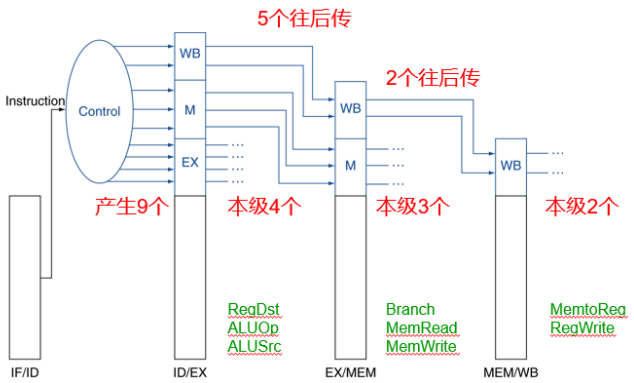
- 分支将决定控制流
  - 下一条要取出的指令取决于分支指令的输出
  - 流水线未必能取到应该执行的下一条指令

流水线寄存器

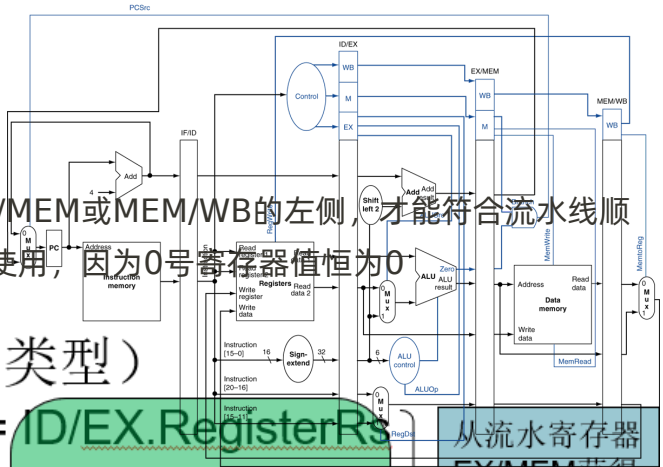
各级之间需要增加流水线寄存器，用于记录前一时钟周期产生的结果



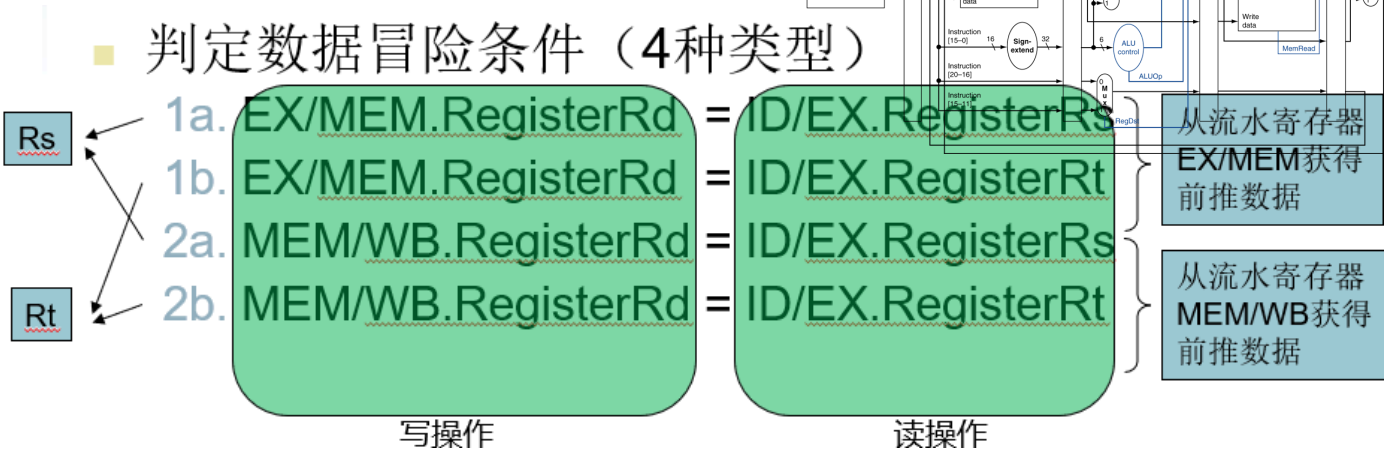
流水线控制信号



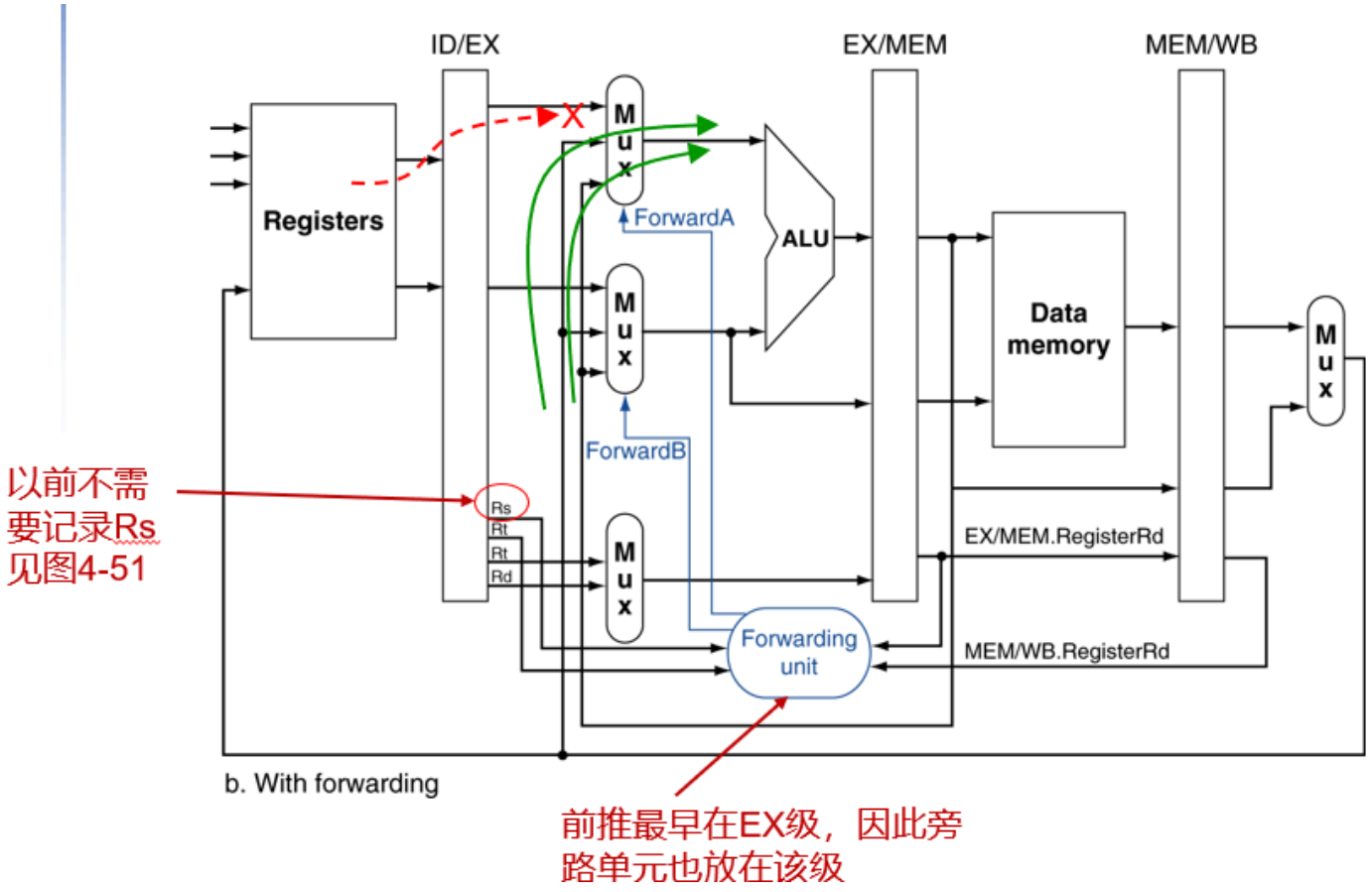
数据冒险&旁路/前推



这个图感觉画的不好，ID/EX流水线寄存器应该在EX/MEM或MEM/WB的左侧，才能符合流水线顺序。需要注意以下的前推在遇到寄存器$zero时无需使用，因为0号寄存器值恒为0



流水线示意图



前推控制信号

▾ 代码实现前推控制信号

```
//EX 冒险（rs/rt在EX/MEM检测到相应的rd）
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))// 1a
  ForwardA = 10
if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))// 1b
  ForwardB = 10

//MEM 冒险（ rs/rt在MEM/WB检测到相应的rd ）
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))// 2a
  ForwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))// 2b
  ForwardB = 01

```

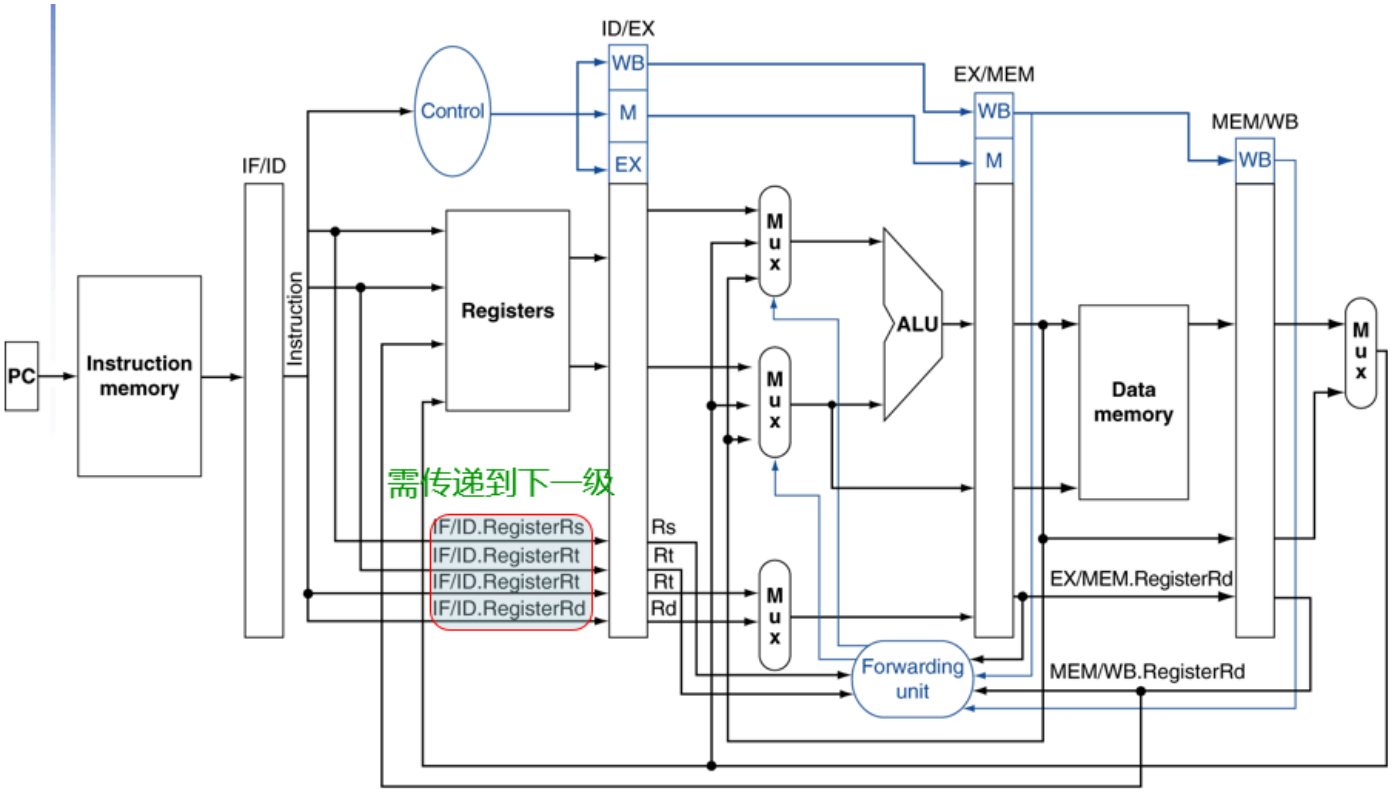
双重数据冒险

- EX冒险和MEM冒险同时存在
  - 使用EX提供的结果，而不是使用更早期的MEM结果
- 修订MEM冒险的的控制信号，只有EX冒险不成立时，才对MEM冒险前推

修订后的MEM前推条件

```
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
            and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) // 2a
  ForwardA = 01
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
            and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))// 2b
  ForwardB = 01

```



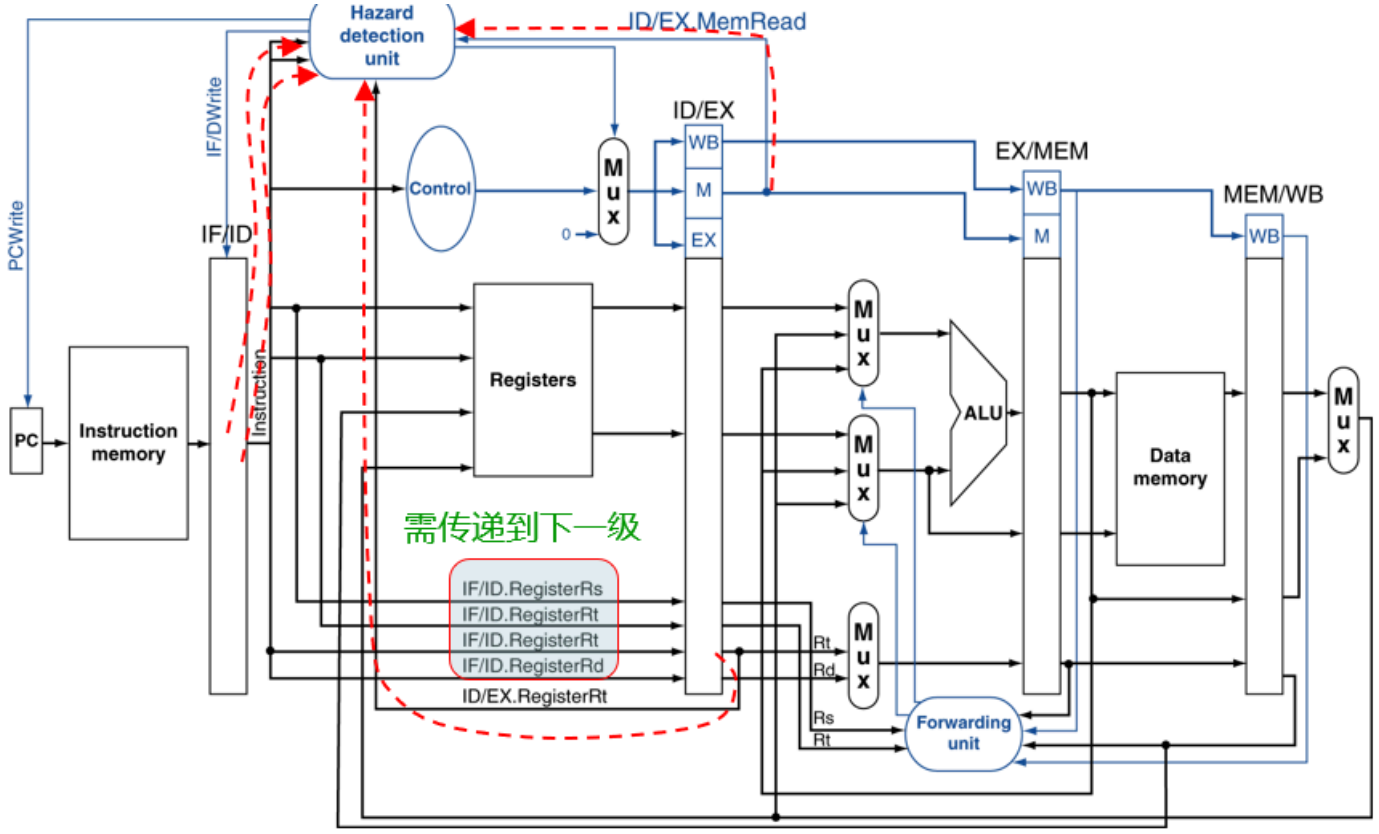
取数lw：冒险与阻塞

- 在指令译码ID阶段进行检测寄存器的使用，发现该类型冒险后，需要阻塞插入气泡

取数冒险检测

```
ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt))
```

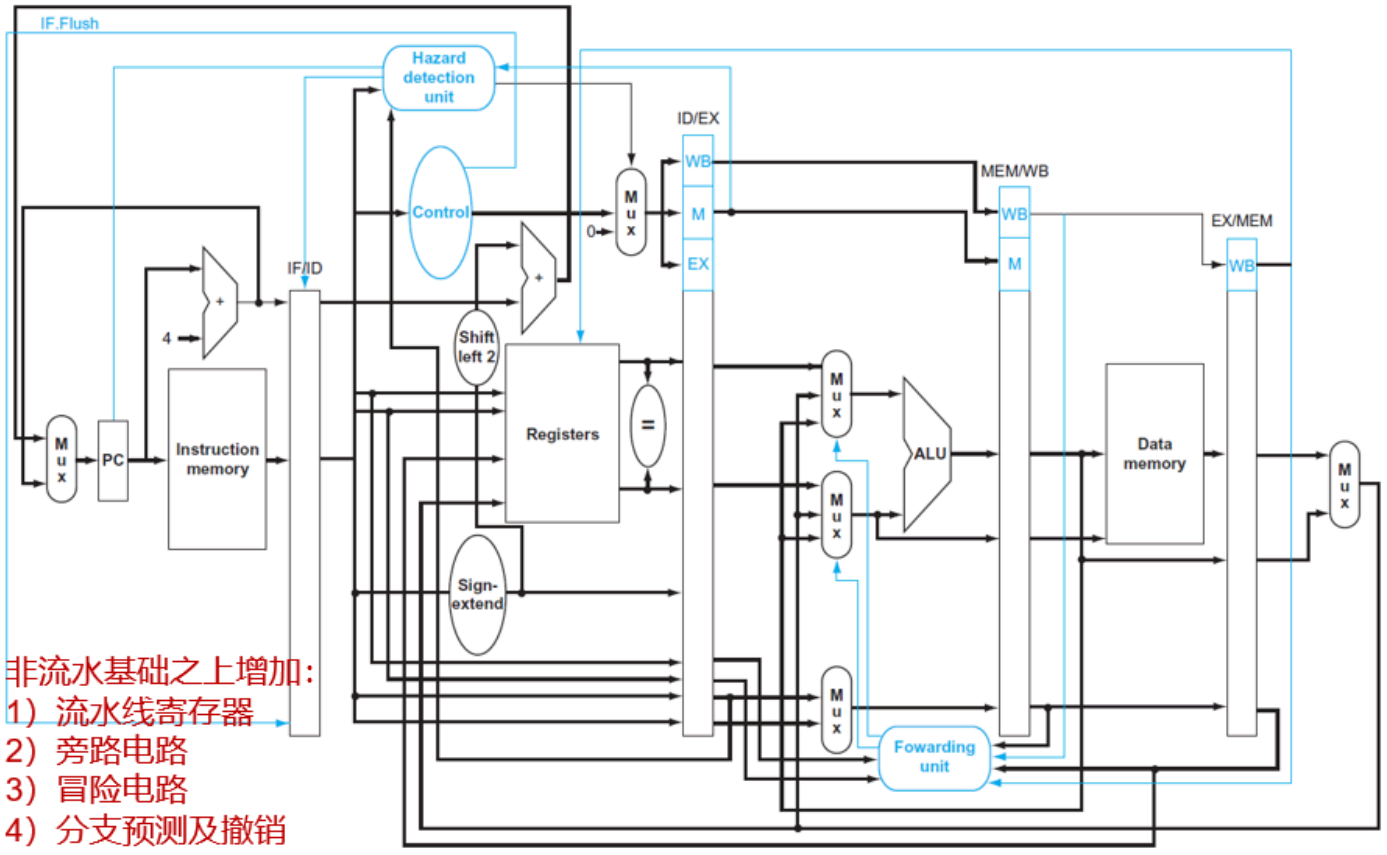
- 实现阻塞：
  - 将ID/EX寄存器控制信号全为0
  - 组织PC和IF/ID寄存器的更新（维持不变）



控制冒险/分支冒险

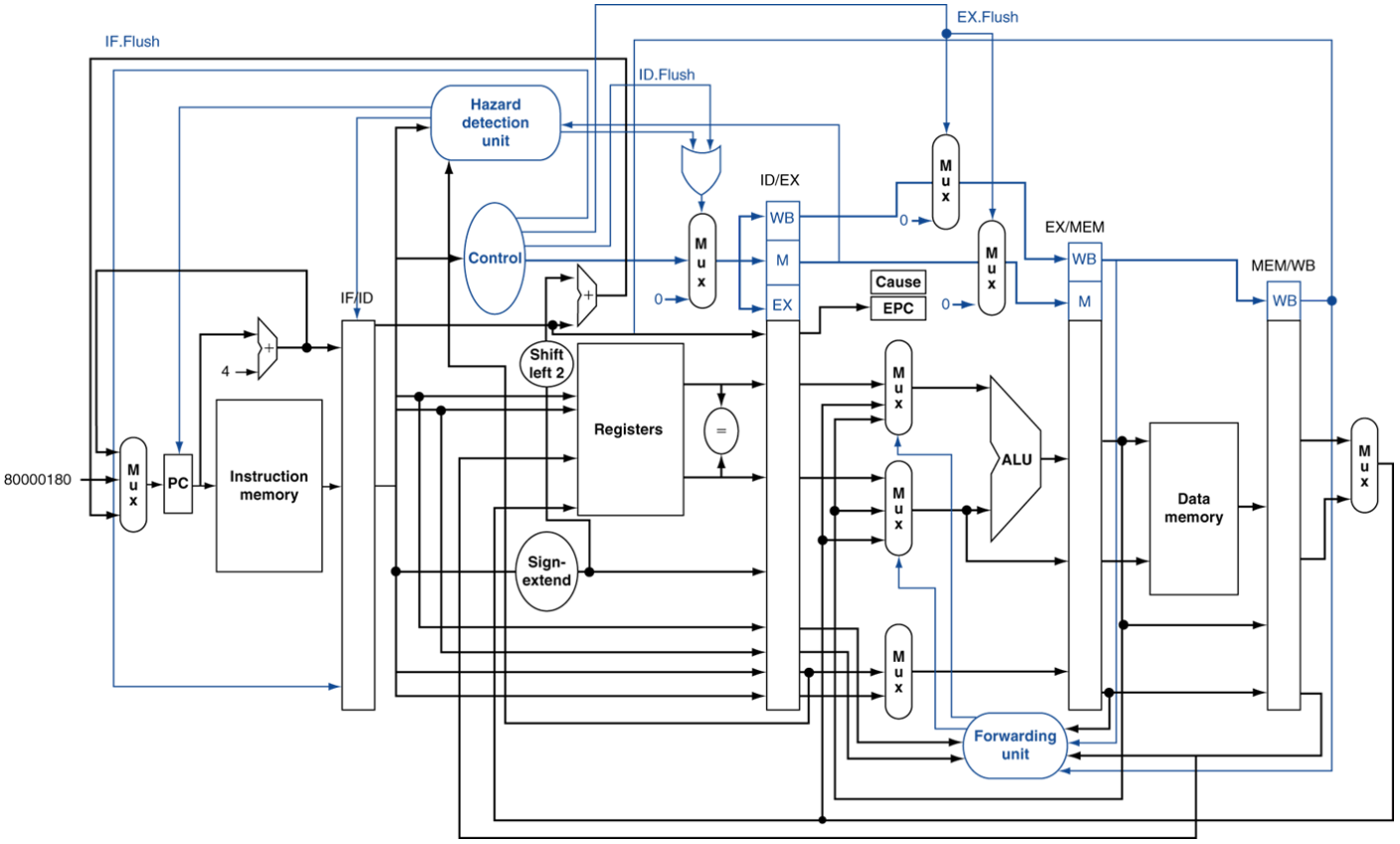
- 分支目标缓存
- 延迟槽

流水线全图



带有异常处理的流水线

- 指令撤销（撤销的是异常以及之后的指令，之前的指令要继续完成）
  - IF级已经有IF-flush
  - ID级已有（阻塞用的）全0的控制信号，新增ID-Flush
  - EX新增EX-Flush
  - 此例子异常不会在MEM和WB产生，无须后续的撤销动作



存储

磁盘访问

- 扇区大小512B, 转速15,000rpm, 平均寻道时间4ms, 数据传输率100MB/s , 控制器延时0.2ms

平均读时间

- 4ms 寻道时间
  + ½ / (15,000/60) = 2ms 旋转延迟
  + 512 / 100MB/s = 0.005ms 传输时间
  + 0.2ms 控制器延时
  = 6.2ms

Cache

直接相连

全相联

多路组相连

替换策略

- 直接相联：没得选，哪里冲突换哪里
- 组相联：
  - 优先替换存储非有效信息（non-valid）的块
  - 如果没有non-valid块，选择组中最近最少使用（LRU）的块
- 随机替换：对于相联度比较高的情况，和LRU获得近似相同的结果

多级Cache

假设

- CPU 基准 CPI = 1, 时钟频率 = 4GHz
- 缺失率/指令 = 2%
- 主存访问时间 = 100ns

仅有L1-cache

- 缺失代价 = 100ns/0.25ns = 400 时钟周期
- 有效 CPI = 1 + 0.02 × 400 = 9

如果增加 L2-cache

- 访问时间 = 5ns
- 对于主存的整体失效率 = 0.5%

L1失效& L2 命中

- 缺失代价 = 5ns/0.25ns = 20 时钟周期

L1失效& L2 失效

- 额外缺失代价 =400时钟周期

CPI = 1 + 0.02 × 20 + 0.005 × 400 = 3.4

性能比 = 9/3.4 = 2.6