目录

浙江大学实验报告

课程名称：计算机组成与设计　　指导老师：屈民军、唐奕　　成绩：_____

实验名称：MIPS 流水线 CPU 设计

一、实验目的

1. 通过设计并实现一个 32 位流水线 MIPS CPU，加深对 MIPS 指令系统和流水线 CPU 结构的理解；
2. 了解用软件方式(指令方法)设计数字系统的方法；
3. 进一步掌握数字系统的设计和调试方法；
4. 从系统实际应用的角度来考虑问题和进行相关设计。

二、实验内容

1. 实现五级指令流水 CPU；
2. 实现 33 条 MIPS 指令操作，包括：
   (1) 算术逻辑运算指令: ADD、ADDU、SUB、SUBU、AND、OR、NOR、XOR、SLT、SLTU
   (2) 移位指令: SLLV、SRLV、SRAV、SLL、SRL、SRA
   (3) 寄存器跳转指令: JR
   (4) 存储器访问指令: LW、SW
   (5) 立即数算术逻辑运算指令: ADDI、ADDIU、ANDI、ORI、XORI、SLTI、SLTIU
   (6) 分支指令: BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ
   (7) 无条件跳转指令: J
3. CPU 能够实现数据旁路、冒险控制单元、分支预测等，更好地解决各种“冲突”问题。

三、主要仪器设备

1. 装有 Modelsim 、vivado 软件的计算机
2. Nexys Video 开发板
3. 3.3V 的 DC 电源适配器

四、实验原理

（一）流水线的通用原理

1. 流水线的通用原理

将指令拆分成不同阶段后，如果每个阶段所用的硬件是相互独立的，那么可以在对应的硬件电路的每个阶段后添加寄存器，从而将单周期处理器改造成了流水线处理器。整个处理过程为：填充流水线---流水线填满---流水线排空。在流水线填满阶段，所有的硬件都处于工作状态，理论上性能为非流水线的 5 倍。

操作	描述
取指(fetch)	从存储器取指令，再更新PC
译码(decode)	从寄存器堆读出寄存器的值
执行(execute)	运算指令：进行算术逻辑运算；访存指令：计算存储器的地址
访存(memory)	load指令：从内存读出数据；store指令：将数据写入内存
写回(write-back)	将数据写回寄存器堆

将指令拆分成不同阶段后，如果每个阶段所用的硬件是相互独立的，那么可以在对应的硬件电路的每个阶段后添加寄存器，从而将单周期处理器改造成了流水线处理器。整个处理过程为：填充流水线---流水线填满---流水线排空。在流水线填满阶段，所有的硬件都处于工作状态，理论上性能为非流水线的 5 倍。

2. 五级流水线数据通路

(1) 取指令(IF)：根据程序计数器 pc 中的指令地址，从存储器中取出一条指令，同时，pc 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 pc，当然得到的“地址”需要做些变换才送入 pc。

(2) 指令译码(ID)：对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。

(3) 指令执行(EXE)：根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。

(4) 存储器访问(MEM)：所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(WB)：指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。

	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				IF	ID	EX	MEM	WB	
I5					IF	ID	EX	MEM	WB

图 1 Pipeline CPU Time Diagram

（二）数据冒险

1. 用转发(forwading)处理数据冒险

转发的思想是，与其暂停流水线直到写完成，不如简单将要写的值传到所需要的地方作为源操作数。对于 ALU 运算的指令，输入数据来自寄存器堆时，可能存在数据冒险。

为了解决这个问题，可以从 EX/MEM 或 MEM/WB 寄存器转发数据到 ALU 输入端。从 EX/MEM 转发的条件是前一条指令需要写非$0 寄存器，且写入地址和欲读取地址相等；从 MEM/WB 转发的条件是前两条指令需要写非$0 寄存器，欲读取的地址和写入地址相等，并且不能从前 EX/MEM 就近转发。

对于 JR 指令，同样存在数据冒险。由于跳转操作在 ID 阶段完成，如果检测到 JR 的目标寄存器和 ID_EX 段写入寄存器相同，需要从 ALUResult 转发数据;如果目标寄存器和 EX_MEM 段写入寄存器相同，需要从 MemReadData 转发数据;如果目标寄存器和 MEM_WB 段写入寄存器相同，则需要从 RegWriteData 转发数据。

2. 将暂停(stall)和转发(forwarding)结合来处理加载/使用冒险(load-use hazard)

上文只用转发的方式，来将还在内存中的值，直接传递给下一条指令作为源操作数。但是如果 lw 指令的写回寄存器和下一条指令读的寄存器是同一个，单纯的转发就解决不了。这时需要插入 bubble 来解决，本实验中使用 stall+forwarding 来解决 load-use hazard.

五、系统设计

根据实验原理，系统设计也分为五个阶段，分别是 IF、ID、EX、MEM 和 WB 阶段，各个阶段实现其不同的功能。下面进行详细介绍。

（一） 数据通路 Data Path

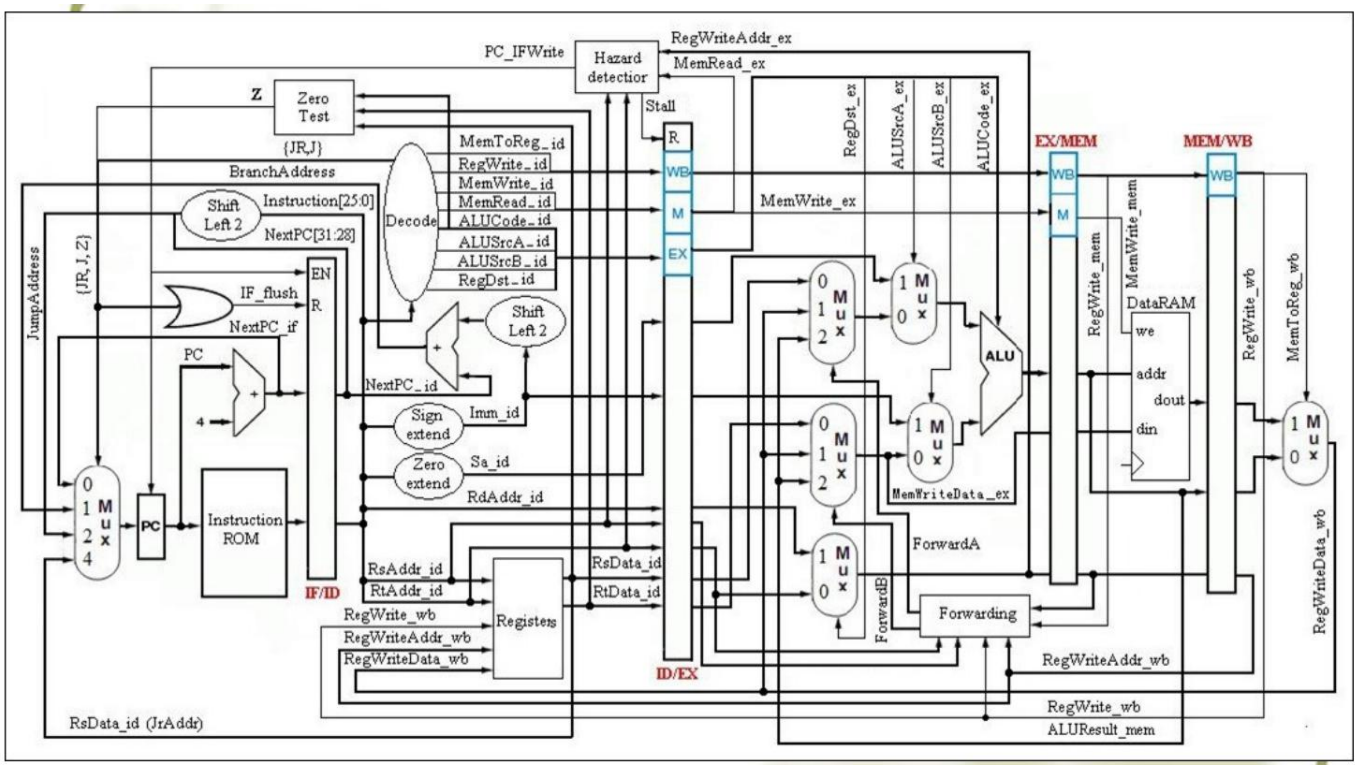


图 2 流水线 CPU 数据通路和控制线路图

上图是一个简单的基本上能够在流水线 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄存器地址（编号），读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在使能信号（RegWrite_wb）为 1 时，在时钟边沿触

发将数据写入寄存器。

控制信号功能如下表所示

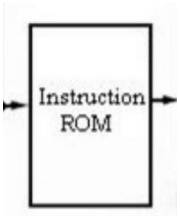
控制信号名	状态“0”	状态“1”
clk	时钟信号	
reset	复位信号，仿真时候，给各个寄存器提供初始值	
IF_flush	无效	IF/ID 寄存器中指令清零
PC_IFWrite	PC 不更改	PC 更改
Stall	无效	(插入流水线气泡)ID/EX 寄存器清零
PCSource[2:0]	000：pc <= NextPC，相关指令：add、addi、sub、or、ori、and、slt、sltiu、sll、sw、lw、beq(zero=0)、bltz(sign=0，或 zero=1)； 001：pc <= NextPC+(sign-extend)immediate，相关指令：beq(zero=1)、bltz(sign=1，zero=0)； 010：pc <= rs，相关指令：jr； 100：pc <= {NextPC[31:28] , addr[27:2] , 2'b00}，相关指令：j、jal；	
ForwardA[1:0]	决定操作数 A 来源 00：RsData_ex 01：RegWriteData_wb（二阶数据相关） 10：ALUResult_mem（一阶数据相关）	
ForwardB[1:0]	决定操作数 B 来源 00：RtData_ex 01：RegWriteData_wb（二阶数据相关） 10：ALUResult_mem（一阶数据相关）	
ALUSrcA	来自 RsData 或者来自转发数据	来自移位数(zero-extend)sa
ALUSrcB	来自 RtData 输出或转发数据	来自 sign 扩展的立即数
MemWrite	无写数据存储器操作	DataRAM 写使能
MemRead	无	数据存储器读使能
MemToReg	写回寄存器的数据来自 ALU 运算结果的输出	来自数据存储器(DataRAM)的输出
RegWrite	不写寄存器堆	寄存器堆寄存器写使能
RegDst	目标寄存器为 rt	目标寄存器为 rd
ALUCode[4:0]	ALU 操作控制信号（进行加减与或等等运算）	

（二） IF 阶段

IF: Instruction Fetch 取指令。用到部件: 4 选 1 多选器，PC 寄存器，Instruction ROM 指令存储器，Adder（32 位超前进位加法器）以及几个简单门电路。

主要子模块说明：

1. Instruction ROM：



端口声明：

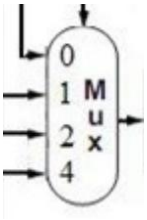
Input ：指令地址输入端口

Output：指令输出端口

功能：指令存储器。输入 PC 值，输出寄存器标号(PC 值)对应的寄存器的值。

实现：32 位的寄存器堆。（老师已经提供代码）

2. MUX 四选一



端口声明：

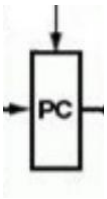
Input[31:0] ：J 指令跳转地址 JumpAddress 、JR 跳转地址 JR Address(RsData)、NextPC（PC+4）、分支地址 BranchAddress

Output：指令地址输出端口，输出信号 PC_MUX

功能：选择 PC 来源

实现：组合逻辑，控制信号：PCSrc

3. PC 寄存器



端口声明：

Input[31:0] ：PC_MUX

Output：PC 指令地址

en：使能端，使能信号 PC_IFwrite，PC_IFwrite=1 时，写 PC 有效；流水线阻塞时（stall）写使能无效（PC_IFwrite = ~stall = 0），PC 值保持不变。

功能：指令地址寄存器

实现：带使能端的 D 触发器

（三） ID 阶段

ID：Instruction Decode，译码，同时也取数。用到部件：指令译码器，寄存器堆，分支检测器，冒险检测器。

主要子模块说明：

1. Decode 译码

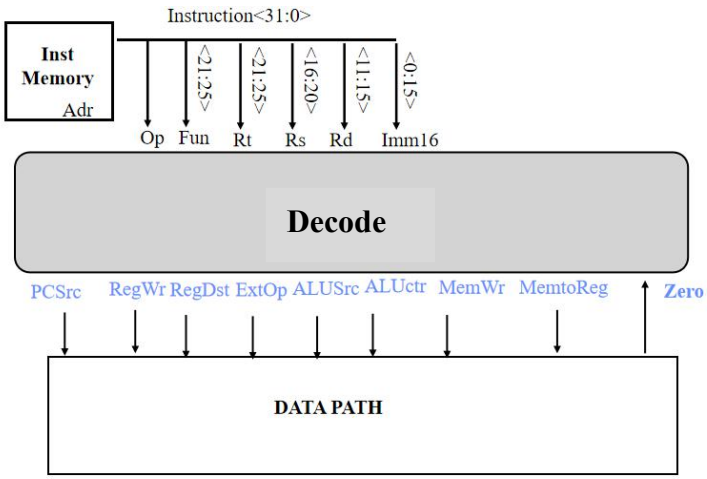


图 3 译码实现过程

端口声明：

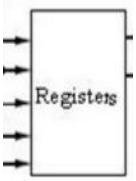
```
module Decode(
  // Outputs
  MemtoReg,
  RegWrite,
  MemWrite,
  MemRead,
  ALUCode,
  ALUSrcA,
  ALUSrcB,
  RegDst,
  J ,
  JR,

  // Inputs
  Instruction   );
```

功能：针对不同类型的 MIPS 指令实现译码，输出各个控制信号给数据通路。

2. RegFiles 寄存器堆

本系统均为上升沿触发，因此为了实现先写后读（Read After Write）功能，需要设计一个 ForwardDector 转发检测单元和两个选择电路来解决三阶数据冒险。（寄存器 Registers.v 代码已经提供）



端口说明：

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

RegWrite，写使能信号

RegWriteAddr，写入数据的地址输入端口

RegWriteData，写入的数据

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

功能：保存通用寄存器 R0~R7 的，以及其他 IN、SP、T、RA 寄存器的值；能够做到先写后读。

实现：1）Registers：读取时采用组合逻辑，输出寄存器标号对应的寄存器的值。写入时采用时序逻辑，时钟上升沿时写入。

2） ForwardDector ：当检测到写回寄存器等于取数寄存器时，即 RegWriteAddr_wb == RsAddr_id 或 RegWriteAddr_wb == RtAddr_id（写回寄存器不能为$0），转发条件成立，控制信号为 RsSel、RsSel，控制 ALU 操作数据来自转发数据，即 RegWriteData_wb。

3. BranchTest 分支检测

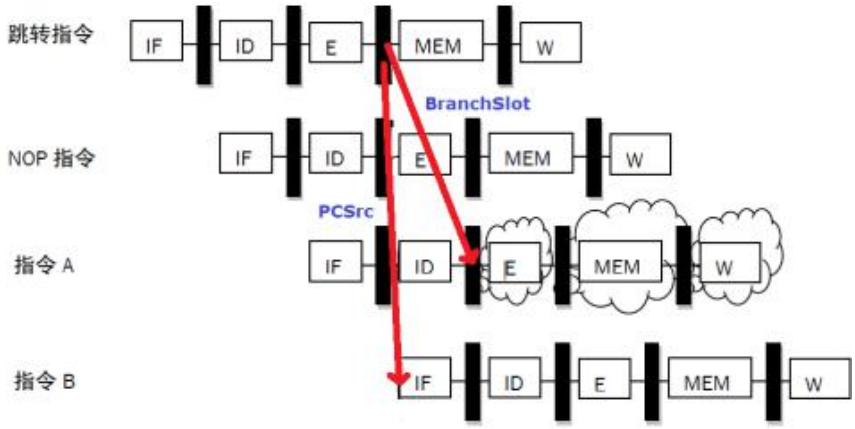


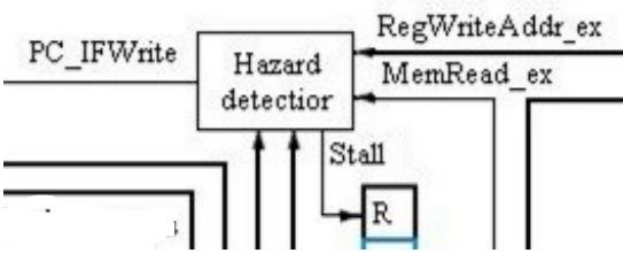
图 4 预测分支不发生失败

对于分支指令，有几种处理方式，插入延迟时间槽，或者分支预测等等。我们一般会默认预测跳转失败，若确实不发生跳转，则指令流水仍按原来顺序正常执行，流水线不受影响；若预测失败，跳转发生，同时插入气泡，使得已经执行至 ID 译码阶段的控制信号全部置零。如上图，若需要跳转时，指令 A 已经执行到了 ID 阶段，这时我们通过检测清空 ID_EXE 寄存器堆，保证运行的正确。

本系统使用另外一种方法，通过把分支检测提前到 ID 级，用花费额外的硬件面积为代价，来降低分支预测失败的代价，可以将流水线阻塞减少到 1 个 stall。

分支检测信号为 IF_flush，当分支条件成立，IF_flush 逻辑 true，将 IF/ID 流水线寄存器的 Intruction 清零，我用 NecxPC_id <= NextPC_if - 4 代替把 NecxPC_id 值清零，因为当系统出现异常时，系统调用处的程序计数器中的值被保存在异常程序计数器（EPC）中，处理器被置于管理态。

4. Hazard Detector 冒险检测



端口声明：

Input: MemRead_ex、 RegWriteAddr_ex、

RsAddr_id、 RtAddr_id

Output: stall、PC_IFWrite

功能：检测是否发生冒险和控制冲突，若有则插入气泡暂停流水线。一方面根据上一条指令是否为读内存且写当前指令使用的寄存器，判断是否需要插入 Load 延迟。另一方面根据前面第三条指令

是否跳转且成功，判断是否需要清空上一条指令的影响（前面第二条指令默认为 NOP 且执行）

实现：Hazard Detectort 根据 ID/EXE 与 EXE/MEM 流水线寄存器中的控制信号进行冒险检测

检测条件为 stall = MemRead_ex &&((RegWriteAddr_ex == RsAddr_id) || (RegWriteAddr_ex == RtAddr_id)) ；若有冒险发生，则 stall 为 1，插入 bubble，清空 ID/EX 流水线寄存器，同时 PC_IFWrite = ~stall = 0，保持 PC 与 IF/ID 寄存器中的值不变。

（四） EX 阶段

EX: execution ,执行，计算内存单元地址。用到部件：ALU，扩展器件（有/无符号扩展），转发单元

主要子模块说明：

1. ALU 算术逻辑单元

端口说明：

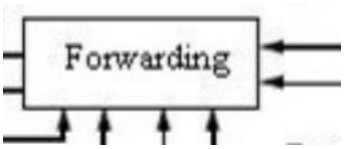
Input: A、B 两个操作数

Output: ALUResult 计算结果

功能：根据操作码执行相应的加减，移位，比较运算等

实现：组合逻辑，依据操作码（ALUCode）分别计算结果

2. Forwarding 转发单元



端口说明：

Input: RsAddr_ex、RtAddr_ex、
RegWrite_mem、RegWriteAddr_mem、
RegWrite_wb、RegWriteAddr_wb、

Output: ForwardA、ForwardB

输出信号 ForwardA、ForwardB 用于 ALU 的源操作数的选择控制信号。

功能：解决数据冲突，发生冲突时通过旁路传递数据，通过加入旁路后，不需要插入额外的 NOP。

根据 MIPS 指令发生冲突的位置可以将数据冲突分为 EXE 段冲突和 MEM 段冲突。其冒险检测条件为：

1a. EXE/MEM.RegisterRd = ID/EXE.RegisterRs

1b. EXE/MEM.RegisterRd = ID/EXE.RegisterRt

2a. MEM/WB.WBRegister = ID/EXE.RegisterRs

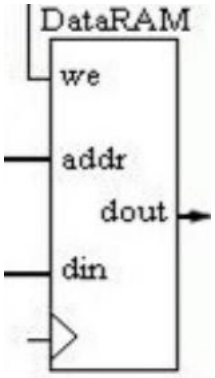
2b. MEM/WB.WBRegister = ID/EXE.RegisterRt

（五） MEM 阶段

用 Xilinx CORE Generator 生成的 DataRAM(2^6x32bit 的单端口 RAM)

对与 DataRAM，始终读取数据到 WB 级的数据选择器，因此 MemRead 读使能信号未作为该存储器的输入信号。

Data RAM：数据存储器



端口说明：

addr，数据地址输入端口

din，存储器数据输入端口

dout，存储器数据输出端口

we，数据存储器写使能信号，为 1 写

实现：由于是按字寻址，因此 addr 接 ALUResult_mem[7:2]，非 4 倍数的地址访问是不允许的。

（六） WB 阶段

WB:Write Back 写回，将数据写到寄存器中。

一个简单的 MUX 组合逻辑电路，通过 MemToReg_wb 控制信号选择写回寄存器的数据来源。

（七） 流水线寄存器模块

包括：IF_ID_Reg、 ID_EXE_Reg、 EXE_MEM_Reg、MEM_WB_Reg

功能：上升沿时存入前一阶段中被后续步骤所需要的数据，实现控制信号和数据的锁存功能，实现 CPU 的流水线功能。

（八） 系统文件清单：

Dictionary	File	Description
/src	/ff_lib.v	寄存器库
	/PipelineCPU.v	流水线 CPU 顶层文件
	/PipelineCPU_tb.v	顶层文件的 testbench
/src/IF	/IF.V	IF 级的顶层文件
	/IF_tb.v	testbench
	/InstructionROM.v	指令寄存器
/src/ID	/ID.v	ID 级顶层文件
	/BranchTest.v	分支检测
	/Decode.v	译码
	/Decode_tb.v	testbench
	/HazardDetector.v	冒险检测
	/RegFiles/ForwardDetector.v	Register 先写后读的转发检测
	/RegFiles/RegFiles.v	寄存器堆顶层文件
	/RegFiles/Registers.v	已提供的寄存器文件
/src/EX	/EX.v	EX 级顶层文件
	/ALU.v	ALU 单元

	/ALU_tb.v	testbench
	/Forwarding.v	解决数据冒险的转发单元
/src/WB	/WB.v	WB 级顶层文件
/src/PipeReg	/EX_MEM_reg.v	各级的流水线寄存器
	/ID_EX_reg.v	
	/IF_ID_reg.v	
	/MEM_WB_reg.v	
/vivado/Pipeline	/sim/DataRam.v	数据存储器
CPU.srcs/source	/DataRam_sim_netlist.v	用于仿真的网表文件
s_1/ip/DataRam	/dist_mem_gen_v8_0.v	时钟模块

六、实验结果测试

（一）ALU 测试

对 ALU 模块进行 modelsim 仿真，通过 list 可以更清晰地观察到 ALU 计算结果。

ns	ALUCode	A	B	ALUResult
0.000	5'h00	32'h00004012	32'h1000200f	32'h10006021
100.000	5'h00	32'h40000000	32'h40000000	32'h80000000
200.000	5'h01	32'hff0c0e10	32'h10df30ff	32'h100c0010
300.000	5'h02	32'hff0c0e10	32'h10df30ff	32'hefd33eef
400.000	5'h03	32'hff0c0e10	32'h10df30ff	32'hffdf3eff
500.000	5'h04	32'hff0c0e10	32'h10df30ff	32'h0020c100
600.000	5'h05	32'h70f0c0e0	32'h10003054	32'h60f0908c
700.000	5'h06	32'hff0c0e10	32'hfffffe0ff	32'h00000010
800.000	5'h07	32'hff0c0e10	32'hfffffe0ff	32'hff0ceeef
900.000	5'h08	32'hff0c0e10	32'hfffffe0ff	32'hff0ceeff
1000.000	5'h10	32'h00000004	32'hfffffe0ff	32'hfffe0ff0
1100.000	5'h11	32'h00000004	32'hfffffe0ff	32'h0fffffe0f
1200.000	5'h12	32'h00000004	32'hfffffe0ff	32'hffffffe0f
1300.000	5'h13	32'hfff000004	32'h700000ff	32'h00000001
1400.000	5'h14	32'hfff000004	32'h700000ff	32'h00000000

图 5 ALU 测试 list 文件结果

我在 testbench 文件添加了如下代码：

```
wire [31:0] ALUResult;

    integer w_file;
     initial
    begin
        w_file = $fopen("ALU_DataOut.txt");
        $fdisplay(w_file,"ALUCode          A          B       ALUResult");
    end
    always @ (ALUCode)
    begin
          $fdisplay(w_file,"%b    ",ALUCode,"%h        ",A,"%h    ",B,"%h",ALUResult);
```

```
end
```

用$fdisplay 将每次结果写入"ALU_DataOut.txt"文件。

ALU operation
add 加
and 与
xor 同或
or 或
nor 异或
sub 减
andi 立即数加
xori 立即数同或
ori 立即数或
sll 逻辑左移
srl 逻辑右移
sra 算数友谊
slt 比较
sltu 无符号比较

ALU_DataOut.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

ALUCode	A	B	ALUResult
00000	00004012	1000200f	10006021
00001	ff0c0e10	10df30ff	80000000
00010	ff0c0e10	10df30ff	100c0010
00011	ff0c0e10	10df30ff	efd33eef
00100	ff0c0e10	10df30ff	ffdf3eff
00101	70f0c0e0	10003054	0020c100
00110	ff0c0e10	ffffe0ff	60f0908c
00111	ff0c0e10	ffffe0ff	00000010
01000	ff0c0e10	ffffe0ff	ff0ceeef
10000	00000004	ffffe0ff	ff0ceeff
10001	00000004	ffffe0ff	fffe0ff0
10010	00000004	ffffe0ff	0ffffe0f
10011	ff000004	700000ff	fffffe0f
10100	ff000004	700000ff	00000001

图 6 ALU_DataOut.txt 文件内容

通过手动计算各个 ALU 操作的结果，与输出文本中的 ALU Result 进行验证。结果正确。

附文件清单：

Dictionary	File	Description
/sim/ALU	/ALU_DataOut.txt	ALU 计算结果
	/list.do	仿真 list 文件

（二）Decode 仿真

对译码模块进行 modelsim 仿真，list 信号值和自己输出的.txt 文件内容如下，对于控制信号结果可为 x 的（0,1 均可，对操作不影响），我都译作 0。

ps delta	Instruction	ALUCode	RegDst	ALUSrcA	ALUSrcB	MemWrite	MemRead	RegWrite	MemtoReg	J	JR
0 +2	32'h0800000b	5'b00000	0	0	0	0	0	0	0	1	0
100000 +1	32'h20080042	5'b00000	0	0	1	0	0	1	0	0	0
200000 +2	32'h01095022	5'b00101	1	0	0	0	0	1	0	0	0
300000 +2	32'h01485825	5'b00011	1	0	0	0	0	1	0	0	0
400000 +2	32'hac0b000c	5'b00000	0	0	1	1	0	0	0	0	0
500000 +1	32'h8d2c0008	5'b00000	0	0	1	0	1	1	1	0	0
600000 +2	32'h000c4080	5'b10000	1	1	0	0	0	1	0	0	0
700000 +2	32'h012a582b	5'b10100	1	0	0	0	0	1	0	0	0
800000 +2	32'h14000001	5'b01011	0	0	0	0	0	0	0	0	0
900000 +2	32'h1000fff4	5'b01010	0	0	0	0	0	0	0	0	0

图 7 Decode 仿真 list 文件内容

data_out.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

操作	op	funct	ALUCode	RegDst	ALUSrcA	ALUSrcB	MemWrite	MemRead	MemtoReg	RegWrite	J	JR
j	000010	001011	00000	0	0	0	0	0	0	0	1	0
addi	001000	000010	00000	0	0	1	0	0	0	1	0	0
sub	000000	100010	00000	1	0	0	0	0	0	1	0	0
or	000000	100101	00101	1	0	0	0	0	0	1	0	0
sw	101011	001100	00011	0	0	1	1	0	0	0	0	0
lw	100011	001000	00000	0	0	1	0	1	1	1	0	0
sll	000000	000000	00000	1	1	0	0	0	0	1	0	0
sltu	000000	101011	10000	1	0	0	0	0	0	1	0	0
bne	000101	000001	10100	0	0	0	0	0	0	0	0	0
beq	000100	110100	01011	0	0	0	0	0	0	0	0	0

图 8 data_out.txt 文件内容

将上述指令的仿真结果与指令的控制信号真值表（表 1）对应，测试结果正确。

	ALUSrcA	ALUSrcB	MemtoReg	RegWrite	MemRead	RegDst
add	0	0	0	1	0	1
addi	0	1	0	1	0	0
sub	0	0	0	1	0	1
ori	0	1	0	1	0	0
and	0	0	0	1	0	1
or	0	0	0	1	0	1
sll	0	0	0	1	0	1
slti	1	1	0	1	0	0
sw	0	1	0	0	0	0
lw	0	1	1	1	1	0
beq	0	0	0	0	0	0
bne	0	0	0	0	0	0
j	0	0	0	0	0	0

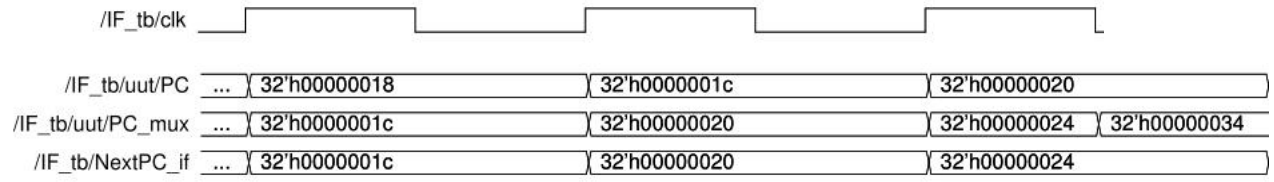
表 1 指令的控制信号真值表

附文件清单：

Dictionary	File	Description
/sim/Decode	/data_out.txt	Decode 结果
	/list.do	仿真 list 文件

（三）IF 测试

1. 地址自增测试

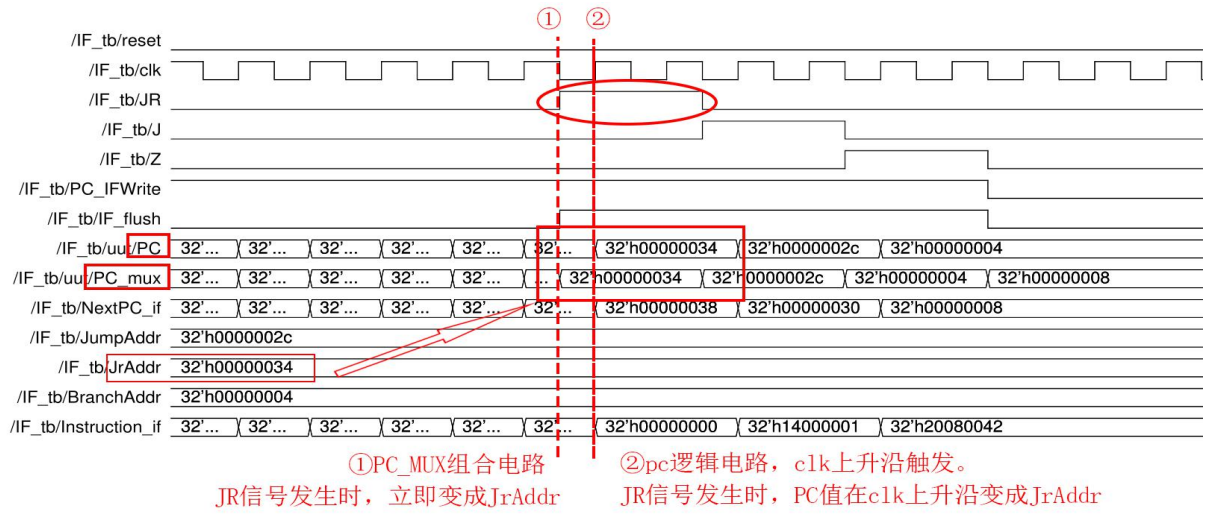


每一个时钟上升沿，PC 自增，NextPC = PC+4，测试正确。

2. JR、J、Branch 指令

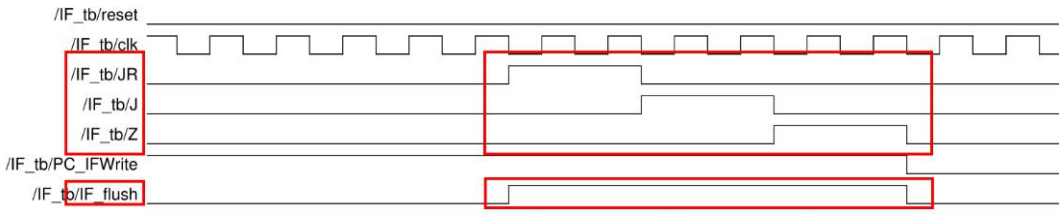
已知 JR/J/Branch 指令发生时，PC 值分别变为对应跳转地址（JrAddr/JumpAddr/BranchAddr）。

由下图分析，当 JR 指令发生时，在时钟上升沿 PC 值变成 JrAddr，即 32’h00000034。J 和 Branch 指令同理。因此对 JR/J/Branch 指令，IF 级逻辑功能正确。



3. IF_flush 控制信号

已知当 JR/J/Branch 指令发生时，IF_flush = JR ‖ J ‖ Z = 1，清空 ID/IF 寄存器内容。



由上图发现，IF_flush = JR ‖ J ‖ Z 测试正确。

（四）顶层文件仿真

前面已经分别对 IF 级的 PC 和 ID 级的译码，EX 级的 ALU 进行了测试，结果均正确。因此这部分主要检测电路的旁路单元对数据冒险的处理和 load-use Hazard 的处理。下面是对测试的程序代码，根据下表中的注释，分别验证各条指令执行结果，对比是否正确。

Instruction	**实例**	**指令代码**				
		op（6）	**rs(5)**	**rt(5)**	**rd(5)/immediate (16)**	**注释**
0x00000000	reset					
0x0800000b	j 11	000010	00000	00000	0000 0000 0000 1011	go to 44
0x00000000	nop	000000	00000	00000	0000 0000 0000 0000	
0x14000001	bne $0, $0, 1	101000	00000	00000	0000 0000 0000 0001	≠,转 NextPC+4
0x1000fff4	beq $0, $0, -12	100000	00000	00000	1111 1111 1111 0100	=,转 NextPC-48
0x00000000	nop	000000	00000	00000	0000 0000 0000 0000	
0x20080042	addi $8, $0, 0x42	001000	00000	01000	0000 0000 0100 0010	$8= $0+ 0x42
0x20090004	addi $9, $0, 0x4	001000	00000	01001	0000 0000 0000 0100	$9= $0+ 0x4
0x01095022	sub $10, $8, $9	000100	01000	01001	0101 0000 0010 0010	$10= $8 - $9
0x01485825	or $11,$10,$8	000000	01010	01000	0101 1000 0010 0101	$11= $10 | $8

0xac0b000c	sw $11,12($0)	101011	00000	01011	0000 0000 0000 1100	memory[12+$0] = $11
0x8d2c0008	lw $12,9($9)	100011	01001	01100	0000 0000 0000 1000	$12= memory[$9+8]
0x000c4080	sll $8, $12, 4	000000	00000	01100	0100 0000 1000 0000	$8 = $12<< 4
0x8d2b0008	lw $11,9($9)	100011	01001	01011	0000 0000 0000 1000	$11= memory[$9+8]
0x012a582b	sltu $9,4($1)	000000	01001	01010	0101 1000 0010 1011	if($9 < $10) $11=1, else $11 =0
0x0800000a	j 10	000010	00000	00000	0000 0000 0000 1010	go to 40
0x00000000	nop	000000	00000	00000	0000 0000 0000 0000	
0x0800000a	j 10					go to 40

表 2 测试程序代码

写成汇编程序如下：

测试程序

```
            j     later          //JAddr=2C
earlier:    addi  $t0,$0,42      //Reg[8]=0+42
            addi  $t1,$0,4       //Reg[9]=0+4
            sub   $t2,$t0,$t1    //Reg[10]=Reg[8]-Reg[9]
                                 //操作数B一阶数据相关，操作数A二阶数据相关
            or    $t3,$t2,$t0    //Reg[11]=Reg[10] |Reg[8]
                                 //操作数A一阶数据相关，操作数B三阶数据相关
            sw    $t3,0C($0)     //Mem[0+0C]=Reg[11]
            lw    $t4,08($t1)    //Reg[12]=Mem[Reg[9]+08]
            sll   $t0,$t4,2      //Reg[8]=Reg[12]<<2
                                 //数据阻塞冒险：Stall=1,PC_IFWrite=0
            lw    $t3,08($t1)    //Reg[11]=Mem[Reg[9]+08]
            sltu  $t3,$t1,$t2    // Reg[11]=Reg[9]<Reg[10]?1:0
done:       j     done           // JAddr=28
later:      bne   $0,$0,end      //分支条件不成立：Z=0
            beq   $0,$0,earlier  //分支条件成立：Z=1,BranchAddr=4
                                 //分支冒险，IF_flush=1
end:        nop
```

1. J 指令测试

Instruction	实例	预期结果	结果
0x0800000b	j 11	JumpFlag= 3’b010 go to 44	PC = 44 , JumpFlag= 3’b010 测试正确

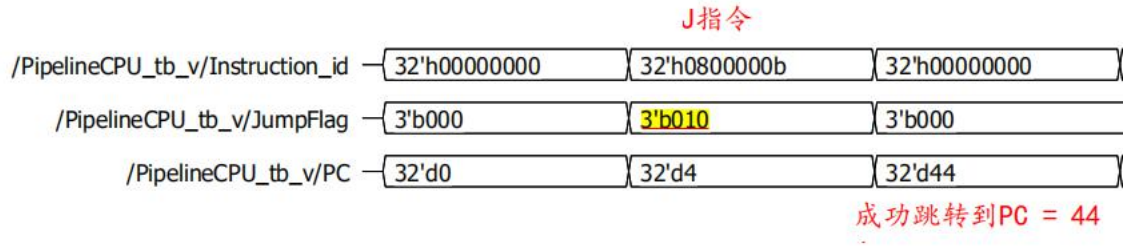


图 9 J 指令测试结果

2. 分支指令测试

Instruction	实例	预期结果	结果
0x14000001	bne $0, $0, 1	分支条件不成立 JumpFlag=0 PC 自增	PC=52 测试正确
0x1000fff4	beq $0, $0, -12	分支条件成立 JumpFlag=3'001 跳转到 NextPC - 48	PC=4 测试正确

图 10 分支指令测试

(对 IF_flush 的测试在上文 IF 仿真时已经测试，在此未作为测试信号）

已知每次将分支检测提前后，若分支发生，代价只有一个时钟周期，插入一个 stall，由下表，每一条成立的分支指令后面都插入了一条空指令（Instruction=32'b0）

JumpFlag	Instruction_id	
3'hx	32'hxxxxxxxx	
3'h0	32'h00000000	
3'h2	32'h0800000b	← J 指令
3'h0	32'h00000000	
3'h0	32'h14000001	
3'h1	32'h1000fff4	
3'h0	32'h00000000	←beq 指令成立
3'h0	32'h20080042	
3'h0	32'h20090004	
3'h0	32'h01095022	
3'h0	32'h01485825	
3'h0	32'hac0b000c	
3'h0	32'h8d2c0008	
3'h0	32'h000c4080	
3'h0	32'h000c4080	
3'h0	32'h000c4080	
3'h0	32'h8d2b0008	
3'h0	32'h012a582b	
3'h2	32'h0800000a	←J 指令
3'h0	32'h00000000	
3'h2	32'h0800000a	

表 3 分支指令测试 2

3. 立即数计算测试

Instruction	实例	预期结果	结果	
0x20080042	addi $8, $0, 0x42	ALUResult = 0x42 regs[8]=0x42	ALUResult = 0x42 regs[8]=32'h00000042	测试结果正确
0x20090004	addi $9, $0, 0x4	ALUResult = 0x4 regs[9]=0x4	ALUResult = 0x4 regs[9]=32'h00000004	

图 11 立即数操作测试

4. sub 指令测试

Instruction	实例	预期结果	结果
0x01095022	sub $10, $8, $9	ALUResult =$8-$9 =0x42-0x4 =0x3e regs[10]=ALUResult=0x3e	ALUResult = 0x42 regs[10]=32'h0000003e 测试结果正确

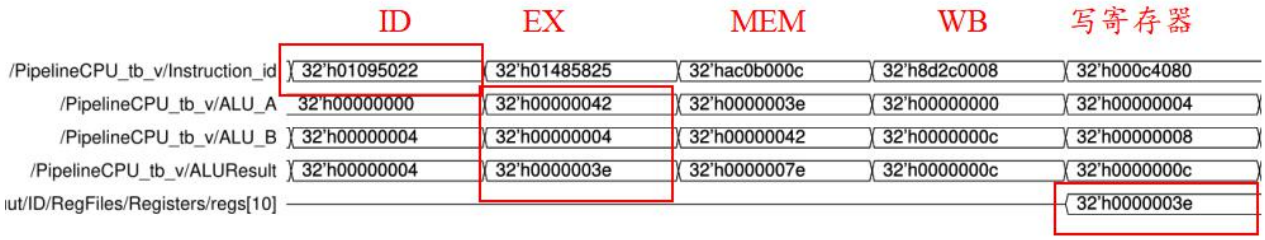


图 12 sub 测试结果

5. 转发测试

or 指令后立即就是 sw，二者存在对寄存器$11 的数据以来，存在冲突。

Instruction	实例	预期结果	结果
0x01485825	or $11,$10,$8		由图 12 可知 ForwardA=2'h0 ForwardB=2'h2 sw 执行结果见下一条测试 测试结果正确
0xac0b000c	sw $11,12($0)	ForwardA=2'b00 操作数 A 来自 RsData_ex 即$0 ForwardB=2'b10，操作数 B 来自 ALUResult_mem sw 指令执行成功，流水线无 nop	

对于解决数据冲突的转发测试，主要通过观察 ForwardA、ForwardB 控制信号的值来测试。

图 13 转发测试结果

6. 存取指令测试

Instruction	实例	预期结果	结果
0xac0b000c	sw $11,12($0)	memory[12+$0] = $11	regs[12]=regs[11] =32'h0000007e sw 和 lw 都成功执行 测试正确
0x8d2c0008	lw $12,9($9)	$12= memory[$9+8] = DataRAM[12] 即$12=$11	

对 sw 指令的测试通过观察 lw 结果来验证，因为 sw 的存数据的位置是 lw 加载数据的地址，均为 DataRAM[12]，因此只要 lw 取出来的数据和$11 寄存器内容一样则 sw 指令正确。

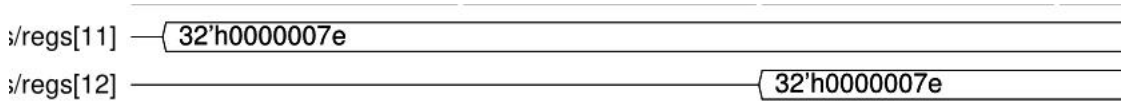


图 14 sw/lw 测试

7. load-use 冒险测试

当 lw 指令之后一条指令依赖 lw 目的寄存器的值时，存在 load-use 冒险冲突。

Instruction	实例	预期结果	结果
0x8d2c0008	lw $12,9($9)	$12= memory[$9+8]	
0x000c4080	sll $8, $12, 4	流水插入一个 bubble ，sll 指令被延迟一拍执行	1）ID/EX 级寄存器清空，流水线阻塞现象： ALU_A=ALU_B=ALUReslut=0 stall=1 、PC_IFWrite=0、IF_flush=1 2）sll 指令 Instrction 不变，被延迟一个 cycle

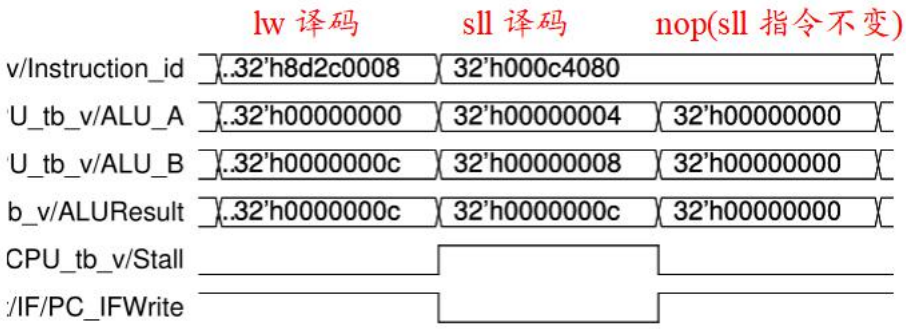


图 15 load-use 测试结果

8. 所有指令测试信号结果

JumpFlag	Instruction_id	ALU_A	ALU_B	ALUResult	PC	MemDout_mem	Stall
3'hx	32'hxxxxxxxx	32'hxxxxxxxx	32'hxxxxxxxx	32'hxxxxxxxx	32'hxxxxxxxx	32'hxxxxxxxx	1'hx
3'h0	32'h00000000	32'h00000000	32'h00000000	32'h00000000	32'h00000000	32'h00000000	1'h0
3'h2	32'h0800000b	32'h00000000	32'h00000000	32'h00000000	32'h00000004	32'h00000000	1'h0
3'h0	32'h00000000	32'h00000000	32'h00000000	32'h00000000	32'h0000002c	32'h00000000	1'h0
3'h0	32'h14000001	32'h00000000	32'h00000000	32'h00000000	32'h00000030	32'h00000000	1'h0
3'h1	32'h1000fff4	32'h00000000	32'h00000000	32'h00000000	32'h00000034	32'h00000000	1'h0
3'h0	32'h00000000	32'h00000000	32'h00000000	32'h00000000	32'h00000004	32'h00000000	1'h0
3'h0	32'h20080042	32'h00000000	32'h00000000	32'h00000000	32'h00000008	32'h00000000	1'h0
3'h0	32'h20090004	32'h00000000	32'h00000042	32'h00000042	32'h0000000c	32'h00000000	1'h0
3'h0	32'h01095022	32'h00000000	32'h00000004	32'h00000004	32'h00000010	32'h00000000	1'h0
3'h0	32'h01485825	32'h00000042	32'h00000004	32'h0000003e	32'h00000014	32'h00000000	1'h0
3'h0	32'hac0b000c	32'h0000003e	32'h00000042	32'h0000007e	32'h00000018	32'h00000000	1'h0
3'h0	32'h8d2c0008	32'h00000000	32'h0000000c	32'h0000000c	32'h0000001c	32'h00000000	1'h0
3'h0	32'h000c4080	32'h00000004	32'h00000008	32'h0000000c	32'h00000020	32'h00000000	1'h1
3'h0	32'h000c4080	32'h00000000	32'h00000000	32'h00000000	32'h00000020	32'h00000000	1'h0
3'h0	32'h000c4080	32'h00000000	32'h00000000	32'h00000000	32'h00000020	32'h0000007e	1'h0
3'h0	32'h8d2b0008	32'h00000002	32'h0000007e	32'h000001f8	32'h00000024	32'h00000000	1'h0
3'h0	32'h012a582b	32'h00000004	32'h00000008	32'h0000000c	32'h00000028	32'h00000000	1'h0
3'h2	32'h0800000a	32'h00000004	32'h0000003e	32'h00000001	32'h0000002c	32'h0000007e	1'h0
3'h0	32'h00000000	32'h00000000	32'h00000000	32'h00000000	32'h00000028	32'h00000000	1'h0
3'h2	32'h0800000a	32'h00000000	32'h00000000	32'h00000000	32'h0000002c	32'h00000000	1'h0

这是顶层 testbench 的信号结果，与老师给出的测试结果真值表对应，测试正确。

七、 讨论、心得

本次实验已经结束，回顾这次实验，收获颇丰。

1. 【前期准备】

除了在理论课上学习的关于 PipelineCPU 的知识，还需花一定时间设计总体的电路框架，首先感喟老师们精妙的设计思维，整个 CPU 基本的运行流程用一张图和几个表就表示清楚了，而且对计算机指令等基本介绍思路清晰，让我们这些 CPU 入门者很受用。

通过这次实验，增强了我对流水线 CPU 运行流程的理解。正所谓“纸上得来终觉，绝知此事要躬行”，实验课将理论知识实例化，实现项目的过程本身就是一个纠错的过程。有些时候理论知识出现了错误，自己可能感觉不到，实验过程波形的错位可以察觉自己对理论知识掌握的错误，因此做这种大项目对于我们学生来说是很有好处的。学以致用才是根本。

2. 【代码编写】和【调试环节】。

关于【代码编写】

1）模块划分

实验中我尽可能的将模块细分，使得每个 module 中的逻辑都相对比较简单。模块的细分有利于程序的编写，同时如果是工程较大，团队合作编写时，有利于分工编写程序之后再使用顶层 module 组装起来。

2）使用 parameter 来定义常量

Verilog 中的 parameter 语句类型 c 语言中的 define 语句，通过对于各种控制信号指定名称，不仅使得程序的可读性提高，同时也利于程序调试。这样写还能够大大减少程序修改的代价。例如我要修改一个控制信号的取值，只需要修改 parameter 定义语句，而不需要修改内部程序。

3）写 verilog 代码要有硬件的概念

关于写代码要考虑实际电路的问题主要有两方面：

一是对组合逻辑与时序逻辑的合理分解，一开始编写程序时我往往直接按照软件编写的思路，只考虑电路逻辑，而不考虑电路能否综合实现。结果导致在仿真时出现各种奇怪的错误。

例如在 else 里面套用 case 语句；一个 if 搭配了两个 else（if...else ... else，应该是 if .. else if ..else..）；还有我在写 always 模块时，其中的逻辑过于混杂，既包含时序逻辑又包含组合逻辑的功能，不符合目前的编译器能够综合的语言风格。因此后来编程时注意分析好电路功能，采用组合逻辑实现逻辑功能

需求，采用时序逻辑控制状态的改变。

要有所要写的 module 在硬件上如何实现的概念，而不是去想编译器如何去解释这个 module. 比如在决定是否使用 reg 定义时，会问自己物理上是不是真正存在这个 register, 如果是，它的 clock 是什么? D 端是什么？Q 端是什么？有没有清零和置位？同步还是异步？对 Verilog，没有“编译”的概念，而只有综合的概念。

二是考虑本次实验实际用到的板子的构造来编写代码。在理论课上，讲到出现 flush 的时候，为了避免写寄存或存储器，只需要将所有的写控制信号（RegWrite 和 MemWrite）置为 0，而不用关心其他的控制信号，但是在询问了老师之后，知道我们用到的板子是用一个多位的寄存器来存储流水线寄存器里面的内容，如果只清除写控制信号，还需要单独清除寄存器的某些位，如果全部清零就简单得许多，所以最终采用了将 ID/EX 寄存器全部清零的处理。

4）当分支条件成立，IF_flush 控制将 IF/ID 流水线寄存器清零，在实际处理中，如果系统出现异常时，CPU 需要处理异常，系统调用处的程序计数器中的值被保存在异常程序计数器（EPC）中，处理器被置于管理态。因此我只清空了 IF/ID 寄存器中的 Intruction，变成空指令，用 NecxPC_id <= NextPC_if - 4 代替把 NecxPC_id 值清零。

关于【调试环节】

1） 遵循实验核心思想，我对各个模块进行单独调试，例如 ALU 模块，译码模块，取址模块，分模块纠错有点类似于“断点法 debug”，很大程度上减小了在 top 仿真出错时再去找 bug 的工作量，如果确保了这几个模块无误就可以缩小 debug 的范围。但自己 debug 还是花了好久好久时间，debug 很沮丧，感觉“剪不断，理还乱”，可见，程序员是十分需要耐心的，这种碎片化知识的整合能力是通过一次次的纠错查错、与 bug 作斗争的过程中培养起来的，这种不骄不躁的品格对我们的人生发展是有好处的。

2） 除了上面几个模块，HazardDetector 和 Forwading 模块也是很容易出错的，我就花了大量时间在后期的各项调试上，确保 30 条指令的正确性。过程中多次遇到了接口不匹配接口缺失的问题，除此之外还有很多大小写问题，老师给出的接口 Stall 和我自己的 stall 首字母大小写不同，NotePad++ 软件的大小写提示性能太差给我带来了很多的额外的调试工作量。由于代码重复性比较高，往往在把一些信号名字复制粘贴之后没有更改而导致很多很难发现的错误，尤其是大量涉及到 Rs 和 Rt 二者的部分。

3） 在调试部分当有些信号出现 xxxx，我在 Objects 窗口采用 Event Traceback -> show Driver 查看该信号的驱动信号，发现 bug 出在它的驱动信号 not logged 或者找不到驱动信号（如下图），导致这种结果有原因接口名字错误（大小写错误等等）；虽然在 module 中定义了 input/output，但是没有写入到 module (...) 函数参量声明括号中。如下