计系3 期末速通教程

4. 处理器

4.1 指令的执行

[指令的执行]

(1) 各指令的前两步相同:

① PC 从指令所在的存储单元取指.

② 根据寄存器编号从寄存器文件中读取寄存器内容.

(2) 后续操作与指令类型有关:

① 除跳转指令外, 其它指令经过 ALU :

(i) 算术逻辑指令的计算.

(ii) 存储访问指令计算访存地址.

(iii) 分支指令计算判定条件.

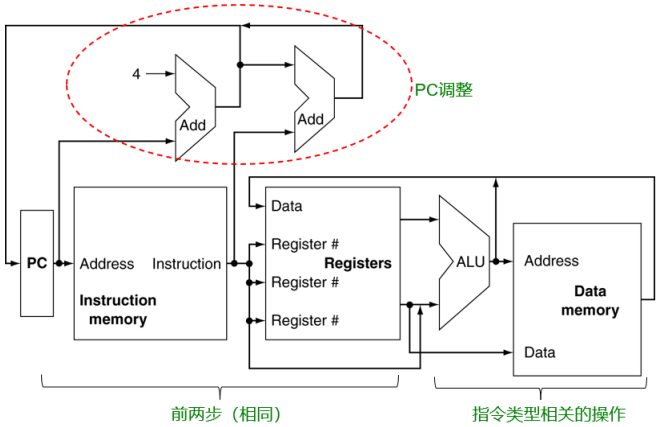
② 存储访问指令读写内存.

③ 算术逻辑指令、`lw` 指令将 ALU 的运算结果写入寄存器.

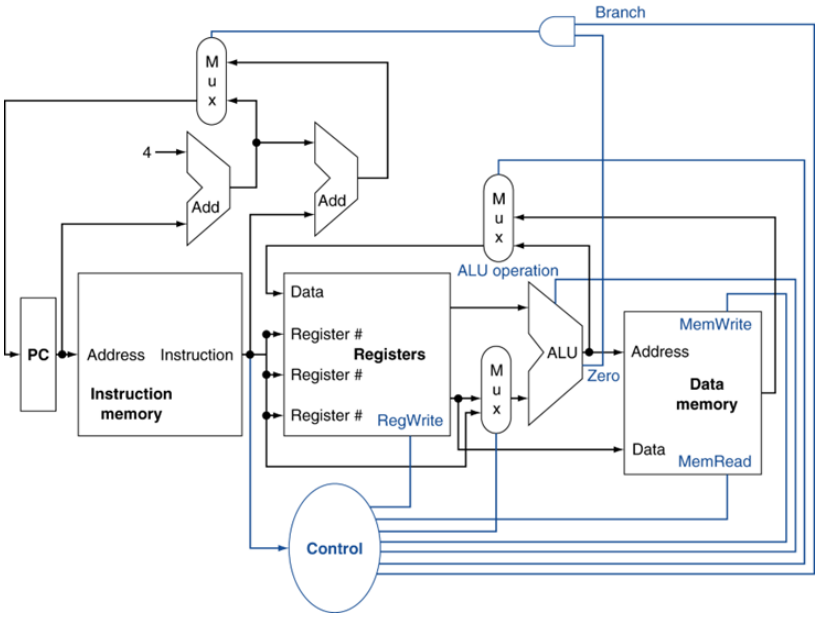
(3) $PC+=4$ 或 $PC=$ 目标跳转地址.

[CPU的抽象视图]

(1) MIPS子集实现的抽象视图:



(2) 在有多个数据来源的地方加复选器, 并加入对应的控制信号:



4.2 数据通路

[数据通路的部件]

(1) 数据通路部件: 用来操作或保存处理器中数据的单元, 如指令存储器、ALU .

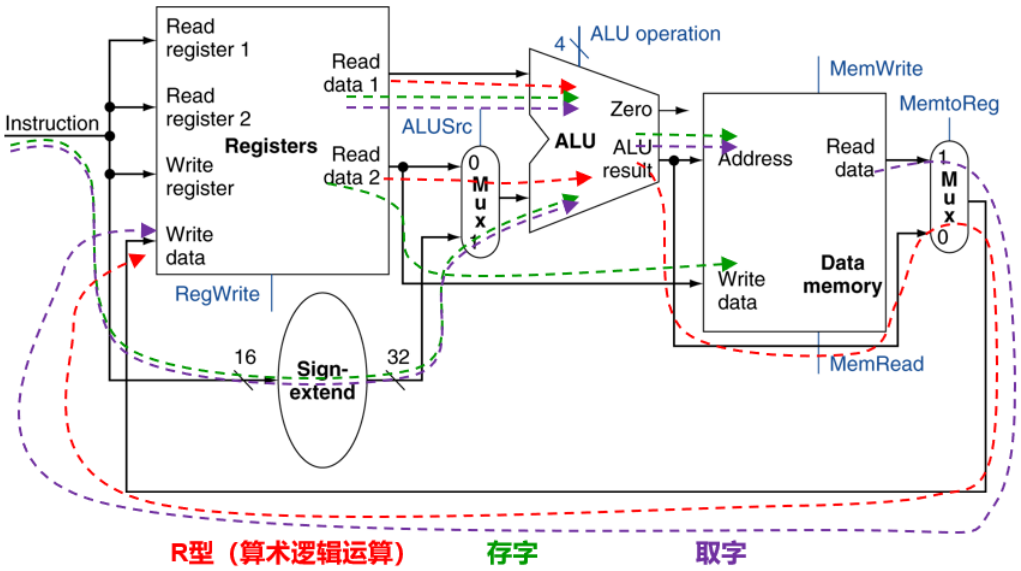
(2) 程序计数器PC: 存放下一条指令地址的寄存器.

[简单的数据通路]

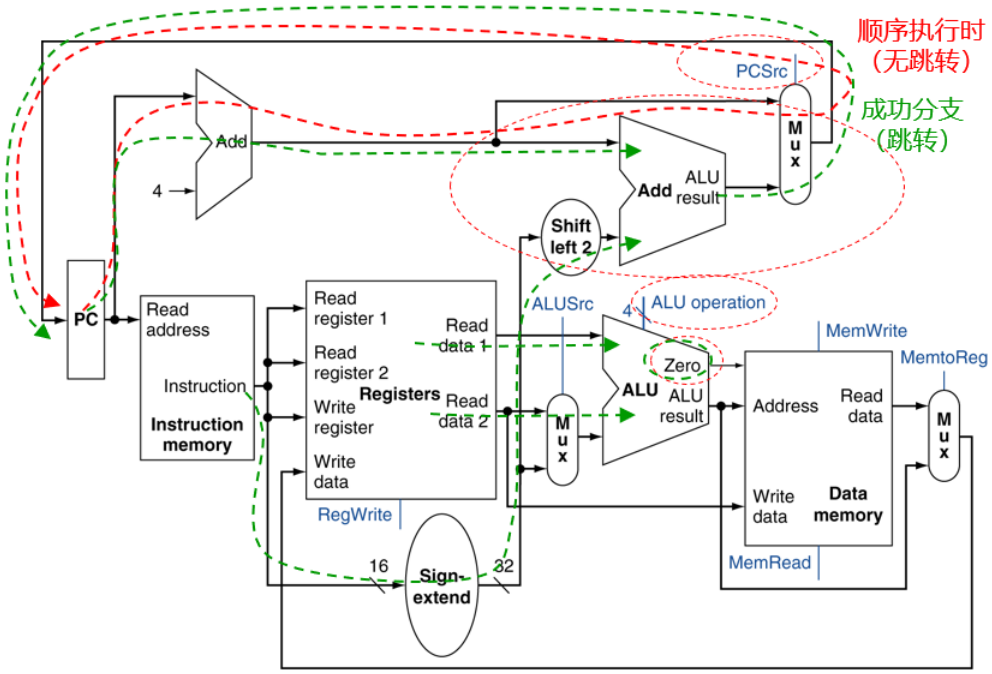
(1) ① 每个数据通路部件在一个时钟周期内只能处理一条指令, 故需独立的数据寄存器和指令寄存器.

② 不同指令的数据来源不同时, 需用复选器.

(2) R型指令和存取指令的数据通路:



(3) 分支指令的数据通路:



4.3 控制单元

[ALU的控制信号]

(1) ALU的控制信号与功能(运算操作):

ALU的控制信号	功能 F (运算操作)
0000	and
0001	or
0010	add
0110	subtract
0111	小于则置 1 (`slt`)
1100	nor

(2) 指令操作码 `op` 产生 2 bit 的 `ALU op` , 用于区分:

① 计算访存地址: add .

② `beq` : subtract .

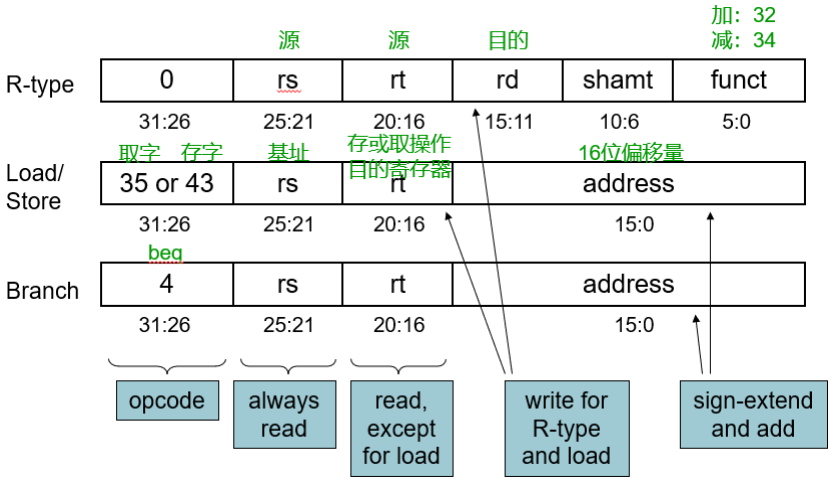
③ 算术逻辑运算: add 、subtract 、and 、or 、 slt .

再由组合逻辑产生ALU的控制信号:

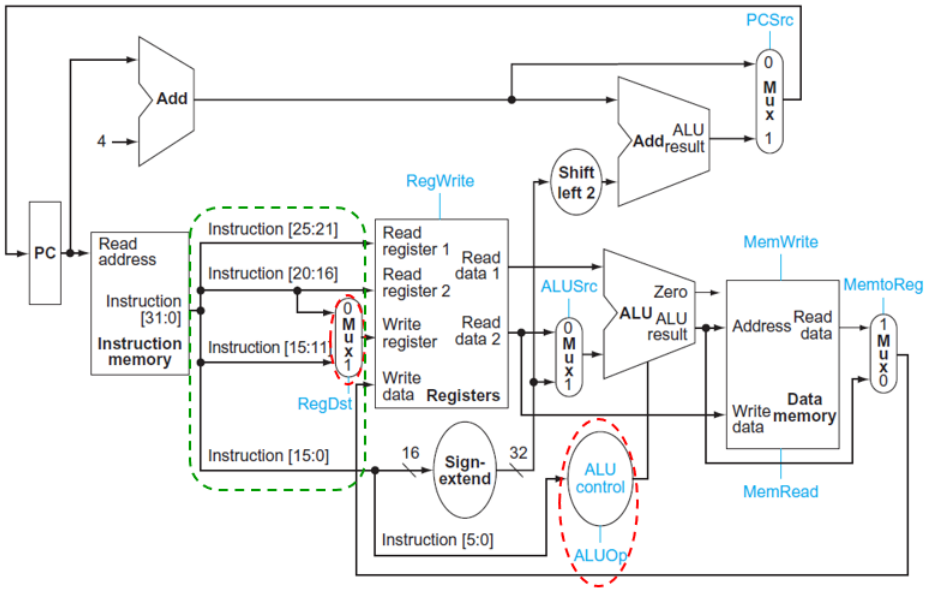
op	ALU op	operation	funct	ALU function	ALU control
`lw`	00	load word	$xxxxxx$	add	0010
`sw`	00	store word	$xxxxxx$	add	0010
`beq`	01	branch equal	$xxxxxx$	subtract	0110
R型指令	10	add	100000	add	0010
R型指令	10	subtract	100010	subtract	0110
R型指令	10	and	100100	and	0000
R型指令	10	or	100101	or	0001
R型指令	10	slt	101010	slt	0111

[主控单元的设计]

(1) 从指令编码产生控制信号:



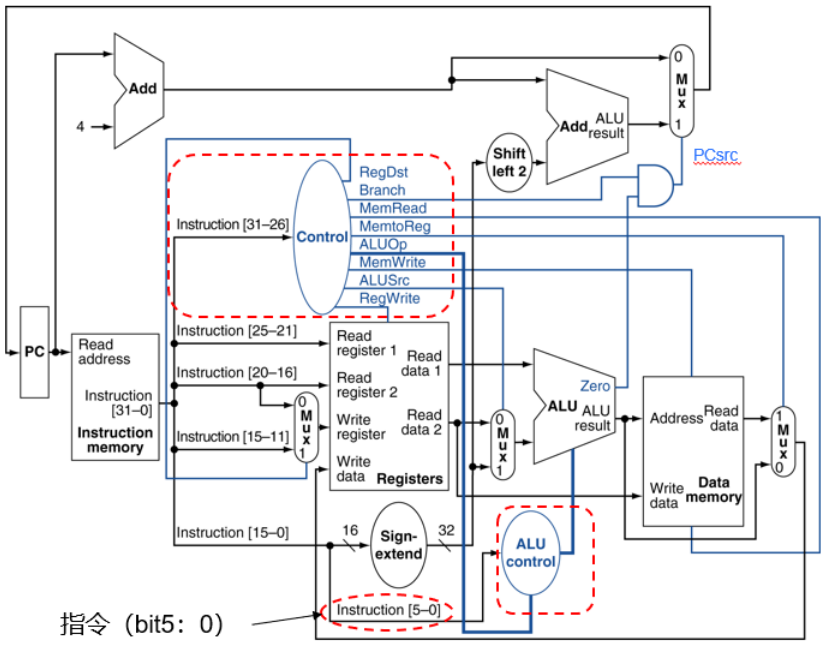
(2) 控制信号:



(3) 控制信号的含义:

除 `PCsrc` 外, 其余信号可在指令读入后通过操作码译码确定. `beq` 指令需确定比较结果后才可生成 `PCSrc` .

信号	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源于指令的 `rt` 字段	写寄存器操作的目标寄存器号源于指令的 `rd` 字段
RegWrite	/	寄存器堆写使能
ALUSrc	ALU的第二个输入源于寄存器堆的第二个输出	ALU的第二个输入源于指令的低 16 位 (目标地址的偏移量)
PCSrc	顺序执行, 取 $PC+4$	跳转, 取目标地址
MemRead	/	数据存储器读使能
MemWrite	/	数据存储器写使能
MemtoReg	写寄存器的值源于ALU	写寄存器的值源于数据存储器



[指令的控制信号]

(1) 控制信号根据指令类型的不同, 控制数据(地址)流经不同部件.

指令	RegDst	ALUSrc	Mem-toReg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp2
R型指令	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	x	1	x	0	0	1	0	0	0
`beq`	x	0	x	0	0	0	1	0	1

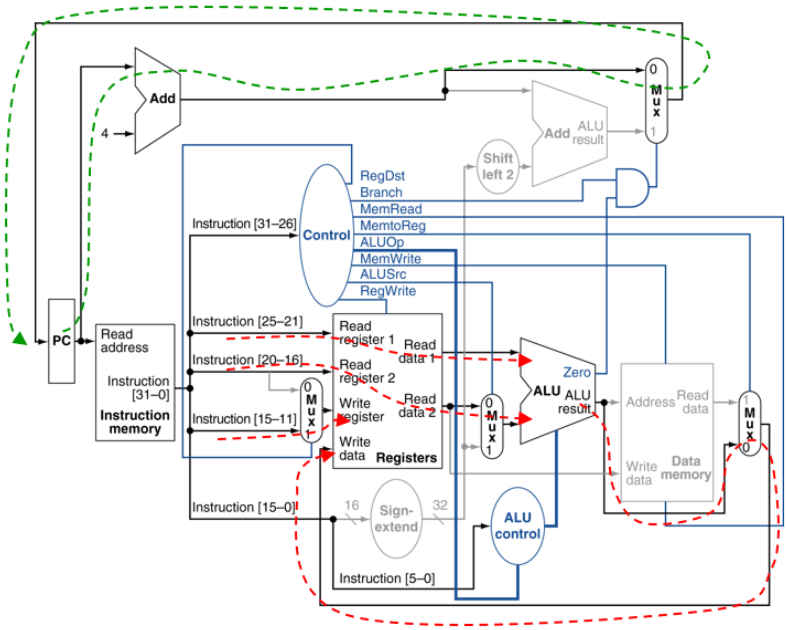
(2) R型指令:

① 取指, PC自增.

② 读两个寄存器, 产生控制信号.

③ ALU根据 `ALUOp` 计算.

④ ALU的结果写回寄存器.



(3) 取字指令: `lw $t1, offset($t2)` .

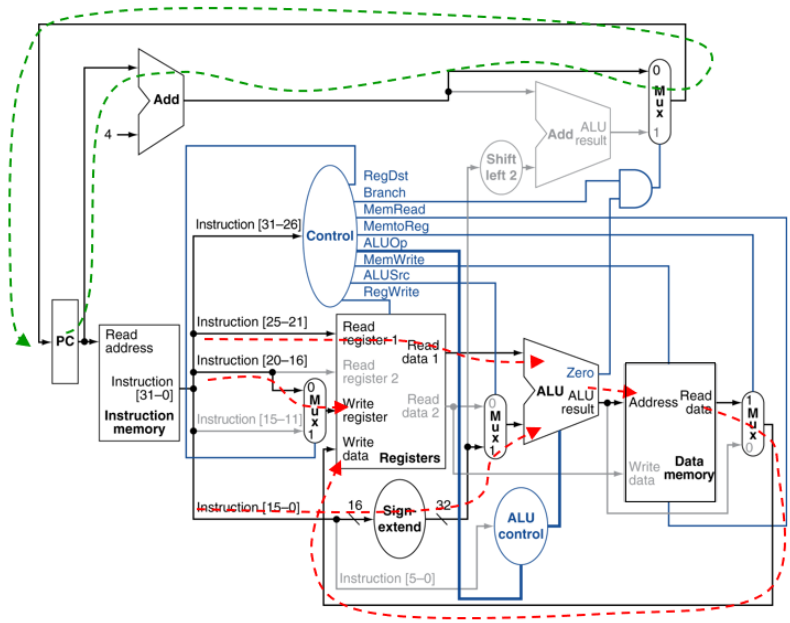
① 取指, PC自增.

② 读寄存器 `t2` , 产生控制信号.

③ ALU根据 `ALUOp` 计算地址.

④ ALU的结果作为存储器地址.

⑤ 数据存储器输出写入到 `t1` .



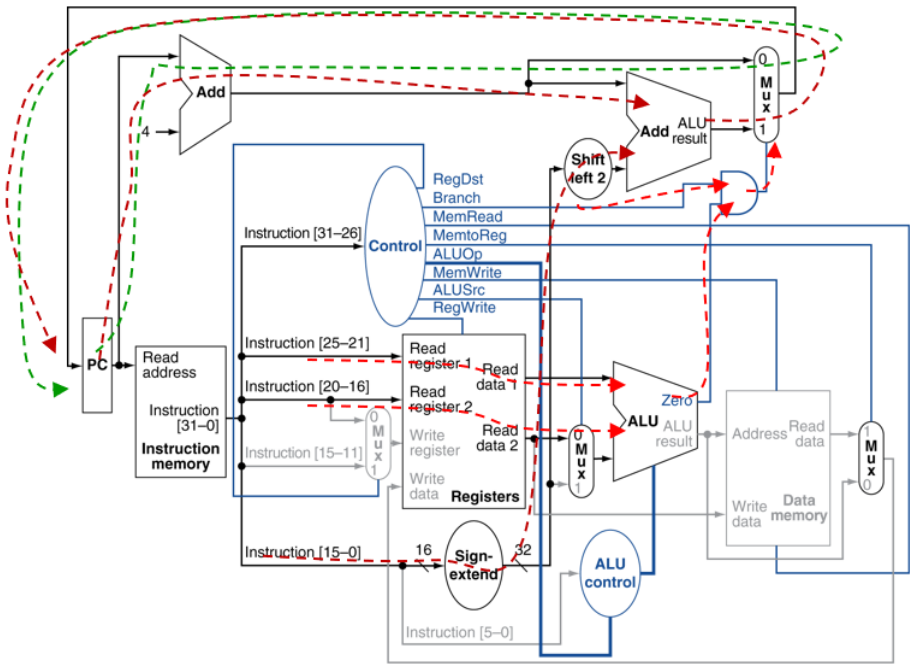
(4) 分支指令: `beq $t1, $t2, offset` .

① 取指, PC自增.

② 读寄存器 `t1` 和 `t2` , 产生控制信号.

③ ALU根据 `ALUOp` 计算减法, 产生标志.

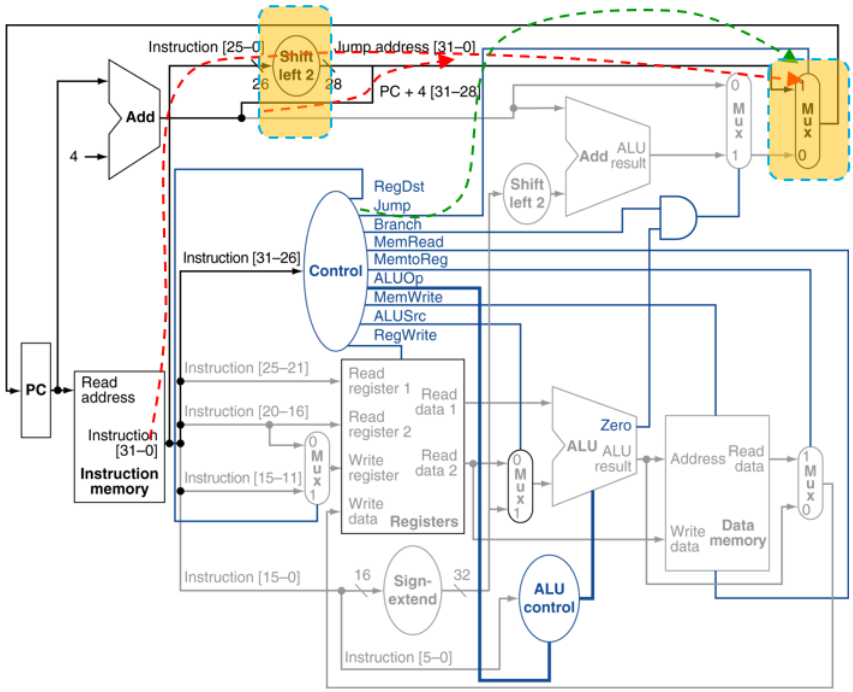
④ ALU的结果位与 `Branch` 产生 `PCSrc` .



(5) `j` 指令: 新的PC值的高 4 位与旧的 $(PC+4)$ 相同, 跳转地址有 26 bits (地址的 $27:2$ 字段), 低 2 位恒为 0 .

2	address
31:26	25:0

加入跳转指令后的数据通路:



4.4 指令的流水处理

[单周期处理器的局限性]

(1) 最慢的路径决定时钟周期.

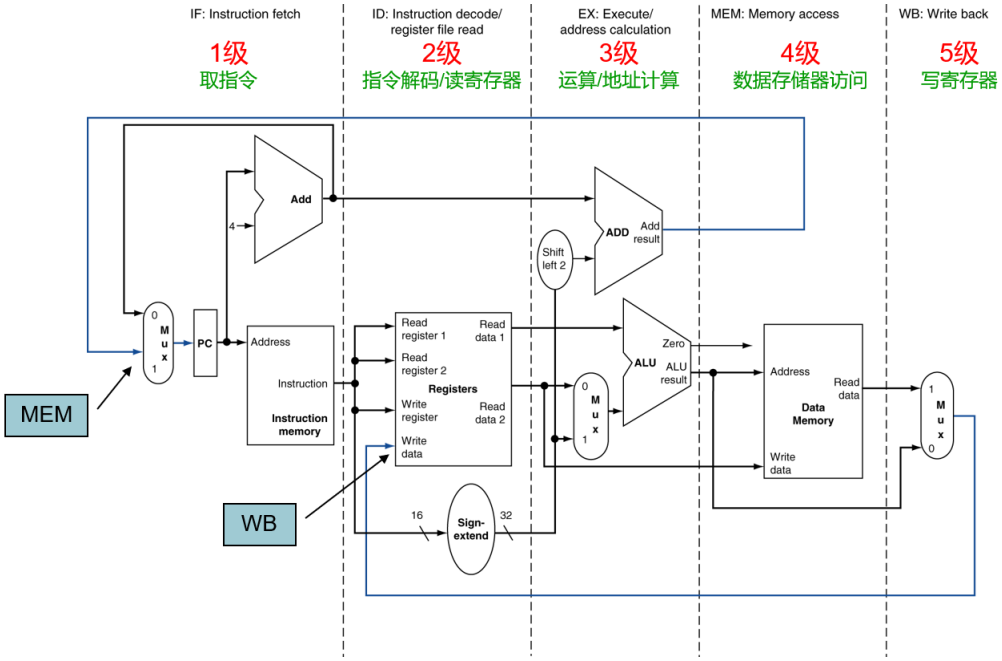
`lw` 指令最慢, 路径: 指令存储器 → 寄存器堆(读) → ALU → 数据存储器 → 寄存器堆(写).

(2) 无法应对不同指令路径(处理时间)长短的差异.

(3) 违背设计原则: 加快经常性事件.

(4) 解决方法: 用流水线处理指令.

[MIPS指令流水]



(1) 5 级流水, 每级完成一个指令步骤:

① IF : 从内存中取指令.

② ID : 指令解码, 读寄存器.

③ EX : 执行运算或计算地址.

④ MEM : 访存.

⑤ WB : 将结果写回寄存器.

(2) 流水线的加速比:

① 若流水线各级等长, 即各级处理时间相等, 假设指令序列无限长, 则 $\text{两条指令间隔}_{\text{流水}} = \dfrac{\text{两条指令间隔}_{\text{非流水}}}{\text{流水线级数}}$.

② 若流水线各级不等长, 则加速比下降. 理想加速比为 $4 \sim 5$, 实际无法达到.

③ 加速原因: 吞吐率提升.

(3) 面向流水线的 ISA 设计:

① 所有指令等长, 易在一个时钟周期内取指和译码.

② 指令格式少且规整: 解码同时可读取寄存器操作数, 否则增加一级流水.

③ 只在 `lw` 和 `sw` 指令中访存, 第 3 级的 "执行部件" 可用于计算地址, 供第 4 级访存使用, 否则需在执行部件和访存间增加一级流水用于计算地址.

④ 操作数内存地址对齐, 内存访问只需一个时钟周期.

[**流水线冒险**]

(1) 冒险现象: 下一周期不能按时执行下一条指令, 性能下降.

(2) 分类:

① 结构冒险(结构相关): 所需的部件忙, 暂不可用.

② 数据冒险(数据相关): 本次计算的输入是前面某条指令的计算结果.

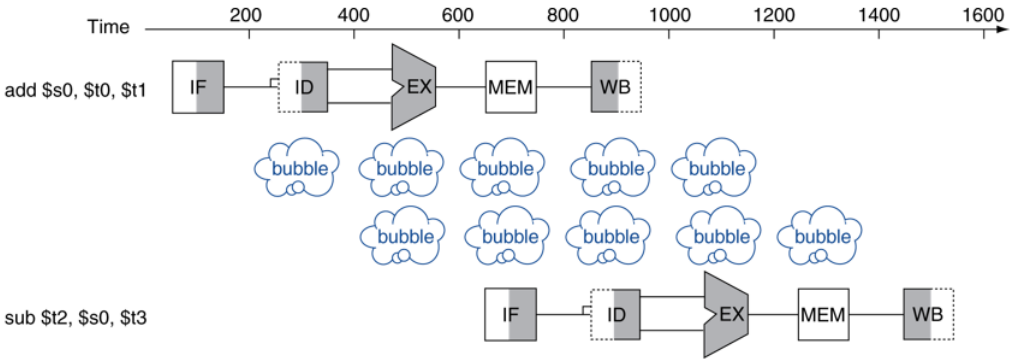
③ 控制冒险(控制相关): 需根据前面某条指令的结果确定分支的选择执行.

[**结构冒险**] 流水实现的MIPS需有分离独立的指令和数据内存, 实际实现中使用独立的 L1 cache .

[**数据冒险**]

(1) 一条指令依赖于前面某条指令的计算结果.

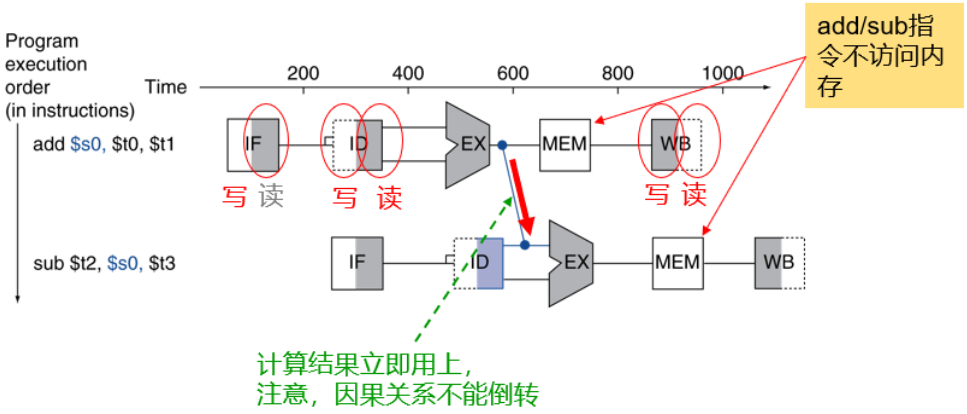
```
1    add $s0, $t0, $t1
2    sub $t2, $s0, $t3
```



若为保持因果关系而推迟执行第二条指令, 则性能下降.

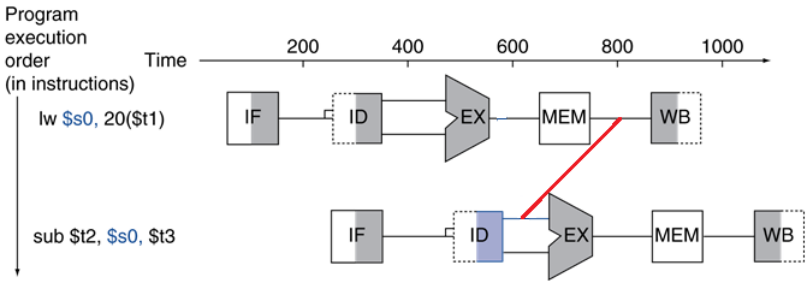
(2) 直接通路(旁路, bypassing)或前推(forwarding): 得到计算结果后即应用, 不等待计算结果写回寄存器.

实现: 从计算结果到 "使用处" 连额外的数据通路.



(3) 取数-使用型冒险: 并非所有的数据冒险都可通过前推避免阻塞, 如需使用数据时还未计算出结果, 此时前推无法也无法避免阻塞, 故只能前推 + 阻塞.

```
    lw $s0, 20($t1)
    sub $s2, $s0, $t3
```



指令调度: 编译器调整指令顺序, 避免在取数后立即使用该数据, 避免阻塞.

下面的代码有 2 处取数-使用型冒险, 共需 13 个时钟周期.

```
    lw $t1, 0($t0)
    lw $t2, 4($t0)
    add $t3, $t1, $t2  # stall
    sw $t3, 12($t0)
    lw $t4, 8($t0)
    add $t5, $t1, $t4  # stall
    sw $t5, 16($t0)
```

调整后的代码如下, 共需 11 个时钟周期.

```
    lw $t1, 0($t0)
    lw $t2, 4($t0)
    lw $t4, 8($t0)
    add $t3, $t1, $t2
    sw $t3, 12($t0)
    add $t5, $t1, $t
    sw $t5, 16($t0)
```

[控制冒险]

(1) 分支将决定控制流:

① 下一条要取出的指令取决于分支指令的输出.

② 流水线未必能取到应执行的下一条指令, 需阻塞(stall).

(2) MIPS流水中在分支指令的 ID 阶段加入硬件, 以支持寄存器比较和分支地址计算, 以尽早地进行寄存器比较和生成目标地址.

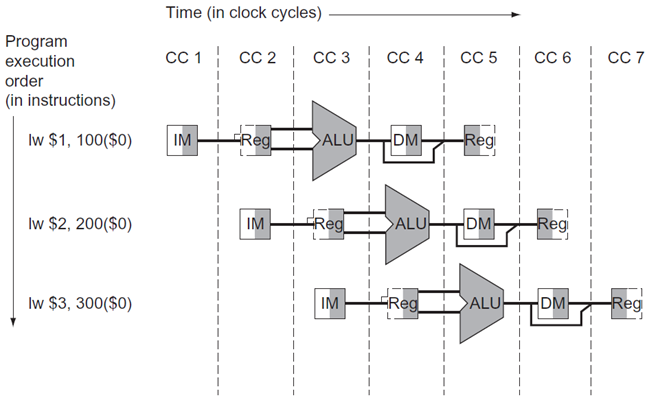
(3) 分支预测: MIPS 预测不跳转的分支, 在分支指令后继续取下一条指令, 无需阻塞, 预测错误时才需阻塞.

① 向后的分支: 预测不发生.

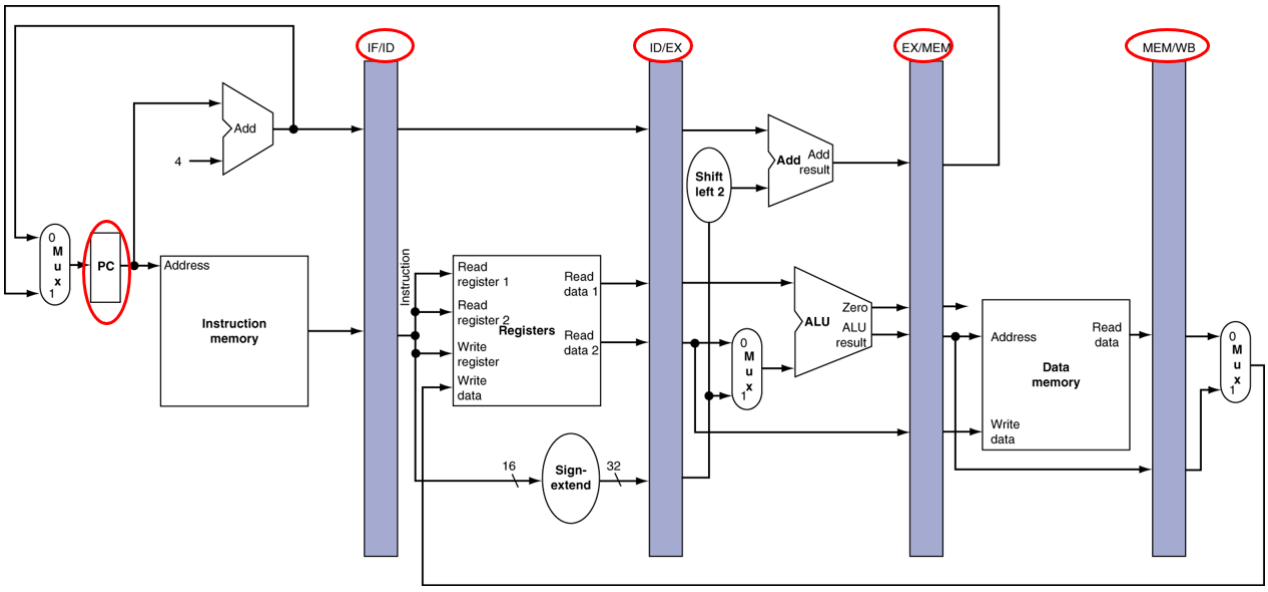
② 向前的分支: 预测发生.

4.5 流水化数据通路

[多周期流水线图] 以指令为中心, 采用 "部件" 而非 "操作" 表示流水过程 .

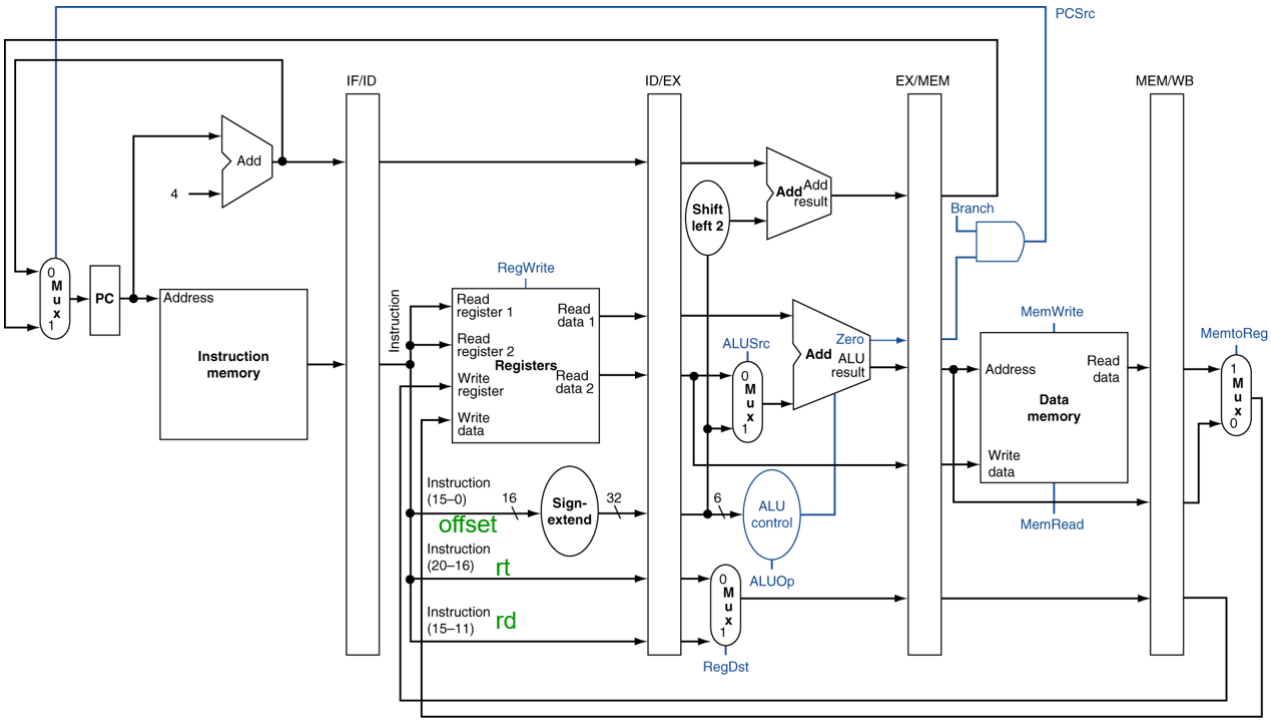


[流水线寄存器] 各级流水间添加流水线寄存器, 记录前一时钟周期产生的结果. 暂不考虑冒险.



[多周期流水线控制信号]

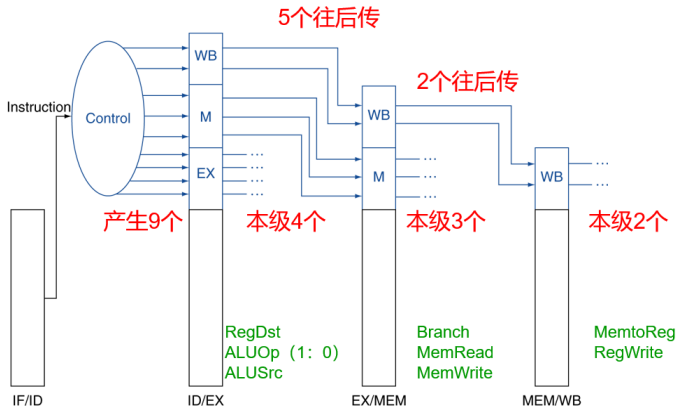
(1) 控制信号分级. 注意 `RegDst` 的位置与单周期流水线不同.



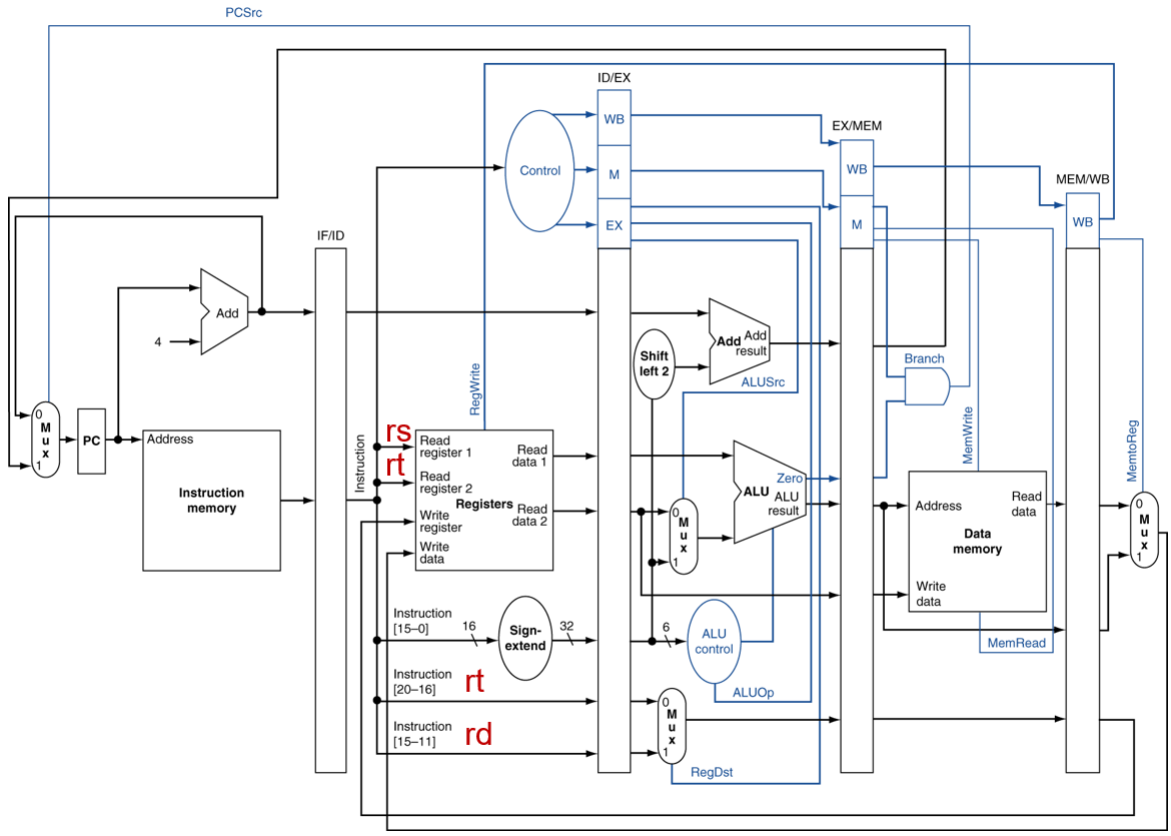
(2) 第三级使用的 ALUOp 信号与单周期流水线相同, 按流水线级分组得:

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

(3) 控制信号保存到流水线寄存器中下传.



(4) 流水线全景:



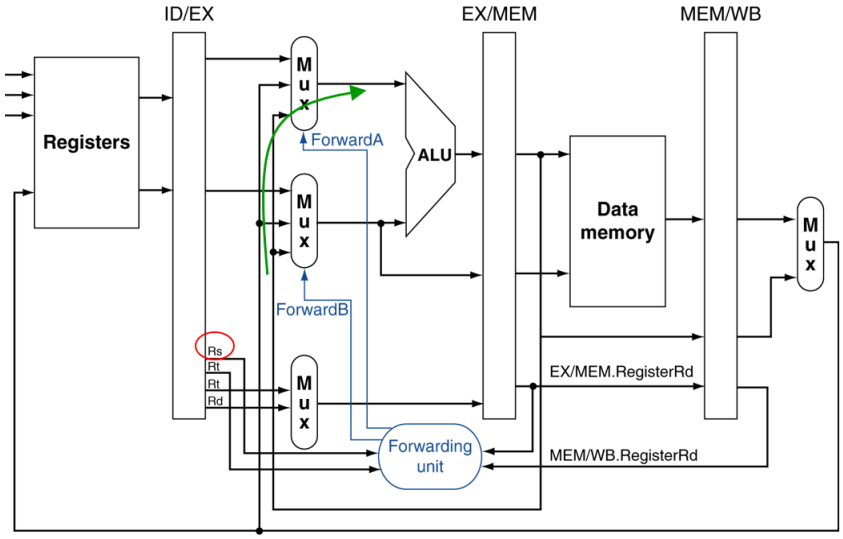
4.6 数据冒险

[数据冒险的判定]

冒险类型	说明 / 判定条件
1 型	从流水线寄存器 `EX/MEM` 获得前推数据
1a	$\mathrm{EX/MEM.Rd} = \mathrm{ID/EX.Rs}$
1b	$\mathrm{EX/MEM.Rd} = \mathrm{ID/EX.Rt}$
2 型	从流水线寄存器 `MEM/WB` 获得前推数据
2a	$\mathrm{MEM/WB.rd} = \mathrm{ID/EX.Rs}$
2b	$\mathrm{MEM/WB.rd} = \mathrm{ID/EX.Rt}$

[**前推通路**] 前推最早在 EX 级, 则前推单元也在该级.

(1) 有前推单元的流水线:



(2) ALU 的 mux 的控制信号:

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

(3) 控制信号的产生: 因 `$zero` 的值恒为 0 , 故无需检测冒险.

① EX 冒险: rs / rt 在 EX/MEM 中检测到相同的 rd .

(i) if (EX/MEM.RegWrite && (EX/MEM.Rd ≠ 0) && (EX/MEM.Rd = ID/EX.Rs)) ForwardA = 10

(ii) if (EX/MEM.RegWrite && (EX/MEM.Rd ≠ 0) && (EX/MEM.Rd = ID/EX.Rt)) ForwardB = 10

② MEM 冒险: rs / rt 在 MEM/WB 中检测到相同的 rd . EX 冒险不成立时, 才前推 MEM 冒险.

(i) if (MEM/WB.RegWrite && (MEM/WB.Rd ≠ 0) && (MEM/WB.Rd = ID/EX.Rs) &&

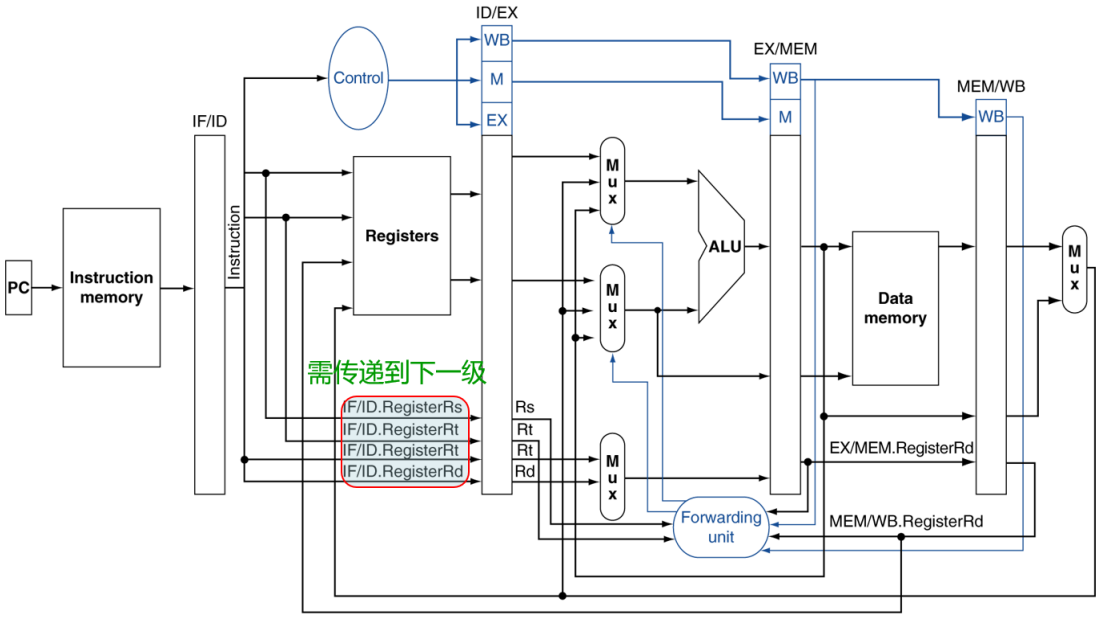
!(EX/MEM.RegWrite && (EX/MEM.Rd ≠ 0) && (EX/MEM.Rd ≠ ID/EX.Rs))) ForwardA = 01

(ii) if (MEM/WB.RegWrite && (MEM/WB.Rd ≠ 0) && (MEM/WB.Rd = ID/EX.Rt) &&

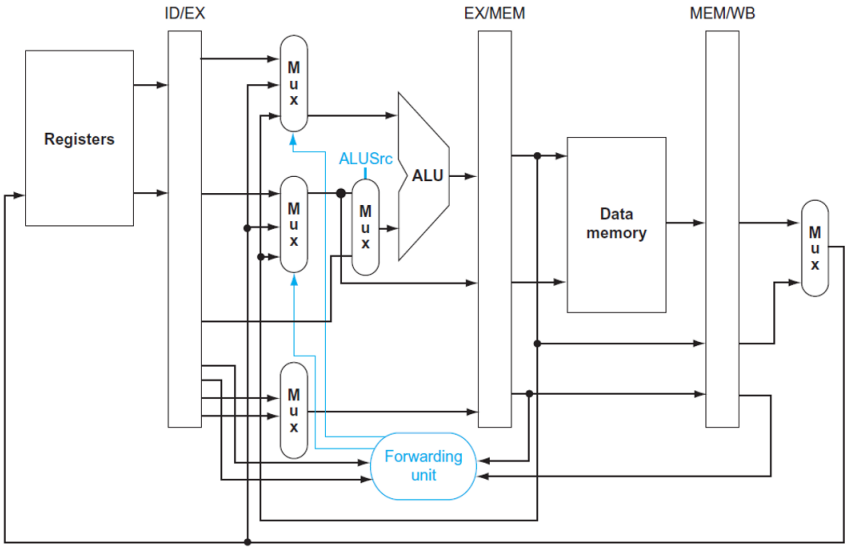
!(EX/MEM.RegWrite && (EX/MEM.Rd ≠ 0) && (EX/MEM.Rd ≠ ID/EX.Rt))) ForwardB = 01

[带前推的流水线]

(1) 忽略 ALU 的立即数的输入:



(2) ALU 的立即数的输入:



[取数-使用型冒险的判定] ID 阶段检测寄存器的使用.

(1) 判定条件: ID/EX.MemRead && ((ID/EX.Rt = IF/ID.Rs) || (ID/EX.Rt = IF/ID.Rt)) .

(2) 发现取数-使用型冒险后, 前推 + 阻塞 1 周期.

[流水线阻塞]

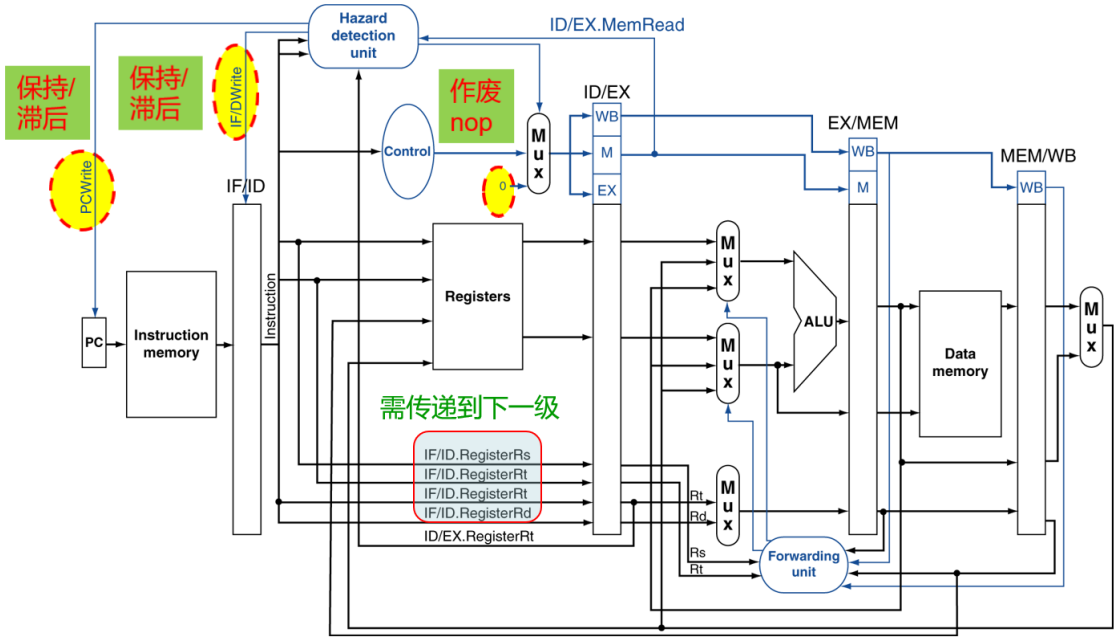
(1) 实现: ID/EX 的控制信号置 0 , EX 、 MEM 、 WB 执行空操作 nop .

(2) 作用:

① 阻止 PC 和 IF/ID 的更新, 指令重新译码, 下一条指令重取.

② 阻塞 1 个周期允许 `lw` 在 MEM 阶段读数据, 前推给下一条指令的 EX 阶段.

[带阻塞的流水线]



4.7 控制冒险

[缩短分支迟延] 增加硬件, 将分支决策提前到 ID 级.

(1) 分支指令在 ID 级译码后, 决定是否将比较数寄存器从前面的指令前推, 来源可为 ALU 输出或内存读出的数据. 若分支发生, 则更新 `PC` .

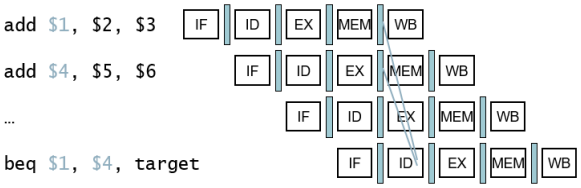
(2) 若分支比较的操作数未准备好, 则分支指令自己阻塞.

① 分支指令的前一条指令是 ALU 指令, 则分支指令阻塞 1 个周期.

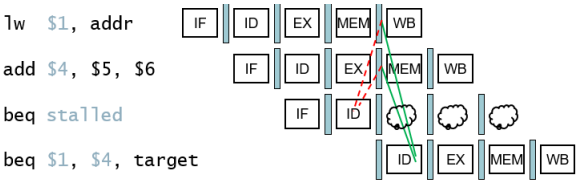
② 分支指令的前一条指令是 `lw` , 则分支指令阻塞 2 个周期.

[分支指令中的数据冒险]

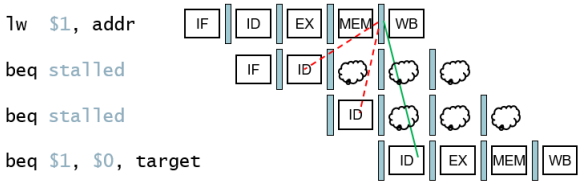
(1) 分支比较的操作数是前 2 、3 条指令的 ALU 输出: 前推.



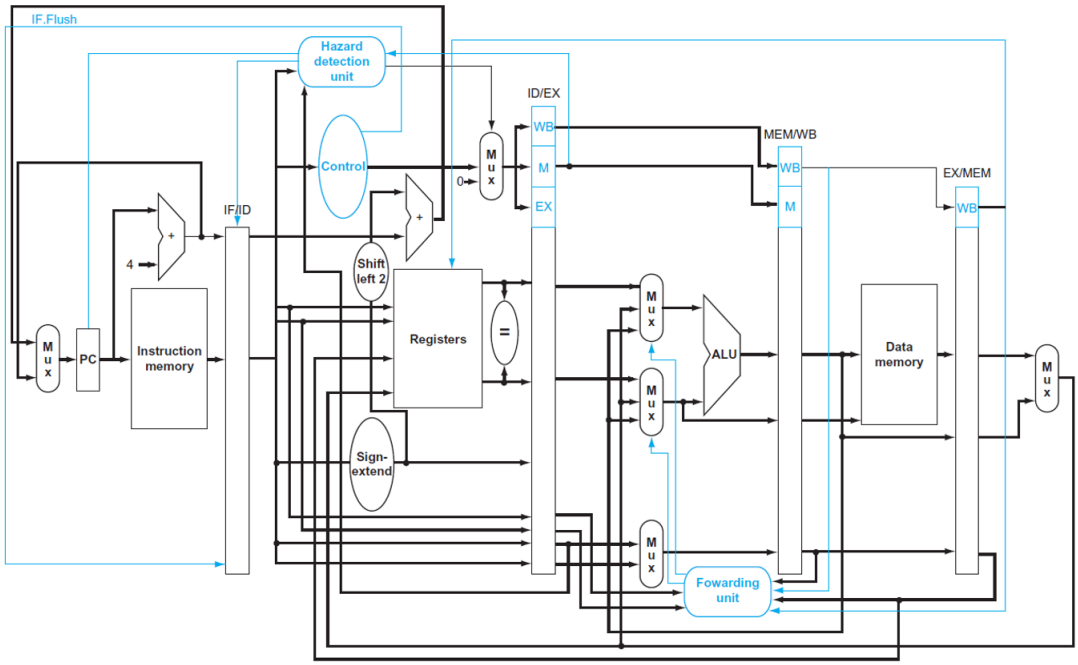
(2) 分支比较的操作数是前 1 条指令的 ALU 输出或前 2 条指令的 `lw` 输出: 阻塞 1 个周期.



(3) 分支比较的操作数是前 1 条指令的 `lw` 输出: 阻塞 2 个周期.



[带冒险检测的流水线]



4.8 异常与中断

[异常与中断]

(1) [**异常**, **Exception**] 源于 CPU 内部事件, 如无效指令、溢出、系统调用指令.

(2) [**中断**, **Interrupt**] 源于外部 I/O 控制器.

(3) 区分内部、外部、异常、中断:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

[MIPS 的异常]

(1) MIPS 的异常由 CP0 (System Control Coprocessor) 处理.

① 保存引起异常(或被中断)的指令的 `PC` 到寄存器 `EPC` (Exception Program Counter).

② 记录引起异常的原因到状态寄存器 `Cause` .

③ 跳转到异常处理入口代码, 地址为 0x8000 0180 .

(2) 指令撤销:

① IF 级已有控制信号 `IF-Flush` .

② ID 级已有(阻塞使用的)全 0 的控制信号, 新增控制信号 `ID-Flush` .

③ EX 级新增控制信号 `EX-Flush` .

④ 假设不在 MEM 级和 WB 级发生异常. 事实上, `lw` 和 `sw` 可在 MEM 级发生异常, 如页表无效.

[带异常处理的流水线]