编号

**南京航空航天大学**

**机组课程设计报告**

|  |  |
| --- | --- |
| 题 目 | 56条指令流水线的实现 |

|  |  |
| --- | --- |
| 学生姓名 | 沈王雄 |
| 学 号 | 161510326 |
| 学 院 | 计算机科学与技术学院 |
| 专 业 | 计算计科学与技术专业 |
| 班 级 | 1615104 |
| 指导教师 |  |

二〇一七年六月

**目录**

1. **实验背景介绍：............................................**
2. **实验要求(功能设计要求)....................................**
3. **模块化和层次化设计说明....................................**
4. **流水线的冒险及处理：......................................**
5. **代码测试与结果分析.........................................**
6. **11**
7. **心得体会 ...................................................**
8. **参考文献.....................................................**
9. **实验背景：**
   1. MIPS指令的优异性：MIPS是世界上很流行的一种RISC处理器。MIPS的意思是“无内部互锁流水级的微处理器”(Microprocessor without interlocked piped stages)，其机制是尽量利用软件办法避免流水线中的数据相关问题。它最早是在80年代初期由斯坦福(Stanford)大学Hennessy教授领导的研究小组研制出来的。MIPS公司的R系列就是在此基础上开发的RISC工业产品的微处理器。这些系列产品为很多计算机公司采用构成各种工作站和计算机系统。精简指令集这种设计思路对指令数目和寻址方式都做了精简，使其实现更容易，指令并行执行程度更好，编译器的效率更高。它能够以更快的速度执行操作。1.统一指令编码（例如，所有指令中的op-code永远位于同样的位元位置、等长指令），可快速解译；2.泛用的暂存器，所有暂存器可用于所有内容，以及编译器设计的单纯化（不过暂存器中区分了整数和浮点数）；3.单纯的寻址模式（复杂寻址模式以简单计算指令序列取代）4.硬件中支援少数资料型别（例如，一些CISC电脑中存有处理字节字串的指令。这在RISC电脑中不太可能出现）。而CISC(复杂指令集计算机)和RISC(精简指令集计算机)是当前CPU的两种架构。它们的区别在于不同的CPU设计理念和方法。早期的CPU全部是CISC架构，它的设计目的是要用最少的机器语言指令来完成所需的计算任务。RISC和CISC是设计制造微处理器的两种典型技术，虽然它们都是试图在体系结构、操作运行、软件硬件、编译时间和运行时间等诸多因素中做出某种平衡，以求达到高效的目的，但采用的方法不同，因此，在很多方面差异很大，它们主要有：

（1）指令系统：RISC设计者把主要精力放在那些经常使用的指令上，尽量使它们具有简单高效的特色。对不常用的功能，常通过组合指令来完成。因此，在RISC机器上实现特殊功能时，效率可能较低。但可以利用流水技术和超标量技术加以改进和弥补。而CISC计算机的指令系统比较丰富，有专用指令来完成特定的功能。因此，处理特殊任务效率较高。

（2）存储器操作：RISC对存储器操作有限制，使控制简单化；而CISC机器的存储器操作指令多，操作直接。

（3）程序：RISC汇编语言程序一般需要较大的内存空间，实现特殊功能时程序复杂，不易设计；而CISC汇编语言程序编程相对简单，科学计算及复杂操作的程序社设计相对容易，效率较高。

（4）中断：RISC机器在一条指令执行的适当地方可以响应中断；而CISC机器是在一条指令执行结束后响应中断。

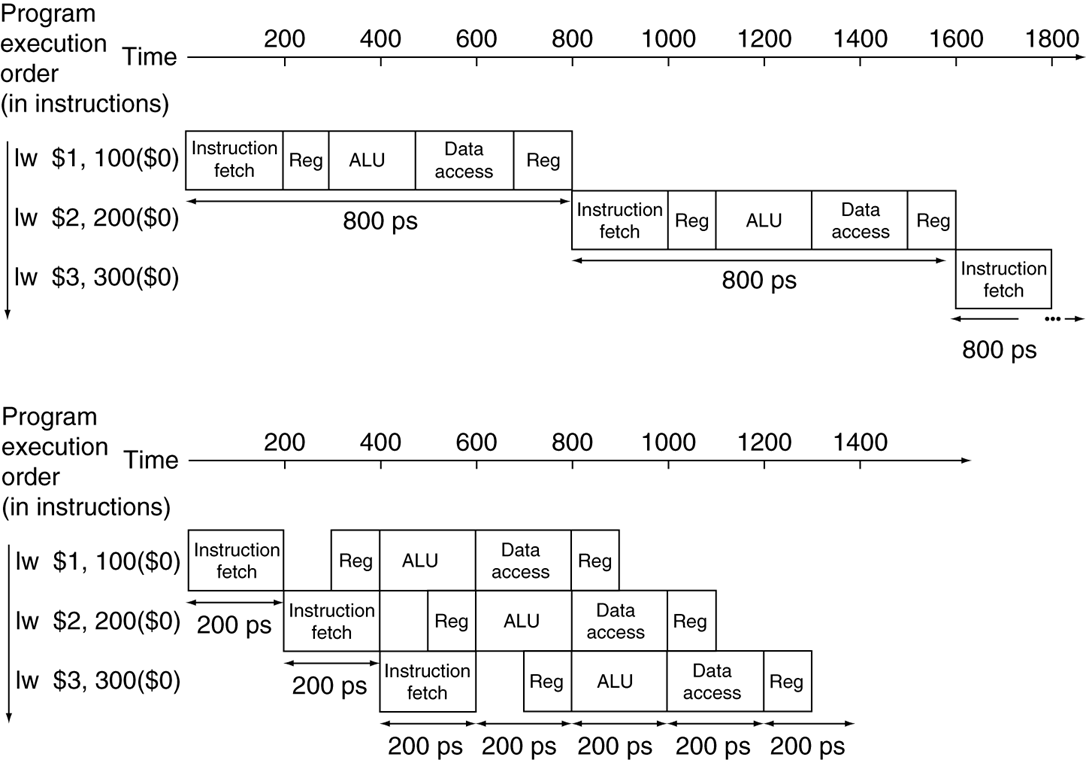
（5）CPU：RISCCPU包含有较少的单元电路，因而面积小、功耗低；而CISCCPU包含有丰富的电路单元，因而功能强、面积大、功耗大。

（6）设计周期：RISC微处理器结构简单，布局紧凑，设计周期短，且易于采用最新技术；CISC微处理器结构复杂，设计周期长。

（7）用户使用：RISC微处理器结构简单，指令规整，性能容易把握，易学易用；CISC微处理器结构复杂，功能强大，实现特殊功能容易。

（8）应用范围：由于RISC指令系统的确定与特定的应用领域有关，故RISC机器更适合于专用机；而CISC机器则更适合于通用机。

2、单周期实验的重要性：基于MIPS的单周期CPU的设计与实现，可以让我们真正的走进计算机CPU的内部世界观察，每一条指令的执行过程，不仅仅是对课堂知识的一个简单的回顾理解，也是从最为根本的门电路去理解计算机。也为之后的多周期，及流水线的实现就是对单周期实现的扩展使得效率的成倍提升。



3、国家安全战略和民生工程等方面，面临挑战：

电站可被远程控制；公务车有处理器后门，可被监听或控制；…….

举例：2012年伊朗核离心装置遭到“震网”病毒攻击，20%的离心机报废，监控录像被篡改，而遭攻击的电脑是物理隔离的

基本现状：国家急需计算机系统方面的人才！！！

4、神威·太湖之光超级计算机

（a）、安装了40960个中国自主研发的“申威26010”众核处理器，采用64位自主申威指令系统，峰值性能为12.5亿亿次/秒，持续性能为9.3亿亿次/秒。2016年6月20日在法兰克福世界超算大会上，国际TOP500组织发布的榜单显示，“神威·太湖之光”超级计算机系统速度比第二名“天河二号”快出近两倍，其效率也提高3倍. “天河二号”比第三名美国“泰坦”快近一倍

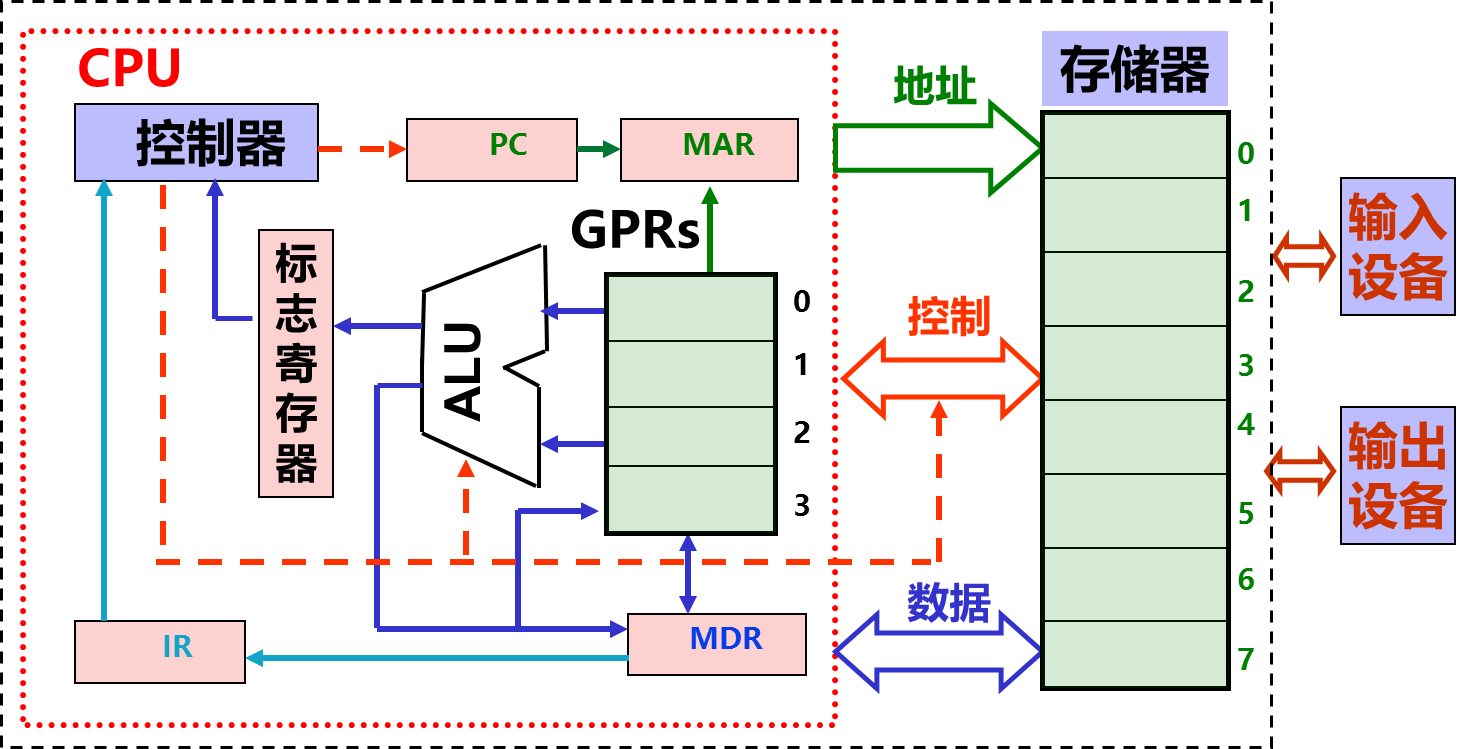
(b)、问题：实用性和效能难令人满意，美国限售Intel高性能处理器芯片

5、龙芯：

(a)、从“北斗”卫星、防火墙等国家安全战略层面，到电子水表、机顶盒等民生项目都有应用，取得长足进步

(b)、软硬件开发人员上万，2013年1.8万片，2014年达35万片

(c)、“长板”与国际先进水平相差2-3年，“短板”相差10年以上。若有10万人基于龙芯开发基础软件，就能很好建立生态系统

6、冯·诺依曼体系结构下图所示：

其存储器结构是将程序指令存储器与数据存储器合并在一起，也就是将指令和数据共同放在一个存储器中，指令存储器和数据存储器共用一条数据总线，这样的结构在早期的微处理器以及单片机中是非常常见。其主要特点有以下几点

（1） 指令与数据共用一个存储器；

（2） 指令与数据的存储地址指向同一存储器的不同物理位置；

（3） 取指令与取数据共用一条总线；

（4） 程序指令和数据的位宽相等。

根据以上总结的特点可以看出冯·诺依曼体系结构其优点在于由于共享数据总线，所以减小了总线的开销，并且能够把 RAM 映射到程序空间。但是其缺点也是显而易见的，处理器在执行命令式，首先必须从存储器中取出指令进行解码，然后再从存储器中取出操作数执行运算，如此操作即使是单指令也要耗费几个甚至几十个时钟周期来执行，所以在高性能和高实时性场合，会在数据传输通道上会出现时序瓶颈，制约微处理器性能。与冯·诺依曼体系结构不同，哈佛结构是一种将程序指令存储和数据存储分开的存储器结构 、

（1） 指令和数据分别存储在指令存储器和数据存储器中；

（2） 指令与数据的地址都是独立编码的；

（3） 指令与数据都是独立访问的；

（4） 微处理器中包含四条独立的总线；

（5） 数据和指令宽度可以不等。

从上面的几点总结可以看出，由于采用地址总线与数据总线相分离的体系结构，所以可以使微处理器能够在一个时钟内同时读取指令和数据，从而缓解了冯·诺依曼体系结构中数据传输通道上会出现的瓶颈效应。从以上对两种体系结构的特点介绍可以发现哈佛结构相对于冯·诺依曼体系结构来说，它具有以下两个特点：首先是它使用两个独立的存储器分别存储指令与数据，其次是它具有两套独立的总线系统，一共四条总线，它们分别担当指令存储器以及数据存储器与微处理之间进行通信的作用，这两套总线系统互相独立。也正是有了这些特点，使得哈佛结构的微处理器拥有较大的存储器带宽，这样可以使得执行指令读取以及数据存储操作时更加方便也更有效率。

因此，为了实现数据的并行处理，采用流水线的方式，提高程序，数据的突出率，使得硬件资源得到充分的利用，进而提高处理器的效率，由于流水线是基于单周期的cpu的设计，因此对于现代流水线的设计依然沿袭单周期的模块设计，但值得注意的是对于流水线的设计，由于要实现对硬件资源的充分，不可避免的就是在数据流动中，出现了数据的逆向流动进而破坏了流水线的数据通路，分别可以分为结构冒险，数据冒险，控制冒险三大类，再设计过程中应该注意避免，以防出现无法预知的错误。

1. **实验要求（功能设计要求）：**

（一）、功能设计说明：

1. 完成以下指令集。

a) 书上的11条指令：add，sub，and，or，slt等多条指令以及lw，sw，beq和J指令。

b) 不支持溢出。

2. 处理器为流水线设计：

a) 数据通路由如下module组成：PC(程序计数器)、NPC(NextPC计算单元)（5.19）、Register File (寄存器文件、寄存器堆)、ALU(算术逻辑单元)（5.13）、EXT(扩展单元)（5.18）、IM(指令存储器)、DM(数据存储器)。

b) IM：容量为4KB(32bit×1024字)。

c) DM：容量为4KB(32bit×1024字)。

d) forwarding单元（处理数据冒险）

e） 数据冒险检测单元

f) IF\_ID\_reg寄存器文件

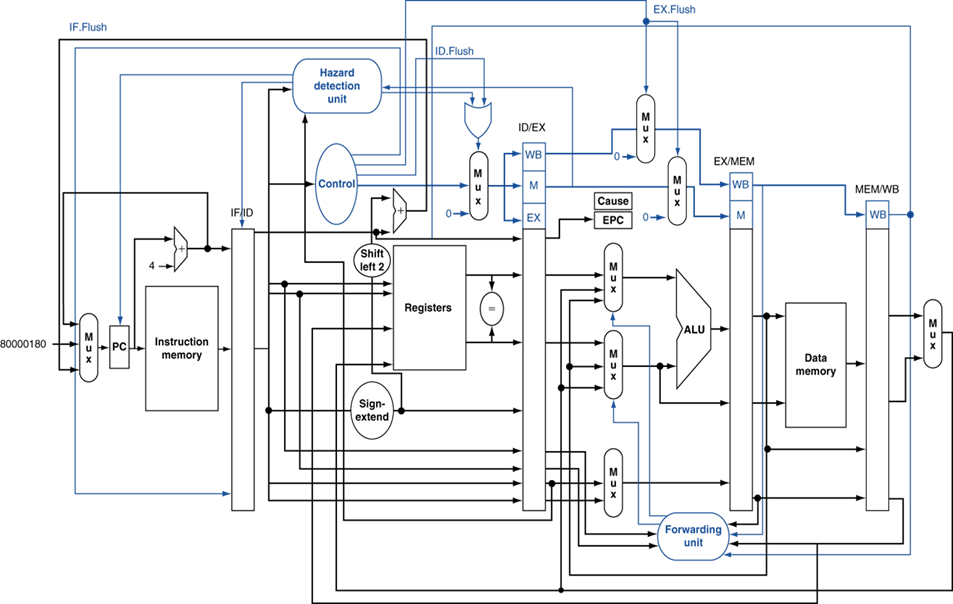
g) ID\_Exe\_reg 寄存器文件

h) Exe\_Mem\_reg 寄存器文件

i) Mem\_wr\_reg 寄存器文件

3、流水线的数据通路架构图（来源于计算机组成与设计:硬件/软件接口）。

* 1. 基于以下的数据通路实现上述11条指令集。数据通路



* 1. 模块分解：

PC

**IF阶段**

Im\_4k

Iunit

IF\_ID\_reg

Im\_test\_code

decode

**ID阶段**

Ctrl

regfile

Hazrd\_det

datapath

Ext

testbench

ID\_EX\_reg

Mux3

ExetUnit

Alu

Exe阶段

Mux1

Frowding

Mux2

EX\_Mem\_reg

Memj阶段

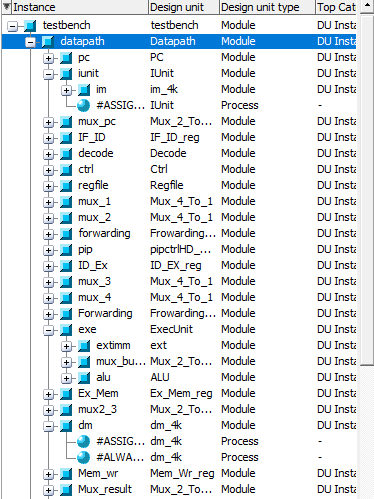
Dm\_4k

RegWr阶段

Mux2

Mem\_wr\_reg

* 1. 实现结果：



4、code.txt中存储的是指令码

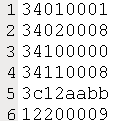
1. 用VerilogHDL建模IM时，必须以读取文件的方式将code.txt中指令加载至IM中。
2. code.txt的格式如 Figure1所示。每条指令占用1行，指令二进制码以文本方式存储。

Figure1code.txt文件格式

1. 为使得代码更加清晰可读，建议多使用宏定义，并将宏定义组织在合理的头文件中。
2. PC复位后初值为0x0000\_3000，目的是与MARS的Memory Configuration相配合。
   1. 也可以用测试程序将通过MARS产生，其配置模式如
   2. Figure2所示。
3. **模块化和层次化设计说明:**

基本模块与单周期的设计基本一致 ,因此在此处只对,流水线中的特有的模块进行分析.因为单周期正在实现的数据通路中的控制信号是不变的,因此,在流水线的设计中可将每一条指令在各个阶段中的控制信号,保存下来以便在之后的数据通路中使用.所以对于五级流水线可设计四个寄存器对中间数据的存储.

(1)、IF\_ID\_reg:

Clk

o\_PC+4

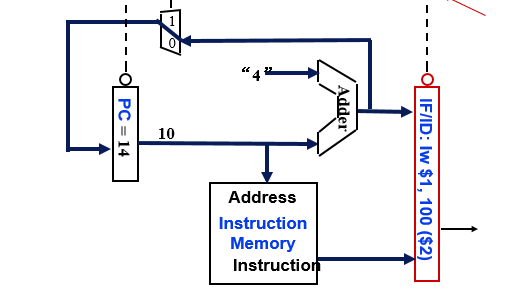
i\_PC+4

o\_instruction

i\_instruction

因此,可设计一个If\_Id\_reg [63:0].的寄存器用于存储取指令阶段产生的pc+4,以及读出的指令。值得注意的是每次都是在时钟下降沿，将数据读出寄存器。

（2）、Iunit模块：取指令并送到IF\_ID\_reg中



取指令部件

(3)、ID\_EX\_reg

W’B

**Ctrl**

M

Ex

o\_imm16

i\_imm16

o\_busA

i\_busA

o\_Rt

o\_busB

i\_busB

i\_Rt

i\_Rs

o\_Rs

Clk

|  |
| --- |
| Memtoreg |
| RegWr |

|  |
| --- |
| Extop |
| ALUsrc |
| ALUCtr |
| RegDst |

其中：

|  |
| --- |
| Memwr |

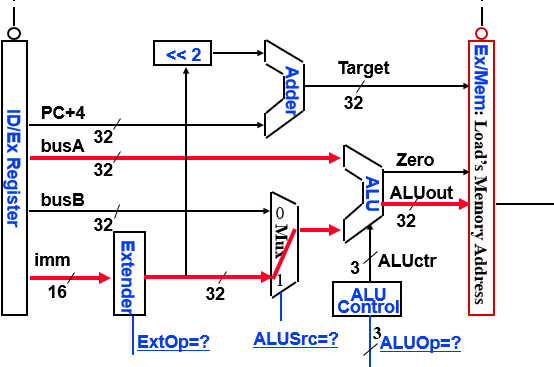
其中，第一个为Mem,

第二个为 Wr

的第三个为Exe

的控制信号。

(4)、ExeUnit模块：



对于基本流水线，Ex阶段分别执行各自的操作：

（1）计算内存地址（lw,sw）

（2）计算转移目标地址(beq,jump)

（3）一般ALU运算(add,sub,or……)

5)、EX\_Mem\_reg

W’B

**Ctrl**

M

O\_overflow

i\_overflow

i\_aluresult

o\_result

o\_RW

i\_Rw

(6)、Mem\_Wr\_reg

W’B

**Ctrl**

O\_overflow

i\_overflow

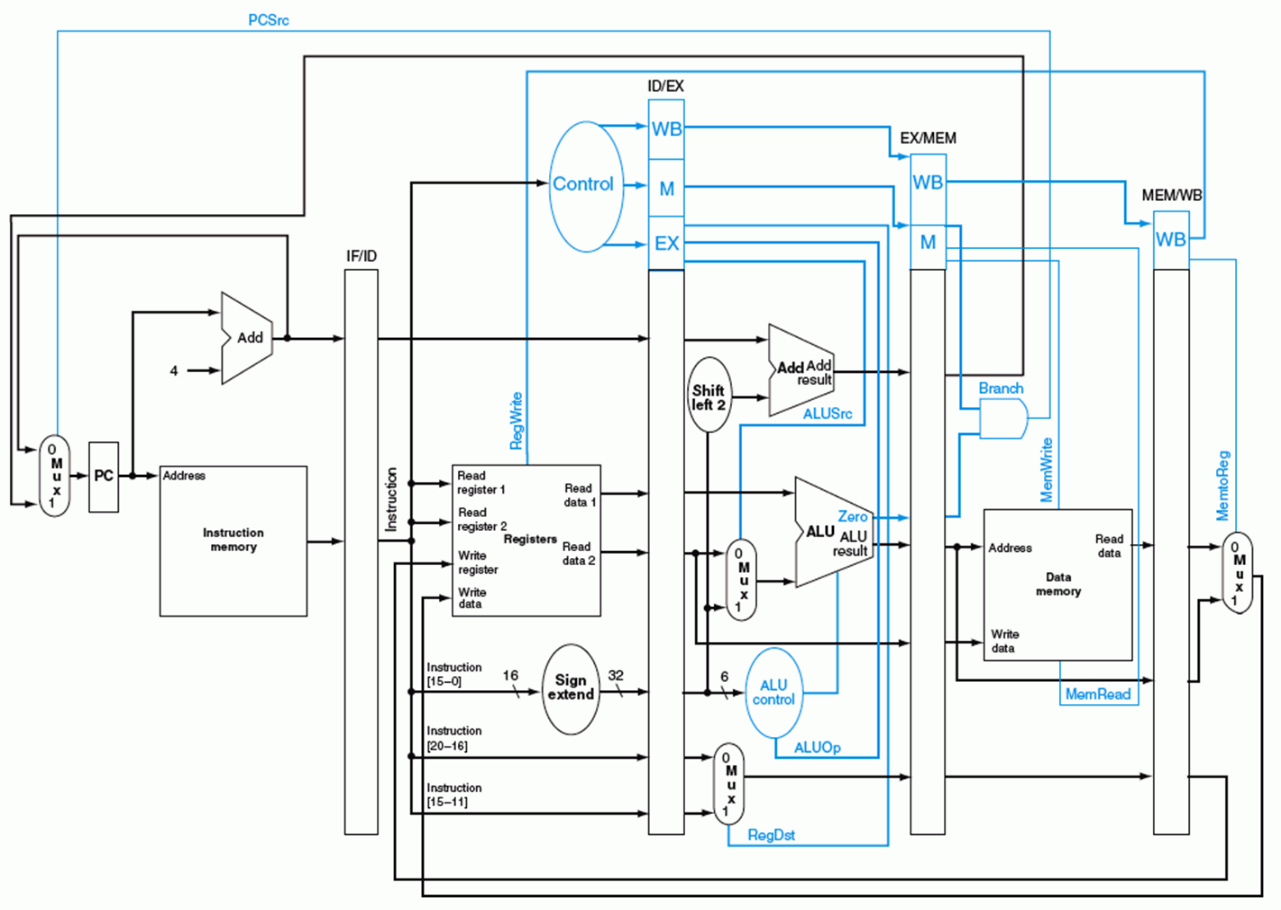
i\_data

o\_data

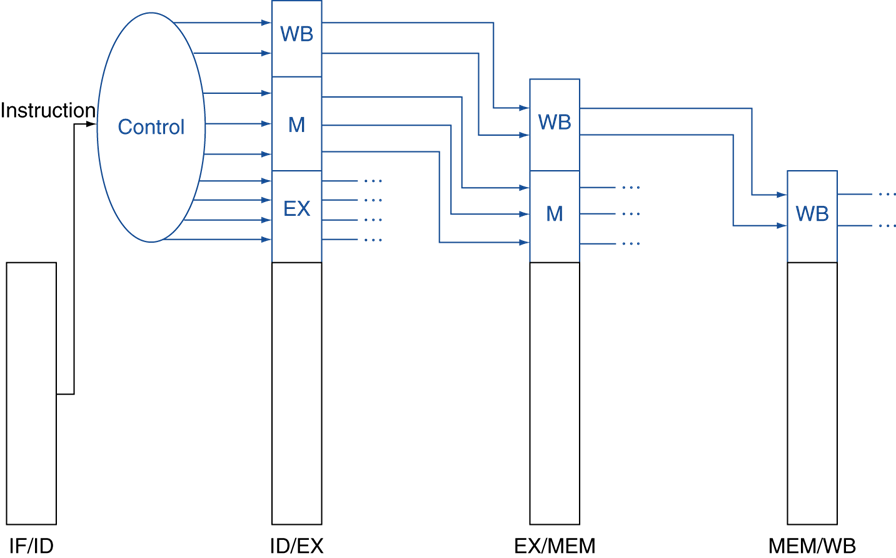
o\_RW

i\_Rw

按照课本上的基本数据通路即可组装为五级基本流水线。



其中对于各个段寄存器的控制指令的变化过程如下：

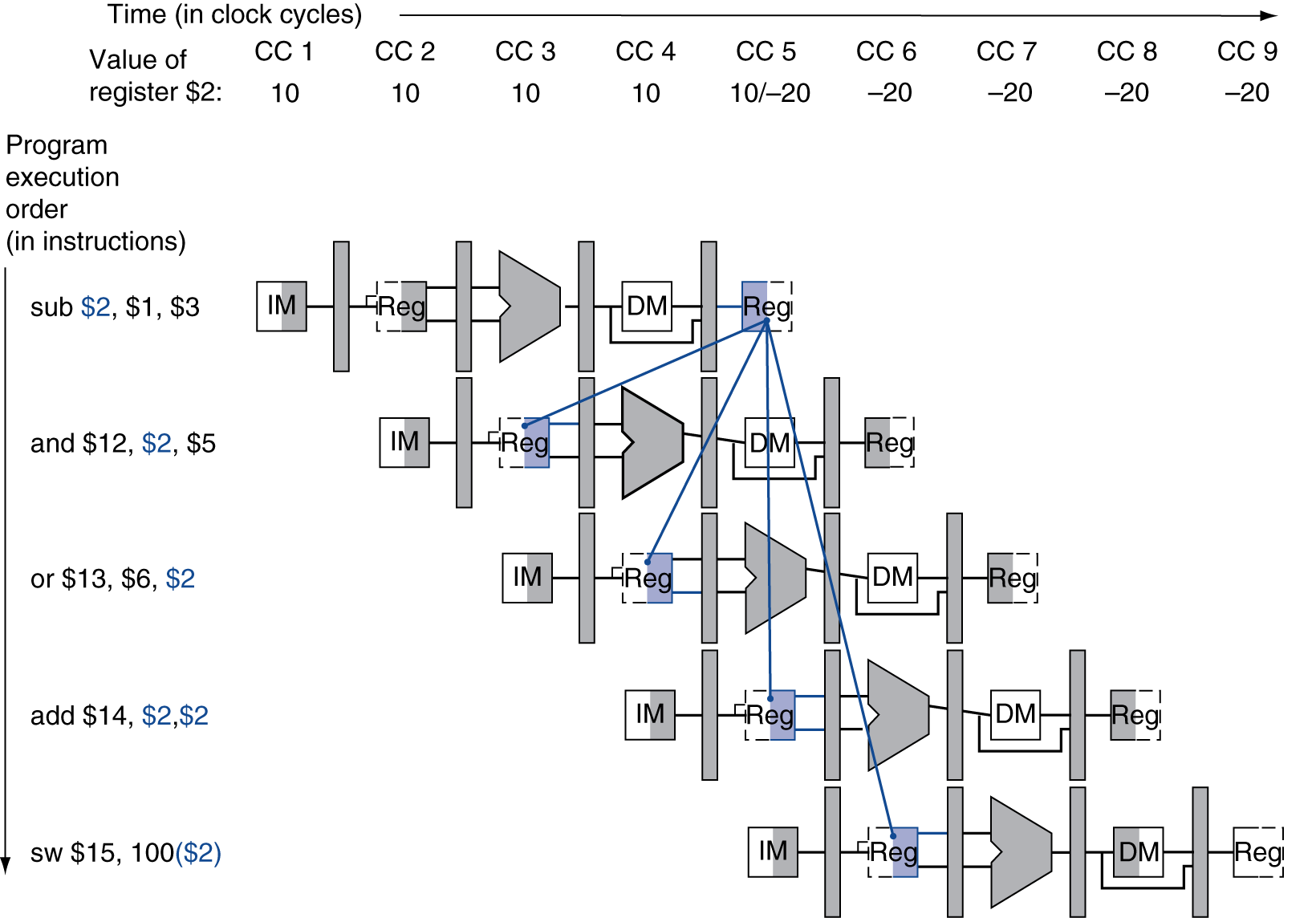


1. **流水线的冒险及处理：**

（1）、结构冒险：同一部件同时被不同指令所使用，也就是由硬件资源竞争造成的。

解决方法：一个部件每条指令只能使用1次，且只能在特定周期使用

设置多个部件，以避免冲突。如指令存储器IM 和数据存储器DM分开。

（2）、数据冒险：后面指令用到前面指令结果数据时，前面指令的结果还没产生

解决方法： 方法1：硬件阻塞（stall）

方法2：软件插入“NOP”指令

方法3：合理实现寄存器堆的读/写操作（不能解决所有数据冒险）:前半时钟周期写，后半时钟周期读，若同一个时钟内前面指令写入的数据正好是后面指令所读数据，则不会发生数据冒险

方法4：转发（Forwarding或Bypassing 旁路）技术

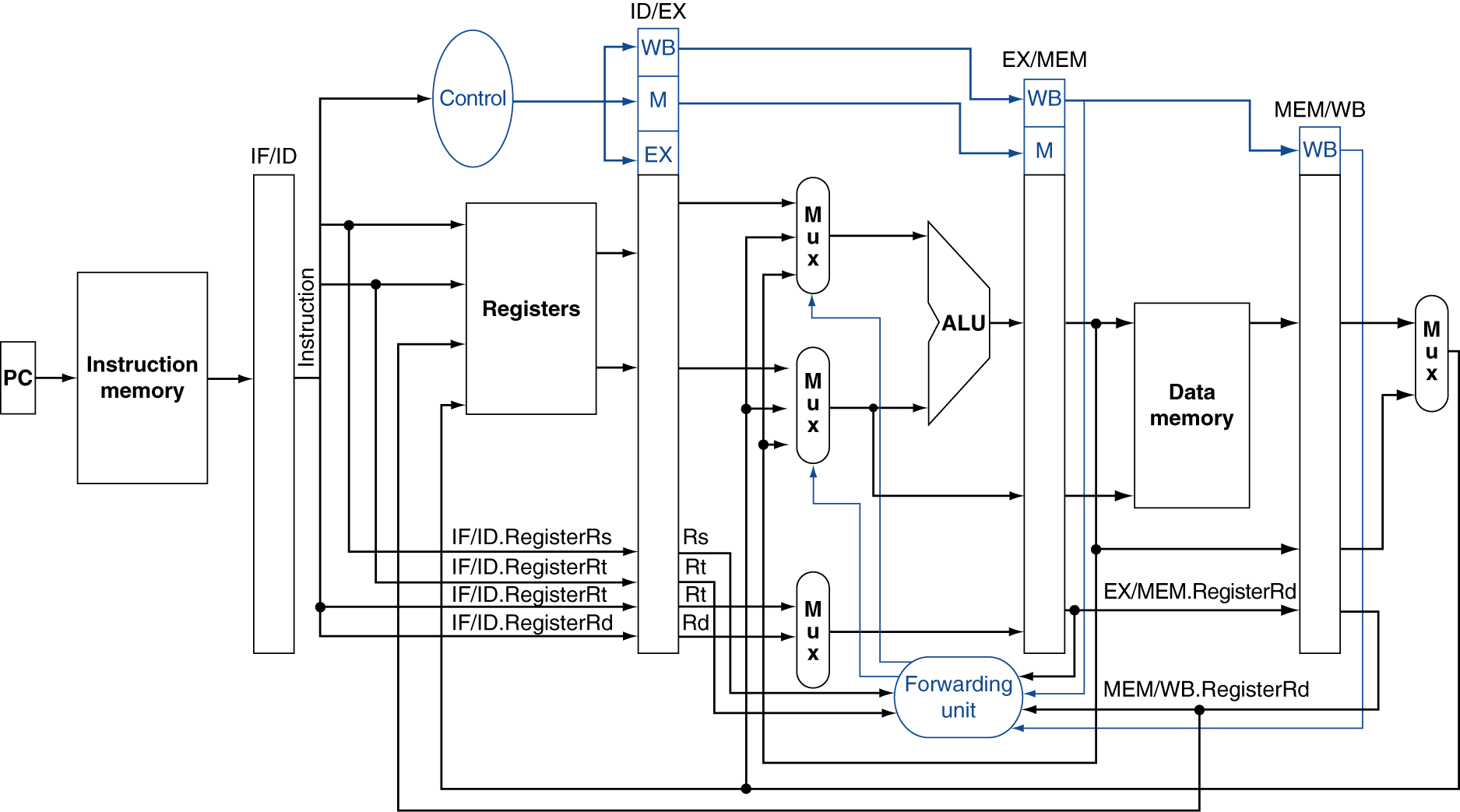
若相关数据是ALU结果， 可通过转发解决

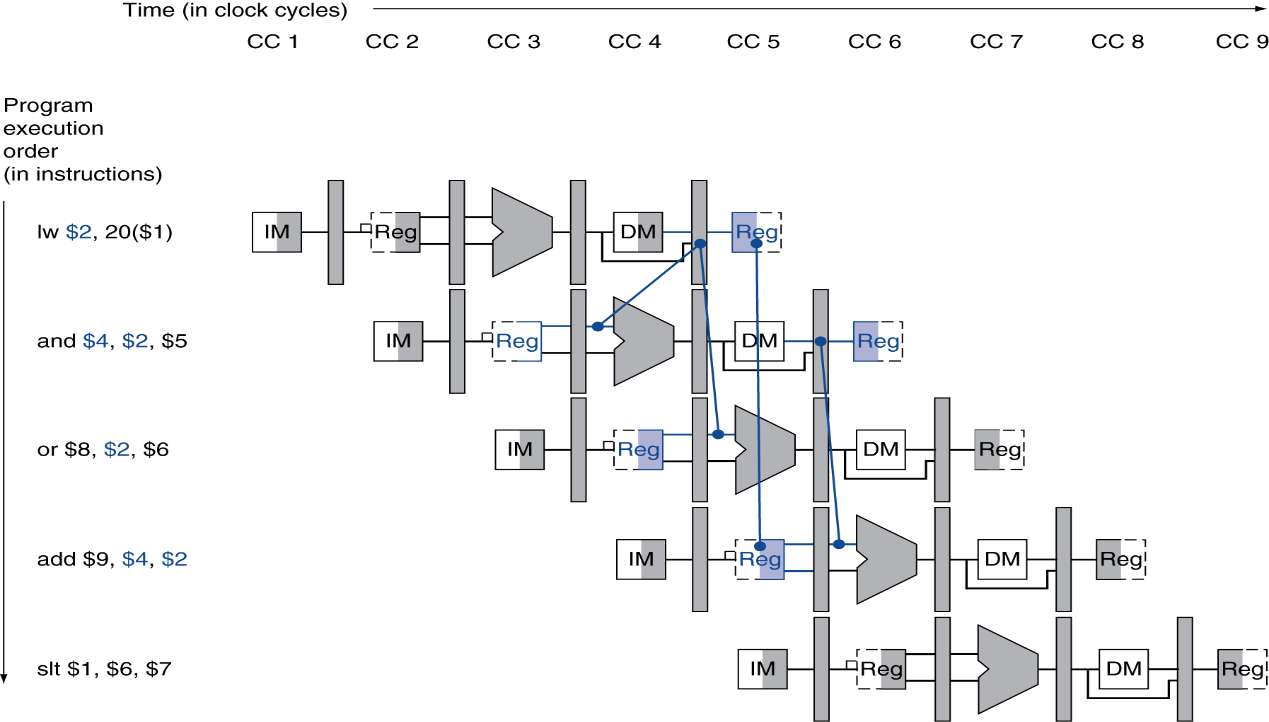
若相关数据是上条指令DM读出内容,不能通过转发解决，随后指令需被阻塞一个时钟 或 加NOP指令

方法5：编译优化：调整指令顺序（不能解决所有数据冒险）

在本次实验中，我们所采用的解决办法为转发加阻塞：

* 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

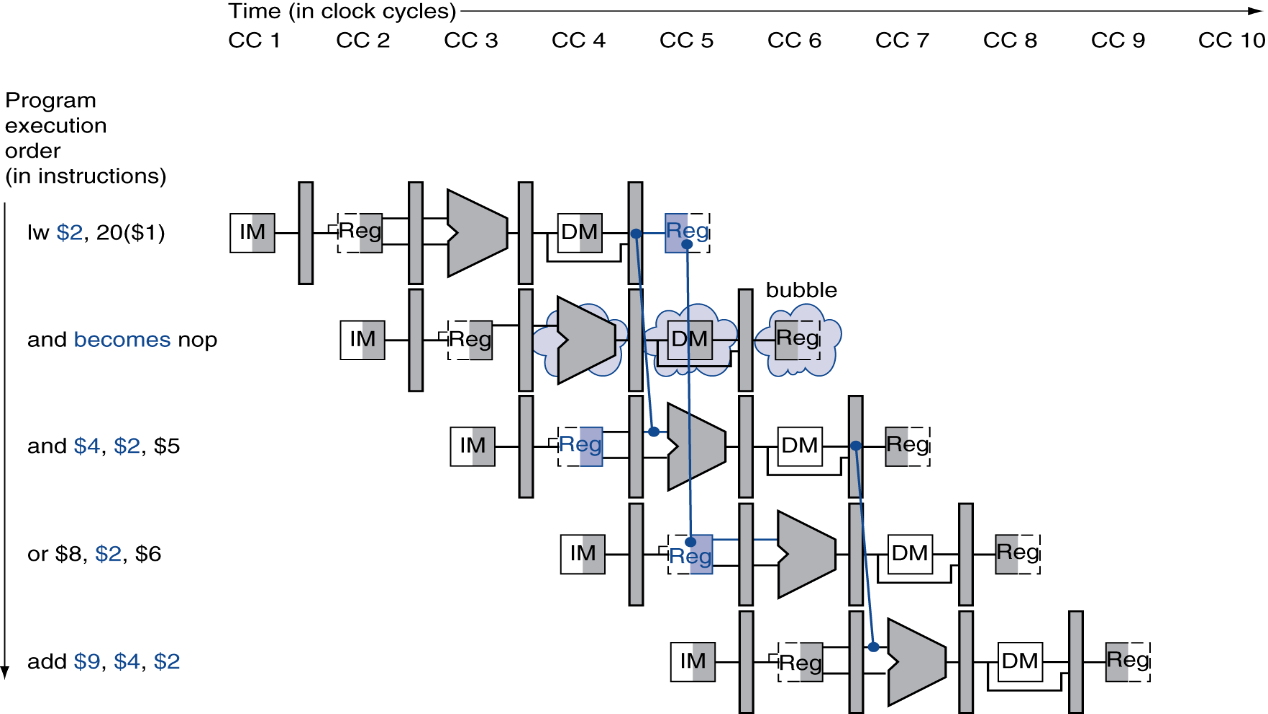


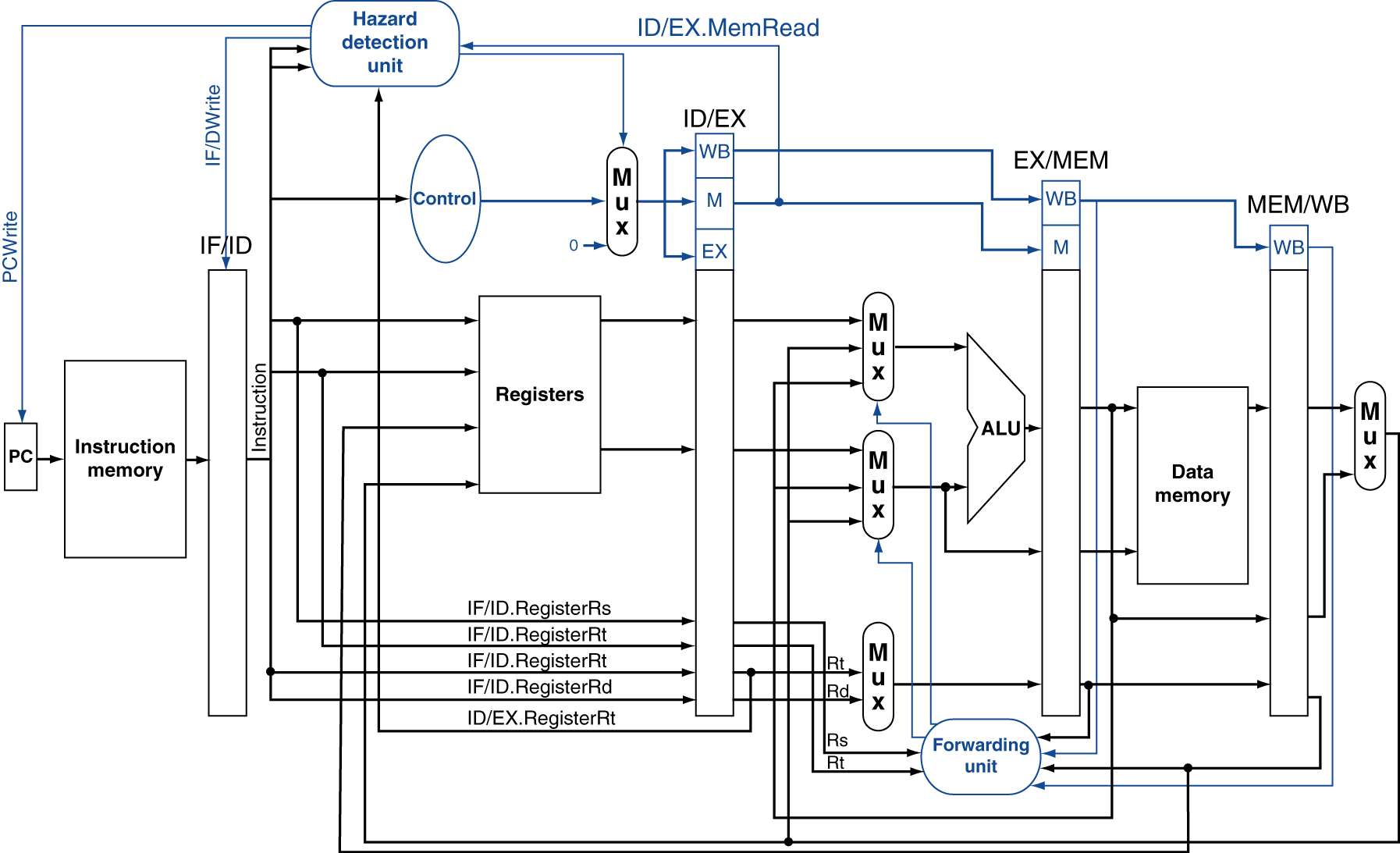


Need to stall for one cycle

但是对于load\_use类型的则需要对其阻塞一个周期：

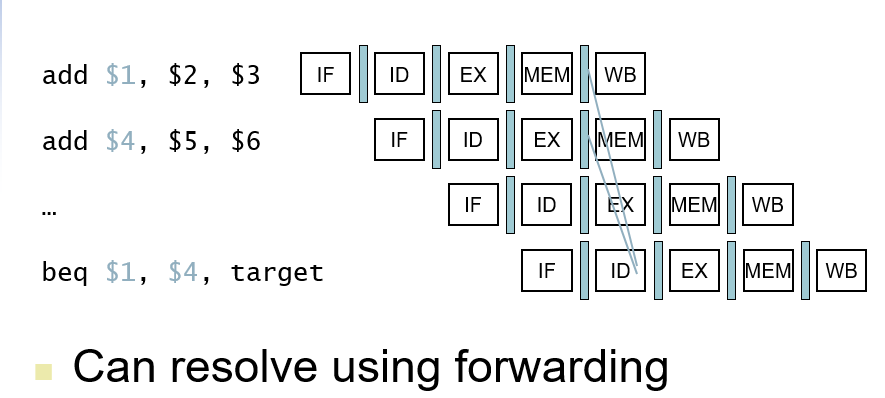
* Check when using instruction is decoded in ID stage
* ALU operand register numbers in ID stage are given by
  + IF/ID.RegisterRs, IF/ID.RegisterRt
* Load-use hazard when
  + ID/EX.MemRead and  
     ((ID/EX.RegisterRt = IF/ID.RegisterRs) or  
     (ID/EX.RegisterRt = IF/ID.RegisterRt))
* If detected,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

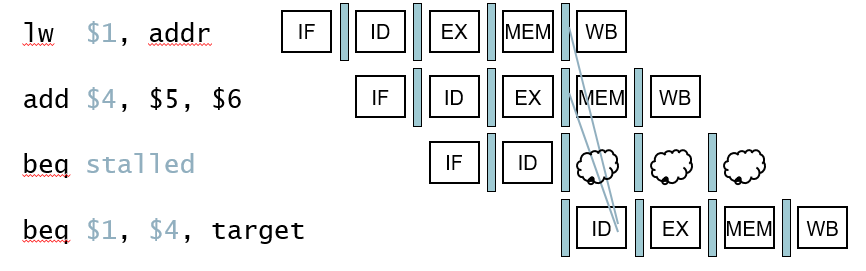




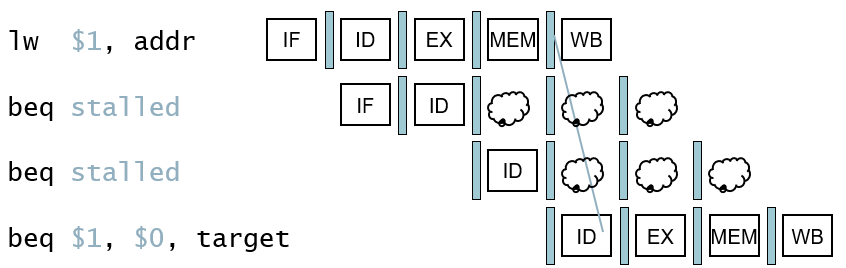
（3）、控制冒险：转移或异常改变执行流程，后继指令在目标地址产生前已被取出；解决方法：采用静态或动态分支预测：编译程序优化指令顺序(分支延迟)

对于branch指令，之前都是在Mem阶段返回值得，导致索取出的指令浪费的情况，因此可将beq指令提前至ID阶段进行执行以减少计算机资源的浪费。但是将beq指令前置，将会导致beq中再次发生数据冒险。

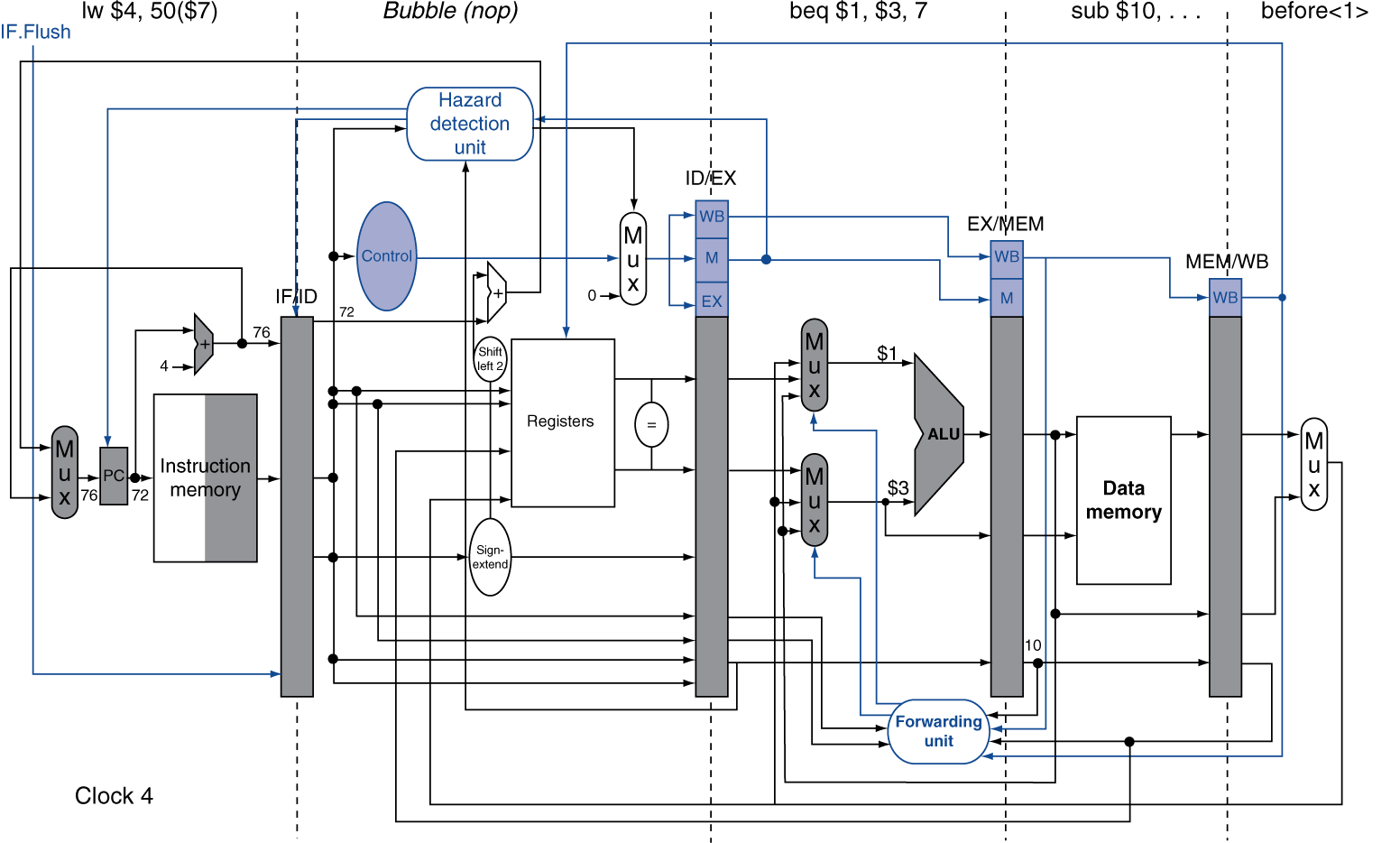


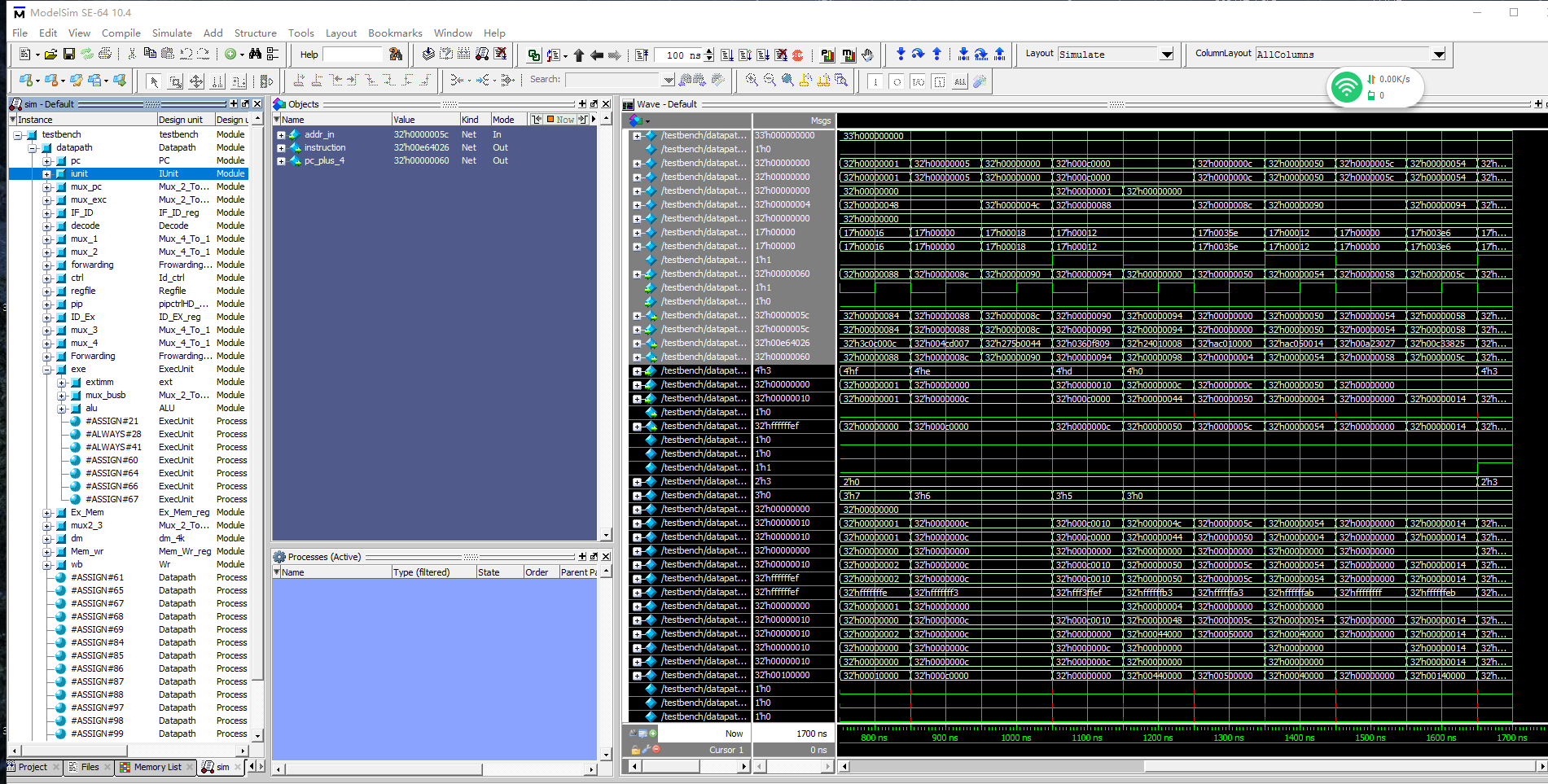


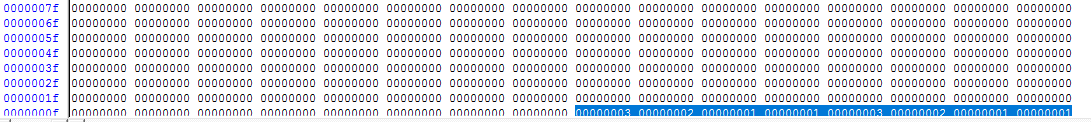
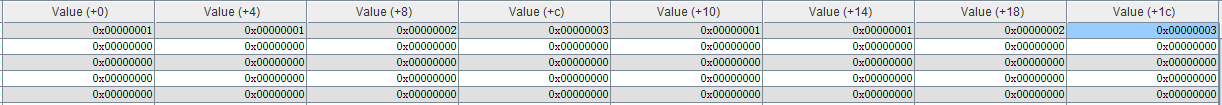
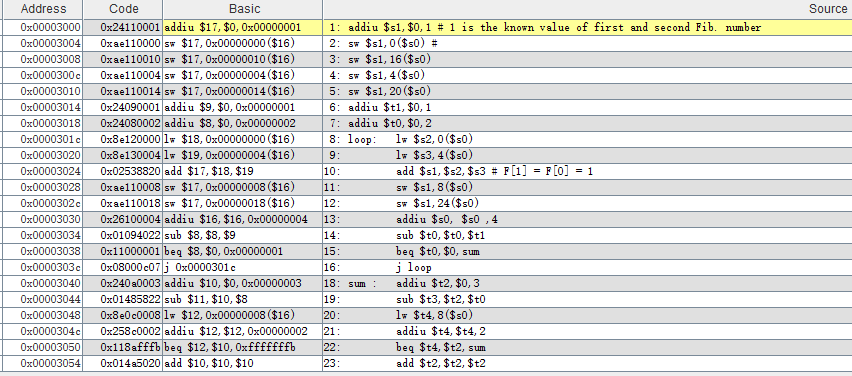
对于以上情况需要阻塞一个周期



需阻塞两个周期



1. **代码测试与结果分析：**



1. **11条指令向56条指令的扩展**

## 数据格式

处理器可处理的数据格式定义如下：

* 比特（bit, b）
* 字节（Byte, 8bits, B）
* 半字（Halfword, 16bits, H）
* 字（Word, 32bits, W）

## 寄存器

处理器包含的软件可见的寄存器种类如下：

* 32 个 32 位通用寄存器，r0~r31。其中有两个被赋予了特殊含义：r0，0 号通用寄存器，值永远为 0； r31，31 号通用寄存器，被 JAL，BLTZAL 和 BGEZAL 指令隐式的用作目标寄存器，存放返回地址。
* HI/LO 寄存器。HI 寄存器存放乘法指令结果的高半部分或是除法指令结果的余数，LO 寄存器存放乘法指令结果的低半部分或是除法指令结果的商。
* 程序计数器（PC）。这个寄存器软件无法直接访问。
* 控制寄存器（CP0）。一组用于中断、例外控制的寄存器。

##### 指令分析：

（1）算术运算指令：

|  |  |
| --- | --- |
| 指令名称格式 | 指令功能简述 |
| ADD rd, rs, rt | 加（可产生溢出例外） |
| ADDI rt, rs, immediate | 加立即数（可产生溢出例外） |
| ADDU rd, rs, rt | 加（不产生溢出例外） |
| ADDIU rt, rs, immeidate | 加立即数（不产生溢出例外） |
| SUB rd. rs, rt | 减（可产生溢出例外） |
| SUBU rd, rs, rt | 减（不产生溢出例外） |
| SLT rd, rs, rt | 有符号小于置1 |
| SLTI rt, rs, immediate | 有符号小于立即数设置1 |
| SLTU rd, rs, rt | 无符号小于设置1 |
| SLTIU rt, rs, immediate | 无符号小于立即数[1]设置1 |
| DIV rs, rt | 有符号字除 |
| DIVU rs,rt | 无符号字除 |
| MULT rs, rt | 有符号字乘 |
| MULTU rs, rt | 无符号字乘 |

（2）逻辑运算指令

|  |  |  |
| --- | --- | --- |
| 指令名称格式 | 指令功能简述 | |
| ANDI rt, rs, immediate | 立即数位与 | |
| LUI rt,immediate | 寄存器高半部分置立即数 | |
| NOR rd, rs, rt | 位或非 | |
| OR rd, rs, rt | 位或 | |
| ORI rt, rs, immediate | 立即数位或 | |
| XOR rd, rs, rt | 位异或 | |
| XORI rt, rs, immediate | 立即数位异或 | |
| AND rd, rs, rt | 位与 |  |

（3）移位指令

|  |  |  |
| --- | --- | --- |
| 指令名称格式 |  | 指令功能简述 |
| SLL rd, rt, sa | 立即数逻辑左移 |  |
| SLLV rd, rs, rt | 变量逻辑左移 |  |
| SRA rd, rt, sa | 立即数算术右移 |  |
| SRAV rd, rs, rt | 变量算术右移 |  |
| SRL rd, rt, sa | 立即数逻辑右移 |  |
| SRLV rd, rs, rt | 变量逻辑右移 |  |

（4）分支跳转指令

|  |  |
| --- | --- |
| 指令名称格式 | 指令功能简述 |
| BEQ rs, rt, offset | 相等转移 |
| BNE rs, rt, offset | 不等转移 |
| BGEZ rs, offset | 大于等于0转移 |
| BGTZ rs, offset | 大于0转移 |
| BLEZ rs, offset | 小于等于0转移 |
| BLTZ rs, offset | 小于0转移 |
| BLTZAL rs, offset | 小于0调用子程序并保存返回地址 |
| BGEZAL rs, offset | 大于等于0调用子程序并保存返回地址 |
| J target | 无条件直接跳转 |
| JAL target | 无条件直接跳转至子程序并保存返回地址 |
| JR rs | 无条件寄存器跳转 |
| JALR rd, rs | 无条件寄存器跳转至子程序并保存返回地址下 |

（6）数据移动指令

|  |  |  |
| --- | --- | --- |
|  | 指令名称格式 | 指令功能简述 |
| MFHI rd |  | HI寄存器至通用寄存器 |
| MFLO rd |  | LO寄存器至通用寄存器 |
| MTHI rs |  | 通用寄存器至HI寄存器 |
| MTLO rs |  | 通用寄存器至LO寄存器 |

（6）自陷指令

|  |  |  |  |
| --- | --- | --- | --- |
| 指令名称格式 |  |  | 指令功能简述 |
| BREAK | 断点 |  |  |
| SYSCALL | 系统调用 |  |  |

注：其中break指令尚未能实现

（7）访存指令

|  |  |  |  |
| --- | --- | --- | --- |
| 指令名称格式 |  | | 指令功能简述 |
| LB rt, offset(base) | 取字节有符号扩展 | |  |
| LBU rt, offset(base) | 取字节无符号扩展 | |  |
| LH rt, offset(base) | 取半字有符号扩展 | |  |
| LHU rt, offset(base) | 取半字无符号扩展 | |  |
| LW rt, offset(base) | 取字 | |  |
| SB rt, offset(base) | 存字节 | |  |
| SH rt, offset(base) | 存半字 | |  |
| SW rt, offset(base) | 存字 |  |  |

（8）自陷指令

|  |  |  |
| --- | --- | --- |
|  | 指令名称格式 | 指令功能简述 |
| ERET |  | 例外处理返回 |
| MFC0 |  | 读CP0寄存器值至通用寄存器 |
| MTC0 |  | 通用寄存器值写入CP0寄存器 |

通过对所有指令的分析，可简单的得出需要对原来11条指令,扩展到56 条需要对原来数据通路中的ALU改进使得ALU可支持更多的运算，当然在这次试验的56条指令中只需要ALU提供14 种运算，因此需要对ALU的控制信号做出相应变化，此外还需要对所有指令进行译码，因此还需要对所有的指令正确的译码，并在第二阶段（Id）阶段作出相应的判断，并生成各个阶段所需要的控制信号，由于在实验的过程对需要将所有的branch以及跳转指令提前至第二周期进行判断，避免资源的浪费，因此在实验的过程中我将所有的branch指令的对busA 以及BusB的值比较多额过程都放到第二周期的额控制信号生成阶段。以下提供在实验过程中所修改的实验代码的内容。

（一）、ALU中所欲操作的定义以及分类：

wire inst\_add, inst\_sub, inst\_slt,inst\_sltu,inst\_subu,inst\_addu;

wire inst\_and, inst\_nor, inst\_or, inst\_xor;

wire inst\_sll, inst\_srl, inst\_sra,inst\_lui;

wire [13:0] alu\_type;

assign inst\_add = add | addi ; // 做加法(考虑溢出)

assign inst\_addu= addu |addiu | inst\_load | inst\_store | inst\_j\_link | inst\_B\_link;

assign inst\_sub = sub; // 减法（需考虑溢出）

assign inst\_subu = subu;

assign inst\_slt = slt | slti; // 有符号小于置位

assign inst\_sltu= sltu | sltiu; // 无符号小于置位

assign inst\_and = And | andi; // 逻辑与

assign inst\_nor = Nor; // 逻辑或非

assign inst\_or = Or | ori; // 逻辑或

assign inst\_xor = Xor | xori; // 逻辑异或

assign inst\_sll = sll | sllv; // 逻辑左移

assign inst\_srl = srl | srlv; // 逻辑右移

assign inst\_sra = sra | srav; // 算术右移

assign inst\_lui = lui; // 立即数装载高位

(二)AlU控制信号的生成：

assign beq = !Op[5] & !Op[4] & !Op[3] & Op[2] & !Op[1] & !Op[0] && !(busA ^ busB);//000100

assign bne = !Op[5] & !Op[4] & !Op[3] & Op[2] & !Op[1] & Op[0] && (busA ^ busB);//000101

assign bgez = !Op[5] & !Op[4] & !Op[3] & !Op[2] & !Op[1] & Op[0] & (Rt == 5'b00001) & (!busA[31]);//000001 //注其中rt=00001 //busA >= 0

assign bgtz = !Op[5] & !Op[4] & !Op[3] & Op[2] & Op[1] & Op[0] & (Rt == 5'b00000) & (!busA[31] & (|busA));//000111 //busA > 0

assign blez = !Op[5] & !Op[4] & !Op[3] & Op[2] & Op[1] & !Op[0] & (Rt == 5'b00000) & ( busA[31] | ~(|busA));//000110 //busA <= 0

assign bltz = !Op[5] & !Op[4] & !Op[3] & !Op[2] & !Op[1] & Op[0] & (Rt == 5'b00000) & ( busA[31]);//000001 //注其中rt=00000 //busA < 0

assign bgezal = !Op[5] & !Op[4] & !Op[3] & !Op[2] & !Op[1] & Op[0] & (Rt == 5'b10001) & (!busA[31]);//000001 //注其中rt=10001 //busA >=0

assign bltzal = !Op[5] & !Op[4] & !Op[3] & !Op[2] & !Op[1] & Op[0] & (Rt == 5'b10000) & ( busA[31]);//000110 //注其中rt=10000//busA < 0

//alu控制信号的生成aluctr

assign alu\_type = {inst\_addu,inst\_add,inst\_or, inst\_nor,inst\_subu,inst\_sub,inst\_sltu,inst\_slt,

inst\_and,inst\_xor,inst\_sll,inst\_srl,inst\_sra,inst\_lui};

always @(\*)

begin

case(alu\_type)

14'b10\_0000\_0000\_0000: ALUctr <= 4'b0000; //addu

14'b01\_0000\_0000\_0000: ALUctr <= 4'b0001; //add

14'b00\_1000\_0000\_0000: ALUctr <= 4'b0010; //or

14'b00\_0100\_0000\_0000: ALUctr <= 4'b0011; //nor

14'b00\_0010\_0000\_0000: ALUctr <= 4'b0100; //subu

14'b00\_0001\_0000\_0000: ALUctr <= 4'b0101; //sub

14'b00\_0000\_1000\_0000: ALUctr <= 4'b0110; //sltu

14'b00\_0000\_0100\_0000: ALUctr <= 4'b0111; //slt

14'b00\_0000\_0010\_0000: ALUctr <= 4'b1001; //and

14'b00\_0000\_0001\_0000: ALUctr <= 4'b1010; //xor

14'b00\_0000\_0000\_1000: ALUctr <= 4'b1011; //sll

14'b00\_0000\_0000\_0100: ALUctr <= 4'b1100; //srl

14'b00\_0000\_0000\_0010: ALUctr <= 4'b1101; //sra

14'b00\_0000\_0000\_0001: ALUctr <= 4'b1110; //lui

default: ALUctr <= 4'b1111;

endcase // alutype

end

（三）、对branch指令的译码的修改（见上面文本框）

（四）、所有指令的译码（由于篇幅有限只给出部分代码）

assign sa\_zero = ~(|shamt);

assign R\_type = !Op[5] & !Op[4] & !Op[3] & !Op[2] & !Op[1] & !Op[0];//000000

//（add,addu,sub,subu,slt,sltu,div,divu,mult,multu,and,nor,or,xor,sllv,sll,srav,sra,srlv,srl）

assign add = R\_type & func[5] & !func[4] & !func[3] & !func[2] & !func[1] & !func[0] & sa\_zero ;//100000

assign addu = R\_type & func[5] & !func[4] & !func[3] & !func[2] & !func[1] & func[0] & sa\_zero ;//100001

assign sub = R\_type & func[5] & !func[4] & !func[3] & !func[2] & func[1] & !func[0] & sa\_zero ;//100010

assign subu = R\_type & func[5] & !func[4] & !func[3] & !func[2] & func[1] & func[0] & sa\_zero ;//100011

assign slt = R\_type & func[5] & !func[4] & func[3] & !func[2] & func[1] & !func[0] & sa\_zero ;//101010

assign sltu = R\_type & func[5] & !func[4] & func[3] & !func[2] & func[1] & func[0] & sa\_zero ;//101011

assign div = R\_type & !func[5] & func[4] & func[3] & !func[2] & func[1] & !func[0] & sa\_zero & (Rd == 5'b00000) ;//011010

assign divu = R\_type & !func[5] & func[4] & func[3] & !func[2] & func[1] & func[0] & sa\_zero & (Rd == 5'b00000) ;//011011

assign mult = R\_type & !func[5] & func[4] & func[3] & !func[2] & !func[1] & !func[0] & sa\_zero & (Rd == 5'b00000) ;//011000

assign multu = R\_type & !func[5] & func[4] & func[3] & !func[2] & !func[1] & func[0] & sa\_zero & (Rd == 5'b00000) ;//011001

assign And = R\_type & func[5] & !func[4] & !func[3] & func[2] & !func[1] & !func[0] & sa\_zero ;//100100

assign Nor = R\_type & func[5] & !func[4] & !func[3] & func[2] & func[1] & func[0] & sa\_zero ;//100111

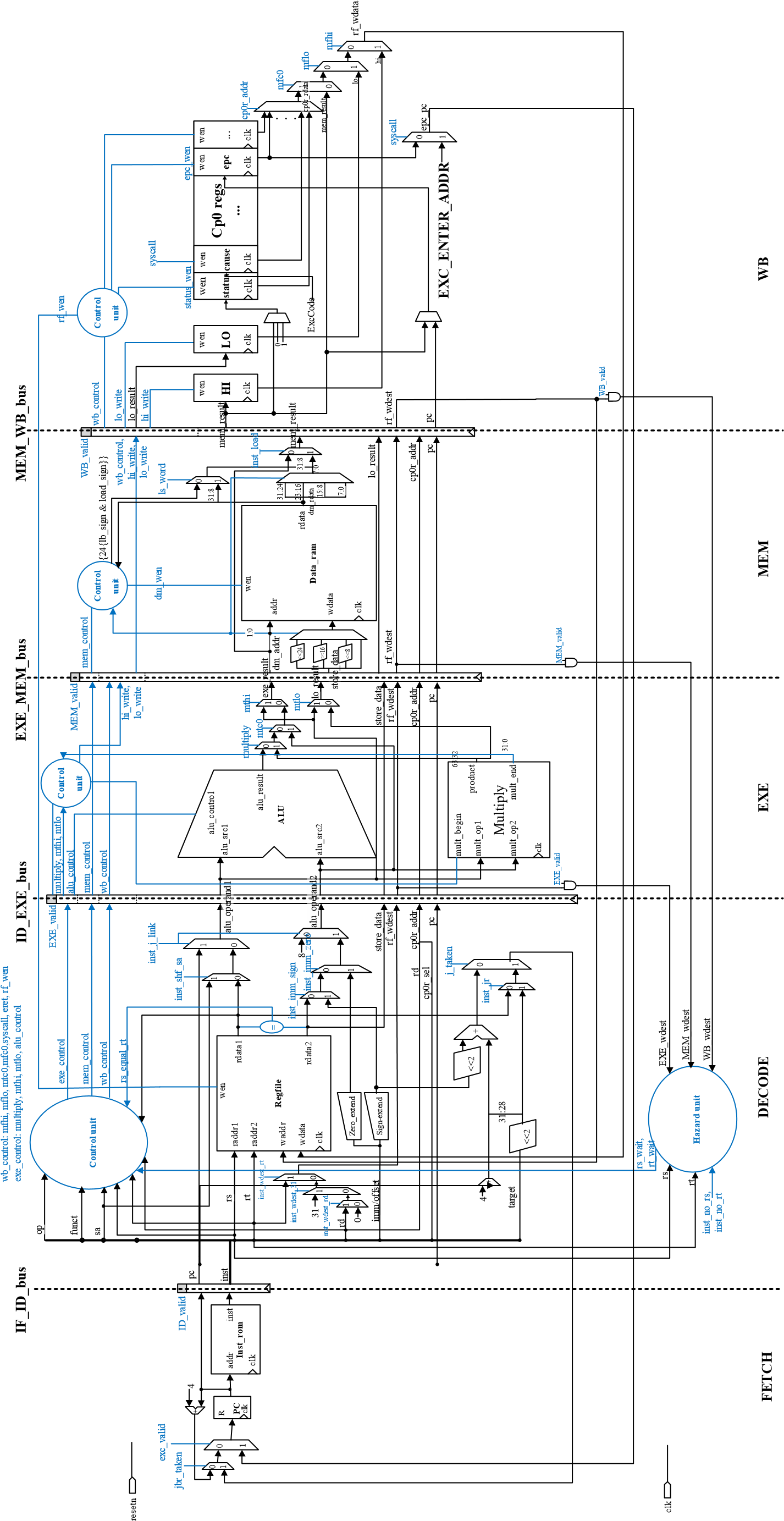
assign Or = R\_type & func[5] & !func[4] & !func[3] & func[2] & !func[1] & func[0] & sa\_zero ;//100101

assign Xor = R\_type & func[5] & !func[4] & !func[3] & func[2] & func[1] & !func[0] & sa\_zero ;//100110

assign sll = R\_type & !func[5] & !func[4] & !func[3] & !func[2] & !func[1] & !func[0] & (Rs == 5'b00000) & !sa\_zero;//000000

assign sllv = R\_type & !func[5] & !func[4] & !func[3] & func[2] & !func[1] & !func[0] & sa\_zero ;//000100

assign srlv = R\_type & !func[5] & !func[4] & !func[3] & func[2] & func[1] & !func[0] & sa\_zero ;//001100

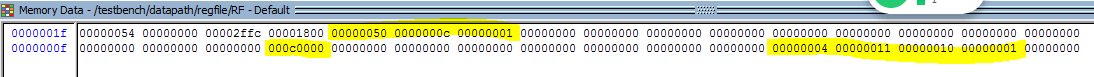


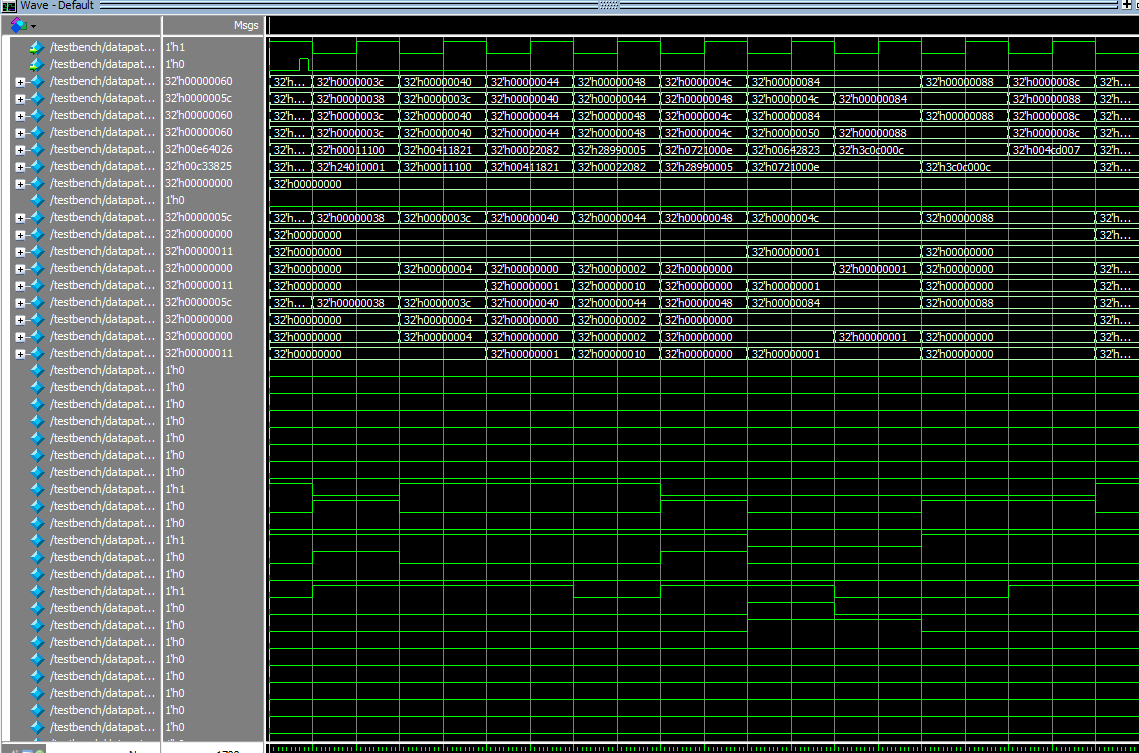
通过上面对指令分析理解可通过以上的数据通路实现所有的指令。

对所有指令集的测设：简单给出前几条的测试结果：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 指令地址 | 汇编指令 |  | 结果描述 | 机器指令的机器码 | |
| 16 进制 | 二进制 |
|  | | Exception 入口地址，在 SYSCALL 指令执行后进入此处执行 | | | |
| 00H | sw $1,#0($0) |  | Mem[0000\_0000H] = 0000\_0008H | AC010000 | 1010\_1100\_0000\_0001\_0000\_0000\_0000\_0000 |
| 04H | sw $2,#4($0) |  | Mem[0000\_0004H]  = 0000\_0010H | AC020004 | 1010\_1100\_0000\_0010\_0000\_0000\_0000\_0100 |
| 08H | sw $3,#8($0) |  | Mem[0000\_0008H]  = 0000\_0011H | AC030008 | 1010\_1100\_0000\_0011\_0000\_0000\_0000\_1000 |
| 0CH | sw $4,#12($0) |  | Mem[0000\_000CH] = 0000\_0004H | AC04000C | 1010\_1100\_0000\_0100\_0000\_0000\_0000\_1100 |
| 10H | sw $5,#16($0) |  | Mem[0000\_0010H] = 0000\_000DH | AC050010 | 1010\_1100\_0000\_0101\_0000\_0000\_0001\_0000 |
| 14H | sw $6,#24($0) |  | Mem[0000\_0018H] = FFFF\_FFE2H | AC060018 | 1010\_1100\_0000\_0110\_0000\_0000\_0001\_1000 |
| 18H | sw $7,#112($0) |  | Mem[0000\_0070H] = FFFF\_FFF3H | AC070070 | 1010\_1100\_0000\_0111\_0000\_0000\_0111\_0000 |
| 1CH | sw $25,#116($0) |  | Mem[0000\_0074H]  = 0000\_0001H | AC190074 | 1010\_1100\_0001\_1001\_0000\_0000\_0111\_0100 |
| 20H | sw $13,#24($0) |  | Mem[0000\_0078H] = 0000\_0000H | AC0D0078 | 1010\_1100\_0000\_1101\_0000\_0000\_0111\_1000 |
| 24H | mfc0 $1, cp0(14.0) |  | [$1] = 0000\_0104H | 40017000 | 0100\_0000\_0000\_0001\_0111\_0000\_0000\_0000 |
| 28H | addiu $1,$1,#4 |  | [$1] = 0000\_0108H | 24210004 | 0010\_0100\_0010\_0001\_0000\_0000\_0000\_0100 |
| 2CH | mtc0 $1, cp0(14.0) |  | cp0(14.0)  = 0000\_0108H | 40817000 | 0100\_0000\_1000\_0001\_0111\_0000\_0000\_0000 |
| 30H | eret |  | 返回 108H | 42000018 | 0100\_0010\_0000\_0000\_0000\_0000\_0001\_1000 |
|  | | CPU 复位地址 0000\_0034H | | | |
| 34H | addiu $1, $0,#1 |  | [$1] = 0000\_0001H | 24010001 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_0001 |
| 38H | sll $2, $1,#4 |  | [$2] = 0000\_0010H | 00011100 | 0000\_0000\_0000\_0001\_0001\_0001\_0000\_0000 |
| 3CH | addu $3, $2,$1 |  | [$3] = 0000\_0011H | 00411821 | 0000\_0000\_0100\_0001\_0001\_1000\_0010\_0001 |
| 40H | srl $4, $2,#2 |  | [$4] = 0000\_0004H | 00022082 | 0000\_0000\_0000\_0010\_0010\_0000\_1000\_0010 |
| 44H | slti $25,$4,#5 |  | [$25] = 0000\_0001H | 28990005 | 0010\_1000\_1001\_1001\_0000\_0000\_0000\_0101 |
| 48H | bgez $25,#14 |  | 跳转到 84H | 0721000E | 0000\_0111\_0010\_0001\_0000\_0000\_0000\_1110 |
| 4CH | subu $5, $3,$4 |  | [$5] = 0000\_000DH | 00642823 | 0000\_0000\_0110\_0100\_0010\_1000\_0010\_0011 |
| 50H | sw $5, #20($0) |  | Mem[0000\_0014H] = 0000\_000DH | AC050014 | 1010\_1100\_0000\_0101\_0000\_0000\_0001\_0100 |
| 54H | nor $6, $5,$2 |  | [$6] = FFFF\_FFE2H | 00A23027 | 0000\_0000\_1010\_0010\_0011\_0000\_0010\_0111 |
| 58H | or $7, $6,$3 |  | [$7] = FFFF\_FFF3H | 00C33825 | 0000\_0000\_1100\_0011\_0011\_1000\_0010\_0101 |
| 5CH | xor $8, $7,$6 |  | [$8] = 0000\_0011H | 00E64026 | 0000\_0000\_1110\_0110\_0100\_0000\_0010\_0110 |
| 60H | beq $8, $3,#2 |  | 跳转到 6CH | 11030002 | 0001\_0001\_0000\_0011\_0000\_0000\_0000\_0010 |
| 64H | sw $8, #28($0) |  | Mem[0000\_001CH] = 0000\_0011H | AC08001C | 1010\_1100\_0000\_1000\_0000\_0000\_0001\_1100 |
| 68H | slt $9, $1,$2 |  | 不执行 | 0022482A | 0000\_0000\_0010\_0010\_0100\_1000\_0010\_1010 |
| 6CH | lw $10,#28($0) |  | [$10] = 0000\_0011H | 8C0A001C | 1000\_1100\_0000\_1010\_0000\_0000\_0001\_1100 |
| 70H | bne $10,$5,#2 |  | 跳转到 7CH | 15450002 | 0001\_0101\_0100\_0101\_0000\_0000\_0000\_0010 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令地址 | 汇编指令 | 结果描述 | 机器指令的机器码 | |
| 16 进制 | 二进制 |
| 74H | and $11,$2,$1 | [$11] = 0000\_0000H | 00415824 | 0000\_0000\_0100\_0001\_0101\_1000\_0010\_0100 |
| 78H | sw $11,#28($0) | 不执行 | AC0B001C | 1010\_1100\_0000\_1011\_0000\_0000\_0001\_1100 |
| 7CH | jal #38 | 跳转到 98H,  [$31] =0000\_0084H | 0C000026 | 0000\_1100\_0000\_0000\_0000\_0000\_0010\_0110 |
| 80H | sw $4, #16($0) | Mem[0000\_0010H] = 0000\_0004H | AC040010 | 1010\_1100\_0000\_0100\_0000\_0000\_0001\_0000 |
| 84H | lui $12,#12 | [$12] = 000C\_0000H | 3C0C000C | 0011\_1100\_0000\_1100\_0000\_0000\_0000\_1100 |
| 88H | srav $26,$12,$2 | [$26] = 0000\_000CH | 004CD007 | 0000\_0000\_0100\_1100\_1101\_0000\_0000\_0111 |
| 8CH | addiu $27,$26,#68 | [$27] = 0000\_0050H | 275B0044 | 0010\_0111\_0101\_1011\_0000\_0000\_0100\_0100 |
| 90H | jalr $27 | 跳转到 50H,  [$31] = 0000\_0098H | 0360F809 | 0000\_0011\_0110\_0000\_1111\_1000\_0000\_1001 |
| 94H | addiu $1, $0,#8 | [$1] = 0000\_0008H | 24010008 | 0010\_0100\_0000\_0001\_0000\_0000\_0000\_1000 |
| 98H | sb $26,#5($3) | MEM[0000\_0014H] = 000C\_000DH | A07A0005 | 1010\_0000\_0111\_1010\_0000\_0000\_0000\_0101 |
| 9CH | sltu $13,$10,$3 | [$13] = 0000\_0000H | 0143682B | 0000\_0001\_0100\_0011\_0110\_1000\_0010\_1011 |
| A0H | bgtz $13,#2 | 不跳转 | 1DA00002 | 0001\_1101\_1010\_0000\_0000\_0000\_0000\_0010 |
| A4H | sllv $14,$6,$4 | [$14] = FFFF\_FE20H | 00867004 | 0000\_0000\_1000\_0110\_0111\_0000\_0000\_0100 |
| A8H | sra $15,$14,#2 | [$15] = FFFF\_FF88H | 000E7883 | 0000\_0000\_0000\_1110\_0111\_1000\_1000\_0011 |
| ACH | srlv $16,$15,$1 | [$16] = 00FF\_FFFFH | 002F8006 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0110 |
| B0H | blez $16,#7 | 不跳转 | 1A000007 | 0001\_1010\_0000\_0000\_0000\_0000\_0000\_0111 |
| B4H | srav $16,$15,$1 | [$16] = FFFF\_FFFFH | 002F8007 | 0000\_0000\_0010\_1111\_1000\_0000\_0000\_0111 |
| B8H | bltz $16,#6 | 跳转到 D4H | 06000006 | 0000\_0110\_0000\_0000\_0000\_0000\_0000\_0110 |
| BCH | sll $11,$26,#4 | [$11] = 0000\_00C0H | 001A5900 | 0000\_0000\_0001\_1010\_0101\_1001\_0000\_0000 |
| C0H | lw $28,#3($10) | [$28]=000C\_000DH /  000C\_880DH | 8D5C0003 | 1000\_1101\_0101\_1100\_0000\_0000\_0000\_0011 |
| C4H | bne $28,$29,#7 | 不跳转/跳转 E4H | 179D0007 | 0001\_0111\_1001\_1101\_0000\_0000\_0000\_0111 |
| C8H | sb $15,#8($5) | Mem[0000\_0014H] = 000C\_8800H | A0AF0008 | 1010\_0000\_1010\_1111\_0000\_0000\_0000\_1000 |
| CCH | lb $18,#8($5) | [$18] = FFFF\_FF88H | 80B20008 | 1000\_0000\_1011\_0010\_0000\_0000\_0000\_1000 |
| D0H | lbu $19,#8($5) | [$19] = 0000\_0088H | 90B30008 | 1001\_0000\_1011\_0011\_0000\_0000\_0000\_1000 |
| D4H | sltiu $24,$15,#0xFFFF | [$24] = 0000\_0001H | 2DF8FFFF | 0010\_1101\_1111\_1000\_1111\_1111\_1111\_1111 |
| D8H | or $29,$12,$5 | [$29] = 000C\_000DH | 0185E825 | 0000\_0001\_1000\_0101\_1110\_1000\_0010\_0101 |
| DCH | jr $11 | 跳转到 C0H | 01600008 | 0000\_0001\_0110\_0000\_0000\_0000\_0000\_1000 |
| E0H | andi $20,$15,#0xFFFF | [$20] = 0000\_FF88H | 31F4FFFF | 0011\_0001\_1111\_0100\_1111\_1111\_1111\_1111 |
| E4H | ori $21,$15,#0xFFFF | [$21] = FFFF\_FFFFH | 35F5FFFF | 0011\_0101\_1111\_0101\_1111\_1111\_1111\_1111 |
| E8H | xori $22,$15,#0xFFFF | [$22] = FFFF\_0077H | 39F6FFFF | 0011\_1001\_1111\_0110\_1111\_1111\_1111\_1111 |
| ECH | mult $12,$29 | [HI] = 0000\_0090H,  [LO] = 009C\_0000H | 019D0018 | 0000\_0001\_1001\_1101\_0000\_0000\_0001\_1000 |
| F0H | mflo $23 | [$23] = 009C\_0000H | 0000B812 | 0000\_0000\_0000\_0000\_1011\_1000\_0001\_0010 |
| F4H | mfhi $30 | [$30] = 0000\_0090H | 0000F010 | 0000\_0000\_0000\_0000\_1111\_0000\_0001\_0000 |
| F8H | mtlo $26 | [LO] = 0000\_000CH | 03400013 | 0000\_0011\_0100\_0000\_0000\_0000\_0001\_0011 |





如图所示每个周期取指令的状态，以及最后一个红圈显示的便是branch发生数据冒险的时候对的状态。诸如此类。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 指令地址 | 汇编指令 | 结果描述 | 机器指令的机器码 | |
| 16 进制 | 二进制 |
| 13CH | srl $17,$17,#4 | [$17] = 0123\_4567H /  0012\_3456H  0001\_2345H  0000\_1234H  0000\_0123H  0000\_0012H /0000\_001H  0000\_0000H | 00118902 | 0000\_0000\_0001\_0001\_1000\_1001\_0000\_0010 |
| 140H | sw $17,#0($2) | Mem[0000\_003CH]  =0123\_4567H  Mem [0000\_0040H]  =0012\_3456H  Mem [0000\_0044H]  =0001\_2345H  Mem [0000\_0048H]  =0000\_1234H  Mem [0000\_004CH]  =0000\_0123H  Mem [0000\_0050H]  =0000\_0012H  Mem [0000\_0054H]  =0000\_0001H  Mem [0000\_0058H]  =0000\_0000H | AC510000 | 1010\_1100\_0101\_0001\_0000\_0000\_0000\_0000 |
| 144H | bne $17,$0,#-3 | 跳转到 13CH /跳转/ 跳转/跳转/跳转/跳转  /跳转 13CH/不跳转 | 1620FFFD | 0001\_0110\_0010\_0000\_1111\_1111\_1111\_1101 |
| 148H | addiu $2,$2,#4 | [$2] = 0000\_0040H /  0000\_0044H  0000\_0048H  0000\_004CH  0000\_0050H  0000\_0054H  0000\_0058H  0000\_005CH | 24420004 | 0010\_0100\_0100\_0010\_0000\_0000\_0000\_0100 |
| 14CH | addiu $6,$0,#68 | [$6] = 0000\_0044H | 24060044 | 0010\_0100\_0000\_0110\_0000\_0000\_0100\_0100 |
| 150H | addiu $7,$0,#100 | [$7] =0000\_0064H | 24070064 | 0010\_0100\_0000\_0111\_0000\_0000\_0110\_0100 |
| 154H | lw $3,#-28($1) | [$3] = 1234\_5678H /  2345\_6780H  3456\_7800H  4567\_8000H  5678\_0000H  6780\_0000H  7800\_0000H | 8C23FFE4 | 1000\_1100\_0010\_0011\_1111\_1111\_1110\_0100 |
| 158H | lw $4,#-4($2) | [$4] = 0000\_0000H /  0000\_0001H  0000\_0012H  0000\_0123H  0000\_1234H  0001\_2345H  0012\_3456H | 8C44FFFC | 1000\_1100\_0100\_0100\_1111\_1111\_1111\_1100 |
| 15CH | or $5,$3,$4 | [$5] = 1234\_5678H /  2345\_6781H  3456\_7812H  4567\_8123H  5678\_1234H  6781\_2345H  7812\_3456H | 00642825 | 0000\_0000\_0110\_0100\_0010\_1000\_0010\_0101 |
| 160H | sb $5,#0($7) | Mem[0000\_0064H]  =0000\_0078H  Mem [0000\_0064H]  =0000\_8178H  Mem[0000\_0064H]  =0012\_8178H  Mem[0000\_0064H]  =2312\_8178H  Mem[0000\_0068H]  =0000\_0034H  Mem[0000\_0068H]  =0000\_4534H  Mem[0000\_0068H]  =0056\_4534H | A0E50000 | 1010\_0000\_1110\_0101\_0000\_0000\_0000\_0000 |
| 164H | addiu $7,$7,#1 | [$7] = 0000\_0065H /  0000\_0066H  0000\_0067H  0000\_0068H  0000\_0069H  0000\_006AH  0000\_006BH | 24E70001 | 0010\_0100\_1110\_0111\_0000\_0000\_0000\_0001 |
| 168H | addiu $1,$1,#4 | [$1] = 0000\_0040H /  0000\_0044H  0000\_0048H  00\_004CH  0000\_0050H  0000\_0054H  0000\_0058H | 24210004 | 0010\_0100\_0010\_0001\_0000\_0000\_0000\_0100 |
| 16CH | bne $2,$6,#-7 | 跳转到 154H/跳转/跳转/跳转/跳转/跳转  154H/不跳转 | 1446FFF9 | 0001\_0100\_0100\_0110\_1111\_1111\_1111\_1001 |
| 170H | ADDIU $2,$2,#-4 | [$2] = 0000\_0058H /  0000\_0054H  0000\_0050H  0000\_004CH  0000\_0048H  0000\_0044H  0000\_0040H | 2442FFFC | 0010\_0100\_0100\_0010\_1111\_1111\_1111\_1100 |
| 174H | addiu $9,$0,#100 | [$9] = 0000\_0064H | 24090064 | 0010\_0100\_0000\_1001\_0000\_0000\_0110\_0100 |
| 178H | lbu $9,#3($9) | [$9] = 0000\_0023H | 91290003 | 1001\_0001\_0010\_1001\_0000\_0000\_0000\_0011 |
| 17CH | addiu $13,$0,#104 | [$13] = 0000\_0068H | 240D0068 | 0010\_0100\_0000\_1101\_0000\_0000\_0110\_1000 |
| 180H | lw $13,#0($13) | [$13] = 0056\_4534H | 8DAD0000 | 1000\_1101\_1010\_1101\_0000\_0000\_0000\_0000 |
| 184H | ll $9,$9,#24 | [$9] = 2300\_0000H | 00094E00 | 0000\_0000\_0000\_1001\_0100\_1110\_0000\_0000 |
| 188H | xori $13,$13,#9 | [$13] = 0056\_453DH | 39AD0009 | 0011\_1001\_1010\_1101\_0000\_0000\_0000\_1001 |
| 18CH | sw $13,#1($7) | MEM[0000\_006CH]=  0056\_453DH | ACED0001 | 1010\_1100\_1110\_1101\_0000\_0000\_0000\_0001 |
| 190H | lw $1,#0($0) | [$1] = 0000\_0008H | 8C010000 | 1000\_1100\_0000\_0001\_0000\_0000\_0000\_0000 |
| 194H | lw $2,#4($0) | [$2] = 0000\_0010H | 8C020004 | 1000\_1100\_0000\_0010\_0000\_0000\_0000\_0100 |
| 198H | lw $3,#8($0) | [$3] = 0000\_0011H | 8C030008 | 1000\_1100\_0000\_0011\_0000\_0000\_0000\_1000 |
| 19CH | lw $4,#12($0) | [$4] = 0000\_0004H | 8C04000C | 1000\_1100\_0000\_0100\_0000\_0000\_0000\_1100 |
| 1A0H | lw $5,#16($0) | [$5] = 0000\_000DH | 8C050010 | 1000\_1100\_0000\_0101\_0000\_0000\_0001\_0000 |
| 1A4H | lw $6,#24($0) | [$6] = FFFF\_FFE2H | 8C060018 | 1000\_1100\_0000\_0110\_0000\_0000\_0001\_1000 |

十分遗憾的由于时间不是太充分，只能在modelSIM上仿真，还未能实现上板子。

1. **心得体会**

这次的课程设计可谓是颇为艰辛，对于这个流水线CPU的设计，虽然书本上有这方面的内容，课上老师也讲了相关知识，但是真的要动手一步步将理论变为现实的时候，还是有些让人手足无措，只好硬着头皮上了。一开始是先是进行数据通路与控制模块的理论设计，这个借助书本上学到的知识与网络上搜集的资料，花了一段时间完了。接下来便开始进行初级流水线的设计即不考虑冒险的五段流水线CPU的设计。虽然之前已经在理论层面设计完了数据通路与控制模块，但是真的开始动手做了起来还是问题不断，对于仿真信号的分析，然后对设计反复调试，花了很久终于完成了初步的流水线CPU设计。接下来便是开始考虑冒险，因为冒险与很多模块都有关联，冒险的处理是基于已有设计的模块之上的，并不是单单添加进去几个模块就可以解决问题的，而且要考虑到很多种冒险——结构冒险，Load-use数据冒险，Branch指令数据冒险，普通数据冒险。经过几番波折调试，最终只完成了九条流水线CPU的设计，接下来是上板的实现，因为有了之前实验的基础，还有老师额外布置给我的任务——改进32位加法器为64为加法器，这一块的实现调试并没有耗费太多的精力，最终也是在老师的悉心教导下完成了课程设计，当然同时也指出了不足之处，我仍需努力去，花费时间精力去改进这些不足。

1. **参考文献：**

袁春风 《计算机系统组成与系统设计》（第二版）清华大学出版社

《深入理解计算机系统》（第三版） 机械工业出版社

《计算机组成——硬件接口与设计》 （第二版） 机械工业出版社