一.引言

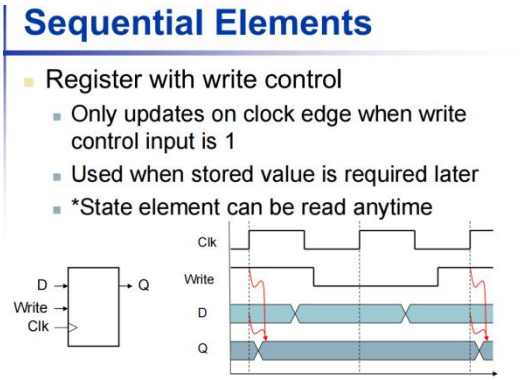
MUX 多选器

对于不同的指令，计算的结果会传到不同的地方，用 Mux 进行选择

控制器

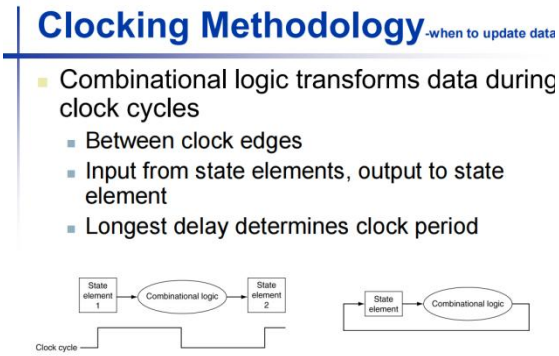
数据的读写（还要受到控制器的控制，不是给个地址就有读的权限），寄存器的写（寄存器的读直接就进行了），分支指令（zero 是用来配合分支指令的，比如 beq 就要与 0 进行比较）都要受到控制器的控制

时序



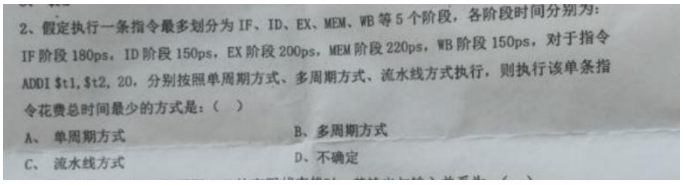
规定什么时间写入，让数据可以保存一段时间（如果没有时序，比如 PC 一直在写，没有一个稳定的值，那么指令执行会混乱），读取数据可以在任何时间段

这里是加了一个 write 的控制信号，也可以不加，那就是那一段时间就是写入



输入经过组合逻辑得到输出，都是在一个时钟周期完成的，是单周期 CPU 的基础，单周期 CPU 只有在这个时钟周期末才能更新（显然这样效率很低）

执行同一条指令所需时间


2、假定执行一条指令最多划分为 IF、ID、EX、MEM、WB 等 5 个阶段，各阶段时间分别为：IF 阶段 180ps，ID 阶段 150ps，EX 阶段 200ps，MEM 阶段 220ps，WB 阶段 150ps，对于指令 ADDI $t1, $t2, 20，分别按照单周期方式、多周期方式、流水线方式执行，则执行该单条指令花费总时间最少的方式是：（ ）

A、 单周期方式　　B、 多周期方式

C、 流水线方式　　D、 不确定

单周期:180+150+200+220+150　多周期：220*4　　流水线：220*（5+1-1）

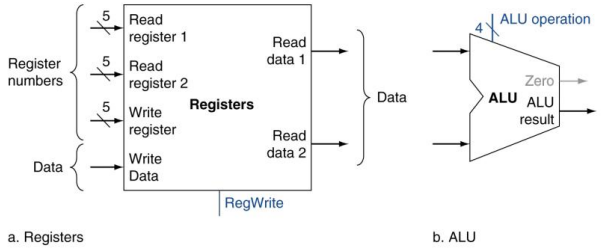
二.建立数据通路

各种指令的认识

R 型指令

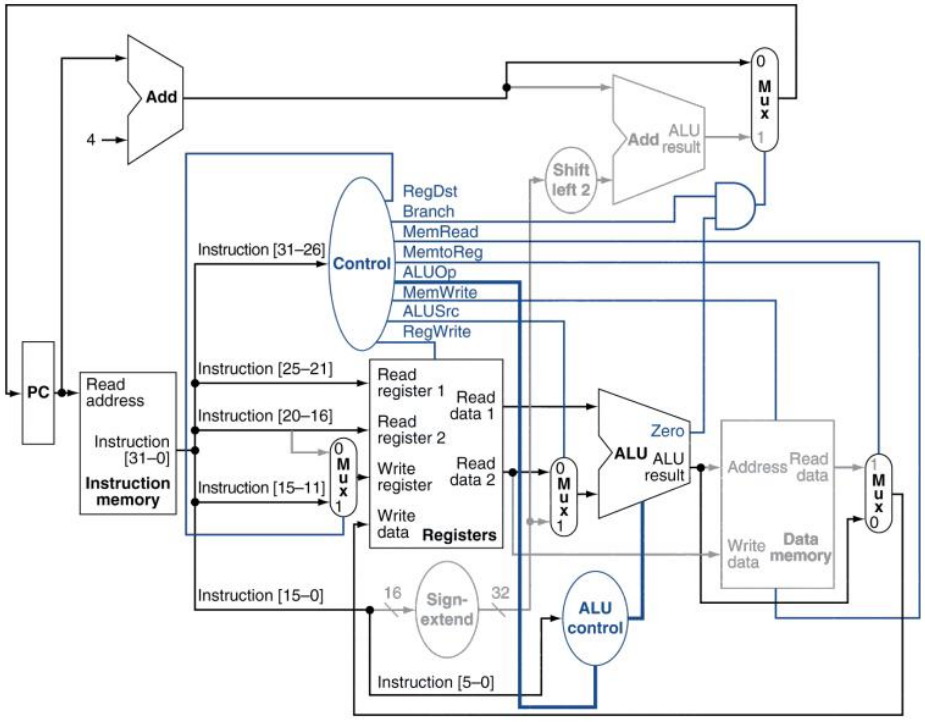
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write result to register



a. Registers b. ALU

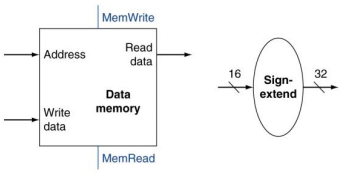
R-Type Instruction



load 和 store 指令

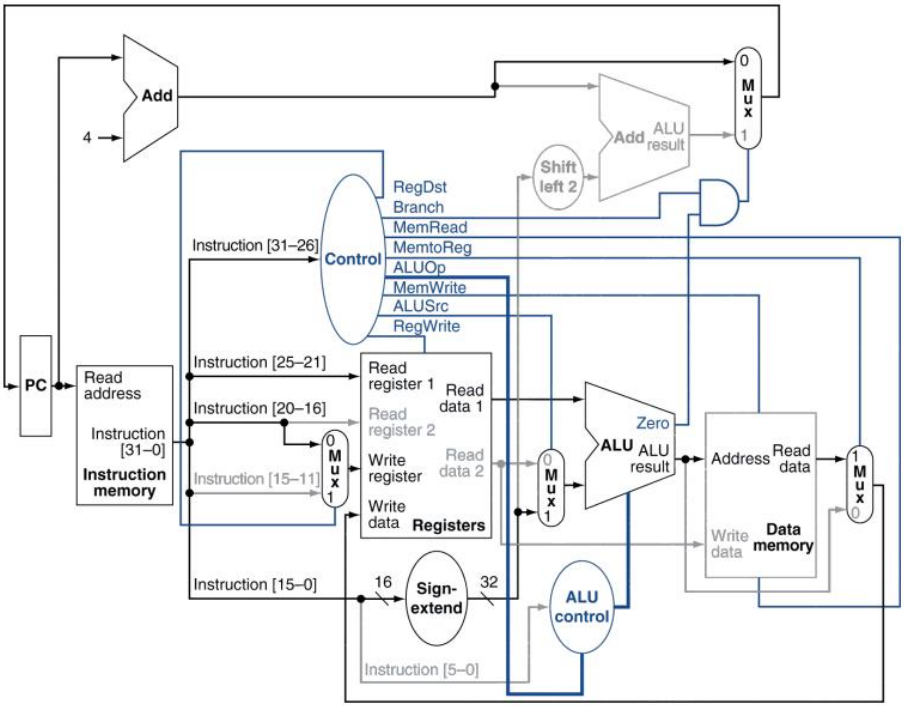
Load/Store Instructions

- Read register operands
- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit b. Sign extension unit

Load Instruction



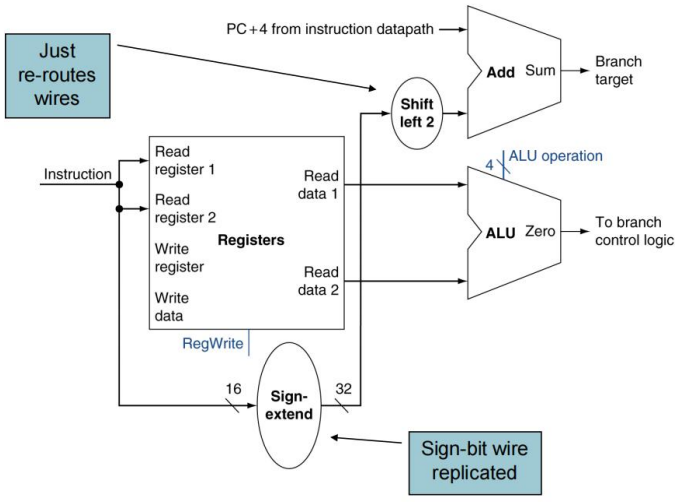
分支指令 beq

Branch Instructions

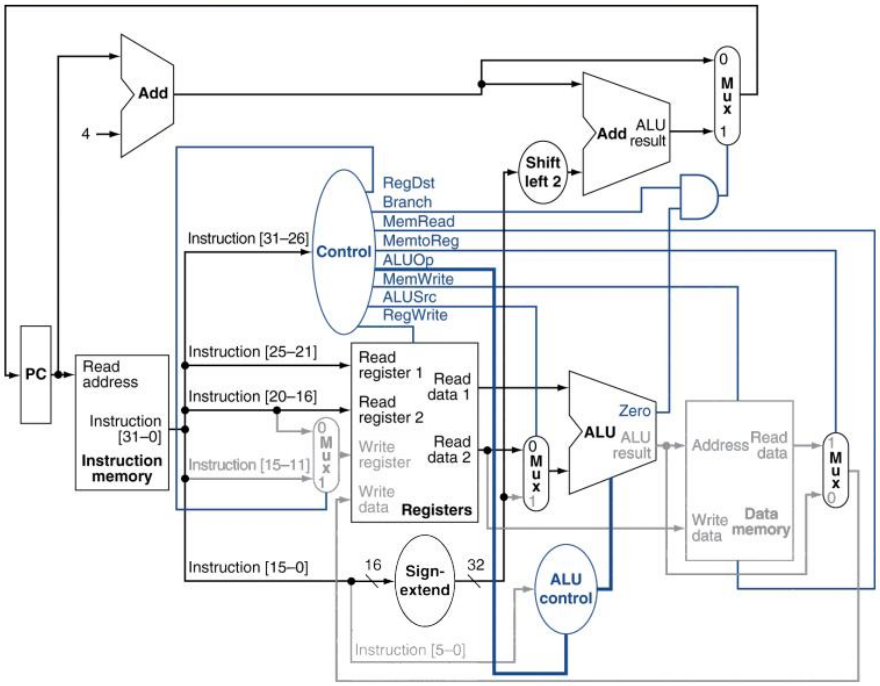
- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
- Calculate target address
  - Sign-extend displacement
  - Shift left 2 places (word displacement)
  - Add to PC + 4
    - *Already calculated by instruction fetch

注意用的是**字地址**，要乘 4

Branch Instructions



Branch-on-Equal Instruction



控制与 ALU

由 opcode（总是 31：26 位）产生的控制信号：RegDst（目的寄存器），Regwrite（寄存器写信号），ALUSrc（ALU 操作数来源），Branch（分支信号），Memread（存储器读信号），Memwrite（存储器写信号），MemtoReg（存储器到寄存器），ALUOp（2 位，决定 ALU 的动作），PCSrc（PC 的来源，但是这个是由 Branch 信号与 0 得到的）

ALU

ALUop 是两位，00 说明是加法，01 说明是减法，10 说明要看 funct 字段的内容来决定 ALU 的功能（ALUop 为 00 或 01 时，ALU 的动作不依赖于 funct 字段）

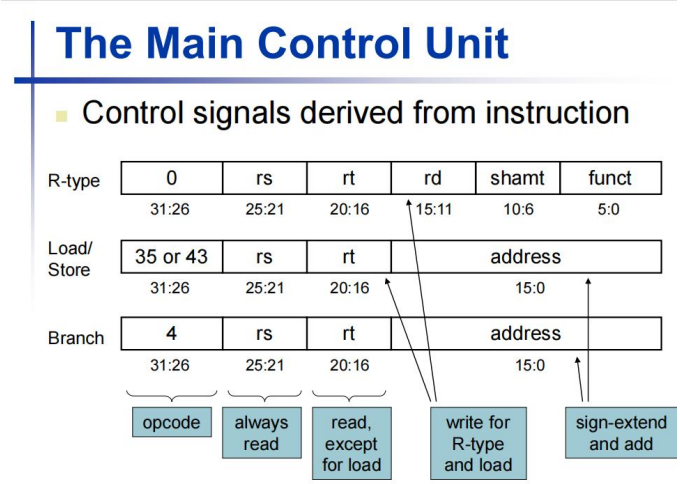
ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

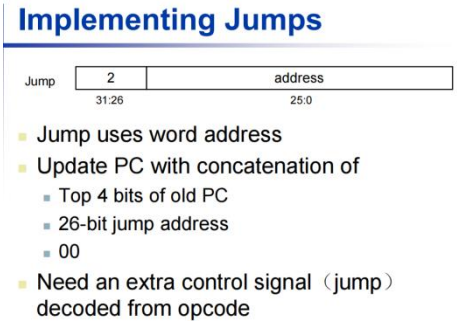
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

各种指令得到的 ALU 控制信号

对于 store,load 指令 如 store $S1,(20)$S2 rs 是$S2,rt 是$S1,因为 rs 只能读，不可以当目的寄存器

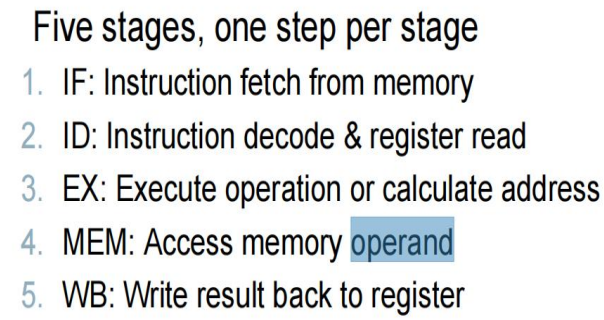


J 型指令



三.旁路与冒险

5 个核心步骤



3 种冒险

Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

1.结构冒险

如果只有一个存储器，load 和 store 指令的 MEM 阶段和 IF 阶段读将会冲突，因此需要单独的指令/数据存储器

Structure Hazards

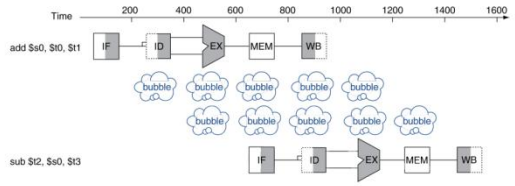
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

2.数据冒险：因无法提供指令执行所需的数据而导致指令不能在预定的时钟周期内执行

寄存器堆或者存储器右半边的阴影表示它们在此步骤中被读取，左半边的阴影表示它们在此步骤中被写入

Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - `add $s0, $t0, $t1`
    `sub $t2, $s0, $t3`

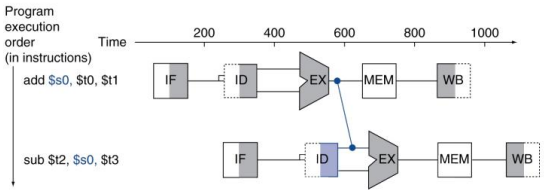


如果要正常运行，要等到 WB 结束才能得到数据

因此可以从内部资源中直接提前得到缺少的运算项的过程称为前推/旁路

Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath

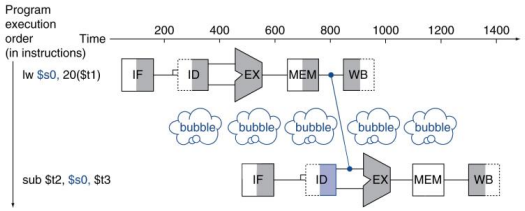


流水线阻塞（气泡）

取数-使用型数据冒险

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



即使有旁路也会阻塞一个周期，因为至少要等到 MEM 后才能得到数据，这点和控制冒险很相似

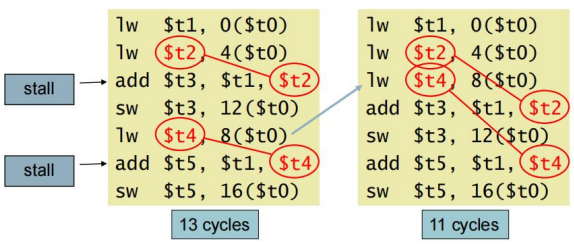
解决方法：

1.采用硬件上检测阻塞（冒险检测单元）

2.重排代码顺序就可以消除阻塞

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A = B + E; C = B + F;



3.控制冒险（也称为分支冒险）：决策依赖于一条指令的结果（分支指令在 MEM 级才能决定是否执行分支，因为在 MEM 级才比较完两个寄存器的值），而其他指令正在执行中，所取的指令不一定是所需要的，因为可能进行了分支

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

解决方法：

1.阻塞：阻塞几个时钟周期，进行寄存器值的比较和目标地址的计算，用硬件做出来

2.预测：分支不执行时，流水线会全速执行；分支执行时流水线才阻塞（算出目标地址）

分支预测

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

静态预测：基于传统的分支表现，默认分支是否发生

动态预测：动态地记录每个分支的最近发生历史，认为分支执行会持续这个趋势

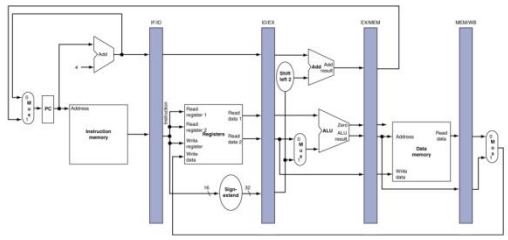
预测正确流水线正常执行；预测错误时流水线控制必须确保错误预测的分支后面的指令执行不会生效，并且必须在正确的分支地址处重新开始启动流水线

四.流水线数据通路及其控制

流水线寄存器

Pipeline registers

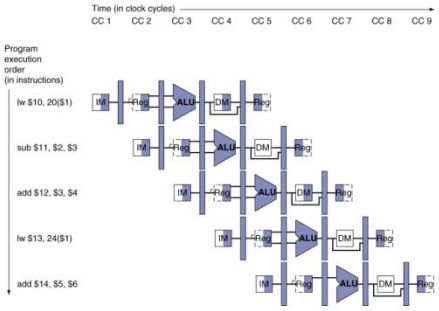
- Need registers between stages
  - To hold information produced in previous cycle



多周期流水线图表

Multi-Cycle Pipeline Diagram

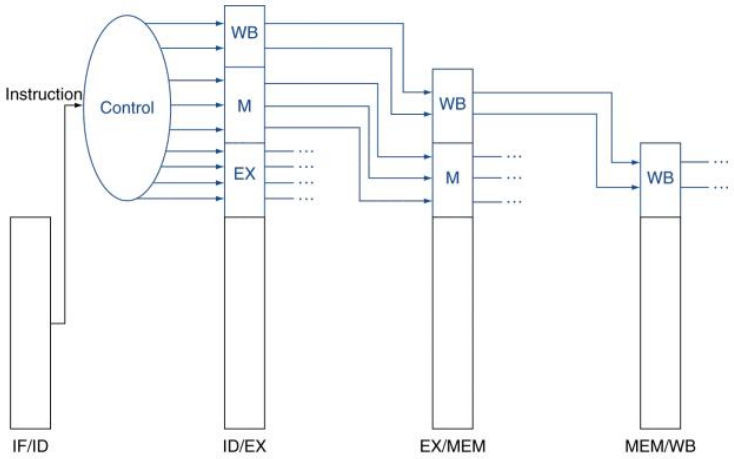
- Form showing resource usage



流水线控制

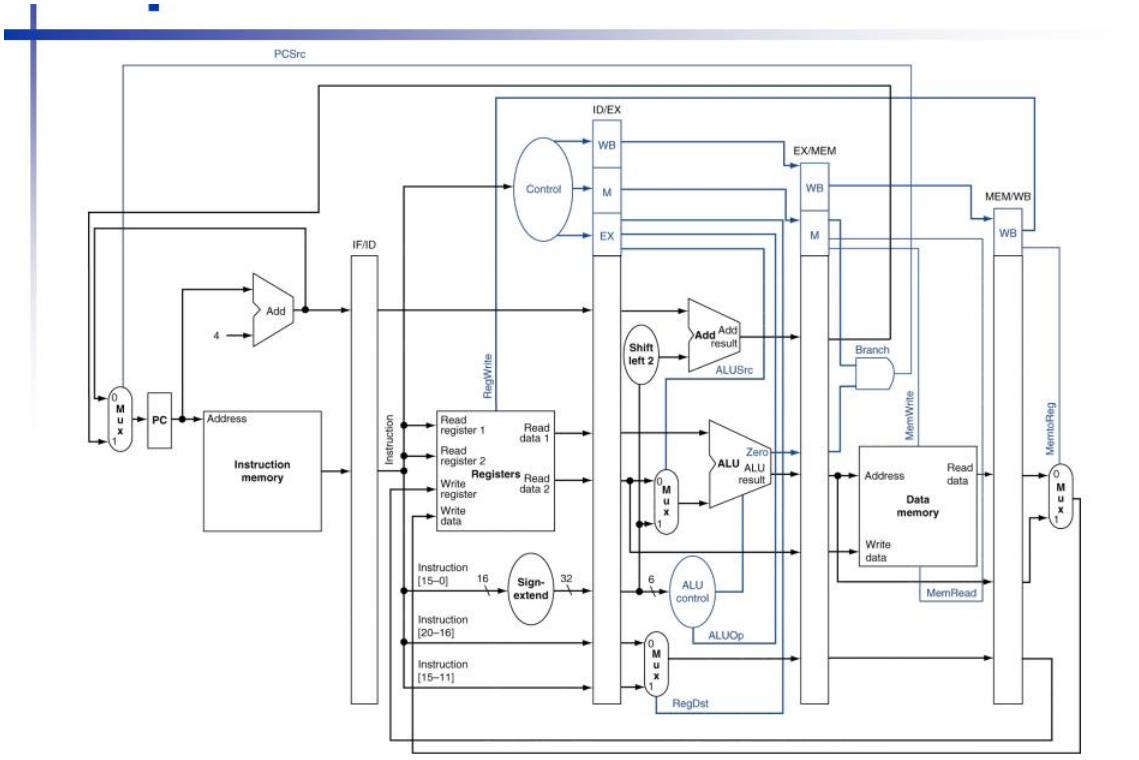
Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



流水线的控制信号从一开始传递到最后

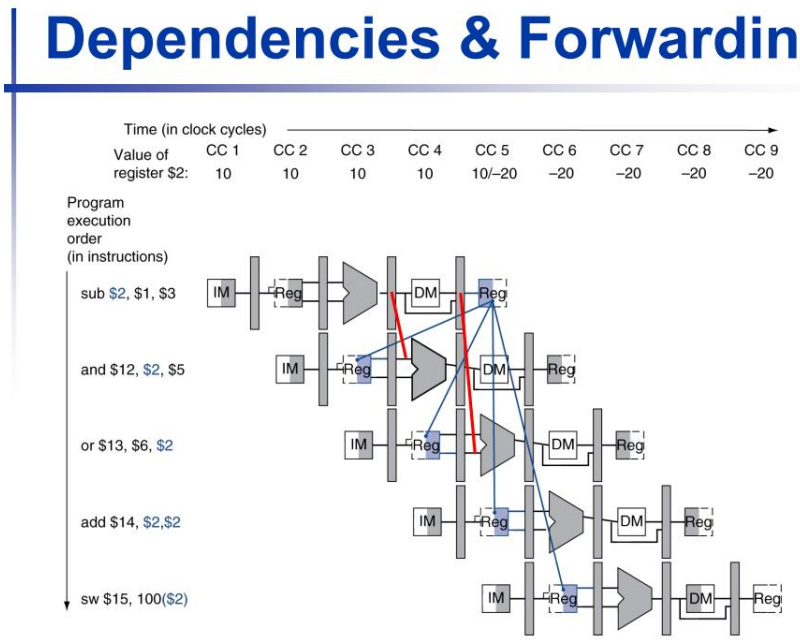
加上控制信号后的数据通路



五.数据冒险：

1.旁路与阻塞 针对 R 型指令

注意$S2，后面的指令要用到 sub 指令的结果$2 的值



数据冒险的条件：

Data hazards when

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

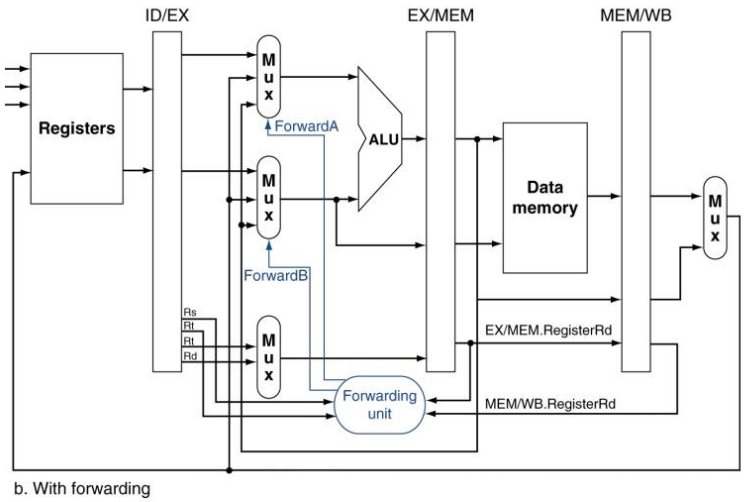
旁路条件：注意要有寄存器的写信号，且 Rd 不为 0

Forwarding Conditions

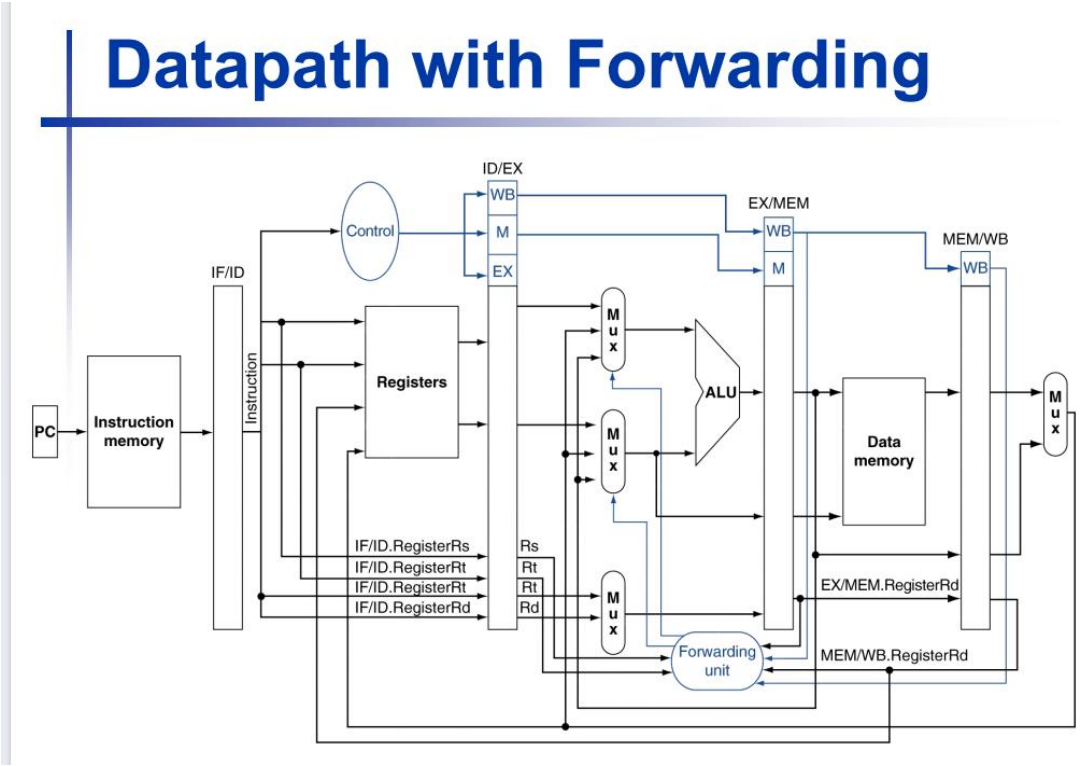
- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

旁路硬件结构

Forwarding Paths



b. With forwarding



Fowarding unit 输入有 ID/EX 的 rs,rt、EX/MEM 的 rd、MEM/WB 的 rd、RegisterWrite 信号，输出有两种旁路信号

双重数据冒险，第三个 add 指令既有 EX 冒险，又有 MEM 冒险（了解即可）

Double Data Hazard

- Consider the sequence:

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

修正措施，当然是选择更近的冒险，也就是 EX 冒险，修正如下

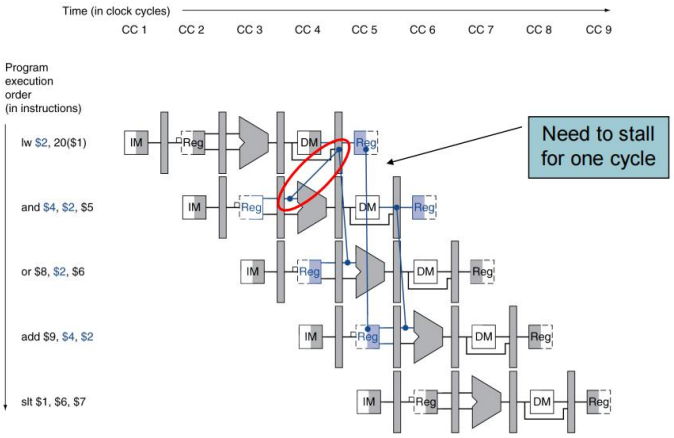
Revised Forwarding Condition

- MEM hazard
  - 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))
    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))
    ForwardB = 01

2.冒险与阻塞 针对 load 型指令（除了有旁路单元外，还要有冒险检测单元，其工作在 ID 级，从而可以在 load 指令与紧随其后需要它的结果的指令间插入阻塞）

如果没有旁路，还需要多阻塞一个时钟周期，因此阻塞几个周期还得看是否有旁路，想不清楚就画画图思考

Load-Use Data Hazard



冒险检测条件：注意 ID/EX 有存储器读信号

Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble

如何阻塞流水线

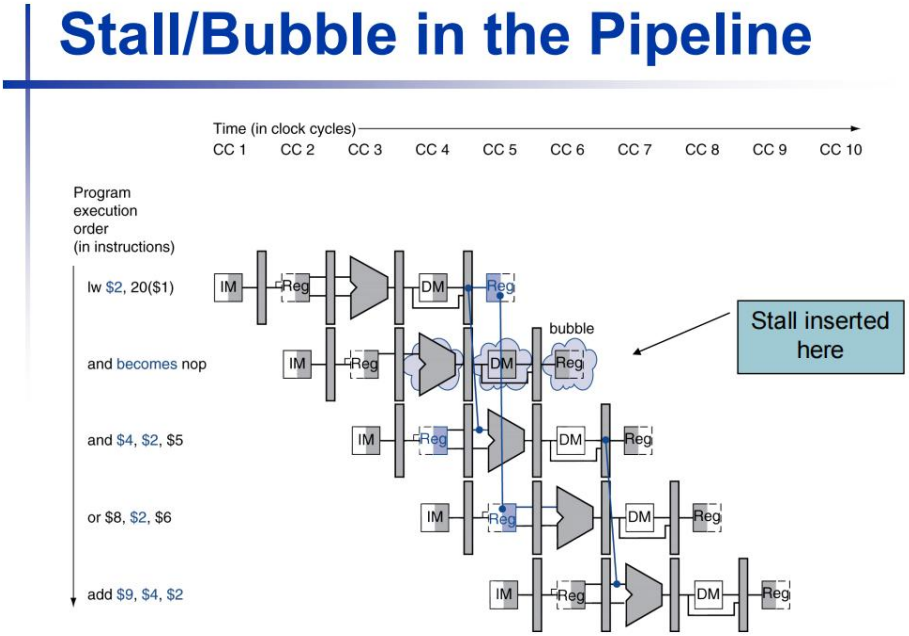
How to Stall the Pipeline

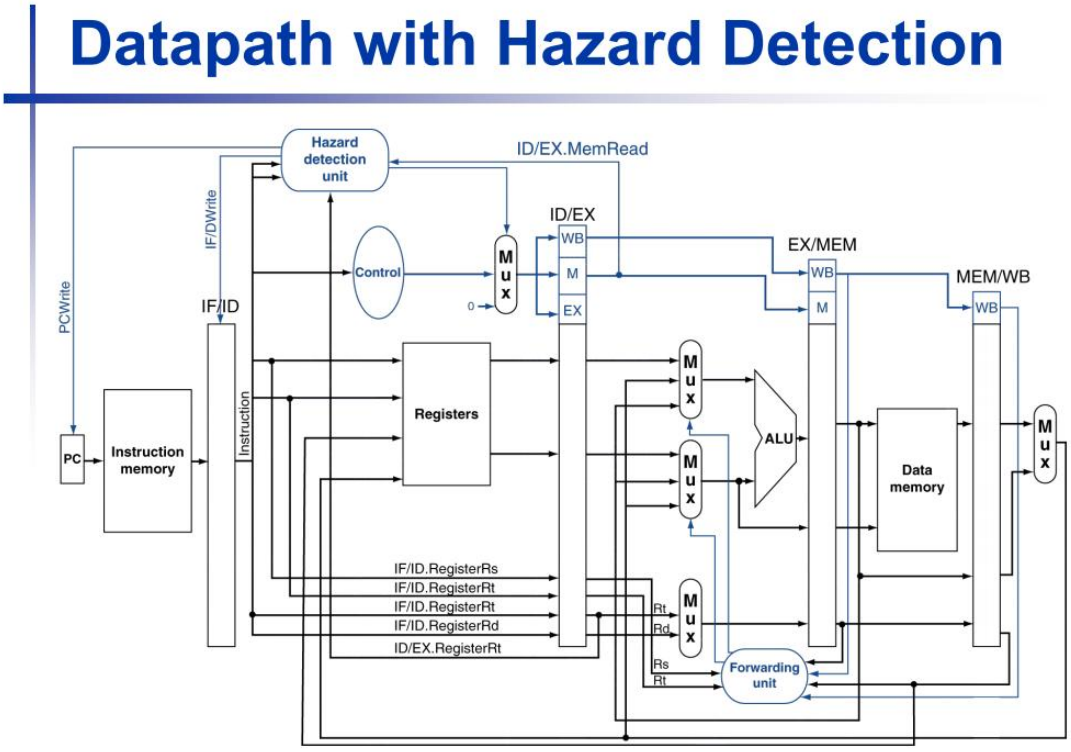
- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

让在 ID/EX 中的控制信号清 0，接下来 EX,MEM,WB 就不会执行任何操作，也就是变成 nop 阻止 PC 和 IF/ID 寄存器的更新（也就是让 PCWrite 为 0，IF/IDWrite 为 0），再次使用得到的指令并且译码，这一个周期的阻塞让上一条指令执行了 MEM 的读取数据的操作，那么我们就可以进行 EX 旁路

书上阐述：如果处于 ID 级的指令被阻塞，那么处于 IF 级的指令也必须被阻塞，否则，已经取到的指令就会丢失（因为 IF/ID 寄存器写入东西就会让之前的指令被覆盖掉）。防止这两条指令继续执行的方法是保持 PC 寄存器和 IF/ID 流水线寄存器不变。如果这些寄存器内容保持不变，在 IF 级的指令将继续使用相同的 PC 取指令，而在 ID 级将继续使用 IF/ID 流水线寄存器中的相同的指令字段读寄存器堆。

如图所示进行阻塞





Hazard detection unit 中输入有 ID/EX.MemRead、IF/ID 的 rs,rt、ID/EX 的 rt，输出有 PCWrite、IF/IDWrite 和 WB,M,EX 的控制信号

3.控制冒险（分支冒险）

假设分支不发生

采用阻塞直到分支判断完毕来处理控制冒险的速度实在太慢。一种比较普遍的提高速度的方法是假设分支不发生，并继续执行顺序的指令流。如果分支发生，就丢弃已经读取并译码的指令，并按分支目标继续执行。如果分支不发生的可能性是 50%，同时丢弃指令的代价很小，那么这种优化方法可以将控制冒险的代价减半。

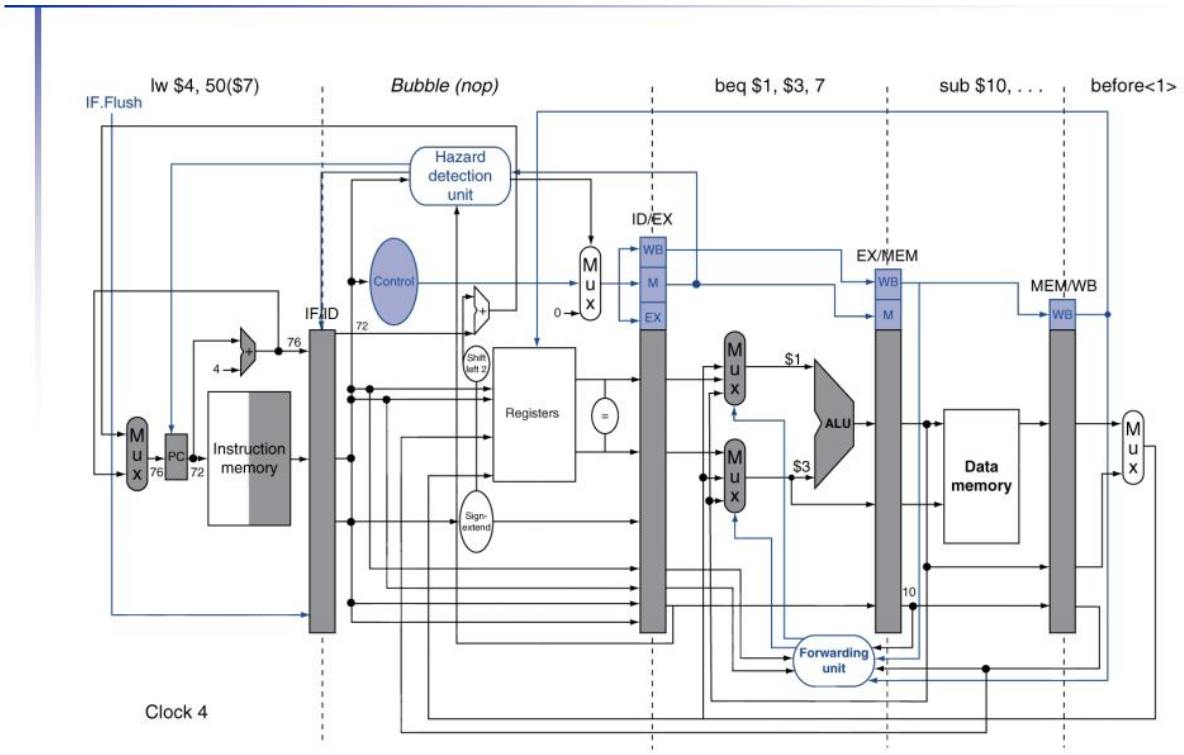
为了丢弃指令，只需要将最初的控制信号置为 0 即可，这一点与阻塞解决 load 指令的数据冒险类似。其不同之处在于当分支到达 MEM 级时必须分别改变在 IF、ID 和 EX 级的三条指令的控制信号，而对于 load 指令的阻塞只需要将 ID 级的控制信号置为 0，并将其从流水线中退出即可。分支冒险中的丢弃指令意味着必须能够将流水线的 IF、ID 和 EX 级的指令都清除（flush)。

缩短分支的延迟：计算分支目标地址、判断分支条件

地址的计算是比较简单的。我们在 IF/ID 流水线寄存器中已经有了 PC 的值和立即数字段，所以只需要**将分支地址计算从 EX 级移到 ID 级就可以了**。当然，尽管分支目标地址对所有指令都会计算，但仅在需要时才会使用。

判断分支条件比较复杂。为了判断分支的执行条件，需要比较从 ID 级取到的两个寄存器的值是否相等。判断相等的方法可以是先将对应的位进行异或操作，然后将结果按位进行或操作。为了把分支条件判断提前到 ID 级，还需要额外的旁路和冒险检测硬件，因为分支条件的判断可能依赖于还在流水线中的结果。例如，为了实现相等则分支（或不等则分支)，我们需要旁路结果至 ID 级进行相等检测。这里有两个比较复杂的因素:1.在 ID 级指令译码后，决定是否需要将所需数据旁路到相等检测单元进行相等检测。如果是分支指令，就可以把 PC 替换为分支目标地址。旁路分支指令的操作数以前是由 EX 旁路单元来完成的，但 ID 级相等检测单元的引人需要一个新的旁路单元。必须注意的是，需要旁路的分支指令源操作数可能来自 EX/MEM 或 MEM/WB 流水线寄存器。2.因为 ID 级进行分支比较所需的数据可能在后面才能产生，因此有可能会发生数据冒险，这样就需要阻塞流水线。例如，如果分支指令前刚好是一条 ALU 指令，而这条 ALU 指令的结果恰是分支指令比较所需要的，那么必然产生阻塞，因为 ALU 指令的 EX 级将在分支指令的 ID 级后发生。再举一个例子，如果分支指令前刚好是一条 load 指令，而 load 指令的结果恰是分支指令判断所需要的，则必须产生两个阻塞，因为装载指令的结果将在 load 指令的 MEM 级结束时产生，但在分支指令的 ID 级开始时就会用到。

在哪一级进行分支预测，那么在这一级要被阻塞

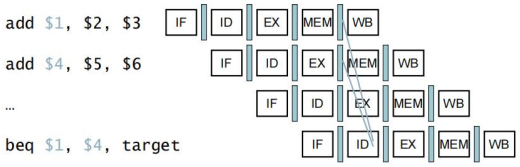


复杂因素的图示：

注意一件事，我们进行分支条件的判断在 ID 级，用相等检测单元，不是用 ALU 做减法，所以旁路是 ID 级和其它的流水寄存器，而不是之前的 ALU 的操作数接根线

Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

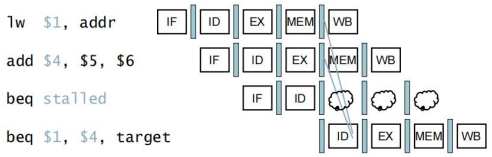


- Can resolve using forwarding

会阻塞一个周期的情况

Data Hazards for Branches

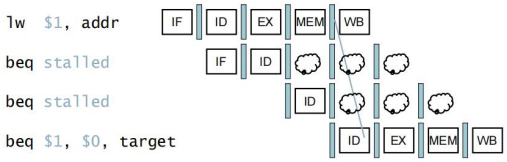
- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle



会阻塞两个周期的情况

Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles



动态分支预测

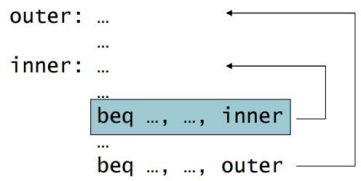
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (1/2-bit predictor: Flag indicates taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from next or target
    - If wrong, flush pipeline and flip prediction

分支预测缓存（又名分支历史记录表）按最近的分支指令地址索引

1bit 预测

1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!



  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

默认都是进行分支，但是在内部循环中，最后一次的预测显然总会是错误的，让后 predictor 又被置为 0，导致下一次内部循环的第一次又是错误的

2bit 预测

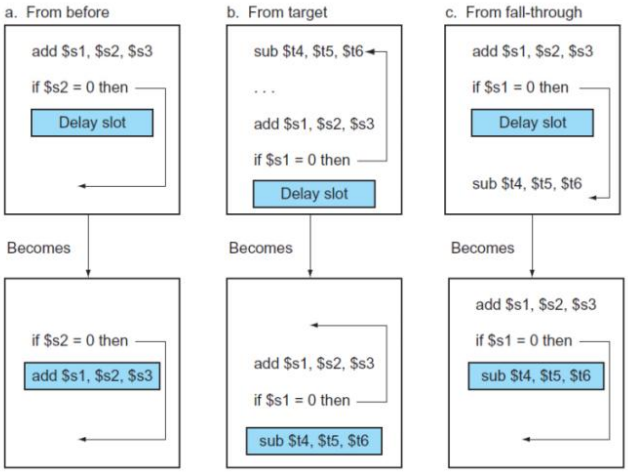
2-Bit Predictor

- Only change prediction on two successive mispredictions
- Status:(11-10-01-00)
- 11(predict taken. if not,change to 10)
- 10(predict taken. if taken, change to 11,
- if not, change to 01)
- 01(predict not taken. if not, change to 00,
- if taken, change to 10)
- 00(predict not taken. if taken, change to 01)

可以用加减法来理解，11 就是一定会进行分支，00 是一定不进行分支，每次根据是否出错来进行加减修改

分支延迟槽

Branch Delay Slot



分支延迟时间槽的调度。每一对方框中的上面一个表示调度前的代码，下面一个表示调度后的代码。在方案 a 中，延迟时间片通过插入分支之前的一条与分支无关的指令实现，这是一种最佳的选择。当方案 a 无法实现时，就使用方案 b 和方案 c。在方案 b 和方案 c 的代码序列中，分支条件中使用了$1，因而不能将 add 指令（它的目的寄存器是$1）移人分支延迟时间槽。方案 b 中的分支延迟时间槽是按照分支目标地址调度的。由于目标指令可以通过其他路径访问到，通常需要将它们进行复制。当分支发生的可能性比较大时，一般选择方案 b，如循环分支。最后，也可能采用方案 c 预测分支不发生的下一条指令进行调度。为了使方案 b 和方案 c 中的优化合法，sub 指令必须在分支预测错误时也能“正常”执行。“正常”意味着虽然有些工作是多余的，但程序依然能够正确执行。例如，当分支预测错误且$t4 是未被使用的临时寄存器时，就是这种情况

无论预测是否正确，分支延迟槽中的指令都要执行，如果预测错误，系统会进行相应的处理修正错误

8.异常和中断

事件类型	来源	对应的 MIPS 术语
I/O 设备请求	外部	中断
用户程序进行操作系统调用	内部	异常
算术溢出	内部	异常
使用未定义的指令	内部	异常
硬件故障	内部或外部	异常或中断

EPC（异常程序计数器）：保存出错指令的地址，把控制权交给操作系统的特定地址
Cause 寄存器（状态寄存器）：其中有一个字段用来记录异常产生的原因

向量中断

另一种方法是使用**向量中断**（vectored interrupt）。在向量中断中，控制权被转移到由异常原因决定的地址处。例如，为处理前面的两种异常，可定义如下的两个异常向量地址：

向量中断：由异常原因决定中断控制转移地址的中断。

异常类型	异常向量地址（十六进制）
未定义指令	$8000\ 0000_{16}$
算术溢出	$8000\ 0180_{16}$

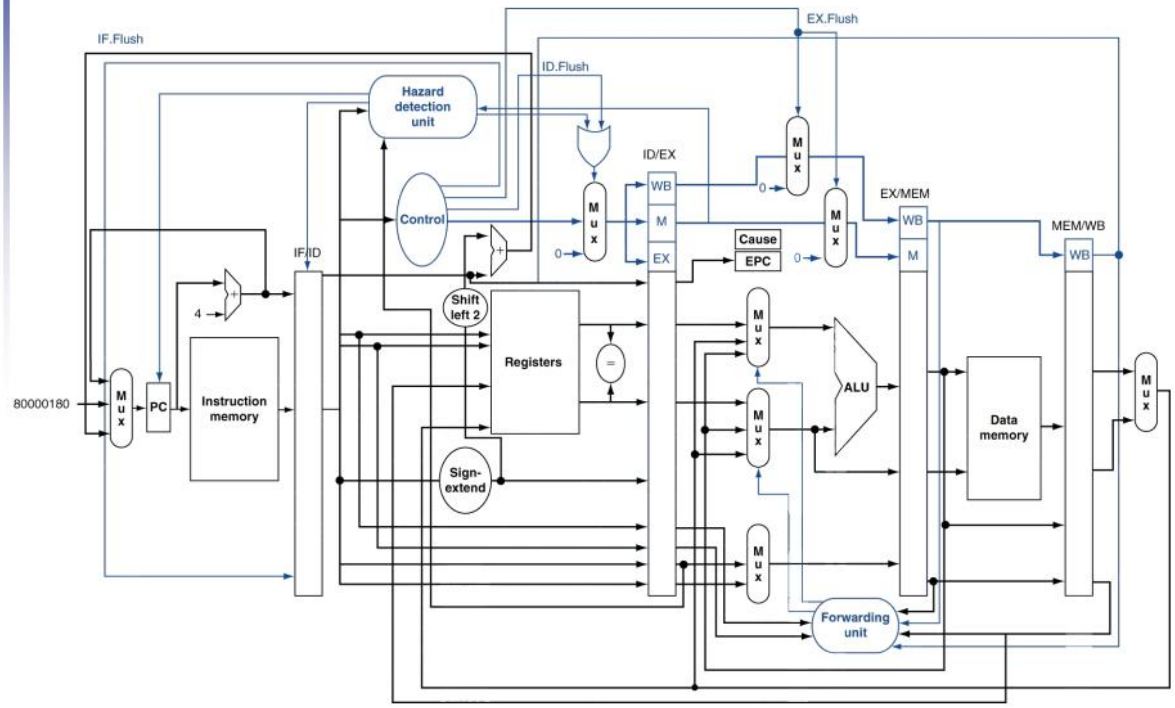
课本内容：

在处理分支预测错误时，我们已经知道如何通过将 IF 级的指令转换成 nop 指令来清除指令。为了清除 ID 级的指令，我们使用 ID 级已有的多选器，将控制信号清零以产生阻塞。一个称为 ID. Flush 的新控制信号与冒险检测单元的阻塞信号相或，可以在 ID 级进行清除。为了清除 EX 级的指令，我们使用一个称为 EX.Flush 的新信号，用它控制新的多选器将控制信号清零。为了从地址 8000 0180,(MIPS 异常地址）开始取指令，只要简单地加入一个额外的输人到 PC 的多选器，由它将 8000 0180,,传递到 PC。图 4-66 具体描述了这种变化。

这个例子指出了异常存在的一个问题，即如果不在指令执行期间中止指令的执行，程序员将无法看到导致溢出的寄存器$1 中的原始值，因为它将作为指令 add 的目标寄存器被冲掉。这一问题可以通过下面的方法解决:**异常溢出在 EX 级检测出来，可用 EX. Flush 信号避免 EX 级的指令在 WB 级写回结果。**许多异常需要我们能够最终正常执行引起异常的指令。做到这一点最简单的方法是先清除这条指令，然后在异常处理完后再重新执行这条指令。异常处理的最后一步是将导致异常的指令的地址保存到 EPC 中。实际上，我们保存的是原始地址+4(EPC=原始地址+4），因此异常处理例程必须先从保存的地址中减去 4。硬件/软件接口、硬件与操作系统必须协同工作才能按照我们期望的方式处理异常。硬件一般暂停指令流中导致异常的指令，同时执行完该指令前的所有指令，清除该指令后的所有指令，并且设置一个寄存器描述异常发生的原因，保存导致异常发生的指令的地址，然后跳转到预先确定的地址开始执行。操作系统则查看异常发生的原因并采取相应的操作。对于一个未定义指令异常、硬件错误异常或算术溢出异常，操作系统通常终止执行的程序并返回原因描述。

结构如图所示

Pipeline with Exceptions



关于流水线的一些理解：

武漢大学 WUHAN UNIVERSITY NO. Date:

书P248 4.13.6

```
add r5,r2,r1   ← 2 nop
lw  r3,4(r5)
lw  r2,0(r2)
or  r3,r5,r3
①  ②  ③  ④  ⑤
IF  ID  EX  MEM  WB
```

	PCwrite	IF/ID write	ID/EX zero
①	1	1	0
②	1	1	0
③	1	1	0
④	0	0	1
⑤	0	0	1

```
    IF  ID  X   X   EX
      • IF  X   X   ID
            X   X   IF
```

阻塞的是周期 对于周期，竖着看；对于每个指令，横着看

阻塞的目的是为了等到合适的时机取到想要的数据，如题是要等到WB后得到r5的正确值