计算机系统结构实验报告 Lab06

简单的类 MIPS 多周期流水化处理器实现

姚宣骋 520021910431

2022 年 4 月 20 日

摘要

本实验以实验 5 实现的类 MIPS 单周期处理器为基础，对部分功能模块进行修改，并且添加了高速缓存模块作为内存与 CPU 之间的过渡，并且将支持的指令扩展到 31 条。并且，在本实验中还将单周期处理器改为流水线处理器，且使用前向通路（forwarding）和流水线停顿（stall）的方式解决可能的流水线冒险，并通过预测不转移（predict-not-taken）策略提高流水线性能。通过软件仿真运行指令，验证实验结果。

目录

1 实验目的

本次实验有 6 个目的：

- 理解 CPU Pipeline，了解流水线冒险 (hazard) 及相关性，设计基础流水线 CPU；
- 设计支持停顿的流水线 CPU，通过检测竞争并插入停顿 (Stall) 机制解决数据冒险、控制竞争和结构冒险；
- 增加前向传递（Forwarding）机制解决数据竞争，减少因数据竞争带来的流水线停顿延时；
- 通过 predict-not-taken 或延时转移策略解决控制冒险/竞争，减少控制竞争带来的流水线停顿延时
- 将 CPU 支持的指令数量从 16 条扩充为 31 条，使处理器功能更加丰富；

- 应用 Cache 原理，设计 Cache Line 并进行仿真验证；

2 原理分析

2.1 主控制器（Ctr）原理分析

主控制器会对指令的最高 6 位的 OpCode 进行解析，初步判定指令的类型并产生相应的处理器信号。在 Ctr 中，OpCode 可以大致分为 R 型指令；I 型指令中的 load 指令（lw）、store 指令（sw）与 branch 指令（beq）；J 型指令中的 jump 指令（j）。控制信号则如表 1 所示。

与实验 5 中不同的是，在单周期处理器中，jrSign 信号又 ALUCtr 产生。但是在流水线处理器中，为了提高流水线执行 JR 指令的性能，需要在指令译码阶段（ID），完成 JR 指令的识别。因此，在本实验中，我们调整主控制器模块的功能，使其能够支持产生 jrSign 信号。

表 1: Ctr 控制信号

信号	对应寄存器	具体说明
regDst	RegDst	目标寄存器的选择信号；低电平：rt 寄存器；高电平：rd 寄存器
aluSrc	ALUSrc	ALU 第二个操作数来源选择信号；低电平：rt 寄存器值，高电平：立即数拓展结果
memToReg	MemToReg	写寄存器的数据来源选择信号；低电平：ALU 运算结果，高电平：内存读取结果
regWrite	RegWrite	寄存器写使能信号，高电平说明当前指令需要进行寄存器写入
memRead	MemRead	内存读使能信号，高电平有效
memWrite	MemWrite	内存写使能信号，高电平有效
aluOp	ALUOp	3 位信号，发送给运算单元控制器 (ALUCtr) 用来进一步解析运算类型的控制信号
extSign	ExtSign	带符号扩展信号，高电平将对立即数进行带符号扩展
luiSign	LuiSign	载入立即数指令 (LUI) 信号，高电平说明当前指令为 LUI 指令
jumpSign	JumpSign	无条件跳转指令 (J) 信号，高电平说明当前指令是 J 指令
jrSign	JrSign	寄存器无条件跳转指令 (JR) 信号，高电平说明当前指令是 JR 指令
jalSign	JalSign	跳转并链接指令 (JAL) 信号，高电平说明当前指令是 JAL 指令
beqSign	BeqSign	条件跳转指令 (BEQ) 信号，高电平说明当前指令是 BEQ 指令
bneSign	BneSign	条件跳转指令 (BNE) 信号，高电平说明当前指令是 BNE 指令

其中 ALUOp 信号为两个二进制为的信号，其含义比一般信号复杂，如表 2 所示。此信号还需要经过 ALUCtr 处理后送入 ALU，才能对 ALU 进行控制。

表 2: ALUOp 信号具体含义与解析方式

ALUOp 的信号内容	指令	具体说明
101	R	ALUCtr 结合指令 Funct 段决定最终操作
000	lw,sw,addi,addiu	ALU 执行加法
001	beq	ALU 执行减法
011	andi	ALU 执行逻辑与
100	ori	ALU 执行逻辑或
111	xori	ALU 执行逻辑异或
010	slti	ALU 执行带符号数大小比较
110	sltiu	ALU 执行无符号数大小比较

OpCode 指令与控制信号的对应关系如表 3 所示一一对应。若出现其

表 3: OpCode 指令与控制信号的对应关系

OpCode	指令	aluOp	aluSrc	memRead	memToReg	memWrite	regDst	regWrite	extSign	branch	jump	jalSign
000000	R 型指令	101	0	0	0	0	1	1	0	0	0	0
100011	lw	000	1	1	1	0	0	1	1	0	0	0
101011	sw	000	1	0	0	1	0	0	1	0	0	0
000100	beq	001	0	0	0	0	0	0	1	1	0	0
000010	j	101	0	0	0	0	0	0	0	0	1	0
000011	jal	101	0	0	0	0	0	1	0	0	1	0
001000	addi	000	1	0	0	0	0	1	1	0	0	0
001001	addiu	000	1	0	0	0	0	1	0	0	0	0
001100	andi	011	1	0	0	0	0	1	0	0	0	0
001010	xori	111	1	0	0	0	0	1	1	0	0	0
001101	ori	100	1	0	0	0	0	1	1	0	0	0
001010	slti	010	1	0	0	0	0	1	1	0	0	0
001011	sltiu	110	1	0	0	0	0	1	0	0	0	0

余未设定指令，我们先暂且将所有控制信号置 0，即为空指令（nop）。

2.2 运算单元控制器（ALUCtr）原理分析

运算单元控制器（ALUCtr）对 ALUOp 信号和指令后 6 位 Funct 进行解析，给出控制 ALU 的控制信号 ALUCtrOut。输入与输出的对应关系如

表 4 所示。此外，为了支持 sll、srl、sra 指令，ALUCtr 还负责产生控制立

表 4: 运算单元控制器（ALUCtr）的解析方式

指令	ALUOp	Funct	ALUCtrOut	具体说明
w,sw,addi,addiu	000	xxxxxx	0010	ALU 执行加法运算
beq	001	xxxxxx	0110	ALU 执行减法运算
stli	010	xxxxxx	0111	ALU 执行带符号数大小比较
stliu	110	xxxxxx	1000	ALU 执行无符号数大小比较
andi	011	xxxxxx	0000	ALU 执行逻辑与运算
ori	100	xxxxxx	0001	ALU 执行逻辑或运算
xori	111	xxxxxx	1011	ALU 执行逻辑异或运算
add	101	100000	0010	ALU 执行加法运算
addu	101	100001	0010	ALU 执行加法运算
sub	101	100010	0110	ALU 执行减法运算
subu	101	100011	0110	ALU 执行减法运算
and	101	100100	0000	ALU 执行逻辑与运算
or	101	100101	0001	ALU 执行逻辑或运算
xor	101	100110	1011	ALU 执行逻辑异或运算
nor	101	100111	1100	ALU 执行逻辑或非运算
slt	101	101010	0111	ALU 执行带符号数大小比较
sltu	101	101011	1000	ALU 执行无符号数大小比较
sll	101	000000	0011	ALU 执行逻辑左移运算
sllv	101	000100	0011	ALU 执行逻辑左移运算
srl	101	000010	0100	ALU 执行逻辑右移运算
srlv	101	000110	0100	ALU 执行逻辑右移运算
sra	101	000011	1110	ALU 执行算术右移运算
srav	101	000111	1110	ALU 执行算术右移运算

即数输入的 shamtSign 信号。此外，如主控制器原理 2.1.1 节中所述，在本实验中，ALUCtr 模块不负责产生 jrSign 信号。

2.3 算术逻辑运算单元（ALU）原理分析

ALU 根据 ALUCtr 产生的控制信号 ALUCtrOut 对两个输入数进行对应的算数逻辑运算，并且输出运算结果以及部分控制信号（zero）。该信号用于与 branch 指令结合判断是否满足转移条件。ALU 执行的算术逻辑运算类型与运算单元控制信号 ALUCtrOut 的对应方式如表 5 所示。

表 5: ALU 的算术逻辑运算类型与 ALUCtrOut 的对应方式

ALUCtrOut	ALU 执行算数逻辑运算类型
0000	AND
0001	OR
0010	add
0011	Left Shift (logic)
0100	Right Shift (logic)
0110	sub
0111	set on less than (signed)
1000	set on less than (unsigned)
1011	XOR
1100	NOR
1110	Right Shift (arithmetic)

2.4 寄存器（Register）原理分析

寄存器（Register）的功能是暂时存储少量的数据，需要同时支持双通道数据的读取、写入功能。该模块内部共有 32 个寄存器，故两个读取地址 Readregister1、Readregister2 和一个写入地址 Writeregister 都是 5 位 2 进制数。此实验中，寄存器内存储的数据是 32 位的二进制数，故写入数据接口 WriteData 和两个读取数据接口 ReadData1、ReadData2 都是 32 位二进制数。寄存器的主要接口如图 1 所示。由于不确定 WriteReg,WriteData,RegWrite

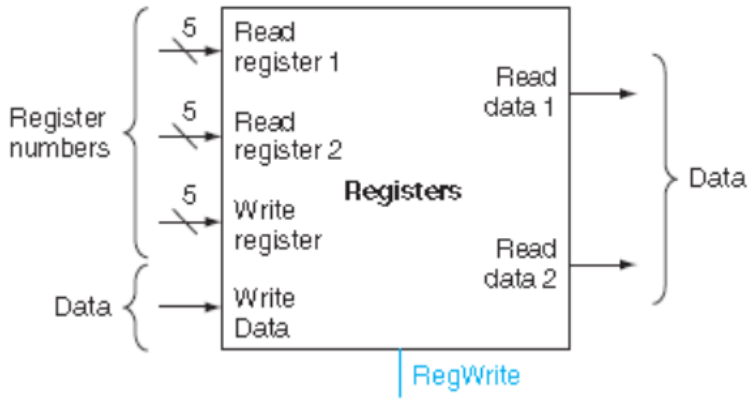

图 1: 寄存器（Register）的主要接口

信号的先后次序，我们采用时钟（样例里命名为 Clk）的下降沿作为写操作的同步信号，防止发生错误。

2.5 高速缓存（Cache）原理分析

高速缓存（Cache）的存在是为了减少处理器访问内存所需要的平均时间。其容量虽然远小于内存，但速度却可以接近处理器的频率。Cache 处于处理器和内存之间，当处理器进行写操作时，将直接写入内存中，且同时将缓存中对应数据块的 valid 位清零。当进行读操作时，首先会检查对应数据块的 valid 位是否有效以及 tag 是否与内存地址一致，若符合以上条件，则称为命中。若缓存命中，则直接返回缓存中存储的值；若未命中，则从内存中读取一个数据块的内容，覆盖缓存中对应的块，并将 valid 修改为有效。高速缓存的原理示意图如图 2 所示。

在本实验中，采用全相联映射策略，块大小微 128 位，共 16 个数据块，总容量位 64 个字。采用双层页表的检索方式，地址末两位表示块内偏移地址，地址末 2 至 5 位表示访问的块的序号。

为保持缓存与内存数据的一致性，本实验中高速缓存（Cache）的输入输出与数据存储器（Data Memory）的输入输出一致。

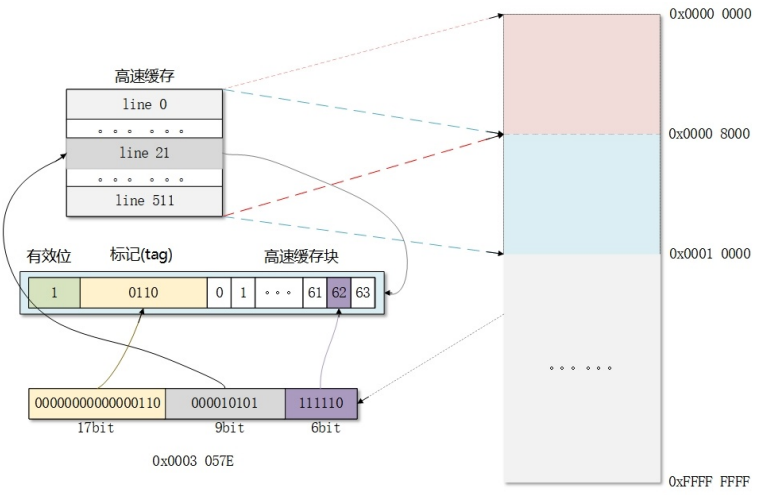

图 2: 高速缓存（Cache）原理示意图

2.6 数据存储器（Data Memory）原理分析

存储器（Data Memory）的功能是存储大量的数据，支持数据读取、数据写入的功能。该模块容量比寄存器和高速缓存都要大，但是访问速度比较慢，因此在本实验中，内存与缓存共同组成了数据存储模块，使得其可以兼顾容量与速度。在本实验中，我们将内存大小设定为 1024 个字，不过为了契合一般处理器的设计，地址输入接受 32 位，不过当且仅当地址大小不大于 1024 时访问才有效。存储器的主要接口如图 3 所示。

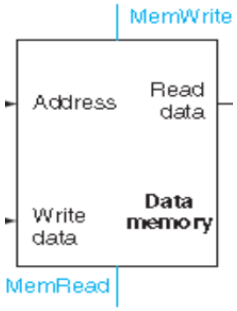

图 3: 数据存储器（Data Memory）的主要接口

对于写操作，设置 writeData 为 32 位大小。同寄存器，我们也采用时钟（样例里命名为 Clk）的下降沿作为写操作的同步信号，防止发生错误。

对于读操作，为了与高速缓存（Cache）中的块对应，本实验中数据存储模块（Data Memory）的输入 readData 为 128 位大小，即一次性返回 4 个字的内容。

2.7 带符号扩展单元（Sign Extension）原理分析

带符号扩展单元（Sign Extension）的功能是将指令中的 16 位立即数拓展为 32 位立即数。因此只需要根据立即数扩展规则，将这个 16 位带符号立即数的符号位填充在 32 位带符号立即数的高 16 位，再将低 16 位复制到 32 位带符号立即数的低 16 位即可。此外，我们用 signExt 的一位二进制信号决定立即数是否带符号。其主要接口如图 4 所示。

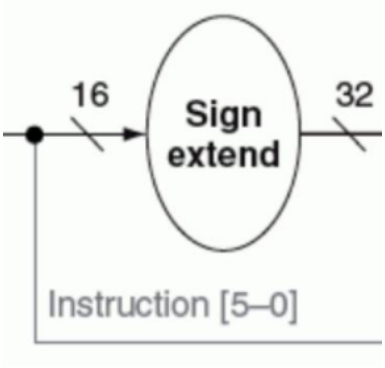

图 4: 带符号扩展单元（Sign Extension）的主要接口

2.8 数据选择器（Mux/RegMux）原理分析

数据选择器的功能即是接受两个输入信号，根据一个选择信号将其中一个输入信号输出。在本实验中，我们有 32 位选择器 Mux 和 5 位选择器 RegMux，分别用于数据和寄存器选取信号的选择。电路模型如图 5 所示。

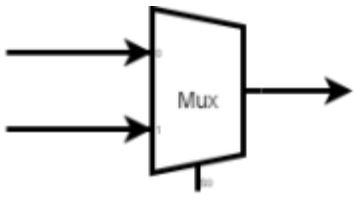

图 5: 数据选择器（Mux/RegMux）的主要接口

2.9 指令存储器 (Instruction Memory) 原理分析

本实验中处理器采取哈佛架构，即指令内存和数据内存相互分离。指令存储器的功能比数据存储器简单，只需要接受一个 32 位地址输入（从 PC）和一条 32 位的指令输出。电路模型如图 6 所示。

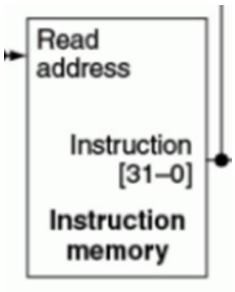

图 6: 指令存储器 (Instruction Memory) 的主要接口

2.10 五阶段流水线处理器-顶层模块 (TOP) 原理分析

2.10.1 流水线五阶段原理分析

五阶段流水线处理器可以将处理器的运行分为**取址-> 译码-> 执行-> 访存-> 回写**五个阶段。各个阶段之间需要用段寄存器临时保留上一阶段的执行结果和控制信号。各个阶段的名称和操作功能如下。

- **取指（IF）**：取指阶段主要包括 PC 寄存器与指令内存（Instruction Memory）模块，指令内存模块根据 PC 寄存器中的地址取出指令，将

指令内容和当前 PC 值存入 IF-ID 段寄存器中。

- **译码（ID）**：译码阶段主要包括主控制器（Ctr）、寄存器（Registers）、符号扩展模块（Sign Extension）。根据 IF-ID 段寄存器中储存的指令，主控制器产生对应的控制信号、寄存器模块读取寄存器数据，以及对于可能的立即数扩展至 32 位。以 R 指令为例，rs 和 rt 都是源寄存器地址，而 rd 是目的寄存器地址，而 I 指令则只有 rs 作为源寄存器地址，rt 作为目的寄存器地址。提前完成从 rt 和 rd 中选择目的寄存器的工作，便于后续的数据保存、传输，便于实现数据冒险判断和前向通路。

  为了提高流水线的执行效率，在译码阶段，处理器将提前完成无条件跳转指令 j、jal、jr 的操作。

  将控制信号、符号扩展结果、rs 与 rt 对应寄存器和目的寄存器编号、funct 和 shamt 的值以及当前指令的 PC 值存入 ID-EX 段寄存器。

- **执行（EX）**：执行阶段主要包括 ALU 控制器（ALUCtr）、ALU 以及一系列用于选择 ALU 输入数据的选择器（Mux/RegMux）。根据 ID-EX 段寄存器中的控制信号以及 ALUCtr 的扩展，ALU 将根据输入数据计算出结果，对于 beq、bne 指令，本阶段也将决定是否跳转。而对于 lui 指令，lui 选择器会将立即数作为运算结果的高 16 位，ALU 的计算结果被丢弃。

  将剩余的控制信号、ALU 运算结果、rt 与目的寄存器的编号存入 EX-MA 段寄存器中。

- **访存（MA）**：访存阶段主要包括高速缓存（Cache）和数据存储器（Data Memory）。根据 EX-MA 中的控制信号，需要访问存储器的指令将在该阶段对缓存进行读操作。

  将剩余控制信号、访存阶段结果或者执行阶段结果、目的寄存器编号存入 MA-WB 段寄存器中。

- **回写（WB）**：回写阶段主要包括寄存器模块，对于需要寄存器写入的指令，本阶段将完成寄存器写操作。其中，写入寄存器的编号在 ID 阶段已经确定，写入的数据为 MA 阶段的结果。

2.10.2 数据前向传递（Forwarding）原理分析

为了解决数据冒险（hazard），我们在电路中添加前向数据通路，使得在 EX 阶段可以从 EX-MA 段寄存器、MA-WB 段寄存器中提前于 WB 阶段就可以获取数据，以此提高效率。

2.10.3 停顿（Stall）机制原理分析

对于“读内存-使用”类型的数据冒险，前向传递无法避免解决。因此在必要时（如遭遇此类数据冒险时或控制冒险、结构冒险时），处理器需要发出 STALL 信号，使得 IF、ID 阶段停顿一个周期。

2.10.4 分支预测（predict-not-taken）原理分析

对于跳转指令可能带来的控制冒险，我们通过预测不转移（predict-not-taken）的方式来减少其带来的流水线停顿。即先预测所有的指令均不跳转，当预测错误的时候，则生成 NOP 信号清空 IF、ID 阶段。对于条件跳转指令和无条件跳转指令，以上清空策略均正确成立。此外，以下两点需要特别说明：

- IF-ID 段寄存器在接收到 NOP 信号是，需比较当前指令地址与跳转目标 PC 是否匹配。若两者匹配，则清空段寄存器，否则则不进行操作。这样设计可以减少近跳转指令的停顿周期。
- 为确保跳转目标 PC 的正确性，我们将条件跳转指令选择器放在无条件跳转指令之后。

2.10.5 顶层模块（Top）原理分析

根据五阶段流水线的原理，顶层模块将之前实现的各个模块相连接，实现完成了流水线 CPU 功能的一个处理器模块。其中的连线包括数据通路和控制通路，分别用于传输数据信号与控制信号。部分电路如图 7 所示。（图片经供参考，请以实际实现细节为准）

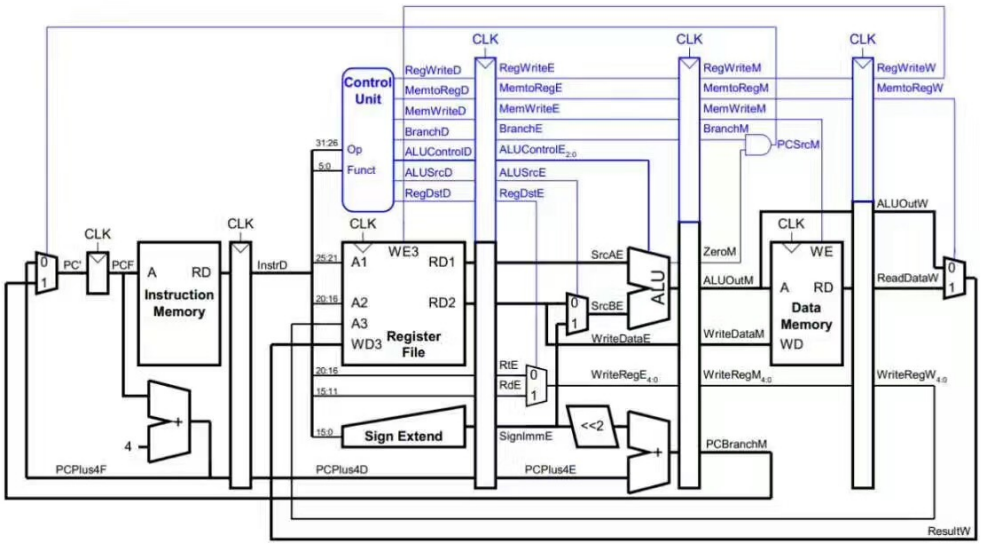

图 7: 顶层模块 (TOP) 原理图

3 功能实现

3.1 主控制器（Ctr）功能实现

根据 2.1 节中的控制信号对应表，使用 Verilog 中的 case 语句分别对各种情况进行实现。相比于实验 5，本实验还增加了部分跳转指令与 lui 指令的控制信号，同时将 jrSign 信号的产生集成到了主控制器中。部分代码如下，完整代码详见附录 A.1。

```
1  always @(opCode)
2      begin
3          case(opCode)
4          6'b000000: //R type
5  ␣␣␣␣␣␣␣␣begin
6  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣RegDst = 1;
7  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUSrc = 0;
8  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemToReg = 0;
9  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemRead = 0;
10 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemWrite = 0;
11 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUOp = 3'b101;
12                 ExtSign = 0;
13                 LuiSign = 0;
14                 JumpSign = 0;
15                 if (funct == 6'b001000) begin
16 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣RegWrite = 0;
17 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣JrSign = 1;
18 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣end else begin
19 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣RegWrite = 1;
20 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣JrSign = 0;
21 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣end
22 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣JalSign = 0;
23 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣BeqSign = 0;
24 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣BneSign = 0;
25 ␣␣␣␣␣␣␣␣end
26 ␣␣␣␣␣␣␣␣6'b000010: //jump
27             begin
28                 RegDst = 0;
```

```
29                  ALUSrc = 0;
30                  MemToReg = 0;
31                  RegWrite = 0;
32                  MemRead = 0;
33                  MemWrite = 0;
34                  ALUOp = 3'b000;
35  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣ExtSign␣=␣0;
36  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣......
37  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣MemWrite␣=␣0;
38  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUOp␣=␣2'b00;
39                  JumpSign = 0;
40                  JrSign = 0;
41              end
42              endcase
43          end
```

3.2 运算单元控制器（ALUCtr）功能实现

根据 2.2 节中的控制信号对应表，使用 Verilog 中的 casex 语句分别对各种情况进行实现。如 3.1 节中所述，与实验 5 相比，jrSign 信号被集成在主控制器单元，ALUCtr 不再负责产生 jrSign 信号。部分代码如下，完整代码详见附录 A.2。

```
1  always @ (aluOp or funct)
2      begin
3          ShamtSign = 0;
4          casex({aluOp,funct})
5              9'b000xxxxxx:␣␣//␣lw,sw,add,addiu
6  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut␣=␣4'b0010;      //add
```

```
7                   9'b001xxxxxx:  //  beq,bne
8   ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b0110;      //sub
9                   9'b010xxxxxx:  ///  stli
10  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b0111;
11                  9'b110xxxxxx:  ///  stliu
12  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b1000;
13                  9'b011xxxxxx:  //  andi
14  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b0000;
15                  9'b100xxxxxx:  //  ori
16  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b0001;
17                  9'b111xxxxxx:  //  xori
18  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b1011;
19                  ......
20                  9'b101100110:  //  xor
21  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b1011;
22                  9'b101100111:  //  nor
23  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b1100;
24                  9'b101101010:  //  slt
25  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b0111;
26                  9'b101101011:  //  sltu
27  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ALUCtrOut = 4'b1000;
28              endcase
29      end
```

3.3 算术逻辑运算单元（ALU）功能实现

根据 2.3 节中的控制信号对应表，使用 Verilog 中的 case 语句分别对各种情况进行实现，并用 if 语句对 zero 信号单独设置。部分代码如下，完整代码详见附录 A.3。

```
always @ (input1 or input2 or aluCtr)
begin
    case(aluCtr)
    4'b0000://AND
    ALURes=input1 & input2;
    4'b0001://OR
    ALURes=input1 | input2;
    4'b0010://ADD
    ALURes=input1 + input2;
    4'b0011://Left-shift
    ALURes=input2 << input1;
    4'b0100://Right-shift
    ALURes=input2 >> input1;
    4'b0101://xor
    ALURes=input1;
    4'b0110://SUB
    ......
    endcase
    if(ALURes==0)
        Zero=1;
    else
        Zero=0;
end
```

3.4 寄存器（Register）功能实现

寄存器(Register)可以一直进行读取操作,但是需要用 RegWrite 信号控制其是否允许写入。同原理 2.4 节中所言,由于不确定 WriteReg,WriteData,RegWrite

信号的先后次序，我们采用时钟（样例里命名为 Clk）的下降沿作为写操作的同步信号，防止发生错误。此外，在模块中还需对 32 个寄存器进行初始化置零处理以及接受 reset 指令后进行清零。部分实现代码如下，详见附录 A.4。

```
initial begin
        for(i=0;i<32;i=i+1)
                RegFile[i] = 0;
    end

    assign readData1 = RegFile[readReg1];
    assign readData2 = RegFile[readReg2];

    always @(negedge clk or reset)
    begin
        if(reset)
        begin
            for(i=0;i<32;i=i+1)
                RegFile[i] = 0;
        end
        else begin
            if(regWrite)
                RegFile[writeReg] = writeData;
        end
    end
```

3.5 高速缓存（Cache）功能实现

对于 CPU 而言，Cache 的接口与正常内存模块没有区别。同原理 2.7 节中所言，在进行读操作时，高速缓存模块首先检查是否命中，若未命中则从内存中读取数据保存至高速缓存内；在进行写操作时，直接写入内存模块，同时将缓存中对应块标记为无效。部分实现代码如下，详见附录 A.5。

```
always @(memRead or address or memWrite)
    begin
        if(memRead)
        begin
        if(validBit[cacheAddr] && tag[cacheAddr]
            == address[31:6])
            ReadData = cacheFile[address[5:0]];
            else begin
            cacheFile[{cacheAddr,2'b00}]
␣␣␣␣␣␣␣␣␣␣␣␣=␣dataFromMemFile[127:96];
␣␣␣␣␣␣␣␣␣␣␣␣if(address[1:0]==2'b00)
            ReadData=dataFromMemFile[127:96];

            cacheFile[{cacheAddr,2'b01}]
␣␣␣␣␣␣␣␣␣␣␣␣=␣dataFromMemFile[95:64];
␣␣␣␣␣␣␣␣␣␣␣␣if(address[1:0]==2'b01)
            ReadData=dataFromMemFile[95:64];

            cacheFile[{cacheAddr,2'b10}]
␣␣␣␣␣␣␣␣␣␣␣␣=␣dataFromMemFile[63:32];
␣␣␣␣␣␣␣␣␣␣␣␣if(address[1:0]==2'b10)
            ReadData=dataFromMemFile[63:32];

```

```
                cacheFile[{cacheAddr,2'b11}]
            = dataFromMemFile[31:0];
            if(address[1:0]==2'b11)
                ReadData=dataFromMemFile[31:0];

                validBit[cacheAddr] = 1;
                tag[cacheAddr] = address[31:6];
                end
        end
    end

    always @(negedge clk)
    begin
        if(memWrite)
            validBit[cacheAddr] = 0;
    end
```

3.6 数据存储器（Data Memory）功能实现

存储器（Data Memory）由 MemRead 和 MemWrite 分别用来控制读取和写入操作。同原理 2.6 节中所言，我们采用时钟（样例里命名为 Clk）的下降沿作为写操作的同步信号，防止发生错误。同样，对模块中的 64 个存储进行初始化置零操作，并且对地址范围加以检查限制，防止访问错误。部分实现代码如下，详见附录 A.6。

```
initial begin
    for(i=0;i<1024;i=i+1)
        memFile[i]=0;
    end
```

```
5       always @(memRead or address or memWrite)
6       begin
7           if(memRead)
8           begin
9               if(address<1023)
10                  ReadData = memFile[address];
11              else
12                  ReadData = 0;
13          end
14      end
15
16      always @(negedge clk)
17      begin
18          if(memWrite)
19              if(address<1023)
20                  memFile[address] = writeData;
21          if(memRead)
22              if(address<1023)
23                  ReadData = writeData;
24      end
```

3.7 带符号扩展单元（Sign Extension）功能实现

有符号扩展单元（Sign Extension）将 16 位有符号立即数扩展位 32 位有符号立即数。如原理 2.7 节中扩展规则所述，我们利用 Verilog 提供的拼接操作完成扩展。部分代码如下，详见附录 A.7。

```
1 assign data = signExt?{{16{inst[15]}},
2       inst[15:0]}:{{16{0}},inst[15:0]};
```

3.8 数据选择器（Mux/RegMux）功能实现

使用 Verilog 自带的三目运算符即可实现数据选择器的功能。核心代码如下，详见附录 A.8。

```
assign out = select?in1:in0;
```

3.9 指令存储器（Instruction Memory）功能实现

指令存储器只需要根据输入的 PC 地址输出对应命令即可。需注意，要对模块中的寄存器变量初始化置零。部分代码如下，详见附录 A.9。

```
initial begin
    for(i=0;i<1024;i=i+1)
        instFile[i]=0;
end
assign inst = instFile[address/4];
```

3.10 顶层模块（Top）功能实现

在顶层模块中，需声明需要的所有连接线路，将其对应连接到相应的模块上。声明代码如下。

```
//IF stage
    wire [31:0] IF_INST;     //INST

    //ID stage
    wire [12:0] ID_CTR_SIGNALS;
    wire [2:0] ID_CTR_SIGNAL_ALUOP;
    wire ID_JUMP_SIG;
    wire ID_JR_SIG;
```

```
9       wire ID_EXT_SIG;
10      wire ID_REG_DST_SIG;
11      wire ID_JAL_SIG;
12      wire ID_ALU_SRC_SIG;
13      wire ID_LUI_SIG;
14      wire ID_BEQ_SIG;
15      wire ID_BNE_SIG;
16      wire ID_MEM_WRITE_SIG;
17      wire ID_MEM_READ_SIG;
18      wire ID_MEM_TO_REG_SIG;
19      wire ID_REG_WRITE_SIG;
20      wire ID_ALU_OP;
21      wire[31:0] ID_REG_READ_DATA1; //REG_READ_DATA1
22      wire[31:0] ID_REG_READ_DATA2; //REG_READ_DATA2
23
24      wire[4:0] WB_WRITE_REG_ID; //WRITE_REG_ID
25      wire[4:0] WB_WRITE_REG_ID_AFTER_JAL_MUX;
26      //WRITE_REG_ID_AFTER_JAL_MUX
27      wire[31:0] WB_REG_WRITE_DATA;  //REG_WRITE_DATA
28      wire[31:0] WB_REG_WRITE_DATA_AFTER_JAL_MUX;
29      //REG_WRITE_DATA_AFTER_JAL_MUX
30      wire WB_REG_WRITE; //REG_WRITE
31
32      ......
33
34      wire BRANCH = EX_BEQ_BRANCH | EX_BNE_BRANCH;
35      //decide at EX stage
36      // forwarding
37      wire[31:0] EX_FORWARDING_A_TEMP;
```

```
38      wire[31:0] EX_FORWARDING_B_TEMP;
```

以高速缓存（Cache）模块的连接代码为例，如下。

```
1  Cache memory(
2          .clk(clk),
3          .address(EX2MA_ALU_RES),
4          .writeData(EX2MA_REG_READ_DATA_2),
5          .memWrite(MA_MEM_WRITE),
6          .memRead(MA_MEM_READ),
7          .readData(MA_MEM_READ_DATA)
8      );
```

此外，对于前向通路的实现，有代码如下

```
1  Mux forward_A_mux1(
2          .select(WB_REG_WRITE & (MA2WB_REG_DEST
3          ==ID2EX_INST_RS)),
4          .in0(ID2EX_REG_READ_DATA1),
5          .in1(MA2WB_FINAL_DATA),
6          .out(EX_FORWARDING_A_TEMP)
7      );
8
9      Mux forward_A_mux2(
10         .select(MA_REG_WRITE & (EX2MA_REG_DEST
11         ==ID2EX_INST_RS)),
12         .in0(EX_FORWARDING_A_TEMP),
13         .in1(EX2MA_ALU_RES),
14         .out(FORWARDING_RES_A)
15     );
16
```

```
17      Mux forward_B_mux1(
18          .select(WB_REG_WRITE & (MA2WB_REG_DEST
19          ==ID2EX_INST_RT)),
20          .in0(ID2EX_REG_READ_DATA2),
21          .in1(MA2WB_FINAL_DATA),
22          .out(EX_FORWARDING_B_TEMP)
23      );
24
25      Mux forward_B_mux2(
26          .select(MA_REG_WRITE & (EX2MA_REG_DEST
27          ==ID2EX_INST_RT)),
28          .in0(EX_FORWARDING_B_TEMP),
29          .in1(EX2MA_ALU_RES),
30          .out(FORWARDING_RES_B)
31      );
```

对于分支预测中的跳转指令判断，有代码如下

```
1       // Jump or branch
2       Mux jump_mux(
3           .select(ID_JUMP_SIG),
4           .in1(((IF2ID_PC + 4) & 32'hf0000000)+
5   ␣␣␣␣␣␣␣␣(IF2ID_INST␣[25␣:␣0]␣<<␣2)),
6   ␣␣␣␣␣␣␣␣.in0(IF_PC␣+␣4),
7   ␣␣␣␣␣␣␣␣.out(PC_AFTER_JUMP_MUX)
8   ␣␣␣␣);
9
10  ␣␣␣␣Mux␣jr_mux(
11  ␣␣␣␣␣␣␣␣.select(ID_JR_SIG),
12  ␣␣␣␣␣␣␣␣.in0(PC_AFTER_JUMP_MUX),
```

```
␣␣␣␣␣␣␣␣.in1(ID_REG_READ_DATA1),
␣␣␣␣␣␣␣␣.out(PC_AFTER_JR_MUX)
␣␣␣␣);

␣␣␣␣//␣EX␣stage
␣␣␣␣Mux␣beq_mux(
␣␣␣␣␣␣␣␣.select(EX_BEQ_BRANCH),
␣␣␣␣␣␣␣␣.in1(BRANCH_DEST),
␣␣␣␣␣␣␣␣.in0(PC_AFTER_JR_MUX),
␣␣␣␣␣␣␣␣.out(PC_AFTER_BEQ_MUX)
␣␣␣␣);

␣␣␣␣Mux␣bne_mux(
␣␣␣␣␣␣␣␣.select(EX_BNE_BRANCH),
␣␣␣␣␣␣␣␣.in1(BRANCH_DEST),
␣␣␣␣␣␣␣␣.in0(PC_AFTER_BEQ_MUX),
␣␣␣␣␣␣␣␣.out(PC_AFTER_BNE_MUX)
␣␣␣␣);
```

对于流水线处理器实现中的段寄存器写入、分支预测、停顿等功能，代码实现如下。

```
always @(reset)
    begin
        if (reset) begin
            // NOP = 1;
            IF_PC = 0;
            IF2ID_INST = 0;
            IF2ID_PC = 0;
            ID2EX_ALUOP = 0;
```

```
9                 ID2EX_CTR_SIGNALS = 0;
10                ID2EX_EXT_RES = 0;
11                ID2EX_INST_RS = 0;
12                ID2EX_INST_RT = 0;
13                ID2EX_REG_READ_DATA1 = 0;
14                ID2EX_REG_READ_DATA2 = 0;
15                ID2EX_INST_FUNCT = 0;
16                ID2EX_INST_SHAMT = 0;
17                ID2EX_REG_DEST = 0;
18                EX2MA_CTR_SIGNALS = 0;
19                EX2MA_ALU_RES = 0;
20                EX2MA_REG_READ_DATA_2 = 0;
21                EX2MA_REG_DEST = 0;
22                MA2WB_CTR_SIGNALS = 0;
23                MA2WB_FINAL_DATA = 0;
24                MA2WB_REG_DEST = 0;
25            end
26        end
27
28        //冒险处理
29        always @(posedge clk)
30        begin
31        NOP = BRANCH | ID_JUMP_SIG | ID_JR_SIG;
32        STALL = ID2EX_CTR_SIGNALS[2] &
33
34        ......
35
36
37        // MA - WB
```

```
38          if (!ID_JAL_SIG)
39          begin
40              MA2WB_CTR_SIGNALS <= EX2MA_CTR_SIGNALS[0];
41              MA2WB_FINAL_DATA <= MA_FINAL_DATA;
42              MA2WB_REG_DEST <= EX2MA_REG_DEST;
43          end
44
45          end
```

详细代码请参见附录 A.10。

4 结果验证

在激励文件 test.v 中，我们使用

```
1 $readmemh("mem_data.dat",top.data_mem.memFile);
2 $readmemb("mem_inst.dat",top.inst_mem.instFile);
```

两条命令分别采用相对路径的形式,从 mem_data.dat 文件和 mem_inst.dat 文件中直接读取数据和指令存入对应的数据存储器（Data Memory）和指令存储器（Instruction Memory）中。这两个文件应当位于.../lab06.sim/sim_1/behav/xsim 文件夹中。

指令代码根据 MIPS 指令对应的汇编代码编写，具体对应关系详见 https://blog.csdn.net/goodlinux/article/details/6731484

将寄存器（Registers）、高速缓存（Cache）、指令存储器（Instruction Memory）以及 PC 信号加入到仿真波形图的监视中，实验结果如下系列图中所示。

可以看出我们设计的类 MIPS 单周期处理器完成了所有功能。激励文件代码详见附录 B。

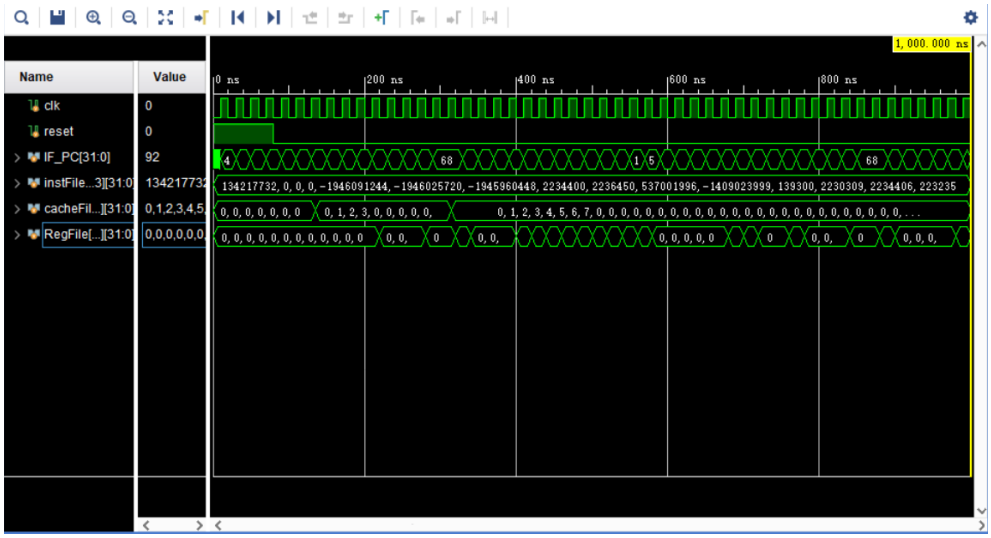

图 8: 总处理器仿真波形图

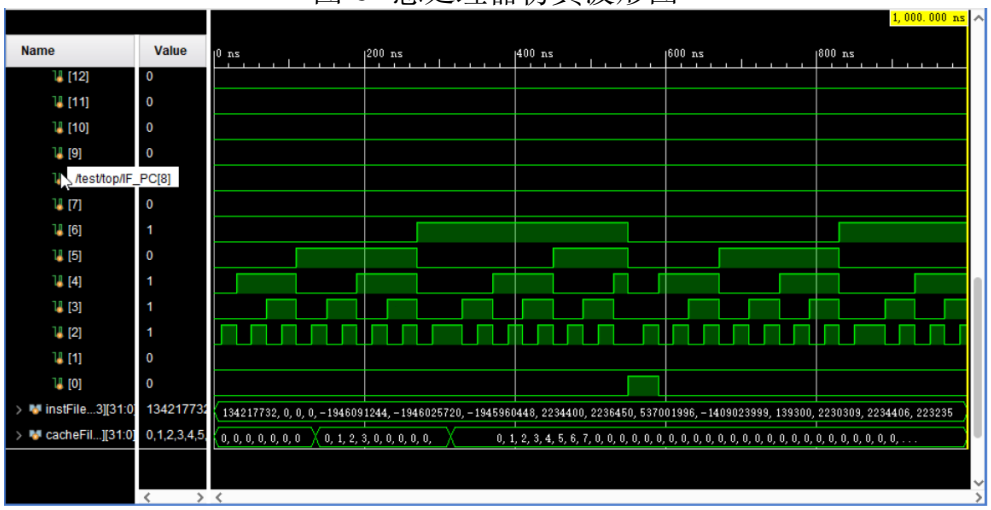

图 9: 仿真 PC 信号波形图

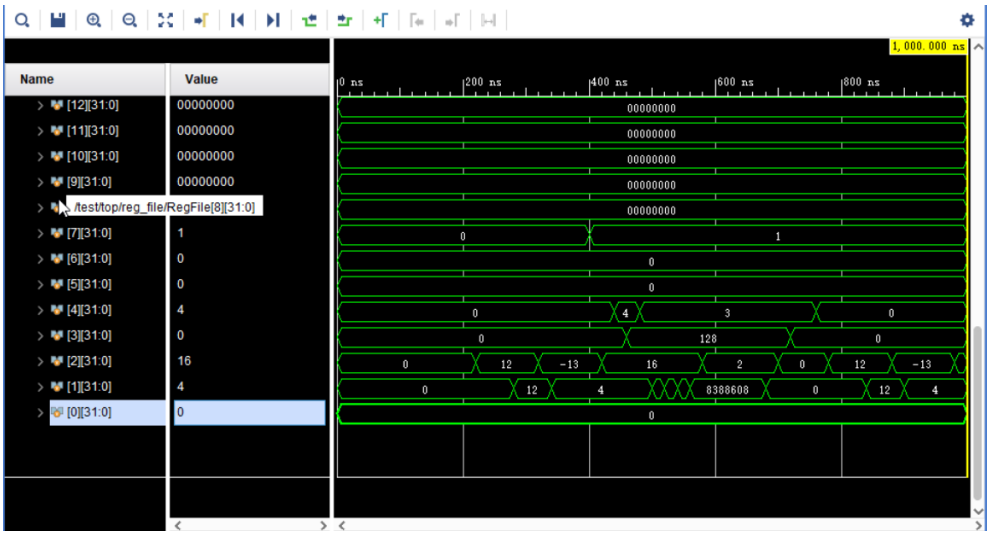

图 10: 仿真寄存器部分信号波形图

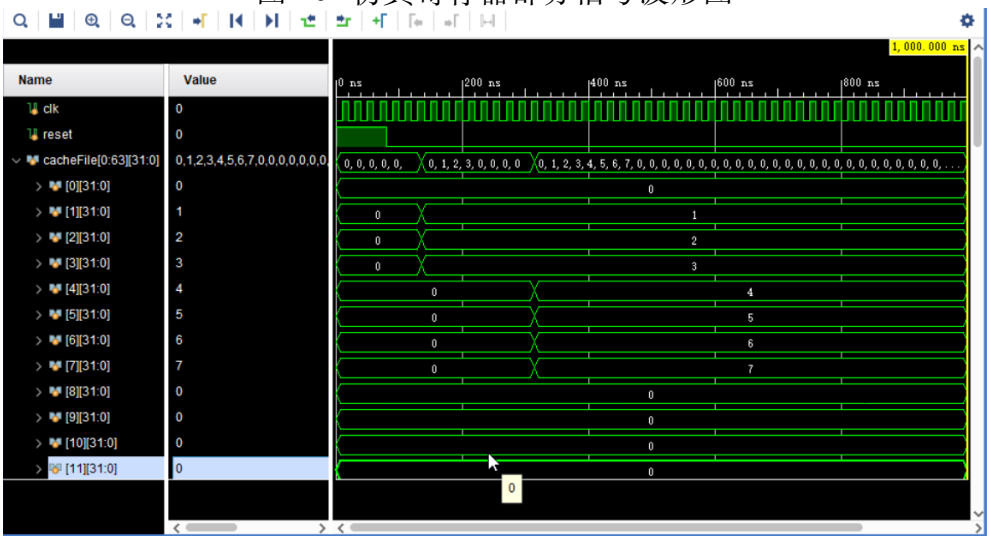

图 11: 仿真高速缓存部分信号波形图

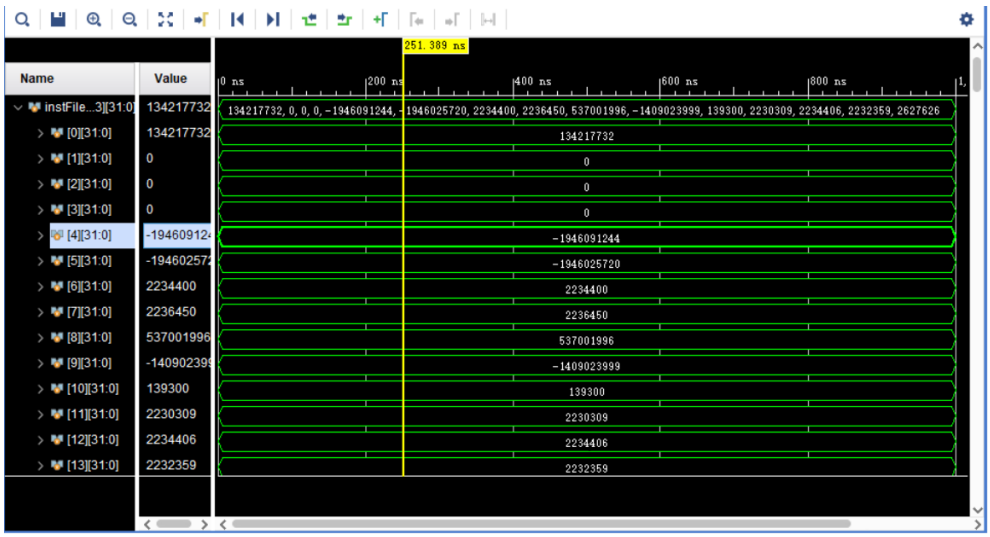

图 12: 仿真指令存储器部分信号波形图

5 总结反思

本次实验在实验 5 的基础上更进一步，将类 MIPS 单周期处理器扩展为类 MIPS 流水线处理器。相对于单周期处理器，流水线处理器的实现更加复杂，不仅仅是多了段寄存器以及更多的连线通路，使得电路的连接更加复杂。更是在对于异常情况（数据冒险、控制冒险、结构冒险）的预测于处理上更加的复杂多变。

然而，初看复杂异常的流水线处理器，在原先单周期处理器的基础上理解，却也不是毫无头绪。在实验指导书、《计算机组成与设计》以及学长经验的帮助下，我逐步将单周期处理器划分连接，并且再添加上前向通路、停顿等异常处理操作，最终完成了流水线处理器的设计与实现。

通过这次实验，我对于流水线处理器的运行过程更加深入地进行了了解，对于控制信号和数据信号在其中的传输与储存更加明晰；此外，还了解了前向通路、停顿。分支预测等对于流水线冒险的处理手段的原理与实现方式，着实让我收获颇多。

此外，经过了整整 6 次的实验，我对于 Verilog 语言和 Vivado 软件仿真从陌生到熟悉，对于硬件的设计更加了解，为我以后相关的学习和实验打下了良好的基础。

6 致谢

感谢本次实验中指导老师在课程微信群里为同学们答疑解惑；

感谢上海交通大学网络信息中心提供的远程桌面资源；

感谢计算机科学与工程系相关老师对于课程指导书的编写以及对于课程的设计，让我们可以更快更好地学习相关知识，掌握相关技能；

感谢电子信息与电气工程学院提供的优秀的课程资源。

A 设计文件完整代码实现

A.1 主控制器（Ctr）的代码实现

参见代码文件 Ctr.v

A.2 运算单元控制器（ALUCtr）的代码实现

参见代码文件 ALUCtr.v

A.3 算术逻辑运算单元（ALU）的代码实现

参见代码文件 ALU.v

A.4 寄存器（Register）的代码实现

参见代码文件 Register.v

A.5 高速缓存（Cache）的代码实现

参见代码文件 Cache.v

A.6 数据存储器（Data Memory）的代码实现

参见代码文件 dataMemory.v

A.7 有符号扩展单元（Sign Extension）的代码实现

参见代码文件 signext.v

A.8 数据选择器（Mux/RegMux）的代码实现

参见代码文件 Mux.v 和 RegMux.v

A.9 指令存储器（Instruction Memory）的代码实现

参见代码文件 InstMemory.v

A.10 顶层模块（Top）的代码实现

参见代码文件 Top.v

B 激励文件完整代码实现

参见代码文件 test.v。