计算机体系结构实验课程第_三_次实验报告

实验名称	多周期 CPU 实现			班级	李雨森老师
学生姓名	蒋薇	学号	2110957	指导老师	董前琨
实验地点	A308		实验时间	2023.10.23	

1、 实验目的

1. 加深对计算机组成原理和体系结构理论知识的理解。

2. 培养对 CPU 设计的兴趣，在理解现有 CPU 架构的基础上，引发对体系结构的思考和创新。

3. 培养创新思维能力，并通过实践验证新想法。

2、 实验内容说明

1. 分析静态 5 级流水 CPU 中的流水线阻塞情况，包括数据相关、控制相关、结构相关等，优化流水线设计，尽可能减少流水线阻塞情况，比如前递技术等。

2. 对于分支跳转指令，mips 架构中有延迟槽指令的设定，利用这一点，在静态 5 级流水 CPU 中，可实现分支指令永远不阻塞后续指令，大家可以检查自己的流水线设定，进行优化实现这一点。

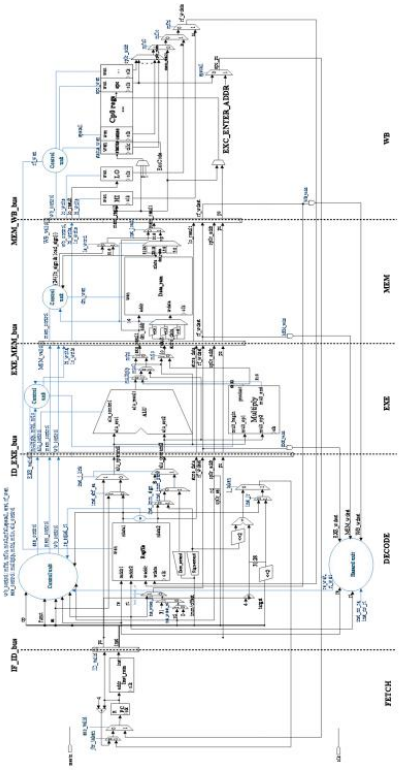
3. 针对第 2 点，此时，分支跳转指令就不需要进行转移猜测了，但大家可以将流水线结构改为 x86 中无延迟槽技术的设定，此时分支跳转指令与后续真正需要执行的指令至少会堵塞一拍，此时可以考虑实现转移预测技术，提升流水线结构。

4. 在学习的过程中，大家一定会有很多自己的想法，比如，为什么取指、译码、执行、访存、写回称为经典的 5 级流水结构，可以实现 3 级、4 级、6 级流水结构的 CPU 吗?答案肯定是可以的，甚至在各类产品的 CPU 中采用经典 5 级流水结构的都很少。所以希望大家尽情地发挥自己的想法，大改流水结构，验证自己关于流水结构的想法，您可以实现 3 级（比如：取指、译码、执行）、4 级、6 级等等各类流水结构的 CPU。

5. 课程设计要求大家实现的 mips 指令有限，大家可以分析其余 mips 指令，加以实现。

6. 目前课程设计实现的指令存储器和数据存储器是同步读的机制的，故在当前拍数（时钟周期）发出读数据的地址请求时，在下一拍才能获得读的数据，因此取值级和访存级的 load 都需要两拍时间。其实发地址请求在下一拍获得读的数据，明显也是一个可以流水做的工作，故可以考虑对流水线设计方案作稍微修改，使得取指级和访存级的 load 不需要多等一拍。

五级流水线 CPU 的实现框图:



五流水线 CPU 测试所用汇编程序：

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

3、 实验步骤

1. 使用前递技术减少阻塞情况

指令相关和流水线冲突

目前流水线结构把一条指令的来去过程分成 5 个阶段，分别是取指（IF）、译码（ID）、执行（EX）、访存（MEM）和写回（WB）。计算 PC 和译码阶段是并行的，可以视作译码的一部分。虽然这样的流水可以提高主频，但由于执行间的相关可能会导致执行结果的错误。如第 N 条指令把结果写到 R1 寄存器，第 N+1 条指令要用到 R1 的值进行运算，由于第 N+1 条指令读 R1 发生在译码阶段，此时 R1 的值还没更新，从而发生错误。

指令相关的分类

数据相关：两条指令访问同一个寄存器或内存单元，且这两条指令中至少有一条是写该寄存器或者内存单元的指令（RAW、WAR、WAW）

控制相关：两条指令中一条是转移指令且另一条指令是否执行取决于该转移指令的执行结果

结构相关：两条指令使用同一个功能部件，如二者都是加法指令

保序执行

由于指令之间的相关，使得指令的执行存在一个序的关系。相关指令之间要隔开满足保序要求的距离才能确保得到正确的结果

数据相关

写后读相关（RAW）

定义：后面的指令要用到前面指令所写的数据，也称真相关，唯一一个不是乱序执行也会发生的相关错误，最常见的相关

解释：下图中，第一条指令的 WB 影响着后面第 2,3,4 条指令的 ID。因为 WB 阶段才能更新通用寄存器堆的值，而 ID 阶段就要读通用寄存器的值

解决方法

阻塞。即在指令译码阶段读取寄存器时，如果发现该寄存器是流水线先前（前 3 时钟内）指令的目标寄存器且没有写回，那么就要在译码阶段等待。

具体方法：ID 流水级指令的两个源寄存器号 rs 和 rt 分别跟 EX、MEM、WB 流水级指令的目标寄存器号 dest 进行相等比较，如果有一个相等且该不是第 0 号寄存器（0 号寄存器的值恒为 0），这条指令就不能前进。（ADDIU 和 LW 指令中的 rt 不是源寄存器，无需比较）。

连锁反应：为了阻塞流水线，需要对 PC 和 IR 的输入使能进行控制，即相关判断逻辑为 1 时，需要控制 PC 和 IR 的输入使能，使 PC 和 IR 保持当前的值不变。而 EX 阶段的流水线则输入指令无效信号，用流水线空泡（bubble）填充 加入阻塞的硬件结构

写后写相关（WAW）

定义：即两个指令写同一个单元，在乱序执行的结构中可能会发生此类错误，静态流水线天然保序，不会出现这种问题

读后写相关（WAR）

定义：在乱序执行的结构或者读写指令流水级不一样时，如果后面的写指令执行得快，在前面的读指令读书之前就把目标单元原来的值覆盖掉了，导致读书指令读到了该单元“未来”的值，从而引起错误

控制相关

冲突的本质：对程序计数器 PC 的冲突访问引起的

解释：在上图中第一条指令的 ID 到第二条指令的 IF 之间的箭头表示控制相关引起的冲突，

即如果第一条是转移指令，第二条不是延迟槽指令的话，则第二条指令等一拍。这里要强调的是，NPC 总是需要计算才能得到，将 PC 相关的计算部件放在了 ID 阶段而非 EX 阶段，就可以仅打一拍完成转移指令的后续操作，否则等写回时才拿到 NPC 的结果就要等 4 拍，从而造成大量的空泡。

数据相关的联动：如果转移指令在译码阶段根据寄存器的值进行条件判断时，该寄存器是 EX、MEM 和 WB 阶段的目标寄存器时，即转移指令与先前的指令存在数据相关，转移指令也会由于数据相关堵在 ID 流水级，延迟槽指令则堵在 IF 流水级

解决方法

加延迟槽这个解决控制相关的方法仅适应于单条静态流水线，而现代处理器的超流水、多发射、乱序执行流水线中，则需要复杂的转移猜测技术

结构相关

问题举例

取指令和访存有单独的存储器，如果共用一个的话，取指和取数都访存时就会冲突，方寸的时候下一条指令就不能取指了。

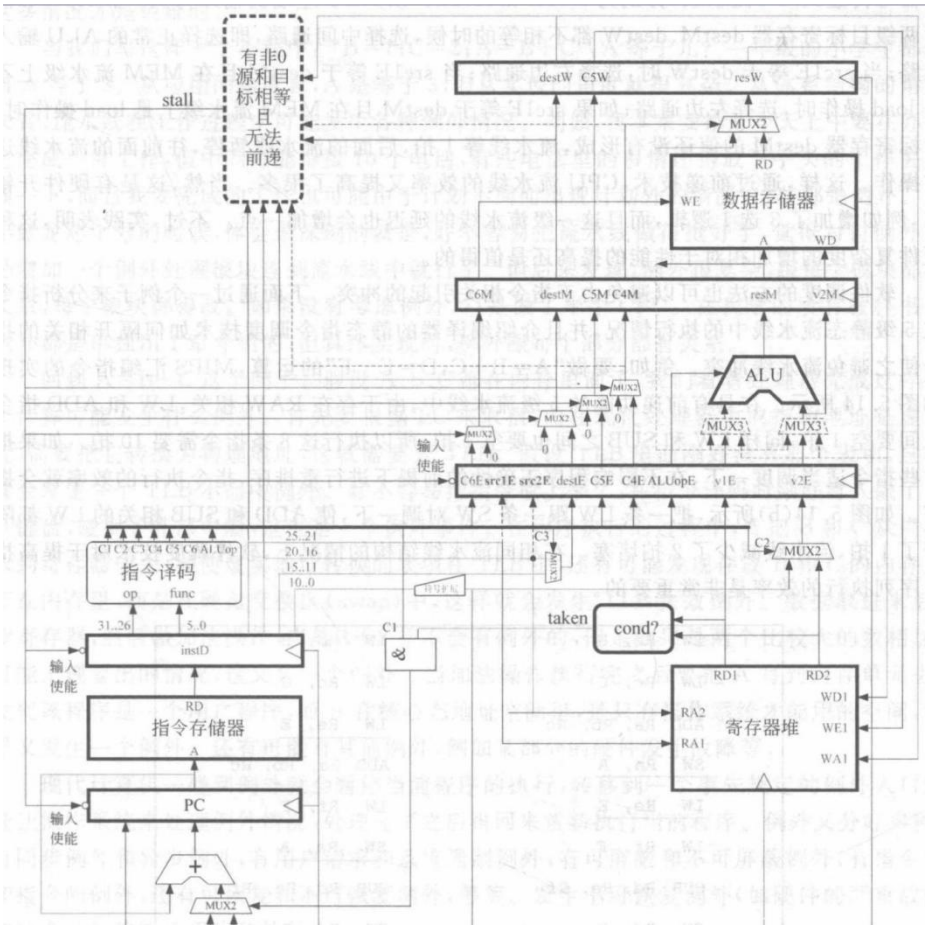
由于除法不能全流水的实现，前一条除法指令在运算时后面的触发指令需要等待

5 级流水中，尽管普通运算指令不用访存，但也要经历 MEM 流水，本质原因是如果有写指令 5 拍写回，而有的 4 拍写回，就会引起通用寄存器堆写端口的结构冲突

流水线的前递技术

5 级静态流水线通过指令的阻塞来保序，但阻塞必然导致流水线效率的降低，可以通过软硬件的方法提高流水线效率。

前递技术（Forward），又称旁路技术（Bypass）



原理：若前后的指令有数据相关，将前面的指令直接把运算结果传给后边的指令而非等到写回寄存器后再给后边的指令结果。

具体实现：在流水线的运算器前通过多路选择直接把前面指令的运算输出作为后面指令的输入。 带有前递技术的流水线下图是在原来流水线的基础上添加了部分前递通路的情况，且只考虑数据前递给 ALU，不考虑前递给存数指令和转移指令的旁路。

ALU 的每个输入端都添加一个 3 选 1 逻辑，3 个输入分别是原来的 ALU 输入、下一级流水线输出的结果（即 EX 流水级 ALU 的运算结果）和再下一级流水线输出的结果（即 MEM 流水级的结果)。**这样后面指令要用到前面指令的运算或访存结果时，就可以直接通过运算器前面的多路选择器选择前面指令的运算或访存结果，不用等到前面指令把结果写回到寄存器后再从寄存器中读取**

前递的判断：需要比较处于 EX 阶段的指令的源寄存器号和处于 MEM 或者 WB 阶段的指令的目标寄存器号是否相等，如果相等且不是 0 号寄存器，则说明处于 EX 流水级的指令与前面的指令有数据相关，需要直接读取前面指令的结果用于运算器的输入。

硬件开销：采用前递技术后就要把指令的两个源寄存器号传递到 EX 流水级，分别是图中的 src1E 和 src2E，

三选一通路：以 ALU 左侧的三选一为例说明

src1E 和前面两级目标寄存器 destM 和 destW 都不相等时，选择中间通路，即正常的 ALU 输入

当 src1E 等于 destW 时，选择右边通路，即 WB 阶段的输出作为 ALU 的输入

当 src1E 等于 destM 时，选择左边通路，但如果 MEM 流水级上是 load 操作时，则意味着 destM

的值还没形成，流水线要等一拍，后面的流水线暂停

1、从执行阶段（EX）前递到译码阶段（ID）：当一个指令在执行阶段计算出结果后，该结果可以直接传递给后续的指令进行译码和操作数读取;2、从访存阶段（MEM）前递到译码阶段（ID）：当一个指令在访存阶段计算出结果后，该结果可以直接传递给后续的指令进行译码和操作数读取;3、从执行阶段（EX）前递到访存阶段（MEM）：当一个指令在执行阶段计算出结果后，该结果可以直接传递给后续的指令进行访存操作。

解决数据冒险问题

（1）使用纯暂停流水线方法解决数据冒险问题：

1.分析数据冒险出现的情况；
2.如何检测数据冒险是否发生；
3.修改流水线 CPU 代码，当数据冒险发生时用暂停流水线的方式处理，保证程序运行结果的正确性。
（2）使用内部前推技术+暂停流水线方法解决数据冒险问题：
1.分析数据冒险出现的情况；
2.如何检测数据冒险是否发生；
3.修改流水线 CPU 代码，当数据冒险发生时用数据前推的方式处理，保证程序运行结果的正确性。
4. 分析在非 Load 指令后产生数据冒险时，是否能通过纯内部前推技术得到正确结果。分析当检测到 Load 指令后数据冒险时，是否能通过内部前推数据+暂停流水线技术得到正确的计算结果。

解决控制冒险问题

1. 修改流水线 CPU 代码，解决无条件跳转指令（JUMP 指令）的控制冒险问题。

   a) 消除无条件跳转指令的后续指令所产生的影响；

2. 修改流水线 CPU 代码，解决条件跳转指令（BNE 与 BEQ 指令）的控制冒险问题。

   a) 当条件跳转指令的 Z 信号还未准备好时，需要暂停流水线；

b) 消除条件跳转指令的后续指令所产生的影响；

2、针对分支跳转指令，实现不阻塞后续指令执行:

在 MIPS 架构中，分支跳转指令会引入分支延迟（branch delay），即指令流水线在分支指令后的一个周期会执行分支指令后的指令而不是下一条指令。这是因为分支指令的条件判断需要在取指阶段完成，而取指阶段之后的指令已经进入流水线。

延迟槽：将分支指令后面的一个指令作为延迟槽指令，将其与分支指令一起执行。这样，在分支指令的条件判断结果出来之前，延迟槽指令可以继续执行，避免了分支指令造成的流水线停顿。

延迟槽是指在分支指令后面的一个位置留出的空槽，可以放置一个指令，这个指令会在分支指令判断结果之前执行。

```
    int condition = 1; // 分支条件
// 分支跳转指令
if (condition) {
    branchDelayInstruction();  // 分支延迟槽中的指令
}
// 后续指令
followingInstruction();
// 分支延迟槽中的指令
void branchDelayInstruction() {
// 延迟槽中的指令，可以是任意的指令，不依赖于分支结果
//延迟槽中的指令为nop（空指令）
asm(“nop”);
}
// 后续指令
void followingInstruction() {
// 后续指令，可以是任意的指令
```

延迟槽中的指令应该是不依赖于分支结果的指令，因为在分支指令条件判断之前，延迟槽中的指令已经开始执行了。如果延迟槽中的指令依赖于分支结果，可能会导致错误的结果。

3、针对分支调整指令，考虑加入分支预测，提升流水线性能:

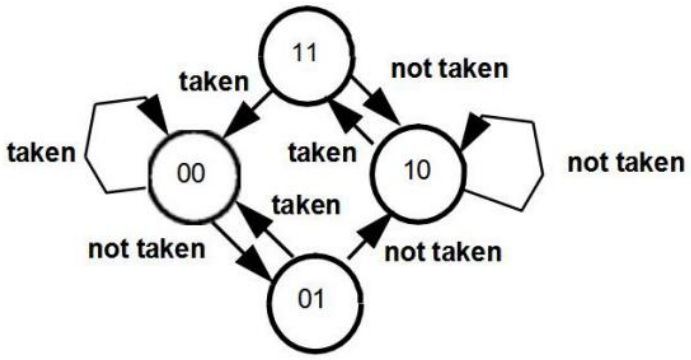
一位：定义一个 bool 类型的变量 branch_prediction 来表示分支预测结果，在 update_branch_prediction 函数中，我们将分支预测的结果存储在 branch_prediction 变量中。

```
#include <stdbool.h>

// 分支预测
bool branch_prediction = false;

// 分支预测器更新
void update_branch_prediction(bool prediction) {
    branch_prediction = prediction;
}

// 分支调整指令
void branch_adjust(int address) {
    // 分支预测
    if (branch_prediction) {
        // 执行分支跳转的操作
        // ...
```

```
    } else {
        // 执行分支不跳转的操作
        // ...
    }

    // 更新分支预测结果
    // 根据实际情况更新分支预测器
    update_branch_prediction(is_branch_taken);
}
```

在 branch_adjust 函数中，我们首先使用 branch_prediction 变量进行分支预测，根据预测结果执行相应的操作。然后，我们通过调用 update_branch_prediction 函数来更新分支预测结果。

两位：转移预测技术是一种通过硬件来预测分支指令的跳转方向的方法，以尽可能减少流水线停顿和分支指令带来的性能损失，动态转移预测：通过硬件中的转移预测缓冲器（Branch Prediction Buffer）来记录分支指令的历史执行信息和模式，并根据这些信息预测下一次的跳转方向，二位饱和计数器（Two-bit Saturating Counter）：使用一个 2 位的计数器来记录分支指令的历史执行情况。根据计数器的值，可以预测分支指令的跳转方向。



In state 1X we will guess not taken. In state 0X we will guess taken.

使用一个 branch_counter_t 枚举类型的变量 branch_prediction 来表示分支预测结果。通过 2 位计数器的方式，我们将分支预测结果分为 4 个状态：STRONGLY_NOT_TAKEN、WEAKLY_NOT_TAKEN、WEAKLY_TAKEN 和 STRONGLY_TAKEN。

```
#include <stdbool.h>

// 2 位计数器
typedef enum {
    STRONGLY_NOT_TAKEN = 0,
    WEAKLY_NOT_TAKEN = 1,
    WEAKLY_TAKEN = 2,
    STRONGLY_TAKEN = 3
} branch_counter_t;
```

```
10. 
11. // 分支预测器
12. branch_counter_t branch_prediction = STRONGLY_NOT_TAKEN;
13. 
14. // 分支预测器更新
15. void update_branch_prediction(bool is_branch_taken) {
16.     if (is_branch_taken) {
17.         if (branch_prediction < STRONGLY_TAKEN) {
18.             branch_prediction++;
19.         }
20.     } else {
21.         if (branch_prediction > STRONGLY_NOT_TAKEN) {
22.             branch_prediction--;
23.         }
24.     }
25. }
26. 
27. // 分支调整指令
28. void branch_adjust(int address) {
29.     // 分支预测
30.     bool is_branch_taken;
31.     switch (branch_prediction) {
32.         case STRONGLY_NOT_TAKEN:
33.         case WEAKLY_NOT_TAKEN:
34.             is_branch_taken = false;
35.             break;
36.         case WEAKLY_TAKEN:
37.         case STRONGLY_TAKEN:
38.             is_branch_taken = true;
39.             break;
40.     }
41. 
42.     // 执行分支操作
43.     if (is_branch_taken) {
44.         // 执行分支跳转的操作
45.         // ...
46.     } else {
47.         // 执行分支不跳转的操作
48.         // ...
49.     }
50. 
51.     // 更新分支预测结果
52.     update_branch_prediction(is_branch_taken);
53. }
```

在 update_branch_prediction 函数中，我们根据实际的分支执行情况更新分支预测器的状态。如果分支被正确预测，则将计数器的值向预测方向移动一位；如果分支没有被正确预测，则将计数器的值向相反方向移动一位。

6.要让取指级和访存级的 load 指令不需要多等一拍，可以对流水线设计方案进行修改，引入流水线握手机制。

1、在取指级（IF）阶段：

在译码级（ID）阶段，判断当前指令是否为 load 指令。

如果是 load 指令，则将读数据的地址请求早一拍发送给数据存储器（DM）。

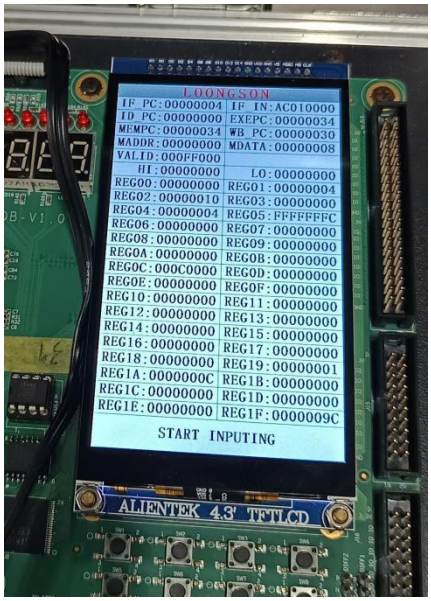
在译码级（ID）阶段，将读数据的地址请求发送给指令存储器（IM）。

2、在访存级（MEM）阶段：

在取指级（IF）阶段，将读数据的地址请求发送给数据存储器（DM）。

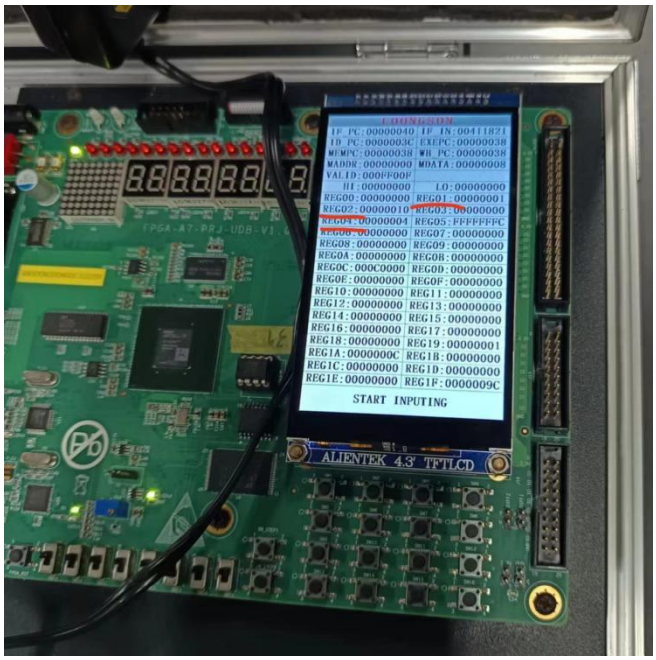
在访存级（MEM）阶段，从数据存储器（DM）中读取数据，并将数据传递给执行级（EX）阶段。

4、 实验结果分析



00H	sw $1,#0($0)	Mem[0000_0000H] =0000_0008H	AC010000	1010_1100_0000_0001_0000_0000_0000_0000

IF_PC 为 00000004，IF_IN 为 AC010000,该指令为 00H 的机器指令机器码，也可知取指 PC、译码 PC、执行 PC、访存 PC、回写 PC，在地址 00000000 处，数据为 00000008，对应 Mem[0000_0000H]=0000_00008H,汇编指令为 sw $1,#0($0),二进制源码为 1010_1100_0000_0001_0000_0000_0000_0000,可继续 clk 单步执行



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
40H	srl $4, $2,#2	[$4] = 0000_0004H	00022082	0000_0000_0000_0010_0010_0000_1000_0010

寄存器$1 结果 0000_0001H:指令地址 34H，二进制源码 0010_0100_0000_0001_0000_0000_0000_0001,指令含义：addiu $1，$0,#1,

寄存器$2 结果 0000_0010H:指令地址 38H，二进制源码 0000_0000_0000_0001_0001_0001_0000_0000,指令含义：sll $2,$1,#4

寄存器 $4 结果 0000_0004H: 指令地址 40H，二进制源码 0000_0000_0000_0010_0010_0000_1000_0010,指令含义：srl $4, $2,#2

5、 总结感想

流水线的设计需要权衡各方面因素，在设计流水线时，需要权衡各种因素，如流水线的深度、分支预测的准确性、前递技术的复杂性等。

数据依赖会导致流水线停顿，控制相关会导致流水线停顿，如果后一条指令依赖于前一条指令的结果，那么前一条指令的结果必须在后一条指令需要时才能被访问到，如果出现分支指令或其他控制指令，流水线中的指令可能会因为分支的结果而无效，导致流水线停顿。

加深了对结构冲突和数据冲突的理解，以及解决结构冲突和数据冲突的方法：

结构冲突：硬件资源满足不了指令重叠执行的要求而发生的冲突。

数据冲突：当指令在流水线中重叠执行时，因需要用到前面的指令的执行结果而发生的冲突。

结构冲突解决方案：设置相互独立的指令存储器和数据存储器，或者将统一的 Cache 分成独立的指令 Cache 和数据 Cache。

数据冲突解决方案：定向技术。某条指令产生计算结果之前，其它指令并不真正立即需要该计算结果，如果能够将该计算结果从其产生的地方直接送到其它指令需要它的地方，那么即可以避免停顿。