合肥工業大學

实 验 报 告

课 程 名 称	系统硬件综合设计
专 业 班 级	物联网工程 17-2 班
学 生 学 号	2017218007
学 生 姓 名	文 华
指 导 教 师	安 鑫、陈 田、李 建 华

目 录

1 课程设计概述

1.1 设计目的

计算机组成原理是计算机专业的核心基础课。该课程力图以“培养学生现代计算机系统设计能力”为目标，贯彻“强调软/硬件关联与协同、以 CPU 设计为核心/层次化系统设计的组织思路，有效地增强对学生的计算机系统设计与实现能力的培养”。课程设计是完成该课程并进行了多个单元实验后，综合利用所学的理论知识，并结合在单元实验中所积累的计算机部件设计和调试方法，设计出一台具有一定规模的指令系统的简单计算机系统。

课程设计属于设计型实验，不仅锻炼学生简单计算机系统的设计能力，而且通过进行中央处理器底层电路的实现、故障分析与定位、系统调试等环节的综合锻炼，进一步提高学生分析和解决问题的能力。

1.2 设计任务

本课程设计的目标总体目标是设计并实现一个满足以下条件的多周期和流水 CPU：

①若干段流水、可以处理冲突；

②三种类型的指令若干条；

③MIPS、ARM、RISC-V 等类型 CPU 都可以；

④下载到 FPGA 上进行验证（选）。

1.3 设计要求

①根据课程设计指导书的要求，制定出设计方案；

②分析指令系统格式，指令系统功能；

③根据指令系统构建基本功能部件，主要数据通路；

④根据功能部件及数据通路连接，分析所需要的控制信号以及这些控制信号的有效形式；

⑤调试、数据分析、验收检查；

⑥课程设计报告和总结。

1.4 技术指标

①支持表 1.4.1 所示的 50 条 32 位 MIPS 指令；

②支持 5 段流水机制，可处理数据冒险，结构冒险，分支冒险；

③能运行由自己所设计的指令系统构成的一段测试程序，测试程序应能涵盖所有指令，程序执行功能正确；

④运行测试程序所耗费的时钟周期数以及每种类型指令执行所需时间。

表 1.4.1 MIPS-C3 指令集

序号	指令	指令说明	指令格式
1	addi	加立即数	addi rt, rs, immediate
2	addiu	加立即数（无符号）	addiu rd, rs, immediate
3	andi	立即数与	andi rt, rs, immediate
4	ori	或立即数	ori rt, rs, immediate
5	sltiu	小于立即数置 1（无符号）	sltiu rt, rs, immediate
6	lui	立即数加载高位	lui rt, immediate
7	xori	异或（立即数）	xori rt, rs, immediate
8	slti	小于置 1（立即数）	slti rt, rs, immediate
9	addu	加（无符号）	addu rd, rs, rt
10	and	与	and rd, rs, rt
11	beq	相等时分支	beq rs, rt, offset
12	bne	不等时分支	bne rs, rt, offset
13	j	跳转	j target
14	jal	跳转并链接	jal target
15	jr	跳转至寄存器所指地址	jr rs
16	lw	取字	lw rt, offset(base)
17	xor	异或	xor rd, rs, rt
18	nor	或非	nor rd, rs, rt
19	or	或	or rd, rs, rt
20	sll	逻辑左移	sll rd, rt, sa
21	sllv	逻辑左移（位数可变）	sllv rd, rt, rs
22	sltu	小于置 1（无符号）	sltu rd, rs, rt

23	sra	算数右移	sra rd, rt, sa
24	srl	逻辑右移	srl rd, rt, sa
25	subu	减（无符号）	sub rd, rs, rt
26	sw	存字	sw rt, offset(base)
27	add	加	add rd, rs, rt
28	sub	减	sub rd, rs, rt
29	slt	小于置 1	slt rd, rs, rt
30	srlv	逻辑右移（位数可变）	srlv rd, rt, rs
31	srav	算数右移（位数可变）	srav rd, rt, rs
32	clz	前导零计数	clz rd, rs
33	divu	除（无符号）	divu rs, rt
34	eret	异常返回	eret
35	jalr	跳转至寄存器所指地址，返回地址保存在 R31 这个寄存器	jalr rs
36	lb	取字节	lb rt, offset(base)
37	lbu	取字节（无符号）	lbu rt, offset(base)
38	lhu	取半字（无符号）	lhu rt, offset(base)
39	sb	存字节	sb rt, offset(base)
40	sh	存半字	sh rt, offset(base)
41	lh	取半字	lh rt, offset(base)
42	mfhi	读 Hi 寄存器	mfhi rd
43	mflo	读 Lo 寄存器	mflo rd
44	mthi	写 Hi 寄存器	mthi rd
45	mtlo	写 Lo 寄存器	mtlo rd
46	mul	乘	mul rd, rs, rt
47	mult	字相乘	mult rs, rt
48	multu	乘（无符号）	multu rs, rt
49	bgez	大于等于 0 时分支	bgez rs, offset
50	div	除	div rs, rt

（续表）

序号	指令	OP 31-26	RS 25-21	RT 20-16	RD 15-11	SA 10-6	FUNCT 5-0	指令码 16 进制
1	addi	001000				00000	100000	20000000
2	addiu	001001						24000000
3	andi	001100						30000000
4	ori	001101						34000000

5	sltiu	001011						2C000000
6	lui	001111	00000					3C000000
7	xori	001110			00000	00000	000000	38000000
8	slti	001010			00000	00000	000000	28000000
9	addu	000000				00000	100001	00000021
10	and	000000				00000	100100	00000024
11	beq	000100						10000000
12	bne	000101						14000000
13	j	000010						08000000
14	jal	000011						0C000000
15	jr	000000					001000	00000009
16	lw	100011						8C000000
17	xor	000000				00000	100110	00000026
18	nor	000000				00000	100111	00000027
19	or	000000				00000	100101	00000025
20	sll	000000	00000				000000	00000000
21	sllv	000000				00000	000100	00000004
22	sltu	000000				00000	101011	0000002B
23	sra	000000	00000				000011	00000003
24	srl	000000	00000				000010	00000002
25	subu	000000				00000	100010	00000022

26	sw	101011						AC000000
27	add	000000				00000	100000	00000020
28	sub	000000				00000	100010	00000022
29	slt	000000				00000	101010	0000002A
30	srlv	000000				00000	000110	00000006
31	srav	000000				00000	000111	00000007
32	clz	011100				00000	100000	70000020
33	divu	000000			00000	00000	011011	0000001B
34	eret	010000	10000	00000	00000	00000	011000	42000018
35	jalr	000000		00000			001001	00000008
36	lb	100000						80000000
37	lbu	100100						90000000
38	lhu	100101						94000000
39	sb	101000						A0000000
40	sh	101001						A4000000
41	lh	100001						84000000
42	mfhi	000000	00000	00000		00000	010000	00000010
43	mflo	000000	00000	00000		00000	010010	00000012
44	mthi	000000		00000	00000	00000	010001	00000011
45	mtlo	000000		00000	00000	00000	010011	00000013
46	mul	011100				00000	000010	70000002
47	mult	000000		00000	00000		011000	00000018
48	multu	000000			00000	00000	011001	00000019

49	bgez	000001		00001				04010000
50	div	000000			00000	00000	011010	0000001A

2 总体方案设计

2.1 MIPS 体系结构

MIPS 是 RISC 处理器中的一种。MIPS 即无内部互锁流水线的微处理器（Microprocessor Without Interlocked Piped Stages），其设计机制是尽量避免处理器中流水线上各种相关问题，使得程序指令可以尽量不停顿的执行。可以说，MIPS 是为了流水线而生的，这也是本设计选择 MIPS 作为目标 CPU 架构的主要原因。接下来介绍与 MIPS 有关的一些基本概念。

2.1.1 MIPS 寄存器

MIPS 寄存器主要包括：通用寄存器、特殊寄存器、浮点寄存器。本设计实现了前两者。

①MIPS 通用寄存器

MIPS32 体系结构总共实现了 32 个 32 比特的通用寄存器，这些寄存器可供指令使用，第 0 号寄存器到第 31 号寄存器。其中，只有两个寄存器较为特殊：第 0 号通用寄存器不论指令向里面写入什么值读出的总是 0，而第 31 号通用寄存器通常被子程序调用指令用来存储返回地址。表 2.1.1 給出了 MIPS 通用寄存器的命名规则。

表 2.1.1 MIPS 通用寄存器命名规则

寄存器编号	助记符	用途
0	zero	固定为 0
1	at	编译暂存寄存器，为编译器保留
2~3	v0, v1	用来存放子程序的返回值（非浮点）
4~7	a0~a3	用来传递子程序的前四个参数（非浮点）
8~15	t0~t7，t8, t9	暂存寄存器，离开时子函数时不需要对其进行存储和恢复
16~23	s0~s7	子函数寄存变量;改变这些寄存器值的子程序必须存储旧的值并在退出前恢复，对调用程序来说值不变
26, 27	k0, k1	为中断/陷阱指令保留
28	gp	全局指针;某些对时间敏感的系统可以使用该寄存器为 static 或 extern 变量提供快速的寻址方式

29	sp	堆栈指针
30	s8/fp	子程序用它来作帧指针
31	ra	子程序返回地址

②MIPS 特殊寄存器

MIPS 体系结构定义了以下 3 个特殊寄存器：

i. PC：程序计数器，保存当前执行程序的地址；

ii. LO：保存乘法指令低位结果或除法指令的商；

iii. HI：保存乘法指令高位结果或除法指令的余数。

③MIPS 浮点寄存器

MIPS 体系结构还定义了以下 3 种浮点寄存器：

i. 32 个 32 比特的浮点寄存器(FPRs)；

ii. 5 个 FPU 控制寄存器用于识别和控制 FPU；

iii. 8 个浮点条件代码，它们是 FCSR 寄存器中的一部分。

浮点寄存器的实现比较复杂且不在本次课程设计的任务要求中，故本设计没有对其进行实现。

2.1.2 MIPS 指令集

每一条 MIPS 指令的长度是 32 位。MIPS 指令集包括三种类型的指令：寄存器类型指令，立即数类型指令和跳转类型指令 1B7。寄存器类型指令只有通用寄存器之间的操作，不包含立即数；立即数类型指令是通用寄存器和 16 位立即数之间的操作；而跳转类型指令包含一个 26 位的指令地址索引，可以进行大范围绝对地址跳转。MIPS 架构这种精简指令集处理器在很大程度上减少指令数目并且极大的简化了指令的译码和执行，程序编译器还可以利用若干个简单的指令合理的组合，形成更加复杂的操作。如图 2.1.2.1 所示为 MIPS 处理器的指令格式，表 2.1.2.1 是对指令中各个域的说明。

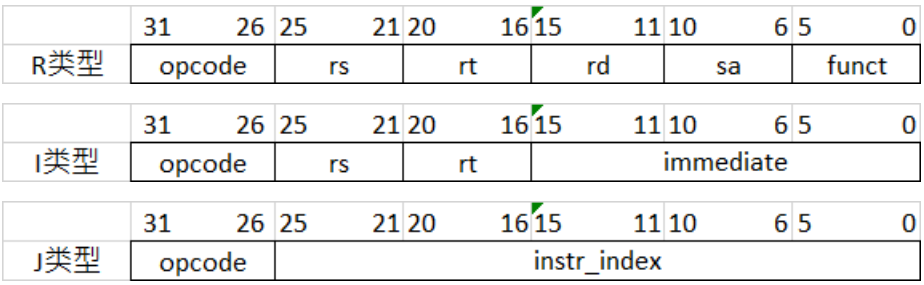


图 2.1.2.1 MIPS 指令格式

表 2.1.2 MIPS 指令域说明

域	说明
opcode	6 位指令主操作码
rs	5 位源寄存器索引
rt	5 位源/目的寄存器索引
rd	5 位目的寄存器索引
sa	5 位移位偏移量
funct	6 位函数功能码，用来指示寄存器指令的具体功能
immediate	16 位的立即数，用做逻辑指令操作数、算术指令操作数、存储器访问
instr_index	指令的地址偏移量和分支指令的跳转目的偏移量 26 位的跳转索引，指示跳转指令的索引

MIPS32 指令分为如下几类：

①存储器访问指令：用于在存储器和通用寄存器之间传递数据。访问存储器的地址是通用寄存器的值和 16 位有符号立即数相加的值，存储器访问指令都属于立即数类型指令。

②运算指令：完成定点的算术、逻辑、移位等运算操作。操作数据既可以是通用寄存器的值，也可以是 16 位立即数。运算指令可以是寄存器类型或立即数类型。

③分支跳转指令：可以改变程序指令的顺序执行。分支指令用 PC 寄存器的值加上 16 位立即数左移 2 位指向目的地址。跳转指令把指令中程序索引或者某个通用寄存器的内容写入 PC。分支指令属于立即数类型，跳转指令可以是寄存器类型或跳转类型。

④系统协处理器指令：进行系统协处理器寄存器的读写操作。

2.1.3 MIPS 五级流水线

流水线是指将计算机指令处理过程拆分为多个步骤，并通过多个硬件处理单元并行执行来加快指令执行速度。本设计的目标是 MIPS 五级流水线，其各个阶段的主要工作如下：

取指阶段（Instruction Fetch，IF）：从指令存储器读出指令，同时确定下一条指令地址。

译码阶段（Instruction Decode，ID）：对指令进行译码，从通用寄存器中读出要使用的寄存器的值，如果指令中含有立即数，那么还要将立即数进行符号扩展或无符号扩展。如果是转移指令，并且满足转移条件，那么给出转移目标，作为新的指令地址。

执行阶段（Execute，EX）：按照译码阶段给出的操作数、运算类型，进行运算，给出运算结果。如果是 Load/Store 指令，那么还会计算 Load/Store 的目标地址。

访存阶段（Memory，MEM）：如果是 Load/Store 指令，那么在此阶段会访问数据存储器，反之，只是将执行阶段的结果向下传递到回写阶段。同时，在此阶段还要判断是否有异常需要处理，如果有，那么会清除流水线，然后转移到异常处理例程入口地址处继续执行

回写阶段（Write Back，WB）：将运算结果保存到目标寄存器。

除上述内容外，MIPS 架构还包括：MIPS 协处理器 0 以及 MIPS 中断与异常等，因本设计并未涉及相关内容，故不作介绍。

2.2 流水 CPU 设计

2.2.1 总体设计

本设计拟实现的 CPU 是全冒险处理的五阶段 CPU，下面对需要解决的问题进行简单地描述。

单周期时即使指令之后的执行不再需要使用前面用到的部件，还仍然霸占着那些部件。合理而高效的做法自然是让指令执行时不再需要使用前面的部件时，应该调入下一条指令执行。所以关键问题是：如何将单周期数据通路进行划分，当指令未执行完而新的指令已经调入时必须使用寄存器保存当前指令的执行信息，如何设置流水寄存器内容。此外，程序在流水线上的执行还会带来一些“流水线专属”的问题，如下所示。

i1. R2 ← R1 + R3

i2. R4 ← R2 + R3

第一条指令是计算一个值，并将其储存在 R2，而第二个指令是使用这个值计算结果并储存在 R4，但是在我们拿出第二步的操作数时，第一步的结果还未被储存。在如此情况下，得到的运算结果是不可预知的。这个问题即是所谓的“数据相关”，又称“数据冒险”，一般发生在指令乱序执行时，可能会发生读取数据与写入数据之间的时序与空间的相关性。数据相关又分三种情况：RAW、WAR、WAW。数据相关可以采用重定向的方法解决，真数据相关也就是 RAW，只需要检测相邻指令读寄存器编号与写寄存器编号一致时就采用定向技术（旁路)，将正确的数据（还未来得及写回）通过新增的数据通路送到需要读该寄存器的地方。有一种特殊的数据相关：load-use 冲突，并不能简单地重定向，有关冲突必需在 ID 阶段监测，否则会造成额外的硬件开销。除了数据相关，流水线中还存在结构相关与分支相关，前者指的是指的是在指令执行的过程中，由于硬件资源满足不了指令执行的要求，发生硬件资源冲突而产生的相关。比如：指令和数据都共享一个存储器，在某个时钟周期，流水线既要完成某条指令对存储器中数据的访问操作，又要完成后续的取指令操作，这样就会发生存储器访问冲突，产生结构相关。解决结构相关可以考虑采用哈佛结构：指令存储器和数据存储器分开（实际商用的处理器是分开的指令 cache 与数据 cache）；后者指的是流水线中的分支指令或者其他需要改写 PC 的指令造成的相关。最后介绍分支相关的解决办法，本设计采用静态分支预测中的预测失败的方法，当监测到分支指令时并不做特别的处理，等到分支成功或者失败的结果计算出来时，若是分支失败则分支指令和普通指令没有任何区别，一旦分支成功，就要清楚误取深度条指令，从新从分支地址出取出正确的指令。当解决上述所有问题后，本设计就实现了全冒险处理的五段流水 CPU。

2.2.2 流水接口部件设计

如 2.1.3 节所述，在 MIPS 五段流水 CPU 上，指令的解释执行可以分为 IF、ID、EX、MEM 与 WB 五个阶段，且这五个阶段在不同的功能部件上执行，故可使五个子任务并发执行。流水寄存器中应该包含指令在后续解释执行阶段所需的所有信号，并且为了便于调试最好将 PCPlus4、IR 信号也全部传递。流水寄存器中传递的信号及在各段的使用情况如图 2.2.2.1 所示。从图 2.2.2.1 上可以看到：IF/ID 寄存器中保存着 PCPlus4 和 IR 信号

也全部传递信号，而 ID/EX 寄存器中包含着指令在 EX 执行所需要的全部信号以及指令在 MEM 和 WB 阶段需要的所有信号，EX/MEM 寄存器包含着指令在 MEM 阶段执行所需要的全部信号以及指令在 WB 段执行的全部信号，MEM/WB 寄存器则包含着指令在 WB 段执行所需要的全部信号，因此流水寄存器不仅需要包含指令在本段执行所需要的信号，还要将指令在后续阶段所需要的信号逐步传递给后续流水寄存器（相当于实现了控制器的复用。考虑到后面要实现流水线的停顿和指令的清空，流水寄存器中还需有使能信号和清零信号。

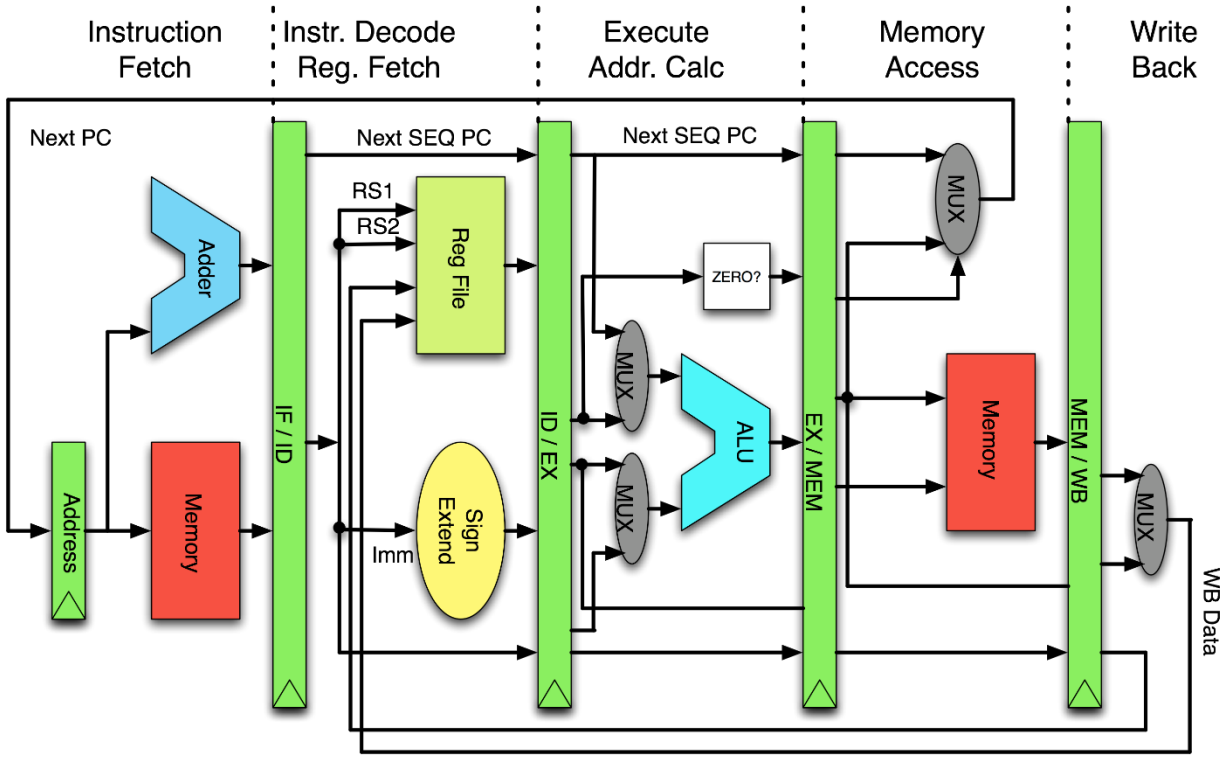


图 2.2.2.1 流水寄接口设计及信号传递示意

2.3 数据转发流水线设计

数据转发技术和重定向、旁路等所指的概念是一样的，是为了解决流水线中存在的 RAW 冲突。后面指令读寄存器编号和前面指令写寄存器编号相同时就会出现真数据相关，该问题存在的原因是执行结果未写回寄存器文件，后续指令就需要从寄存器文件中读取还未来得及写入的寄存器的值，但实际上这些需要写入的值都已经在数据通路中计算出来了，只是没有写回而已，故本设计的解决方法是：通过构建额外的数据旁路来将这些数

据送到需要使用它们的地方，其原理如图 2.3.1。

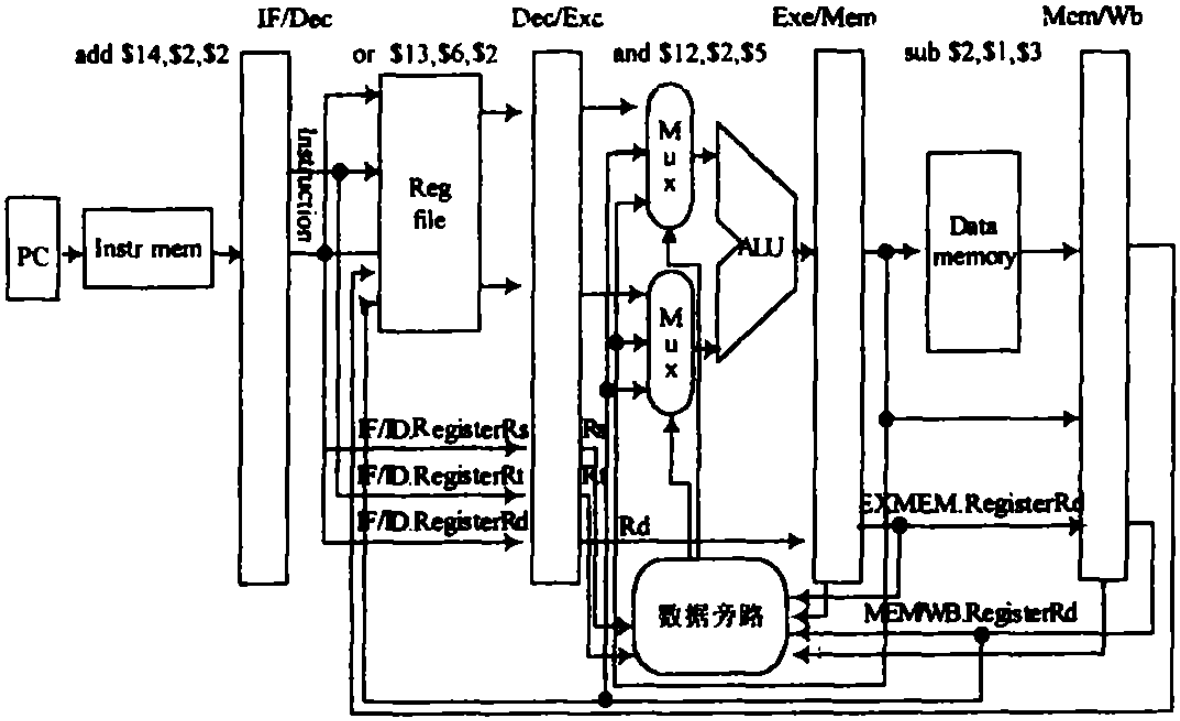


图 2.3.1 数据旁路示意

发生数据相关单元应该检测 EX 段读的寄存器编号（最多可能是两个)，与 MEM 和 WB 段指令写寄存器编号是否出现相同的情况，若出现，则数据冲突发生，所以需要将数据送到 EX 段对应位置，如果编号不相同，则正常执行，无需重定向。此处有两个问题需要注意，当读、写 0 号寄存器时，是不需要重定向的，因 0 号寄存器内容始终为 0，一定不会出现数据冲突。如果读寄存器编号与 MEM 和 WB 段写寄存器编号均相同，应该重定向那个段的数据，显示应该重定向 MEM 段的的数据，因此 MEM 段的数据重定向具有更高的优先级，在实现的时候只需将输入端为 MEM 段数据的二路选择器放在后面即可。当上述问题解决后，数据转发流水线也就完成了，重定向具体实现如图 2.3.2 所示，流水寄存器信号在每段流转情况如图 2.3.3 所示。

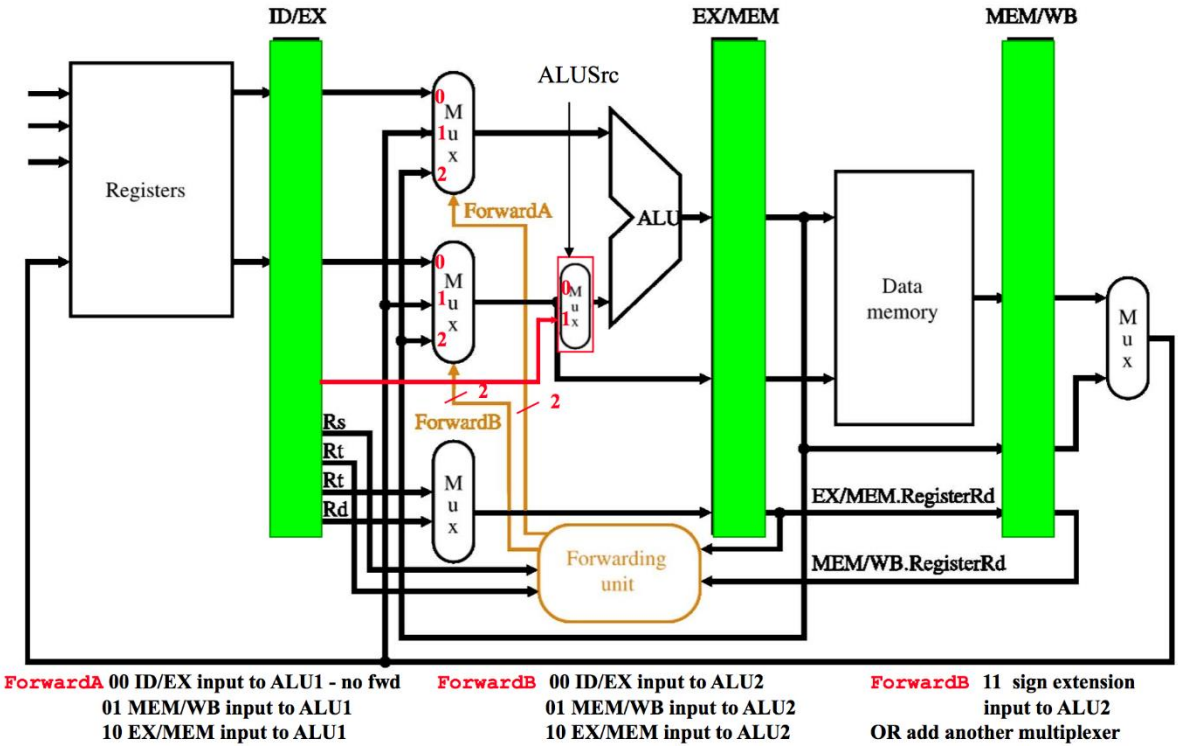


图 2.3.2 流水寄接口设计与重定向实现

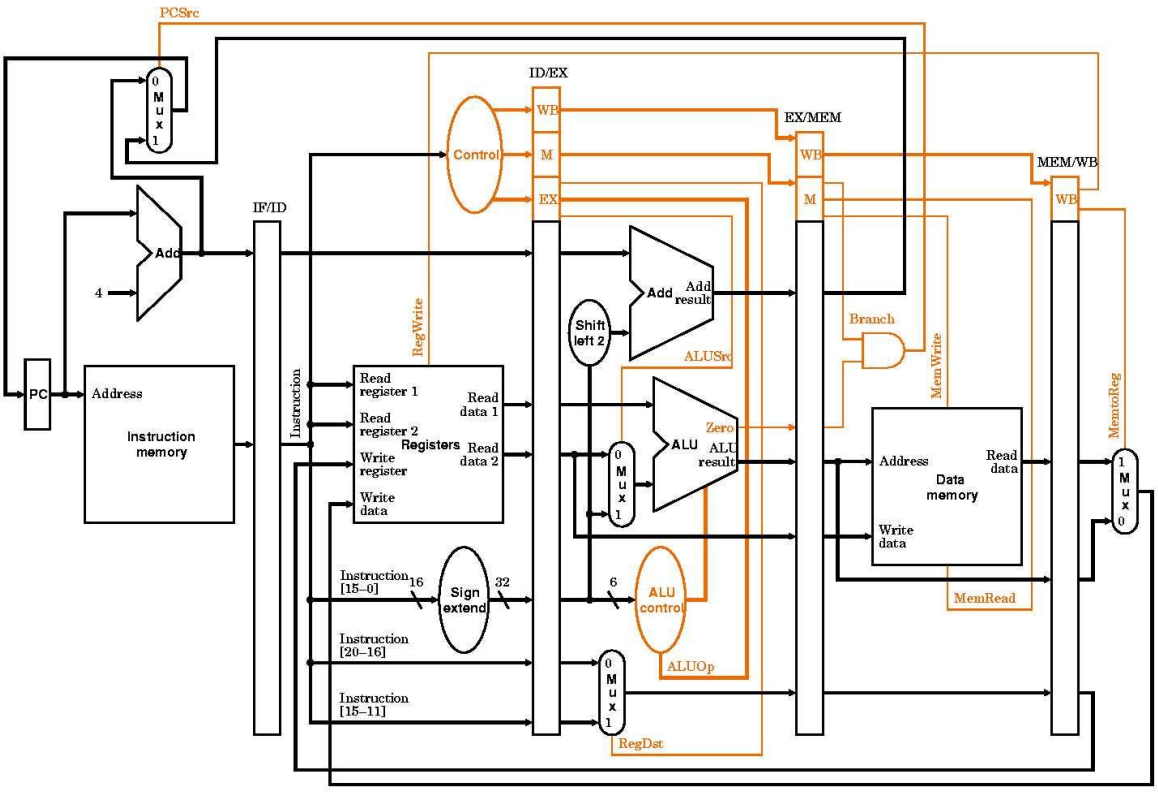


图 2.3.3 流水线寄存器信号在每段流转情况

2.4 气泡式流水线设计

这一节讨论 load-use 相关的问题。当连续两条指令依次执行，其中前一条是 lw 指令，而后面的一条指令的源寄存器编号与 lw 指令的目标寄存器编号相同时，会发生 load-use 冲突（RAW 相关），如图 2.4.1 所示。此时，EX 段的时延将是访问与 ALU 运算串行进行的时间，若在这种情况下仍然直接对指令进行重定向，在不发生错误的前提下，将会造成 CPU 频率降低，而这对于某些工作环境下的 CPU 是无法接受的，故本设计需要使用其它方法来解决这个问题。

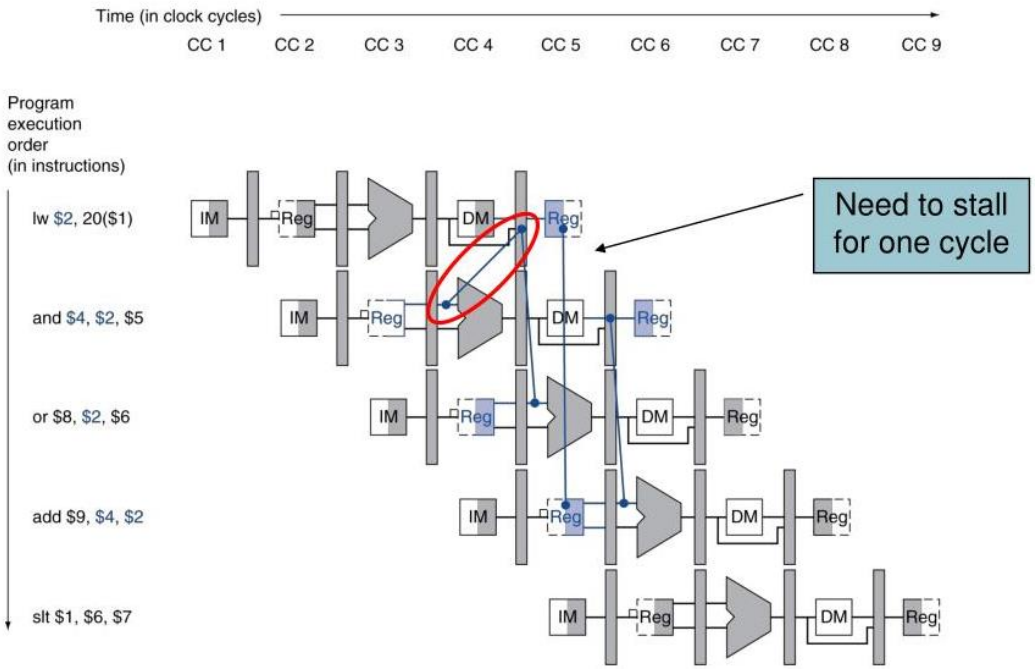


图 2.4.1 load-use 冲突示意图

为了解决 load-use 冲突，可以采用以下方法：当指令流水处于 ID 阶段时，需要插入气泡并暂停流水线，在重定向的检测上外加判定需要重定向的指令的前一条指令是否为 lw 指令即可。否则，将必须以增加硬件开销为基础来保证程序不出错。原因如下所述。

假设在 EX 段检测，考虑以下情况：MEM 段和 WB 段执行的指令都为 lw 指令，且两条 lw 指令的目的寄存器编号正好与 EX 段指令源寄存器编号相同，若插入一个气泡，WB 段 lw 指令将被“挤走”，重定向的数据被丢失，而 EX 段指令又未从寄存器文件中取得正确的源寄存器值，因此就造成了程序执行可能出现错误。考虑在 ID 段检测 load-use 冲突，若在 EX 段与 MEM 段的指令出现与在 EX 段检测 load-use 冲突相同的情况，插入一个气泡并监测 ID 段及 IF 段流水线，那么最先的 lw 指令将会流动到 WB 段，将寄存器文件改造了为下降沿写入，故下一个时钟上升沿来临时，正确的源寄存器值将被锁入 ID/EX 流水寄存器中。

综上，load-use 冲突需要插入气泡并暂停流水线，且关于 load-use 冲突的检测必须在 ID 段。如图 2.4.2 所示为气泡流水线原理图。

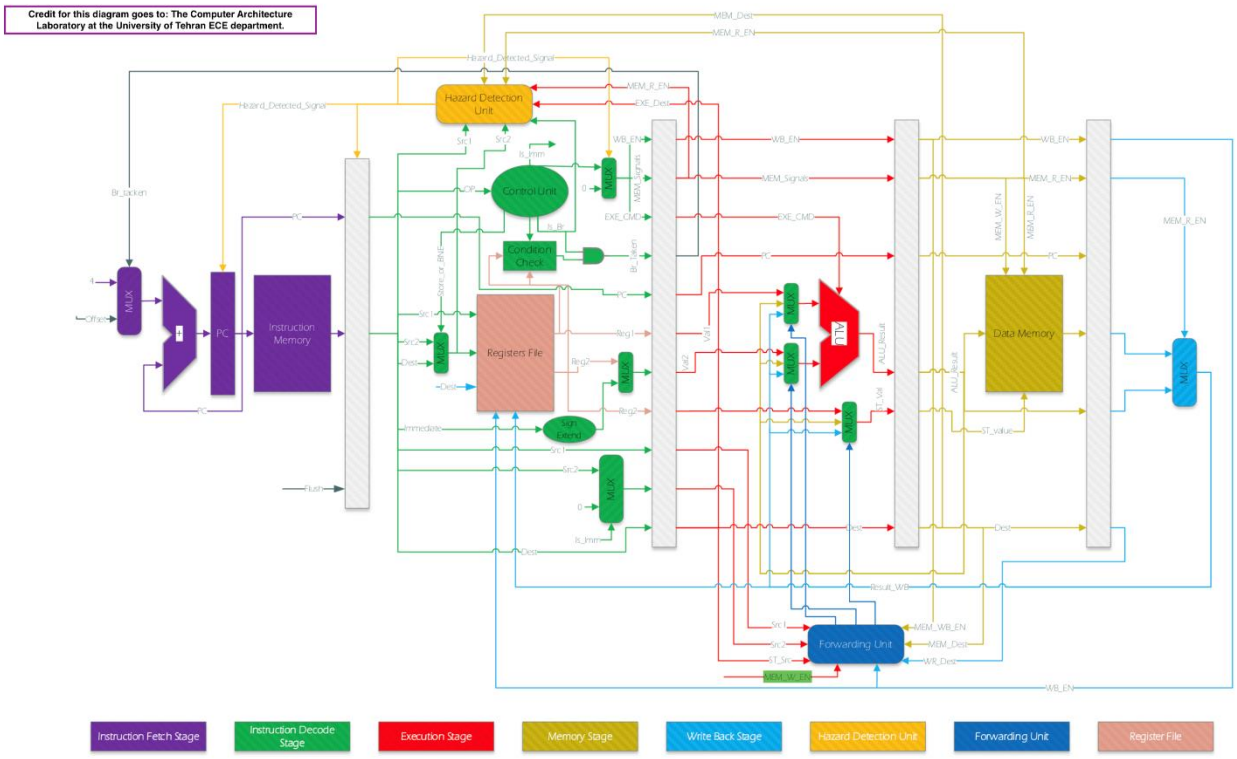


图 2.4.2 气泡流水线原理图

3　详细设计与实现

3.1　流水 CPU 实现

3.1.1　流水接口部件实现

①流水线部件内部接口

所谓的流水接口部件即是流水寄存器，由于本设计拟构建的是经典五段流水线，故共需要设置 4 个流水寄存器：IF/ID、ID/EX、EX/MEM、MEM/WB。这 4 个流水寄存器中应包含哪些信号用于向下一级寄存器传递，已在 2.2.2 节介绍过，在此不赘述，仅給出各个寄存器的内部接口图片。

如图 3.1.1.1 所示为 IF/ID 段流水寄存器内部接口。

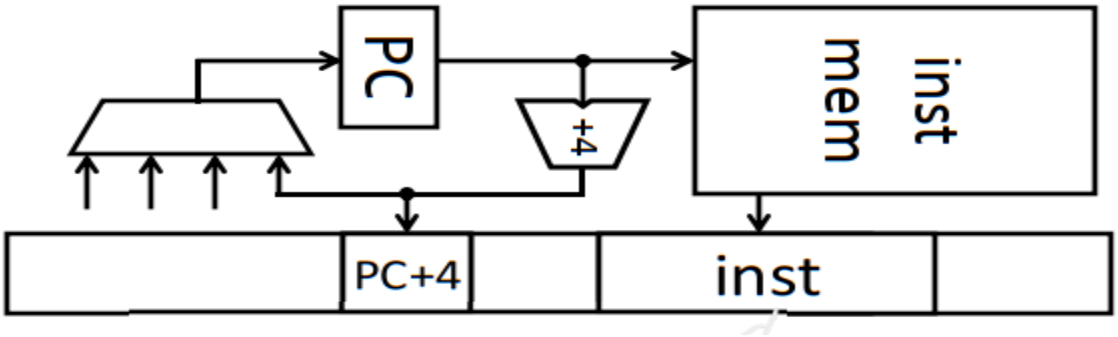


图 3.1.1.1 IF/ID 段流水寄存器内部

如图 3.1.1.2 所示为 ID/EX 段流水寄存器内部接口。

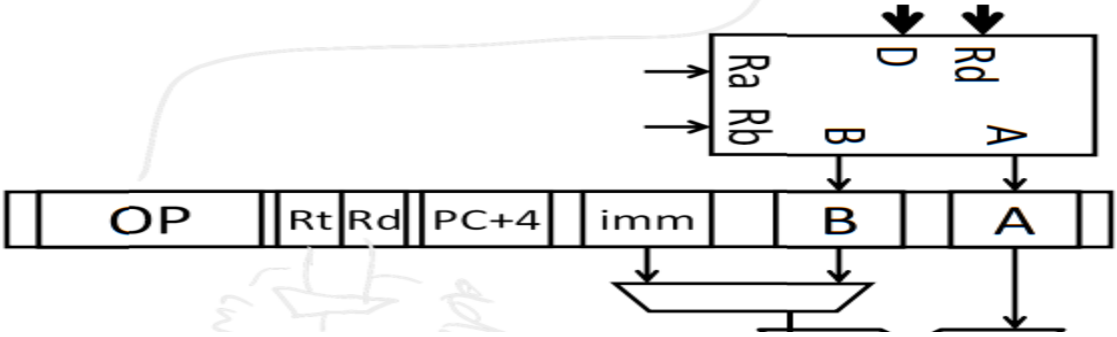


图 3.1.1.2 ID/EX 段流水寄存器内部

如图 3.1.1.3 所示为 EX/MEM 段流水寄存器内部接口。

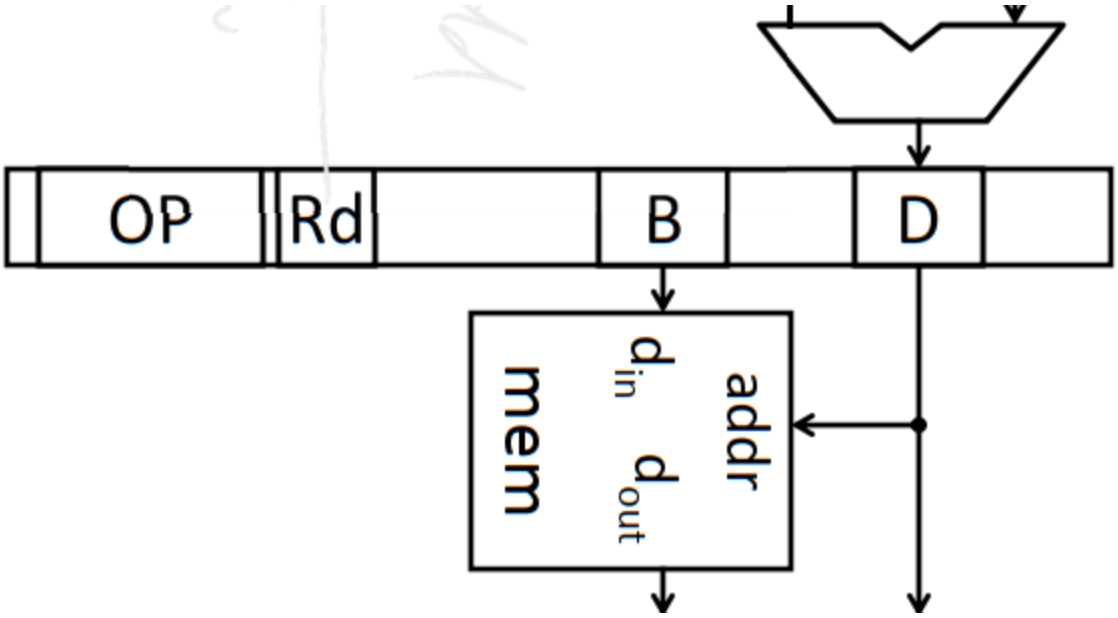


图 3.1.1.3 EX/MEM 段流水寄存器内部

如图 3.1.1.4 所示为 MEM/WB 段流水寄存器内部接口。

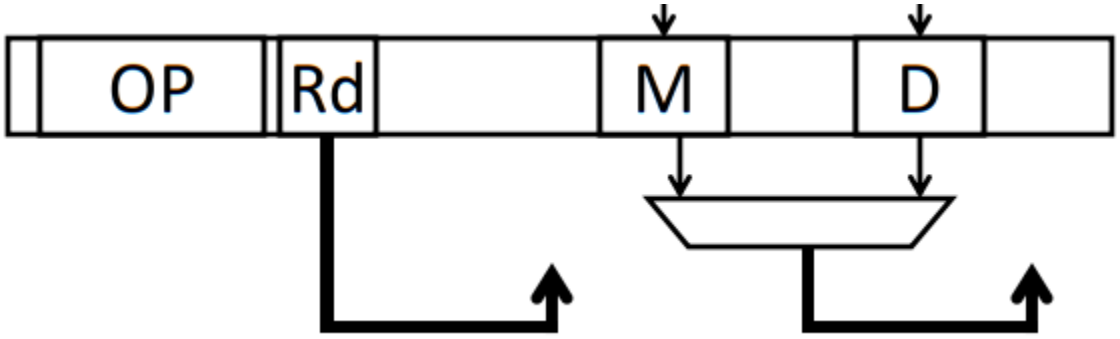


图 3.1.1.4 MEM/WB 段流水寄存器内部接口

②流水线部件 Verilog 实现

可以说，4 个流水线寄存器的工作原理与实现方式大同小异，只是向其下一级传递的信号略有不同，此处仅給出 IF/ID 段流水寄存器的 Verilog 实现。

```
module RegD(                                                    //IF/ID 阶段的寄存器，根据 IF 的指令与 PC 确定 ID 时其对应值
  input             clk, rst, En,
  input      [31:0]  InstrF, PCPlus4F,          //IF（取指令）周期 时的指令 / 下一条指令的 PC 处
  output  reg [31:0]  InstrD, PCPlus4D                 //ID（指令译码）周期 时的指令 / 下一条指令的 PC 处
```

```
  );

  initial
  begin
    InstrD <= 0;
    PCPlus4D <= 0;
  end

  always @(posedge clk)
    if (rst)
    begin
      InstrD <= 0;
      PCPlus4D <= 0;
    end
    else if (En)
    begin
      InstrD <= InstrF;
      PCPlus4D <= PCPlus4F;
    end
    else
    begin
      InstrD <= InstrD;
      PCPlus4D <= PCPlus4D;
    end

endmodule
```

3.1.2 数据通路实现

本设计所构建五段流水线 CPU 的数据通路如图 3.1.2.1 所示。

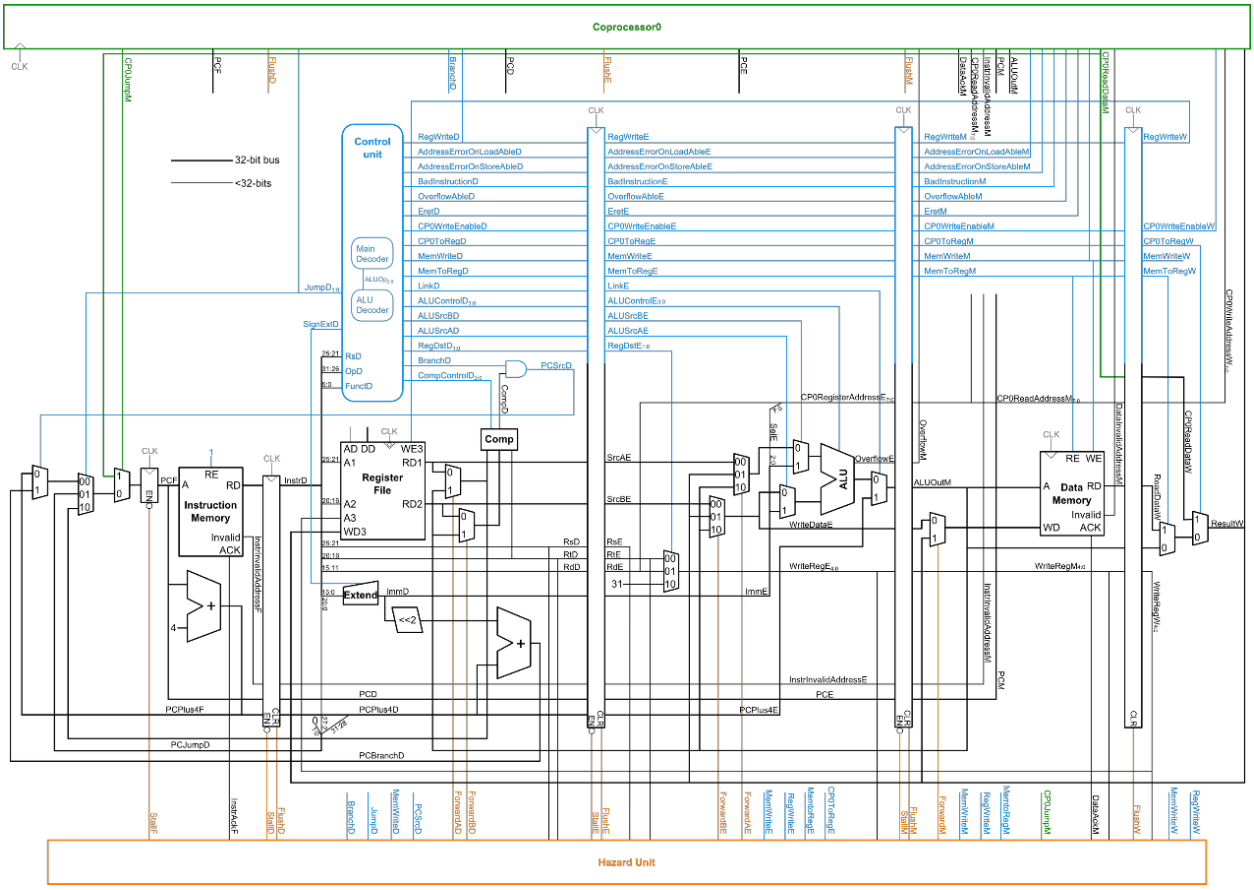


图 3.1.2.1 五段流水线 CPU 数据通路

3.1.3 五段流水线寄存器模块

根据数据通路，可以自然地得到 Verilog 实现的各个模块，下面給出本设计所构建五段流水线 CPU 的有关模块的定义接口。各个模块的接口的作用大同小异，不一一注释。

如表 3.1.3.1 所示为 RegD 模块的定义接口。

表 3.1.1 RegD 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	clk	1	输入	
2	rst	1	输入	
3	En	1	输入	
4	InstrF	32	输入	IF（取指令）周期 时的指令
5	PCPlus4F	32	输入	IF（取指令）周期 时下一条指令的 PC 处
6	InstrD	32	输出	ID（指令译码）周期 时的指令
7	PCPlus4D	32	输出	ID（取指令）周期 时下一条指令的 PC 处

如表 3.1.3.2 所示为 RegE 模块的定义接口。

表 3.1.2 RegE 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	clk	1	输入	
2	rst	1	输入	
3	FlushE	1	输入	
4	PCPlus4D	32	输入	
5	SrcA2D	32	输入	
6	SrcB2D	32	输入	
7	RsD	5	输入	
8	RtD	5	输入	
9	RdD	5	输入	
10	ShamtD	5	输入	
11	ImmD	32	输入	
12	RegWriteD	1	输入	
13	RegDstD	1	输入	
14	MemWriteD	1	输入	
15	MemToRegD	1	输入	
16	ALUControlD	4	输入	
17	ALUSrcD	2	输入	
18	StartD	1	输入	
19	IsJJalD	1	输入	
20	IsJrJalrD	1	输入	
21	IsLbSbD	1	输入	
22	IsLhShD	1	输入	
23	IsUnsignedD	1	输入	
24	HiLoWriteD	1	输入	
25	HiLoD	1	输入	
26	IsShamtD	1	输入	
27	MdOpD	2	输入	
28	PCPlus8E	32	输出	
29	SrcA2E	32	输出	
30	SrcB2E	32	输出	
31	RsE	4	输出	
32	RtE	4	输出	
33	RdE	4	输出	
34	ShamtE	4	输出	
35	ImmE	32	输出	

36	RegWriteE	1	输出	
37	RegDstE	1	输出	
38	MemWriteE	1	输出	
39	MemToRegE	1	输出	
40	ALUControlE	4	输出	
41	ALUSrcE	2	输出	
42	StartE	1	输出	
43	IsJJalE	1	输出	
44	IsJrJalrE	1	输出	
45	IsLbSbE	1	输出	
46	IsLhShE	1	输出	
47	IsUnsignedE	1	输出	
48	HiLoWriteE	1	输出	
49	HiLoE	1	输出	
50	IsShamtE	1	输出	
51	MdOpE	2	输出	

如表 3.1.3.3 所示为 RegM 模块的定义接口。

表 3.1.3 RegM 模块的定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	clk	1	输入	
2	rst	1	输入	
3	PCPlus8E	32	输入	
4	ALUOutE	32	输入	
5	WriteDataE	32	输入	
6	RtE	5	输入	
7	WriteRegE	5	输入	
8	RegWriteE	1	输入	
9	MemWriteE	1	输入	
10	MemToRegE	1	输入	
11	IsJJalE	1	输入	
12	IsJrJalrE	1	输入	
13	IsLbSbE	1	输入	
14	IsLhShE	1	输入	
15	IsUnsignedE	1	输入	
16	PCPlus8M	32	输出	
17	ALUOutM	32	输出	
18	WriteDataM	32	输出	
19	RtM	5	输出	

20	WriteRegM	5	输出	
21	RegWriteM	1	输出	
22	MemWriteM	1	输出	
23	MemToRegM	1	输出	
24	IsJJalM	1	输出	
25	IsJrJalrM	1	输出	
26	IsLbSbM	1	输出	
27	IsLhShM	1	输出	
28	IsUnsignedM	1	输出	

如表 3.1.3.4 所示为 RegW 模块的定义接口。

表 3.1.4 RegW 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	clk	1	输入	
2	rst	1	输入	
3	PCPlus8M	32	输入	
4	ALUOutM	32	输入	
5	ReadDataM	32	输入	
6	WriteRegM	5	输入	
7	RegWriteM	1	输入	
8	MemToRegM	1	输入	
9	IsJJalM	1	输入	
10	IsJrJalrM	1	输入	
11	IsUnsignedM	1	输入	
12	BEOutM	4	输入	
13	PCPlus8W	32	输出	
14	ALUOutW	32	输出	
15	ReadDataW	32	输出	
16	WriteRegW	5	输出	
17	RegWriteW	1	输出	
18	MemToRegW	1	输出	
19	IsJJalW	1	输出	
20	IsJrJalrW	1	输出	
21	IsUnsignedW	1	输出	
22	BEOutW	4	输出	

如表 3.1.3.5 所示为 RegFile 模块的定义接口。

表 3.1.5 RegFile 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	clk	1	输入	
2	rst	1	输入	
3	WE3	1	输入	
4	A1	5	输入	第 一个读寄存器地址
5	A2	5	输入	第 二个读寄存器地址
6	A3	5	输入	第 三 个读寄存器地址
7	WD3	32	输入	要写入的数据
8	SrcAD	32	输入	从寄存器读出的数据
9	SrcBD	32	输出	

3.2 数据转发流水线实现

3.2.1 数据冲突检测

如 2.3 节所述，对于非 load-use 的 RAW 真数据相关，可采用重定向技术避免流水线暂停直待数据写回。

本设计使用 6 路并发比较进行 RAW 相关检测，其中 4 路是为了检测 ID 段读寄存器（最多两个）的编号和 EX 段、MEM 段写寄存器编号是否相同，另外 2 路并发比较是如果 ID 段读寄存器编号为 0 的话，即使检测到与前面两条指令的写寄存器编号相同也不必重定向，因为 0 号寄存器内容永远为 0。

本设计使用 Verilog 实现数据冲突检测如下。

```
assign ForwardAD = ((RsD == WriteRegM) && RegWriteM && (IsJJalM || IsJrJalrM) &&
(WriteRegM != 5'b00000)) ? 2'b11 :
                   ((RsD == WriteRegM) && RegWriteM && (WriteRegM != 5'b00000)) ? 2'b01 :
                   ((RsD == WriteRegW) && RegWriteW && (WriteRegW != 5'b00000)) ? 2'b10 :
2'b00 ;
 assign ForwardBD = ((RtD == WriteRegM) && RegWriteM && (WriteRegM != 5'b00000)) ?
2'b01 :
                   ((RtD == WriteRegW) && RegWriteW && (WriteRegW != 5'b00000)) ? 2'b10 :
2'b00 ;
 assign ForwardAE = ((RsE == WriteRegM) && RegWriteM && (WriteRegM != 5'b00000)) ?
2'b01 :
                   ((RsE == WriteRegW) && RegWriteW && (WriteRegW != 5'b00000)) ? 2'b10 :
2'b00 ;
 assign ForwardBE = ((RtE == WriteRegM) && RegWriteM && (WriteRegM != 5'b00000)) ?
2'b01 :
                   ((RtE == WriteRegW) && RegWriteW && (WriteRegW != 5'b00000)) ? 2'b10 :
```

```
2'b00 ;
  assign ForwardM  = ((RtM == WriteRegW) && RegWriteW && (WriteRegW != 5'b00000)) ?
1 : 0;
```

3.2.2 数据旁路

在上一节中，本设计构建数据冲突检测机制得到了重定向信号，仅有信号是不足以完成重定向的工作，还必须增加新的数据旁路才能实现重定向：通过加入 4 个二路选择器将 MEM 段和 WB 段数据在发生数据冒险时通过旁路送到对应位置。

本设计使用 Verilog 实现数据重定向如下。

```
module MUX2 #(parameter WIDTH = 32)
          (input  [WIDTH-1:0] d0, d1,                                    //双路选择器
           input              s,
           output [WIDTH-1:0] y);

  assign y = s ? d1 : d0;

endmodule

module MUX3 #(parameter WIDTH = 32)                                      //三路
选择器
          (input  [WIDTH-1:0] d0, d1, d2,
           input  [1:0]       s,
           output [WIDTH-1:0] y);

  assign y = s[1] ? d2 : (s[0] ? d1 : d0);

endmodule

module MUX4 #(parameter WIDTH = 32)                                      //四路
选择器
          (input  [WIDTH-1:0] d0, d1, d2, d3,
           input  [1:0]       s,
           output [WIDTH-1:0] y);

  assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0);

endmodule
```

3.3 气泡式流水线实现

最后，介绍如何实现解决 load-use 冲突与分支冲突。

load-use 冲突必须在 ID 段检测，具体操作是：在 EX 段插入一个气泡，后暂停 IF 和 ID 段流水线，如此便能使 lw 指令访存结束后在 WB 段进行重定向。

对于分支指令，本设计采用的是静态预测失败的方式进行处理：总是假定分支失败，在 EX 段得到分支结果，若分支失败，则非顺序代码被加载执行；若分支成功，则需要清除误取的指令并从分支目标处重新取值执行。

本设计使用 Verilog 实现起泡式流水如下。

```
assign NPCD = (IsJJalD)          ? {PCPlus4D[31:28], ImmD_26, 2'b00} :
              (IsJrJalrD)        ? Reg_32 :
              (BranchD && CompD)  ? PCPlus4D + ImmD_32 : PCPlus4D + 4;
assign IsJBrD = ((BranchD && CompD) || IsJJalD || IsJrJalrD);
```

至此，本设计已经实现了全冒险处理机制的五段流水 CPU。为了便于介绍下一步的工作，此处給出本设计使用所构建全冒险处理机制的五段流水 CPU 的 Verilog 实现的模块接口，部分模块的定义接口已在 3.1.3 节展示，此处不赘述。

如表 3.3.1 所示为 ALU 模块的定义接口。

表 3.3.1 ALU 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	A	32	输入	操作数 A
2	B	32	输入	操作数 B
3	ALUControlE	4	输入	算术或逻辑指令操作符
4	ALUOutE	32	输出	算术或逻辑输出

如表 3.3.2 所示为 BECtrl 模块的定义接口。

表 3.3.2 BECtrl 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	ByteOffset	2	输入	DM 低二位地址
2	IsLhShM	1	输入	
3	IsLbSbM	1	输入	
4	BEOutM	4	输出	DM 字节写入控制信号

如表 3.3.3 所示为 Comp 模块的定义接口。

表 3.3.3 Comp 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	A	32	输入	操作数 A
2	B	32	输入	操作数 B
3	CompOpD	3	输入	比较运算符

4	CompD	1	输出	比较结果

如表 3.3.4 所示为 Ctrl 模块的定义接口。

表 3.3.4 Ctrl 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	OpD	6	输入	输入指令的 OPCode，指定是哪种寄存器运算
2	FunctD	6	输入	输入指令的 FunctCode，32 位指令的 FunctD 段
3	RtD	5	输入	写回寄存器的数据流的选择信号
4	RegWriteD	1	输出	指令译码阶段（ID）寄存器写输出
5	MemWriteD	1	输出	指令译码（ID）存储器写
6	MemToRegD	1	输出	指令译码阶段（ID）存储器向寄存器
7	RegDstD	1	输出	指令译码阶段（ID）写回寄存器的地址的选择信号
8	BranchD	1	输出	分支指令
9	IsJJalD	1	输出	跳转指令
10	IsJrJalrD	1	输出	寄存器跳转指令
11	IsLbSbD	1	输出	
12	IsLhShD	1	输出	
13	IsUnsignedD	1	输出	
14	HiLoWriteD	1	输出	
15	HiLoD	1	输出	
16	IsMdD	1	输出	
17	IsShamtD	1	输出	
18	MdOpD	2	输出	
19	ALUControlD	4	输出	
20	ALUSrcD	2	输出	
21	ExtOpD	2	输出	
22	CompOpD	3	输出	

如表 3.3.5 所示为 DM_4k 模块的定义接口。

表 3.3.5 DM_4k 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	A	10	输入	地址
2	WD	32	输入	写入的数据
3	clk	1	输入	时钟信号
4	WE	1	输入	数据存储器写使能
5	BE	4	输入	写回的位置
6	RD	32	输出	写回输出

如表 3.3.6 所示为 DMExt 模块的定义接口。

表 3.3.6 DMExt 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	ReadDataW	32	输入	读取即将写的 32 位数据
2	BEOutW	4	输入	指定读取数据
3	IsUnsignedW	1	输入	是否无符号数
4	DMExtOutW	32	输出	扩展后的 32 位数

如表 3.3.7 所示为 Ext 模块的定义接口。

表 3.3.7 Ext 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	ImmD_16	16	输入	无符号扩展
2	ExtOpD	2	输入	扩展类型
3	ImmD_32	32	输出	立即数

如表 3.3.8 所示为 Hazard 模块的定义接口。

表 3.3.8 Hazard 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	OpD	6	输入	读寄存器周期时的操作数
2	RsD	5	输入	ID / EX 阶段的地址
3	RtD	5	输入	
4	RsE	5	输入	
5	RtE	5	输入	
6	RtM	5	输入	
7	WriteRegE	5	输入	EX / MEM / WB 阶段的地址
8	WriteRegM	5	输入	
9	WriteRegW	5	输入	
10	ALUSrcD	2	输入	
11	IsJrJalrD	1	输入	
12	BranchD	1	输入	
13	IsMdD	1	输入	
14	BusyE	1	输入	
15	IsJJalM	1	输入	
16	IsJrJalrM	1	输入	
17	MemToRegE	1	输入	
18	MemToRegM	1	输入	
19	MemWriteM	1	输入	

20	RegWriteE	1	输入	
21	RegWriteM	1	输入	
22	RegWriteW	1	输入	
23	StallF	1	输出	IF 阶段访存阶段是否暂停
24	StallD	1	输出	ID 阶段访存阶段是否暂停
25	FlushE	1	输出	EX 阶段是否清除流水线
26	ForwardAD	2	输出	
27	ForwardBD	2	输出	
28	ForwardAE	2	输出	
29	ForwardBE	2	输出	
30	ForwardM	1	输出	

如表 3.3.9 所示为 IM_2k 模块的定义接口。

表 3.3.9 IM_2k 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	A	9	输入	地址
2	RD	32	输出	指令

如表 3.3.10 所示为 MD 模块的定义接口。

表 3.3.10 MD 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	clk	1	输入	时钟信号
2	rst	2	输入	复位信号
3	D1	32	输入	操作数 D1
4	D2	32	输入	操作数 D2
5	WE	1	输入	写使能
6	HiLo	1	输入	HI/LO 写使能
7	Start	1	输入	运行使能
8	Op	2	输入	操作符
9	Busy	1	输出	标记是否忙
10	Hi	32	输出	HI 寄存器输出
11	Lo	32	输出	LO 寄存器输出

如表 3.3.11 所示为 MIPS 模块的定义接口。

表 3.3.11 MIPS 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	clk	1	输入	时钟信号
2	rst	2	输入	复位信号

如表 3.3.12 所示为 MUX2 模块的定义接口。

表 3.3.12 MUX2 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	d0	32	输入	数据或地址
2	d1	32	输入	
3	s	1	输入	非冲突信号
4	y	32	输出	非冲突的数据或地址

如表 3.3.13 所示为 MUX3 模块的定义接口。

表 3.3.13 MUX3 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	d0	32	输入	数据或地址
2	d1	32	输入	
3	d2	32	输入	
4	s	1	输入	非冲突信号
5	y	32	输出	非冲突的数据或地址

如表 3.3.14 所示为 MUX4 模块的定义接口。

表 3.3.14 MUX4 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	d0	32	输入	数据或地址
2	d1	32	输入	
3	d2	32	输入	
4	d3	32	输入	
5	s	1	输入	非冲突信号
6	y	32	输出	非冲突的数据或地址

如表 3.3.15 所示为 NPC 模块的定义接口。

表 3.3.15 NPC 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	PCPlus4D	32	输入	PC+4 的值
2	ImmD_26	26	输入	指令的低 26 位
3	ImmD_32	32	输入	分支指令的偏移
4	Reg_32	32	输入	
5	IsJJalD	1	输入	跳转类型 分支类型
6	IsJrJalrD	1	输入	

7	BranchD	1	输入	
8	CompD	1	输入	
9	NPCD	32	输出	下一条指令的地址
10	IsJBrD	1	输出	是否跳转/分支指令

如表 3.3.16 所示为 PC 模块的定义接口。

表 3.3.16 PC 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	clk	1	输入	
2	rst	1	输入	
3	En	1	输入	
4	IsJBrD	1	输入	是否跳转、分支指令
5	NPCD	32	输入	PC 按条件自增后的值
6	PCPlus4F	32	输出	实际的 PC（下一条指令处）值
7	PCF	32	输出	PC 当前值

如表 3.3.17 所示为 StartCtrl 模块的定义接口。

表 3.3.17 StartCtrl 模块定义接口

序 号	接 口 名	宽度（bit）	输入/输出	作 用
1	InstrD	32	输入	ID 阶段读入的指令
2	BusyE	1	输入	指示是否忙
3	StartD	1	输出	是否启动控制模块

4 实验过程与调试

4.1 实验环境

4.1.1 硬件

Dell G579 笔记本电脑。

4.1.2 软件

ModelSim SE-64 2019.2、MARS 4.5、Vivado 2020.1。

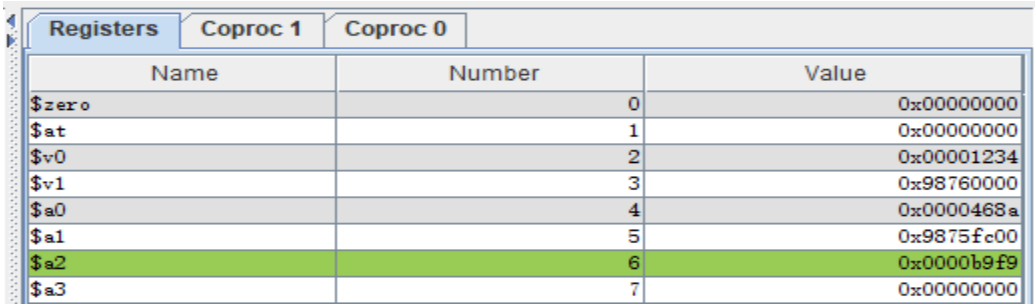
4.2 测试用例与功能测试

在完成全冒险处理机制的五段流水 CPU 的设计与实现后，下面通过实验对本设计所构建 CPU 进行样例测试与功能测试，样例测试是指检测 CPU 在运行指定程序时的输出与标准输出以及 CPU 各时序状态与预期是否一致。本设计所用 MIPS 测试样例程序在附录中給出。功能测试是指 CPU 能否达到设计的性能指标。

4.2.1 样例测试

本设计所构建 CPU 支持 MIPS-C3 指令集的全部 50 条指令，接下来选择有代表性的指令进行波形测试。为了便于实验结果对比，将同 CPU 的波形输出同 MARS 4.5 运行相同程序的结果进行对比。

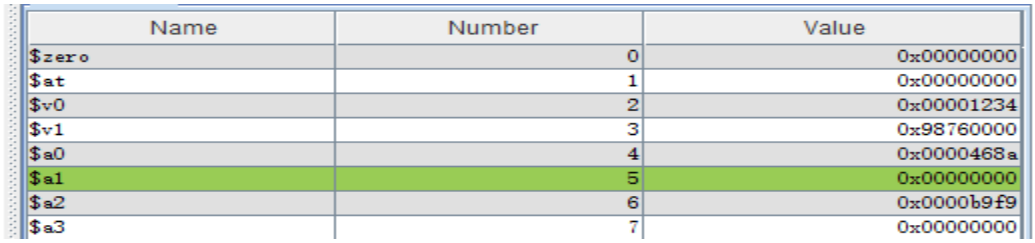
①理想流水线测试



Registers | Coproc 1 | Coproc 0

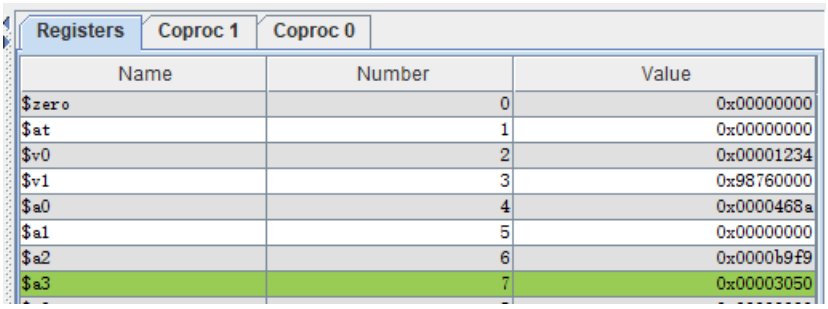
Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00001234
$v1	3	0x98760000
$a0	4	0x0000468a
$a1	5	0x9875fc00
$a2	6	0x0000b9f9
$a3	7	0x00000000

图 4.2.1.1 MARS 界面的寄存器-1



Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00001234
$v1	3	0x98760000
$a0	4	0x0000468a
$a1	5	0x00000000
$a2	6	0x0000b9f9
$a3	7	0x00000000

图 4.2.1.2 MARS 界面的寄存器-2



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00001234
$v1	3	0x98760000
$a0	4	0x0000468a
$a1	5	0x00000000
$a2	6	0x0000b9f9
$a3	7	0x00003050

图 4.2.1.3 MARS 界面的寄存器-3

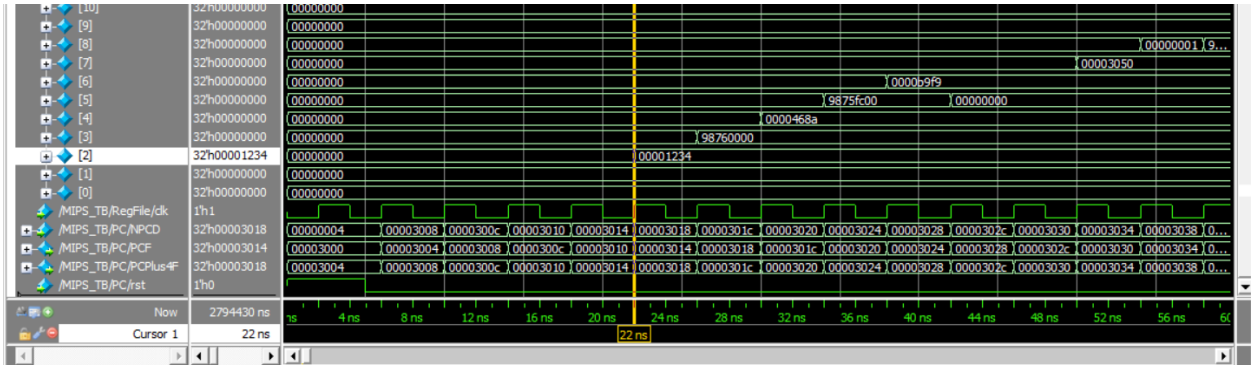


图 4.2.1.4 ModelSim 的仿真波形图-1

图 4.2.1.5 MARS 界面的寄存器-4

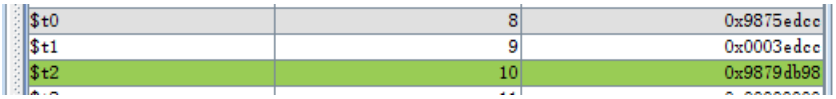


图 4.2.1.6 MARS 界面的寄存器-5

图 4.2.1.7 MARS 界面的寄存器-6

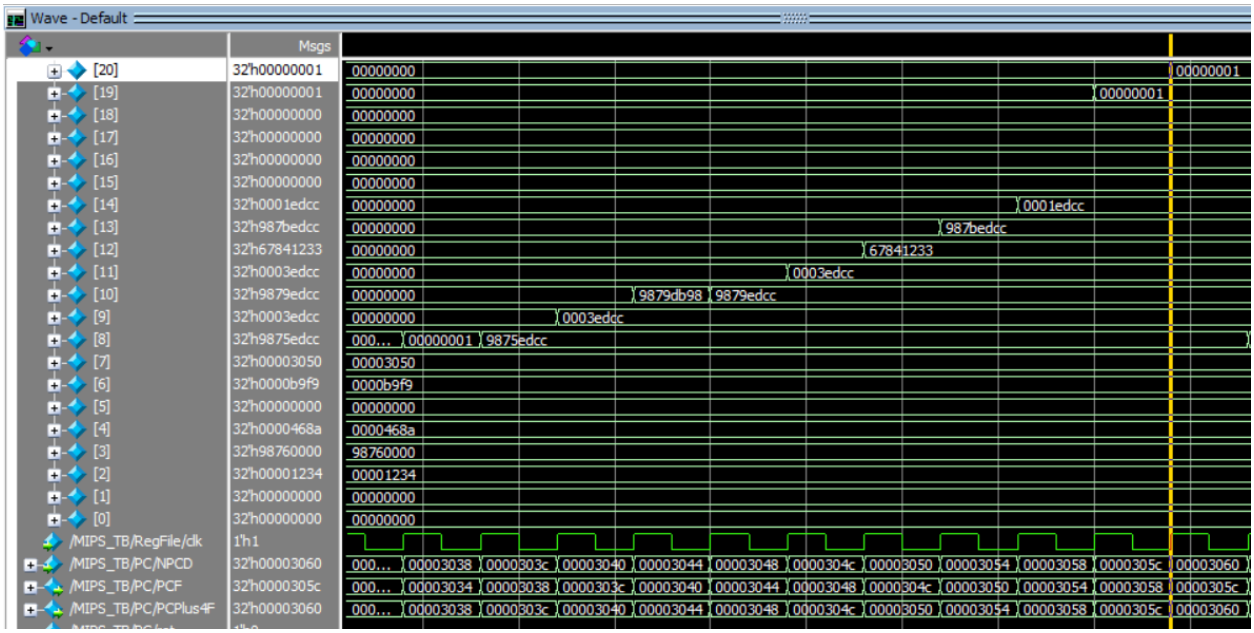


图 4.2.1.8 ModelSim 的仿真波形图-2

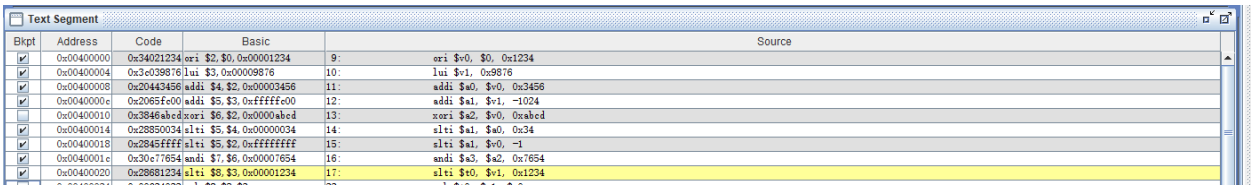


图 4.2.1.9 MARS 界面的寄存器-7

如图 4.1.1.1 至图 4.1.1.9 所示，使用无冲突的 MIPS 程序对 CPU 进行理想流水线测试，由实验结果可知，本设计所构建 CPU 运行样例程序所得波形与 MARS 显示的寄存器数据完全一致。

②跳转指令测试（以 j 指令为例）

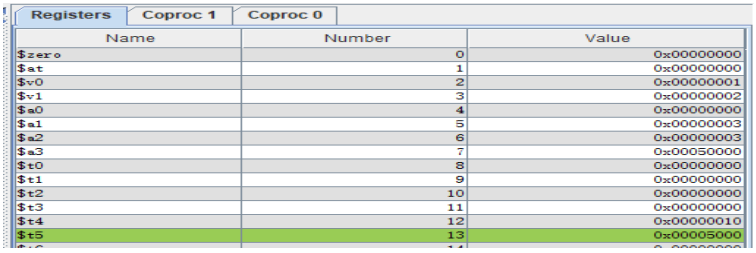


图 4.2.1.10 MARS 界面的寄存器-8

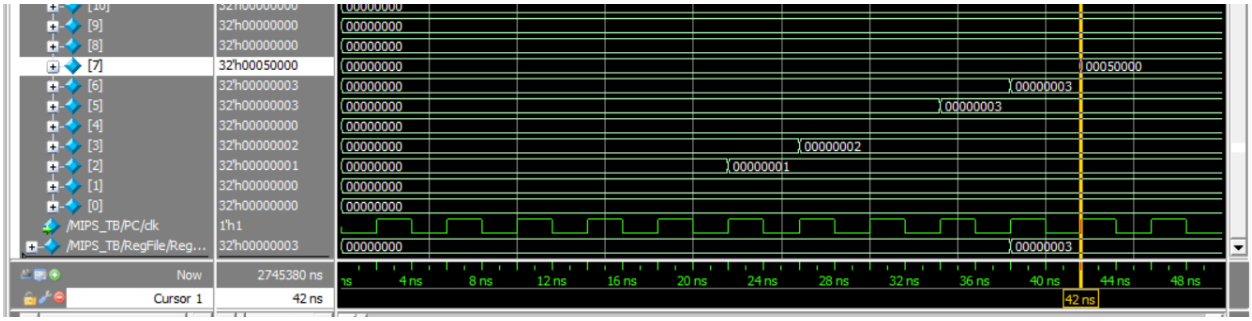


图 4.2.1.11 ModelSim 的仿真波形图-3

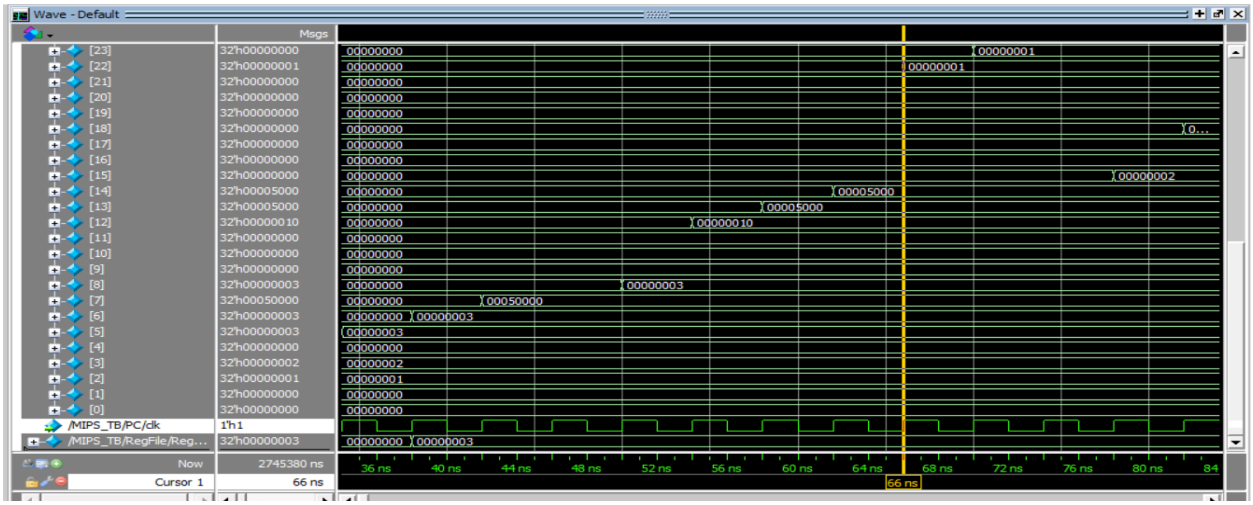
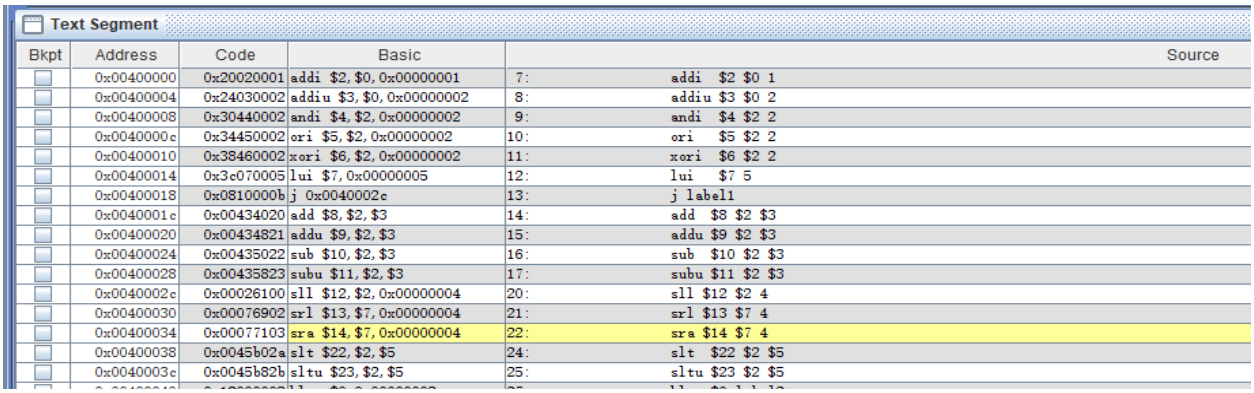


图 4.2.1.12 ModelSim 的仿真波形图-4



Text Segment

Bkpt	Address	Code	Basic		Source
	0x00400000	0x20020001	addi $2, $0, 0x00000001	7:	addi $2 $0 1
	0x00400004	0x24030002	addiu $3, $0, 0x00000002	8:	addiu $3 $0 2
	0x00400008	0x30440002	andi $4, $2, 0x00000002	9:	andi $4 $2 2
	0x0040000c	0x34450002	ori $5, $2, 0x00000002	10:	ori $5 $2 2
	0x00400010	0x38460002	xori $6, $2, 0x00000002	11:	xori $6 $2 2
	0x00400014	0x3c070005	lui $7, 0x00000005	12:	lui $7 5
	0x00400018	0x0810000b	j 0x0040002c	13:	j label1
	0x0040001c	0x00434020	add $8, $2, $3	14:	add $8 $2 $3
	0x00400020	0x00434821	addu $9, $2, $3	15:	addu $9 $2 $3
	0x00400024	0x00435022	sub $10, $2, $3	16:	sub $10 $2 $3
	0x00400028	0x00435823	subu $11, $2, $3	17:	subu $11 $2 $3
	0x0040002c	0x00026100	sll $12, $2, 0x00000004	20:	sll $12 $2 4
	0x00400030	0x00076902	srl $13, $7, 0x00000004	21:	srl $13 $7 4
	0x00400034	0x00077103	sra $14, $7, 0x00000004	22:	sra $14 $7 4
	0x00400038	0x0045b02a	slt $22, $2, $5	24:	slt $22 $2 $5
	0x0040003c	0x0045b82b	sltu $23, $2, $5	25:	sltu $23 $2 $5

图 4.2.1.13 MARS 界面的 MIPS 指令执行-1

如图 4.1.1.10 至图 4.1.1.13 所示，使用含 j 跳转指令的 MIPS 程序对 CPU 进行测试，

程序运行到第 11 个时钟周期时发生了指令跳转，由实验结果可知，本设计所构建 CPU 运行样例程序所得波形与 MARS 显示的寄存器数据完全一致。

③数据前推

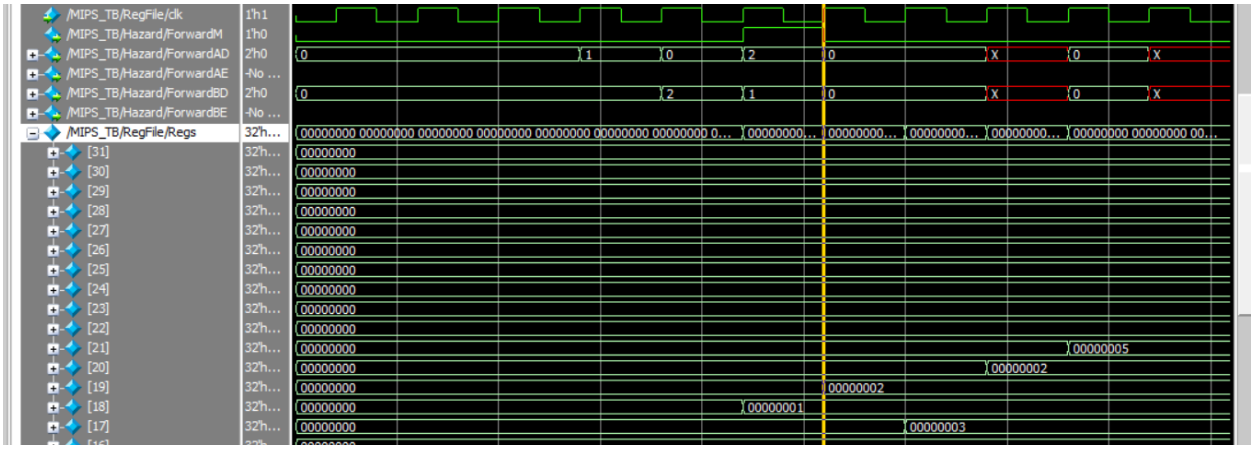


图 4.2.1.14 ModelSim 的仿真波形图-5

$s1	17	0x00000003
$s2	18	0x00000001
$s3	19	0x00000002
$s4	20	0x00000002
$s5	21	0x00000005

图 4.2.1.15 MARS 界面的寄存器-9

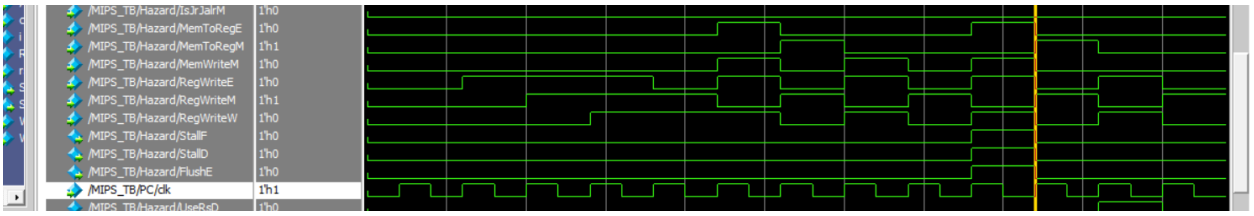


图 4.2.1.16 ModelSim 的仿真波形图-6

如图 4.1.1.14 至图 4.1.1.16 所示，使用含需要进行数据前推方能计算得出正确结果的 MIPS 程序进行 CPU 测试，程序运行到第 6 个周期时发生了数据前推。由实验结果可知，最终，本设计所构建 CPU 运行样例程序所得波形与 MARS 显示的寄存器数据完全一致。

④数据冲突

i. load-use 冲突

$s4	20	0x00000000
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc

图 4.2.1.17 MARS 界面的寄存器-10

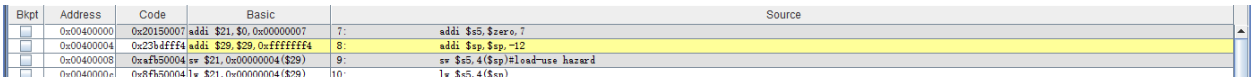


图 4.2.1.18 MARS 界面的 MIPS 指令执行-2

$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0

图 4.2.1.19 MARS 界面的寄存器-11

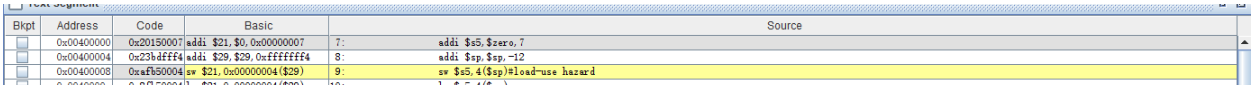


图 4.2.1.20 MARS 界面的 MIPS 指令执行-3

$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0

图 4.2.1.21 MARS 界面的寄存器-12

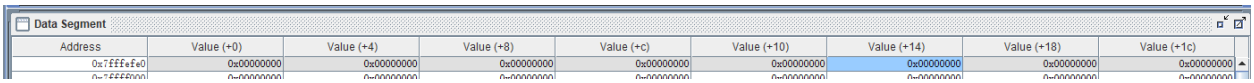


图 4.2.1.22 MARS 界面的存储器-1

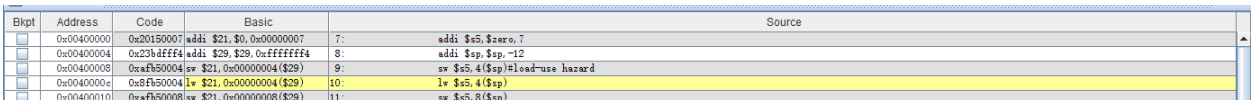


图 4.2.1.23 MARS 界面的 MIPS 指令执行-4

$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0

图 4.2.1.24 MARS 界面的寄存器-13

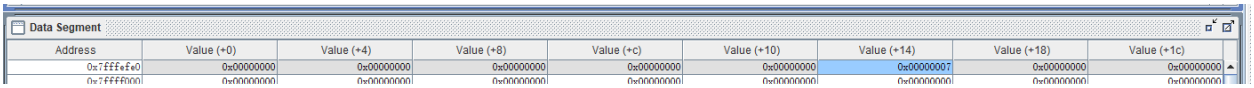


图 4.2.1.25 MARS 界面的存储器-2

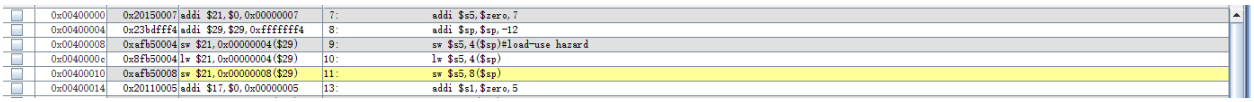


图 4.2.1.26 MARS 界面的 MIPS 指令执行-5

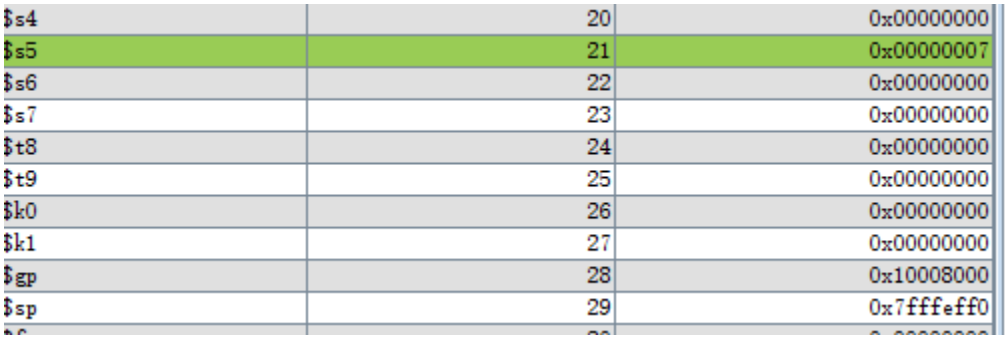


图 4.2.1.27 MARS 界面的寄存器-14

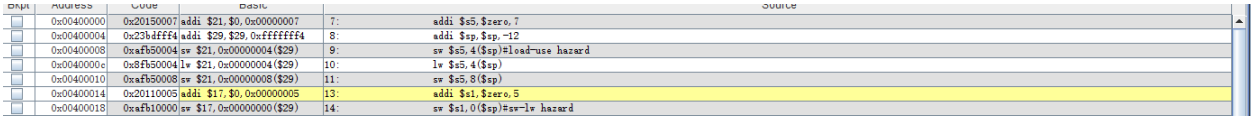


图 4.2.1.28 MARS 界面的 MIPS 指令执行-6

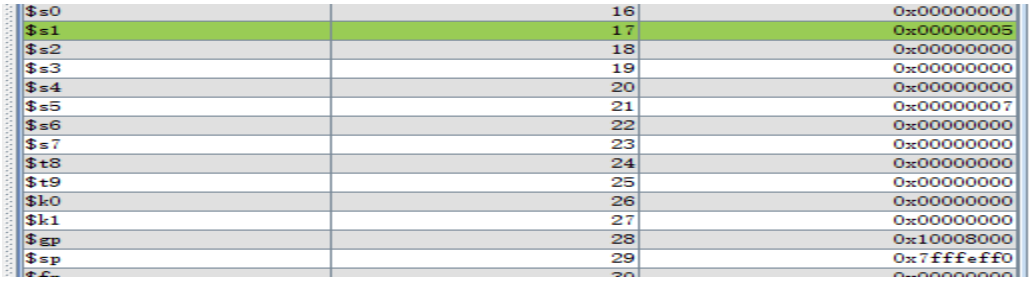


图 4.2.1.29 MARS 界面的寄存器-15

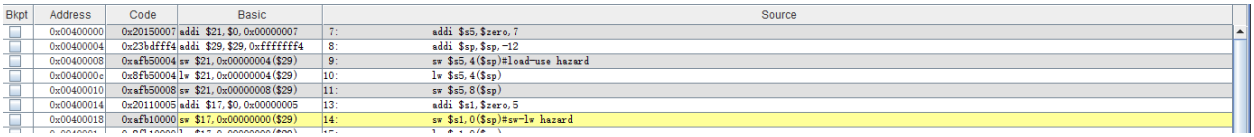


图 4.2.1.30 MARS 界面的 MIPS 指令执行-7

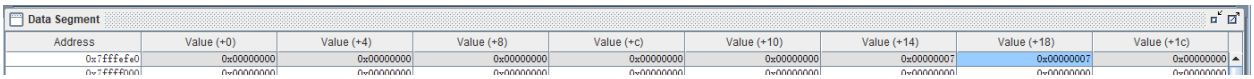


图 4.2.1.31 MARS 界面的存储器-3

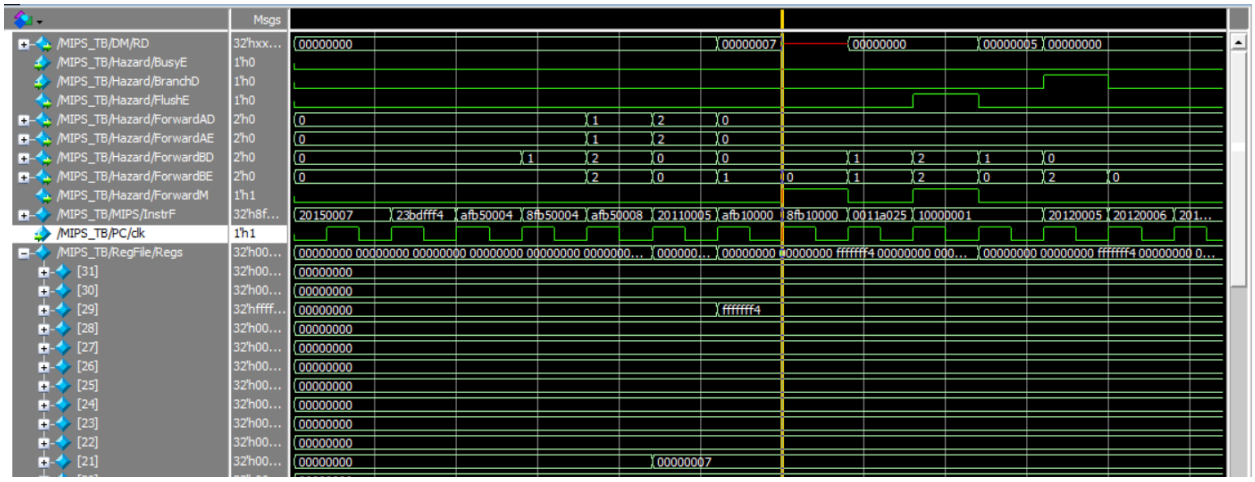


图 4.2.1.32 ModelSim 的仿真波形图-7

如图 4.1.1.17 至图 4.1.1.32 所示，使用含 load-use 冲突的 MIPS 程序进行 CPU 测试，程序运行到第 3 个周期时发生了 load-use 冲突。由实验结果可知，最终，本设计所构建 CPU 运行样例程序所得波形与 MARS 显示的寄存器数据完全一致。

ii. sw-lw 冲突

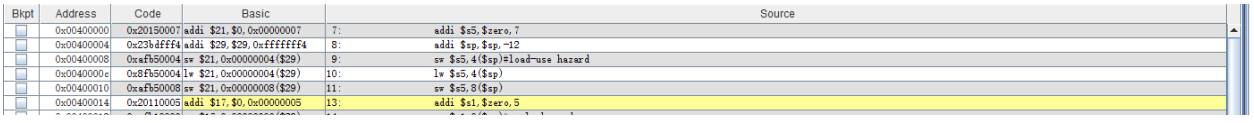


图 4.2.1.33 MARS 界面的 MIPS 指令执行-8

$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0

图 4.2.1.34 MARS 界面的寄存器-16

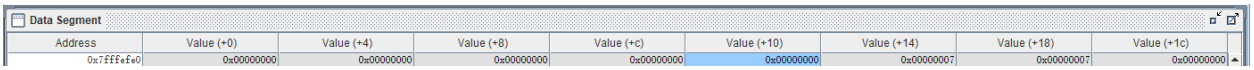


图 4.2.1.35 MARS 界面的存储器-4

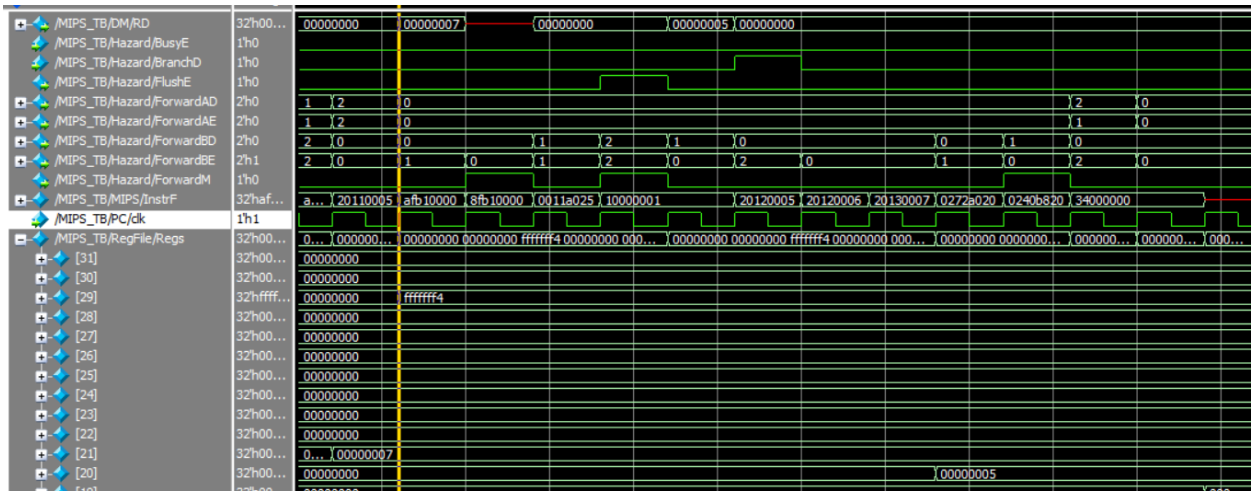


图 4.2.1.36 ModelSim 的仿真波形图-8

$s1	17	0x00000005
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0

图 4.2.1.37 MARS 界面的寄存器-17

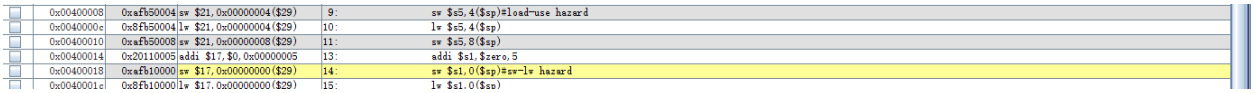


图 4.2.1.38 MARS 界面的 MIPS 指令执行-9

$s1	17	0x00000005
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x0040001c

图 4.2.1.39 MARS 界面的寄存器-18

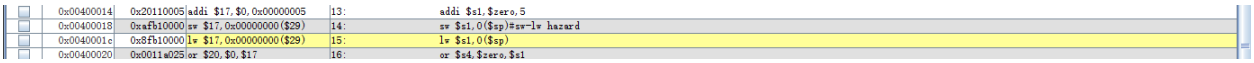


图 4.2.1.40 MARS 界面的 MIPS 指令执行-10

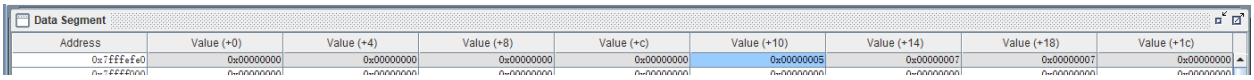


图 4.2.1.41 MARS 界面的存储器-5

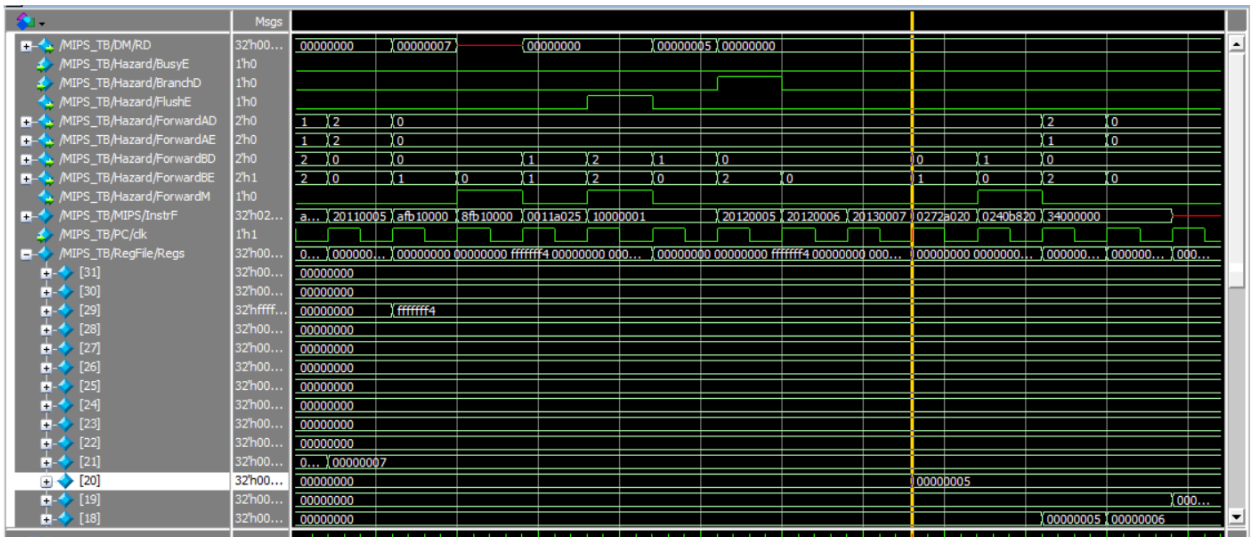


图 4.2.1.42 ModelSim 的仿真波形图-9

如图 4.1.1.33 至图 4.1.1.42 所示，使用含 sw-lw 冲突的 MIPS 程序进行 CPU 测试，程序运行到第 7 个周期时发生了 load-use 冲突。由实验结果可知，最终，本设计所构建 CPU 运行样例程序所得波形与 MARS 显示的寄存器数据完全一致。

iii. RAW 冲突

$s1	17	0x00000005
$s2	18	0x00000006
$s3	19	0x00000000
$s4	20	0x00000005
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400030

图 4.2.1.43 MARS 界面的寄存器-19

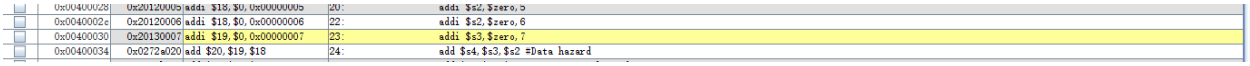


图 4.2.1.44 MARS 界面的 MIPS 指令执行-11

$s1	17	0x00000005
$s2	18	0x00000006
$s3	19	0x00000007
$s4	20	0x00000005
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400034

图 4.2.1.45 MARS 界面的寄存器-20

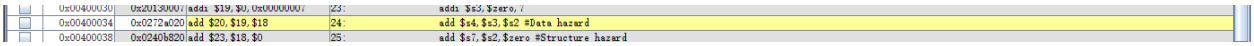


图 4.2.1.46 MARS 界面的 MIPS 指令执行-12

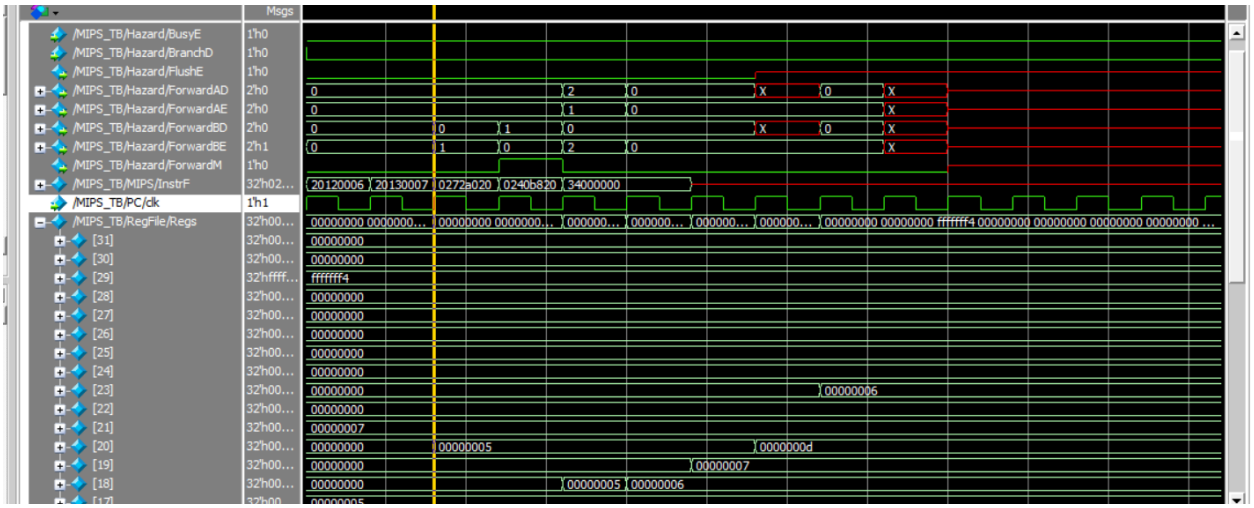


图 4.2.1.47 ModelSim 的仿真波形图-10

$s1	17	0x00000005
$s2	18	0x00000006
$s3	19	0x00000007
$s4	20	0x0000000d
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400038

图 4.2.1.48 MARS 界面的寄存器-21

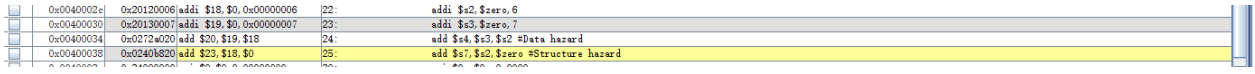


图 4.2.1.49 MARS 界面的 MIPS 指令执行-13

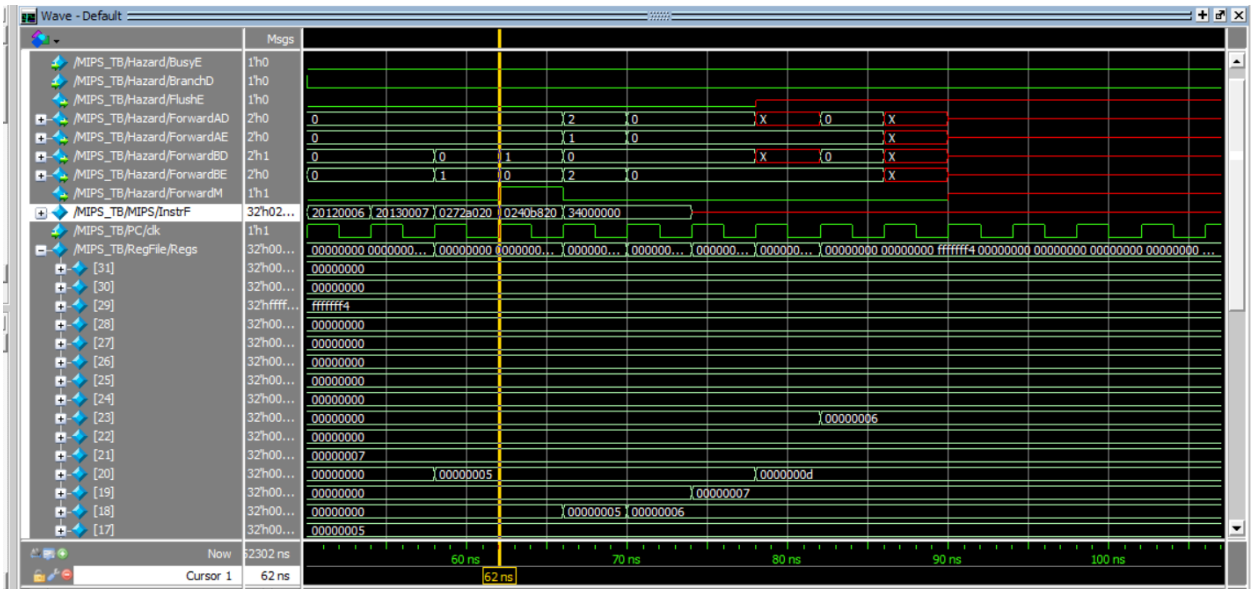


图 4.2.1.50 ModelSim 的仿真波形图-11

如图 4.1.1.43 至图 4.1.1.50，使用含 sw-lw 冲突的 MIPS 程序进行 CPU 测试，程序运行到第 14 个周期时发生了 load-use 冲突。由实验结果可知，最终，本设计所构建 CPU 运行样例程序所得波形与 MARS 显示的寄存器数据完全一致。

iv. 结构冲突

$s1	17	0x00000005
$s2	18	0x00000006
$s3	19	0x00000007
$s4	20	0x0000000d
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400038

图 4.2.1.51 MARS 界面的寄存器-22

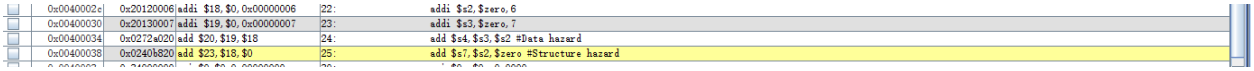


图 4.2.1.52 MARS 界面的 MIPS 指令执行-14

$s0	16	0x00000000
$s1	17	0x00000005
$s2	18	0x00000006
$s3	19	0x00000007
$s4	20	0x0000000d
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400038

图 4.2.1.53 MARS 界面的寄存器-23

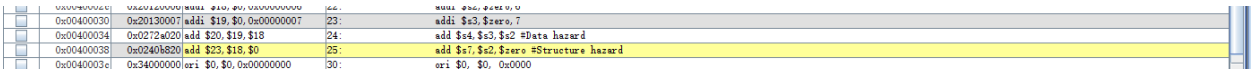


图 4.2.1.54 MARS 界面的 MIPS 指令执行-15

$s0	16	0x00000000
$s1	17	0x00000005
$s2	18	0x00000006
$s3	19	0x00000007
$s4	20	0x0000000d
$s5	21	0x00000007
$s6	22	0x00000000
$s7	23	0x00000006
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeff0
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x0040003c
hi		0x00000000

图 4.2.1.55 MARS 界面的寄存器-24

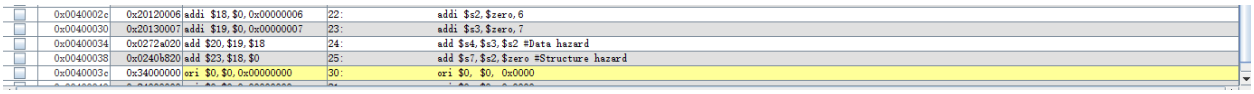


图 4.2.1.56 MARS 界面的 MIPS 指令执行-16

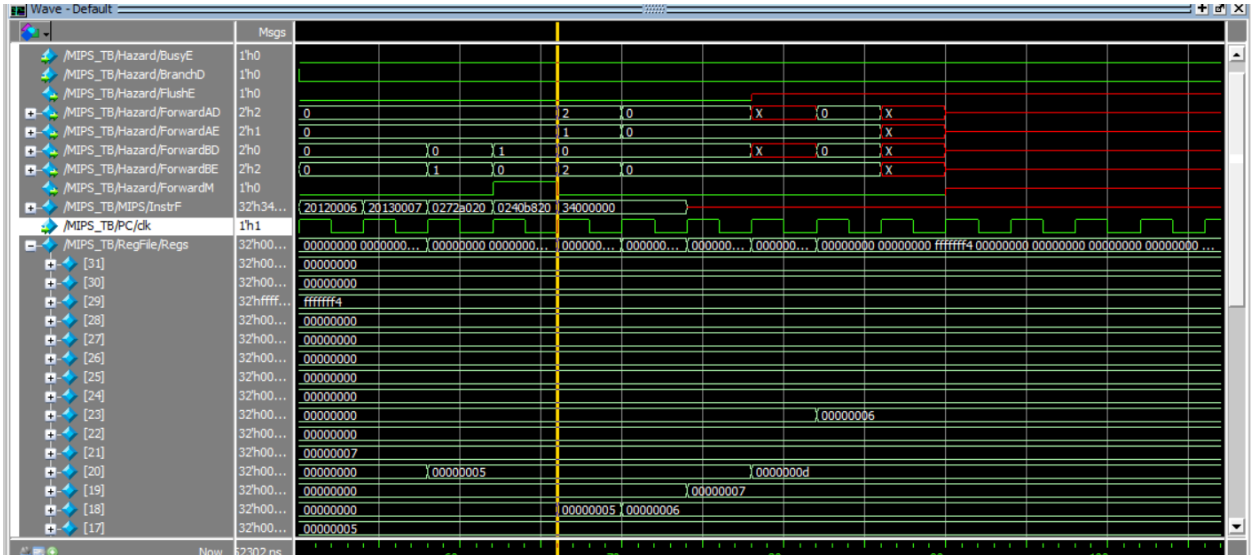


图 4.2.1.57 ModelSim 的仿真波形图-12

如图 4.1.1.51 至图 4.1.1.57，使用含结构冲突的 MIPS 程序进行 CPU 测试，程序运行

到第 15 个周期时发生了 load-use 冲突。由实验结果可知，最终，本设计所构建 CPU 运行样例程序所得波形与 MARS 显示的寄存器数据完全一致。

4.3 性能分析

①采用单周期方式（开源项目代码）下执行 benchmark 时钟周期数为：2698，而采用静态分支预测分支失败的五段流水方式 benchmark 时钟周期数为：3870。假定单周期方式时钟周期为五段流水方式的 5 倍，则加速比为：

$$S = \frac{2698 \times 5\Delta t}{3870 \times \Delta t} \approx 3.49$$

②正确进行数据转发与阻塞；

③执行乘法需 5 个时钟周期，执行除法需 10 个时钟周期，该功能使用一个计数器模拟实现。

4.4 主要故障与调试

4.4.1 保持复位信号为 1 导致时序不正确

复位信号 reset 只应在系统开始运行后的极短时间内为 1，其后处于 0 状态。

故障现象：寄存器中未观察到预期的数据，时序“不正确”

原因分析：复位信号 reset 用于将系统恢复至初始状态，只能在系统开始运行后的一小段时间内为 1，使系统各个指标回复至初始态，便于数据的载入以及系统的正式运行。

解决方案：reset 最初赋值为 1，至#5 时改赋为 0，此后若系统未遇特殊异常引起系统中断复位，则不再改变。

4.4.2 load-use 冲突的处理未达预期效果

因未在 ID 段检测信号，导致一条 lw 指令被挤掉。

故障现象：指令执行后，寄存器中的数据并非预期的值。

原因分析：benchmark 中存在两条 lw 指令的目的寄存器编号分别与后一条指令两个源寄存器编号相同的情况，若在 EX 段才检测 load-use 冲突，则因为插入一个气泡，将第一条 lw 挤出流水线，故 EX 段时钟有效时，其需要的数据已经正确写回，从而无法通过定向技术获得。

解决方案：按照标准的处理方法，将 load-use 的检测提前至 ID 段。

4.4.3 算术右移计算结果不正确

如题，CPU 在进行算术右移时所得计算结果不正确。

故障现象：ALU 在进行算术右移运算时，输入数据与控制信号都正确，但结果不对。

原因分析：查阅有关资料得知被移位的对象必须定义为 reg 类型，而本设计实现起初是直接对输出信号进行 assign 赋值。

解决方案：使用 reg 类型的临时变量进行转存。

4.4.4 其他 BUG

实验过程中遇到了各种各样稀奇古怪的 bug，不一而足，“不足为外人道也”。

故障现象：数组越界、语法错误、负载没有读进去，等等。

原因分析：对 Verilog 不熟悉、前期准备不足。

解决方案：查阅有关资料，求助同学。

5 总结与心得体会

5.1 课设总结

本设计实现了全冒险处理机制的 MIPS 五段流水 CPU，可以运行 MIPS-C3 指令集的所有指令，比较全面地解决了数据冲突、结构冲突、控制冲突等问题，支持数据阻塞和转发。本设计所实现的 CPU 执行乘法需要 5 个时钟周期，执行除法需要 10 个时钟周期，该功能使用一个计数器模拟实现。CPU 在进行乘除法运算时，可以并行执行其他非乘除法指令，冒险检测单元会根据 ID 级的指令类型以及乘除法部件的 busy 值来判断是否对流水线进行暂停和冲刷。为和测试代码保持一致，方便进行测试，本设计所实现 CPU 支持一个延迟槽。值得一提的是，为简化控制信号，没有对 ALU 单独设置控制器，而是直接在主控制器中发出 ALU 控制信号。

5.2 课设心得

在完成本次课程设计得过程中，我们通过查阅有关资料，完成了既定任务。早在大三《计算机组成原理》开课时，陈田老师就曾说过“这门课有一个课设，是让你们自己造一台 CPU”，那时觉得是天方夜谭：本科生怎么可能做得出来 CPU？后面经过学习与主动了解，明白了“造 CPU”的具体要求，畏难情绪少了许多。计组算得上是对于课程考试并不十分擅长的我学得比较透彻的一门课，个中原因可能是自身对硬件比较喜爱吧。本次课程设计的一个遗憾是：我们没有足够的时间来将 CPU 烧到 FPGA 上面，完成硬件演示，功底还是差了许多，李荣浩的那句经典歌词“要是能从来，我要学李白……”。总之，CPU 课设挺好玩的，哈哈哈！😄

老师真帅，給个优吧！🤣😂良也行啊……QwQ

参考文献

[1]唐朔飞. 计算机组成原理（第 2 版）[M]. 北京：高等教育出版社，2008.

[2]张晨曦，王志英等. 计算机体系结构（第 2 版）[M]. 北京：高等教育出版社，2014.

[3] https://zh.wikipedia.org/zh-hans/%E6%B5%81%E6%B0%B4%E7%BA%BF_(%E8%AE%A1%E7%AE%97%E6%9C%BA)

[4]雷思磊. 自己动手写 CPU[M]. 北京：电子工业出版社，2014.

[5]https://zh.wikipedia.org/wiki/%E5%86%92%E9%99%A9_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E4%BD%93%E7%B3%BB%E7%BB%93%E6%9E%84)

[6]王泽坤. MIPS 架构 CPU 设计及 SoC 系统实现[D]. 沈阳：东北大学，2014.

[7]蔡晓燕，袁春风，张泽生. MIPS 架构多周期 CPU 的设计[D]. 计算机教育：第 17 期：93-96，2014.

[8] 李东泽，曹凯宁，曲明，王富昕. 五级流水线 RISC-V 处理器软硬件协同仿真验证[J]. 吉林大学学报（信息科学版）：Vol. 35 NO.6：612-616，2017.

[9]Charles Price. MIPS IV Instruction Set[S]. MIPS Technologies, Inc. 1995.

附录

项目源码已经托管至 GitHub 网站，网址：

https://github.com/25thengineer/HFUT_2020_MIPS_CPU

测试文件 TESTBENCH：

```
#case 1
        .org 0x0
        .set noat
        .set noreorder                  #不进行指令调度
        .set nomacro
        .global __start
__start:
        # 注意，MIPS 编译时，会将 rs 和 rt 的二进制位置互换，写法上是 rt,rs，指令码是
opcode rs rt(写入 rt)
        #
        # 用 nop 和 ori 指令作为开始标志
        # start code
        # 3400 0000 3400 0000
        # ori $0, $0, 0x0000
        # ori $0, $0, 0x0000
        # 注意由于，编译装入问题，此处不用原先指令开头
        # 3403 8000
        # ori  $3, $0, 0x8000

        # 此处开始书写代码
        ###################
        # Test Subset 2  #
        ori $v0, $0, 0x1234
        lui $v1, 0x9876
        addi $a0, $v0, 0x3456
        addi $a1, $v1, -1024
        xori $a2, $v0, 0xabcd
        slti $a1, $a0, 0x34
        slti $a1, $v0, -1
        andi $a3, $a2, 0x7654
        slti $t0, $v1, 0x1234

        ###################
        # Test Subset 1  #
```

```
        sub $t0, $v1, $v0
        xor $t1, $t0, $v1
        add $t2, $t1, $t0
        add $t2, $t2, $v0
        sub $t3, $t2, $v1
        nor $t4, $t3, $t2
        or  $t5, $t3, $t2
        and $t6, $t3, $t2
        slt $s3, $t5, $t4
        slt $s4, $t5, $t4

        ### Test for shift
        sll $t0, $t0, 3
        srl $t1, $t0, 16
        sra $t2, $t0, 29
        addi $t3, $0, 0x3410


        # end code
        # 3400 0000 3400 0000
        ori $0, $0, 0x0000
        ori $0, $0, 0x0000
```

#case 2

```
        .org 0x0
        .set noat
        .set noreorder                  #不进行指令调度
        .set nomacro
        .global __start
__start:
        # 注意，MIPS 编译时，会将 rs 和 rt 的二进制位置互换，写法上是 rt,rs，指令码是
opcode rs rt(写入 rt)
        #
        # 用 nop 和 ori 指令作为开始标志
        # start code
        # 3400 0000 3400 0000
        # ori $0, $0, 0x0000
        # ori $0, $0, 0x0000
        # 注意由于，编译装入问题，此处不用原先指令开头
        # 3403 8000
        # ori  $3, $0, 0x8000

        # 此处开始书写代码
        addi $18 $0 1
        addi $19 $0 2
        add $17 $18 $19
```

```
        sub $20 $17 $18
        add $21 $19 $17


        # end code
        # 3400 0000 3400 0000
        ori $0, $0, 0x0000
        ori $0, $0, 0x0000
```

#case 3

```
        .org 0x0
        .set noat
        .set noreorder                  #不进行指令调度
        .set nomacro
        .global __start
__start:
        # 注意，MIPS 编译时，会将 rs 和 rt 的二进制位置互换，写法上是 rt,rs，指令码是
opcode rs rt(写入 rt)
        #
        # 用 nop 和 ori 指令作为开始标志
        # start code
        # 3400 0000 3400 0000
        # ori $0, $0, 0x0000
        # ori $0, $0, 0x0000
        # 注意由于，编译装入问题，此处不用原先指令开头
        # 3403 8000
        # ori  $3, $0, 0x8000

        # 此处开始书写代码
        addi $s5,$zero,7
        addi $sp,$sp,-12
        sw $s5,4($sp)#load-use hazard
        lw $s5,4($sp)
        sw $s5,8($sp)

        addi $s1,$zero,5
        sw $s1,0($sp)#sw-lw hazard
        lw $s1,0($sp)
        or $s4,$zero,$s1


        beq $zero,$zero,label #beq
        addi $s2,$zero,5
label:
        addi $s2,$zero,6
        addi $s3,$zero,7
```

```
        add $s4,$s3,$s2 #Data hazard
        add $s7,$s2,$zero #Structure hazard


        # end code
        # 3400 0000 3400 0000
        ori $0, $0, 0x0000
        ori $0, $0, 0x0000
```