计算机系统结构实验 实验报告

实验6 简单的类 MIPS 多周期流水线处理器设计与实现

陈文迪 519021910071

摘 要

本实验建立在实验3、实验4与实验5的基础之上，实现了一个支持31条基本MIPS指令的多周期流水线处理器。通过修改已有的功能模块，增加段寄存器、冒险检测、前向通路等新模块，新增信号与数据通路的方式，支持了停顿（Stall）、前向通路（Forwarding）、分支提前预测以及预测不转移（predict-not-taken）机制，解决了数据和控制冒险，降低了流水线停顿延时，提高了处理器性能。此外，通过修改数据内存单元模块，为处理器增加了缓存（Cache），丰富了存储层级，降低了访存延时。本实验通过功能仿真来验证实验结果。

目录

1 实验目的

1. 理解 CPU Pipeline，了解流水线冒险（hazard）及相关性，设计基础流水线CPU。
2. 设计支持 Stall 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、控制冒险和结构冒险。
3. 在2.的基础上，增加 Forwarding 机制解决数据竞争，减少应数据竞争带来的流水线停顿延时，提高流水线处理器性能。可以考虑将 Stall 与 Forwarding 结合起来实现。
4. 在3.的基础上，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，进一步提高处理器性能。可以考虑 将2.、3.和4.结合起来设计。
5. 在4.的基础上，将CPU支持的指令数量从16条扩充到31条，使处理器功能更加丰富。（选做）
6. Cache的设计。（选做）
7. 使用功能仿真验证正确性。

2 原理分析

2.1 处理器基础模块的设计

2.1.1 主控制器Ctr的重新设计

主控制器Ctr通过解析指令的opCode字段（31：26）给处理器的其他部件输出正确的控制信号。在实验5中，我们已经设计了主控器的大部分控制信号，但由于本次实验需要将16条指令扩展到31条，因此需要对信号进行进一步扩展。事实上，经过对比指令我们可以发现，对主控器而言，仅需要新增一个BranchNot信号用于区分条件跳转指令是beq还是bne即可。最终所设计的主控制器信号如**表1**。

表 1 主控制器控制信号及其含义

控制信号	含义
RegDst	控制有关写入寄存器的多路选择器（0：选择rt；1：选择rd）
ALUSrc	控制ALU二号输入口的多路选择器（0：选择寄存器读出的第二个数据；1：选择立即数）
MemtoReg	控制写入寄存器的多路选择器（0：选择ALU输出；1：选择内存读出的数据）
RegWrite	寄存器写使能（0：否；1：是）
MemRead	内存读使能（0：否；1：是）
MemWrite	内存写使能（0：否；1：是）
Branch	是否为条件跳转指令（0：否；1：是）
BranchNot	条件跳转指令是beq还是bne（0：beq；1：bne）
Jump	是否为无条件跳转指令（0：否；1：是）
ALUOp[3:0]	传递给ALUCtr的控制信号
Jal	是否为Jal指令（0：否；1：是）
Sign	是否需要进行有符号扩展（0：否；1：是）

其中加粗的信号即为本次实验中所新增的BranchNot信号。此外，在本实验中，我们还需要为新增加的指令指定ALUOp。对于ALUOp的具体设计我们沿用实验5中所采用如下方法：对于大部分I型指令，将opCode字段的最后四位作为ALUOp，再交由ALUCtr作具体解析；对于小部分后四位ALUOp被其他指令占用的指令，可以为其分配空余的ALUOp；对于R型指令，则直接指定其ALUOp为0010。此外，由于我们已经通过BranchNot信号区分beq和bne，此处的ALUOp二者可以共用。我们所设计的ALUOp与指令的对于关系如**表2**。

表 2 各指令ALUOp的定义及其含义

指令	ALUOp	含义
lw	0011	表示ALU应进行加法
sw	1011	表示ALU应进行加法
beq	0100	表示ALU应进行减法
bne	0100（与beq共用）	表示ALU应进行减法
addi	1000	表示ALU应进行有符号加法
addiu	1001	表示ALU应进行无符号加法
andi	1100	表示ALU应进行按位与
ori	1101	表示ALU应进行按位或
xori	1110	表示ALU应进行按位异或
slti	1010	表示ALU应进行有符号比较置位
sltiu	0001（后四位被占用）	表示ALU应进行无符号比较置位
lui	1111	表示ALU应进行高位填充
R型、j、jal	0010	表示该条指令为R型指令，ALU的操作需要通过Funct字段来判断

其中加粗的指令是本次实验中新增的I型指令。对于除上述31条指令的其他输入，我们将所有信号都置为0，事实上依照我们对信号的定义，信号为0时不会产生任何**写操作**，因此不会对寄存器、内存中的内容产生任何影响。

2.1.2 运算单元控制器ALUCtr的重新设计

运算单元控制器ALUCtr通过接收主控制器Ctr所产生的ALUOp信号，结合指令的Funct段来进一步解析指令。此处的处理与实验5中类似，因此不再赘述。我们所设计的信号解析逻辑如**表3**。

表 3 运算单元控制器的信号解析逻辑

指令	ALUOp	Funct	ALUCtr	shamt	jr	含义
and	0010	100100	0000	0	0	表示ALU应进行按位与
or	0010	100101	0001	0	0	表示ALU应进行按位或
xor	0010	100110	1001	0	0	表示ALU应进行按位异或
nor	0010	100111	0101	0	0	表示ALLU应进行按位或非
add	0010	100000	0010	0	0	表示ALU应进行有符号加法
addu	0010	100001	1010	0	0	表示ALU应进行无符号加法
sub	0010	100010	0110	0	0	表示ALU应进行有符号减法
subu	0010	100011	1110	0	0	表示ALU应进行无符号减法
slt	0010	101010	0111	0	0	表示ALU应进行有符号比较置位
sltu	0010	101011	1111	0	0	表示ALU因进行无符号比较置位
sll	0010	000000	0011	1	0	表示ALU应进行逻辑左移操作
srl	0010	000010	0100	1	0	表示ALU应进行逻辑右移操作
sra	0010	000011	1000	1	0	表示ALU应进行算术右移操作
sllv	0010	000100	0011	0	0	表示ALU应进行逻辑左移操作
srlv	0010	000110	0100	0	0	表示ALU应进行逻辑右移操作
srav	0010	000111	1000	0	0	表示ALU应进行算术右移操作
jr	0010	001000	0000	0	1	表示该条指令为jr指令
addi	1000	xxxxxx	0010	0	0	表示ALU应进行有符号加法
addiu	1001	xxxxxx	1010	0	0	表示ALU应进行无符号加法
andi	1100	xxxxxx	0000	0	0	表示ALU应进行按位与
ori	1101	xxxxxx	0001	0	0	表示ALU应进行按位或
xori	1110	xxxxxx	1001	0	0	表示ALU应进行按位异或
slti	1010	xxxxxx	0111	0	0	表示ALU应进行有符号比较置位
sltiu	0001	xxxxxx	1111	0	0	表示ALU应进行无符号比较置位
lui	1111	xxxxxx	1100	0	0	表示ALU应进行高位填充
lw	0011	xxxxxx	0010	0	0	表示ALU应进行加法
sw	1011	xxxxxx	0010	0	0	表示ALU应进行加法
beq、**bne**	0100	xxxxxx	0110	0	0	表示ALU应进行减法
j、jal	0010	xxxxxx	0000	0	0	表示ALU应进行按位与

其中加粗的指令是本次实验中新增的指令。至此，我们通过Ctr和ALUCtr对全部的31条指令完成了解析，并产生所有所需的信号量。

2.1.3 算术逻辑单元ALU的重新设计

算术逻辑单元ALU通过接受ALUCtr所输出的控制信号，执行所指定的运算类型，并产生Zero信号（由于分支提前预测模块的存在，此处的Zero信号并未发挥作用）。此外，为了区分有符号运算与无符号运算，我们新增了一个overflow信号用于检测溢出：当有符号运算发生溢出时，overflow为1；未发生溢出时，overflow则为0。我们所设计的算术逻辑单元运算类型对照表如**表4**。该部分内容将在3.1.1节中给出具体的实现。

表 4 算术逻辑单元运算类型对照表

ALUCtr	ALU运算类型
0000	按位与（and）
0001	按位或（or）
0010	有符号加法（add）
0110	有符号减法（sub）
0111	有符号比较置位（slt）
0011	逻辑左移（sll）
0100	逻辑右移（srl）
1001	按位异或（xor）
0101	按位或非（nor）
1010	无符号加法（addu）
1110	无符号减法（subu）
1111	无符号比较置位（sltu）
1000	逻辑右移（sra）
1100	高位填充（lui）

2.1.4 寄存器Registers的重新设计

本次实验中，我们基本上可以直接采用在实验4中所设计的寄存器模块。但是，在一些实现细节上需要进行修改，例如：由于jr指令和jal指令的存在，我们需要新增新的信号通路，以实现在进行读操作时对jr指令进行特殊处理，在进行写操作时对jal指令进行特殊处理。该部分内容将在3.1.2节中给出具体的实现。

2.1.5 数据内存单元Data Memory与缓存Cache的重新设计

对于数据内存单元，我们需要为实验4中已设计的模块新增缓存功能。对于数据内存单元与缓存的映射策略，我们采用**直接映射**（如**图1**），使用数据内存地址的最后四位作为Tag；对于缓存的更新策略我们采用**Write-back**策略并对每一缓存块使用一个脏位（dirty bit）予以实现；此外，我们还使用一个有效位（valid bit）来表示某一缓存块是否有效。该部分内容将在3.1.3节中给出具体的实现。

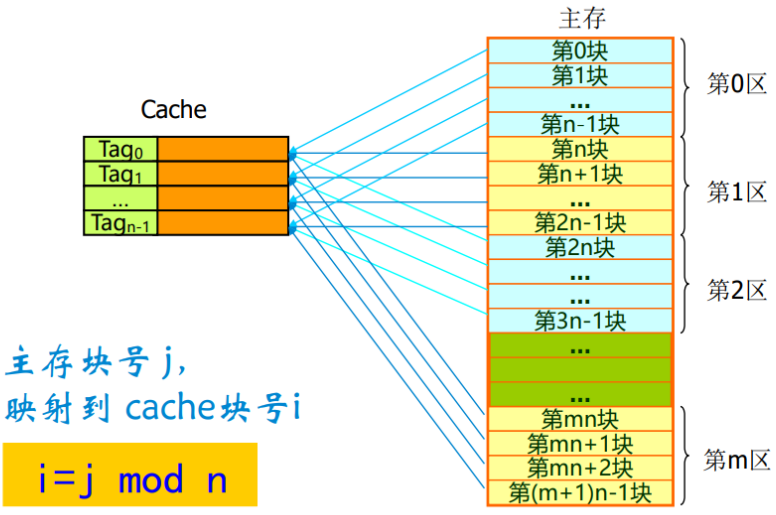

图 1 缓存直接映射原理图

2.1.6 指令内存单元Instruction Memory的设计

该部分内容与实验5基本一致，故不再赘述。

2.1.7 有符号扩展模块Sign Extension的设计

该部分内容与实验5基本一致，故不再赘述。

2.1.8 无符号扩展模块Zero Extension的设计

该模块与有符号扩展模块基本类似，但是其仅需要支持无符号扩展模式即可。

2.1.9 多路选择器Mux的重新设计

本实验在实验五的基础上新增了两种新的多路选择器类型。其一时用于程序计数器PC的输入口的四选一32位宽度多路选择器；其二是用于前向通路Forwarding的三选一32位宽度多路选择器。

2.1.10 程序计数器PC的设计

本实验中，程序计数器PC的设计与实验五中基本类似，不过由于我们需要实现停顿Stall功能，我们需要新增一个Stall信号用于控制PC的更新。该部分内容将在3.1.4节中给出具体的实现。

2.2 流水线的设计及冒险的解决

本实验中，我们继续采用模块化设计的思想。如**图2**，为了实现流水线处理器，我们需要将单周期处理器分割为IF（Instruction fetch），ID（Instruction decode），EX（Execute），MEM（Memory access），WB（Write back）五大部分，并在两个部分之间添加段寄存器以实现信号、数据在各个阶段之间的传递。此外，为了解决冒险，我们需要增加如下模块：冒险检测模块（Hazard detect）、分支提前预测模块（Zero test）和前向通路模块（Forwarding）。

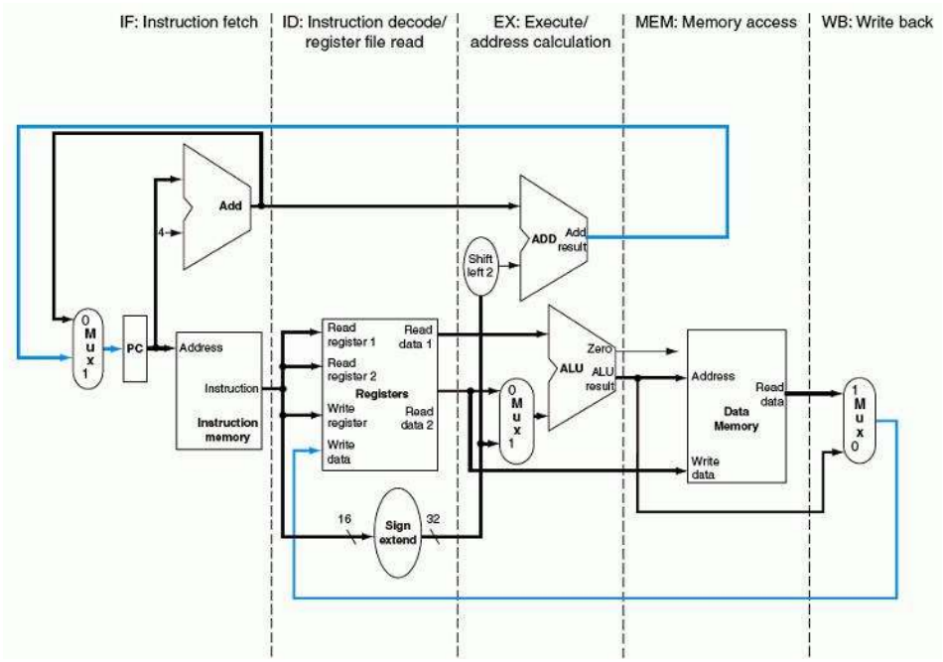

图 2 流水的主要结构

2.2.1 流水线各阶段的设计

在设计段寄存器之前，我们需要先思考每个阶段所包含的处理器模块和处理器所需要执行的任务，以此来确定需要向后阶段传递的信号和数据。如下是对流水线各阶段的分析。

- 取指IF（Instruction fetch）阶段：该阶段主要包含程序计数器、指令内存单元和所需的多路选择器。处理器完成以下工作：按照跳转信号选择下一条指令的地址；依据输出的指令地址从指令内存单元中取出对应的指

令。

- 译码ID（Instruction decode）阶段：该阶段主要包含主控制器、运算单元控制器、寄存器、有符号扩展单元和无符号扩展单元。处理器完成以下工作：依据上一阶段传递过来的指令使用主控制器和运算单元单元控制器完成相应控制信号的解析；完成寄存器的读与写；完成有符号扩展和无符号扩展。
- 执行EX（Execution）阶段：该阶段主要包括算术逻辑运算单元和所需的多路选择器。处理器主要完成以下工作：按照控制信号选择对应的运算数；执行相应的运算；选择产生目标寄存器编号。
- 访存MEM(Memory Access)阶段：该阶段主要包括数据内存单元。处理器主要完成如下工作：依照控制信号完成对指定内存数据单元的读取和写入操作。
- 写入WB（Write Back）阶段：该阶段主要包括多路选择器。处理器主要完成如下工作：依照控制信号选择要求的写入数据。

2.2.2 段寄存器的设计

在明确了各阶段处理器各阶段所要完成的工作之后，我们便可以以此为依据设计各阶段所需要传递信号和数据，也即各段寄存器所需要保存的信号和数据。如下是我们给出的段寄存器设计。

- IF/ID段寄存器：当前指令（32位）与下一条指令的地址（32位）。
- ID/EX段寄存器：WB控制信号（RegWrite与MemToReg，下同）、M控制信号（MemWrite与MemRead，下同）、EX控制信号（ALUCtr、ALUStrcA、ALUStrB、RegDst，下同）、下一条指令地址（32位）、有符号扩展结果（32位）、无符号扩展结果（32位）、rd的编号（5位）、rs的编号（5位）、rt的编号（5位）、rs读出的数据（32位）与rt读出的数据（32位）。
- EX/MEM段寄存器：WB控制信号、M控制信号、ALU运算结果（32位）、需要写入数据内存单元的数据（32位）与目标寄存器编号（5位）。
- MEM/WB段寄存器：WB控制信号、ALU运算结果(32位)、从数据内存单元读出数据（32位）与目标寄存器编号（5位）。

对于每个段寄存器，我们设计了一组输入寄存器和一组输出寄存器。在每次时钟的上跳沿，将输入寄存器的值赋给输出寄存器以完成更新。

2.2.3 冒险解决的总体设计

我们按照以下思路来思考如何解决冒险。

首先，我们需要解决结构冒险。事实上，在实验5中，我们已经解决了结构冒险。我们提供了独立了读写接口，并且规定了写入操作在时钟的下跳沿进行（即前半周期）而读取操作则在后半周期进行。

其次，我们需要解决数据冒险。为了尽量减少停顿，我们设计了一个前向通路Forwarding模块用于解决“写后读数据冒险”（Read after write data hazard）。但由于前向通路无法解决“读存储器-使用冒险”（Load-use data hazard）冒险，因此我们仍需设计一个冒险检测模块Hazard detect用于检测“读存储器-使用冒险”并产生一个周期的停顿。该停顿也是唯一一处由于数据冒险而产生的停顿。

最后，我们需要解决控制冒险。对于j、jal与jr指令，我们可以直接在ID阶段进行判断并将IF/ID段寄存器中的指令清零（flush）即可。而对于beq与bne指令，我们采用分支提前预测和预测不转移的策略，在ID段设计了一个新模块Zero test用于判断需不需要进行跳转，并让IF段中的PC默认采用不跳转的策略，直到需要跳转时将IF/ID段寄存器中的指令清零（flush）即可。

2.2.4 前向通路模块Forwarding的设计

需要前向通路的情况有两种。第一种是EX/MEM段寄存器输出的写入寄存器编号与ID/EX段寄存器读出的寄存器编号一致并且RegWrite为1时，需要将此时MEM段中的ALU结果传递给ALU；第二种是MEM/WB段寄存器输出的写入寄存器编号与ID/EX段寄存器读出的寄存器编号一致并且RegWrite为1时，需要将此时WB段中的准备写入寄存器的数据传递给ALU。需要注意的是，之所以会发生第二种情况，是因为“读存储器-使用”冒险的存在，因此这两种情况是有优先级的，我们应该先判断第一种情况，而第二种情况只有第一种情况不满足时再进行判断。

我们所设计的前向通路模块Forwarding模块的接口如下：对于MEM阶和WB阶段，分别接收RegWrite信号和写入寄存器编号RegWriteAddr以及EX段的rs和rt作为输入，最后产生相应的多路器控制信号用于选择ALU的输入数据。

2.2.5 冒险检测模块Hazard detect的设计

冒险检测模块的实现是简单的，因为只有“读存储器-使用”冒险会导致停顿。我们只需要先判断EX段中的MemRead信号是否为真，并将ID段的rs与rt与EX段产生的写入寄存器地址进行比较即可。停顿信号将被传递给程序计数器PC、IF/ID段寄存器与ID/EX段寄存器。

2.2.6 分支提前预测模块Zero test的设计

分支提前预测模块的设计同样是简单的。我们只需先判断Branch与BranchNot信号，再将寄存器读出的两个数据进行比较即可。需要注意的是，由于分支提前预测模块需要访问寄存器中的数值，所以其本身也会导致数据冒险，所以我们类似于前向通路的实现，将ALU的结果传递给分支预测模块。

2.3 顶层模块Top的设计

在完成以上分析后，我们便可以开始设计顶层模块。我们可以参考实验5中的顶层模块原理图，并向其中添加我们新设计的模块与信号、数据通路以解决冒险。其中有两个部分的通路较为复杂。第一个是PC的输入端，由于分支指令的存在，PC的输入端有4种可能性，需要通过ID段解码的结果进行选择；第二个是ALU的输入段，此处不仅涉及到各指令本身所导致的数据选择，还涉及到由于两种前向通路而导致的数据选择。最终，我们所设计的顶层模块Top的原理图如**图3**。

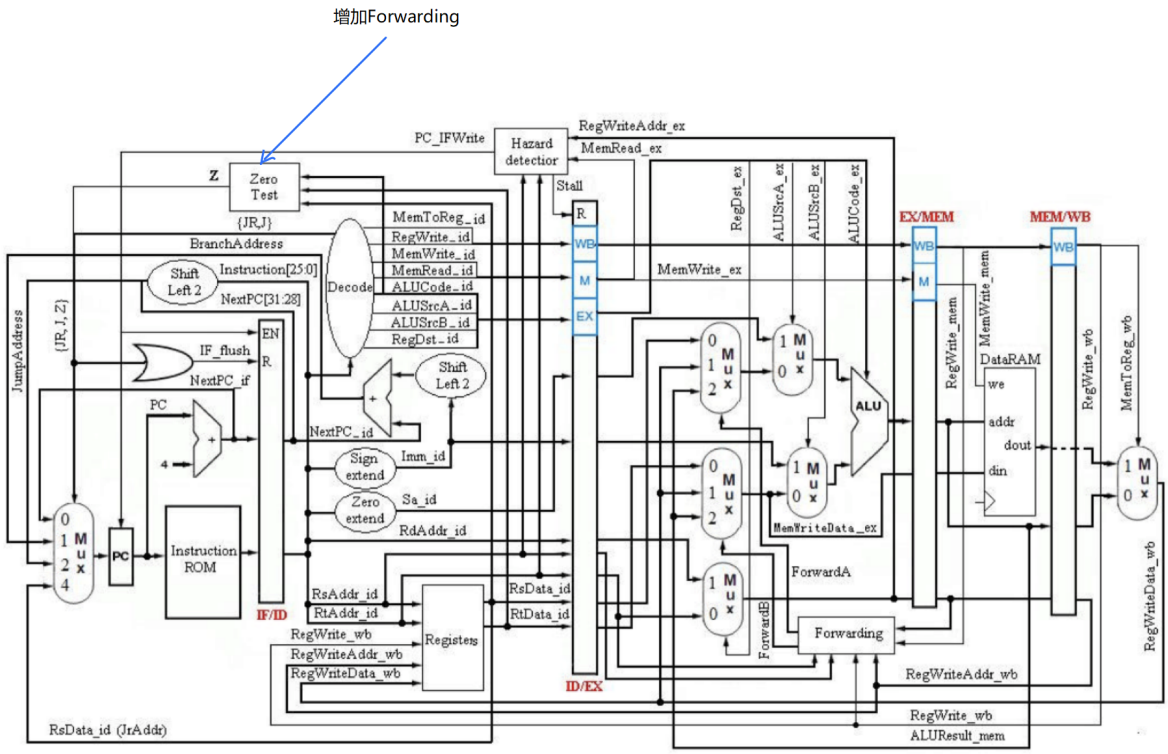

图 3 顶层模块Top原理图

3 功能实现

3.1 处理器基础模块的实现

由于本节中大部分具体实现与实验5极为类似，故仅展示部分关键模块的实现。

3.1.1 运算逻辑单元ALU的实现

运算逻辑单元中最大的变化在于对于有符号运算和无符号运算的区分。

对于需要产生overflow信号的运算，我们采取的方法是：若输入数据的符号位相等，而输出数据的符号位与输入数据的不等，则认为发生了溢出。下方代码以add与sub为例。

```
4'b0010:    //add
    begin
        aluRes = input1 + input2;
        if ((input1[31]) == (input2[31]) && (input1[31]) != (aluRes[31]))
            overflow = 1;
        else
            overflow = 0;
    end
4'b0110:    //sub
    begin
        aluRes = input1 - input2;
        temp = (~input2)+1;
        if ((input1[31]) == (temp[31]) && (input1[31]) != (aluRes[31]))
            overflow = 1;
        else
            overflow = 0;
    end
```

对于比较类运算，我们则通过$signed标识符予以区分。下方代码以slt为例。

```
4'b0111:    //set on less than
    begin
        if ($signed(input1) < $signed(input2))
            aluRes = 1;
        else
            aluRes = 0;
    end
```

3.1.2 寄存器Registers的实现

按照2.1.4节中的讨论，我们需要对jr和jal指令进行特殊处理。具体来说，就是要新增jr和jal信号量，并且要在读出和写入操作时对其进行判断。该模块的核心代码如下。

```
always @(readReg1 or readReg2 or writeReg or jr)
begin
    if (jr)
        readData1 = regFile[31];
    else
        readData1 = regFile[readReg1];
```

```
    readData2 = regFile[readReg2];
end

integer k;
always @(negedge clk or reset )
begin
    if (reset)
        for(k=0;k<32;k=k+1)
            regFile[k] = 0;
    else
        if(jal)
            begin
            regFile[31] = pcPlus4;
            regFile[writeReg] = writeData;
            readData1 = regFile[readReg1];
            readData2 = regFile[readReg2];
            end
        else if (regWrite)
            begin
            regFile[writeReg] = writeData;
            readData1 = regFile[readReg1];
            readData2 = regFile[readReg2];
            end
end
```

3.1.3 数据内存单元Data Memory与缓存Cache的实现

为了实现2.1.5中的设计，我们需要使用新的寄存器用于实现缓存、条目地址、脏位与有效位。此外，在每次读数据时：若cache hit，则直接从Cache中读出数据；若cache miss，则需要先将数据从数据内存单元移动到Cache中再从Cache中读取数据。每次写入数据时：若cache hit，则修改Cache内容，将脏位置位；若cache miss，则需要依据脏位将被替换的Cache内的数据写进内存，并将新数据写入Cache。该模块的核心代码如下。

```
always @(memRead or address)
begin
    if (memRead)
    begin
        if(address<1024)
            begin
                temp = address & 4'hf;
                if(valid[temp] && addr[temp] ==address)
                    begin
                        readData = cache[temp];
                        cache_hit = 1;
                    end

                else if (valid[temp] && addr[temp]!=address)
                    begin
                        if(dirty[temp])
```

```
                            begin
                                memFile[addr[temp]] = cache[temp];
                            end
                        cache[temp] = memFile[address];
                        addr[temp] = address;
                        dirty[temp] = 0;
                        cache_hit = 0;
                    end
                else
                    begin
                        valid[temp] = 1;
                        cache[temp] = memFile[address];
                        addr[temp] = address;
                        dirty[temp] = 0;
                        cache_hit = 0;
                    end
                readData = cache[temp];
            end
        else
            readData = 0;
    end
end

always @(negedge clk)
begin
    if(memWrite && address<1024)
        begin
            temp = address & 4'hf;
            if(valid[temp] && addr[temp] ==address)
                begin
                    dirty [temp] = 1;
                    cache[temp] = writeData;
                    cache_hit = 1;
                end
            else if (valid[temp] && addr[temp]!=address)
                begin
                    if(dirty[temp])
                        begin
                            memFile[addr[temp]] = cache[temp];
                        end
                    cache[temp] = writeData;
                    addr[temp] = address;
                    dirty[temp] = 1;
                    cache_hit = 0;
                end
            else
                begin
                    valid[temp] = 1;
```

```
                    cache[temp] = writeData;
                    addr[temp] = address;
                    dirty[temp] = 1;
                    cache_hit = 0;
                end
        end
end
```

3.1.4 程序计数器PC的实现

为了让程序计数器支持Stall功能，我们只需在每次更新PC时加以判断即可。该模块的核心代码如下。

```
always @(posedge clk or reset)
begin
    if(reset)
        addrOut = 0;
    else
        if(~stall)
        begin
            addrOut = addrIn;
        end
end
```

3.2 流水线的实现

3.2.1 段寄存器的实现

由于程序计数器本质上也是个段寄存器，因此，段寄存器的实现与程序计数器非常类似。我们使用两组寄存器来表示输入与输出，并在时钟周期的上跳延更新输出寄存器。为了代码的可读性，寄存器的命名与其所连接的处理器阶段开头。下面以IF/ID段寄存器为例，给出具体的实现代码。

```
module IFID(
        //--------------------input----------------------//
        input clk,
        input reset,
        input stall,
        input flush,
        input [31:0] IF_inst,
        input [31:0] IF_PC_plus_4,
        //--------------------output---------------------//
        output reg [31:0] ID_inst,
        output reg [31:0] ID_PC_plus_4
    );
    always @ (posedge clk or reset)
        if(reset)
        begin
            ID_inst <= 0;
            ID_PC_plus_4<=0;
        end
```

```
        else
        begin
            if (flush)
            begin
                ID_inst<=0;
                ID_PC_plus_4<=0;
            end
            else if (~stall)
            begin
                ID_inst<=IF_inst;
                ID_PC_plus_4<=IF_PC_plus_4;
            end
        end

endmodule
```

3.2.2 前向通路Forwarding的实现

依照2.2.4中的设计思路与执行逻辑，我们只需要使用三目运算符予以实现即可。注意我们在2.2.4中关于两种前向通路的优先级的表述。该模块的具体实现代码如下。

```

module Forwarding(
        input regWriteMEM,
        input regWriteWB,
        input [4:0] regWriteAddrMEM,
        input [4:0] regWriteAddrWB,
        input [4:0] rsAddr,
        input [4:0] rtAddr,
        output [1:0] FORMUXA,
        output [1:0] FORMUXB
    );

    assign FORMUXA[0] = (
        regWriteMEM?0:(regWriteWB?(rsAddr == regWriteAddrWB?1:0):0)
    );

    assign FORMUXA[1] = (
        regWriteMEM?(rsAddr == regWriteAddrMEM?1:0):0
    );

    assign FORMUXB[0] = (
        regWriteMEM?0:(regWriteWB?(rtAddr == regWriteAddrWB?1:0):0)
    );

    assign FORMUXB[1] = (
        regWriteMEM?(rtAddr == regWriteAddrMEM?1:0):0
    );

```

```
28
29
30  endmodule
31
```

3.2.3 冒险检测模块Hazard detect的实现

依照2.2.5中的讨论，我们需要比较相应的信号与寄存器编号。我们仍然采用三目运算符予以实现。该模块的具体实现代码如下。

```
1   module hazardDetect(
2           input memRead,
3           input [4:0] regWriteAddr,
4           input [4:0] rsAddr,
5           input [4:0] rtAddr,
6           output stall
7       );
8
9       assign stall = (memRead?(((regWriteAddr==rsAddr)||(regWriteAddr==rtAddr))?1:0):0);
10  endmodule
```

3.2.4 分支提前预测模块Zero test的实现

依照2.2.6中的讨论，我们需要在执行比较的同时实现一个简单的前向通路以保证分支提前预测的正确性。此处，我们增加了新的寄存器接口用于获取ALU结果和写入目标寄存器，并增加了额外的判断逻辑以实现前向通路。该模块的具体实现代码如下。次数的实现逻辑较为复杂，需要仔细编写。

```
1   module zerotest(
2           input Branch,
3           input BranchNot,
4           input [31:0] readData1,
5           input [31:0] readData2,
6           input [31:0] aluOut,
7           input [4:0] rsAddr,
8           input [4:0] rtAddr,
9           input [4:0] forAddr,
10          output zero
11      );
12      assign zero = (Branch?
13      ((forAddr==rsAddr)?((aluOut==readData2) ? (BranchNot?0:1) : (BranchNot?1:0)):
    ((forAddr==rtAddr)?((readData1==aluOut) ? (BranchNot?0:1) : (BranchNot?1:0)):
    ((readData1==readData2) ? (BranchNot?0:1) : (BranchNot?1:0))))
14      :0);
15  endmodule
```

3.3 顶层模块Top的实现

由于我们采用模块化设计的思想，我们只需按照2.3中所设计的顶层模块完成连线即可。为了代码的可读性，每根线的变量名都以其所属的阶段开头。该模块的部分实现代码如下。

```
module Top(
    input reset,
    input clk
    );

    wire [31:0] IF_PC_In;
    //------------------IF-start-----------------//
    //-------------------------------------------//

    wire [31:0] IF_PC_Out;


    wire [31:0] IF_INST_Out;
    instMemory inst_mem(
        .readAddr(IF_PC_Out),
        .inst(IF_INST_Out)
    );
    //-------------------------------------------//
    //------------------IF-end-------------------//
    wire [31:0] ID_INST;
    wire [31:0] ID_PC_plus_4;


    //------------------ID-start-----------------//
    //-------------------------------------------//
    wire [31:0] ID_IMM;
    wire [31:0] ID_SA;
    wire [31:0] ID_RS_DATA;
    wire [31:0] ID_RT_DATA;


    //----------some WB wires are defined here-----------//
    wire WB_WRITE_REG;
    wire [4:0] WB_WRITE_REG_ADDR;
    wire [31:0] WB_WRITE_REG_DATA;
    //-----------------WB wires end---------------------//

    zeroext zext(
        .shamt(ID_INST),
        .data(ID_SA)
    );

    wire ID_SIGN;
```

```
44
45      signext sext(
46          .sign(ID_SIGN),
47          .inst(ID_INST),
48          .data(ID_IMM)
49      );
50  // we omit some code here
```

4 结果验证

本次仿真与实验5类似。我们需要使用$readmemb语句和$readmemh语句将我们的二进制指令和16进制数据载入到指令内存单元和数据内存单元，再使用clk作为时钟信号来触发每一周期指令的执行。我们所采用的激励代码如下。

```
1   module MIPS_pipe_tb(
2
3       );
4       reg clk, reset;
5
6       Top processor(
7           .clk(clk),
8           .reset(reset)
9       );
10
11      initial begin:ProcessorInit
12          clk = 1;
13          reset = 1;
14          $readmemb("D:/PersonalFile/courses/CS145-Experiments-in-Computer-
    Organization/lab06/lab06.srcs/data/inst.dat", processor.inst_mem.memFile);
15          $readmemh("D:/PersonalFile/courses/CS145-Experiments-in-Computer-
    Organization/lab06/lab06.srcs/data/data.dat", processor.data_mem.memFile);
16      end
17
18      always #25 clk = !clk;
19
20      initial begin:ProcessorRunning
21          #20;
22          reset = 0;
23      end
24
25  endmodule
```

全部的31条指令已经在课上经过现场检查。为了简单明了地展示该处理器的特点，我们的测试分为两部分。第一部分用于验证处理器的基本功能，第二部分用于验证31条指令的完备性。

4.1 处理器基本功能验证

此部分将通过设计汇编代码来验证处理器的如下功能：

- 遇到Load-use data hazard时采取单周期停顿（Stall）
- 遇到“写后读数据冒险”时采取前向通路（Forwarding）
- 对有符号运算遇到溢出时产生overflow信号
- 预测不转移（predict-not-taken）策略和分支指令时的刷新（Flush）
- cache miss时对Cache的操作和Cache的Write-back策略

汇编代码：

```
0: lw $1 1($0)
4: addi $2 $1 1 // Load-use data hazard
8: sw $2 3($0) // cache miss
c: add $3 $1 $2
10: beq $0 $3 2 // predict not taken & Zero test forwarding
14: sub $3 $3 $1
18: j 10 // flush
1c: and $4 $1 $2
20: or $5 $1 $2
24: slt $6 $1 $2
28: andi $3 $1 2
2c: ori $3 $1 2
30: jal 40
34: sll $3 $1 4
38: srl $3 $3 2
3c: j 44
40: jr $31
44: lw $2 2($0)
48: add $3 $1 $2 // overflow
4c: add $4 $3 $3 // overflow & forwarding
50: sw $4 19($0) // write-back
54: lw $2 1($0) // cache hit
```

二进制代码：

```
10001100000000010000000000000001
00100000001000100000000000000001
10101100000000010000000000000011
00000000001000100001100000100000
00010000011000000000000000000010
00000000011000010001100000100010
00001000000000000000000000000100
00000000001000100010000000100100
00000000001000100010100000100101
00000000001000100011000000101010
00110000001000110000000000000010
00110100001000110000000000000010
```

```
00001100000000000000000000010000
00000000000000010001100100000000
00000000000000011000110001000010
00001000000000000000000000010001
00000011111000000000000000001000
10001100000000100000000000000010
00000000001000100001100000100000
00000000011000110010000000100000
10101100000001000000000000010011
10001100000000100000000000000001
```

对于前三个内存单元，我们使用如下数据进行填充：

```
00000000
00000001
7FFFFFFF
```

测试结果如**图4**。下面我们分功能进行展示。

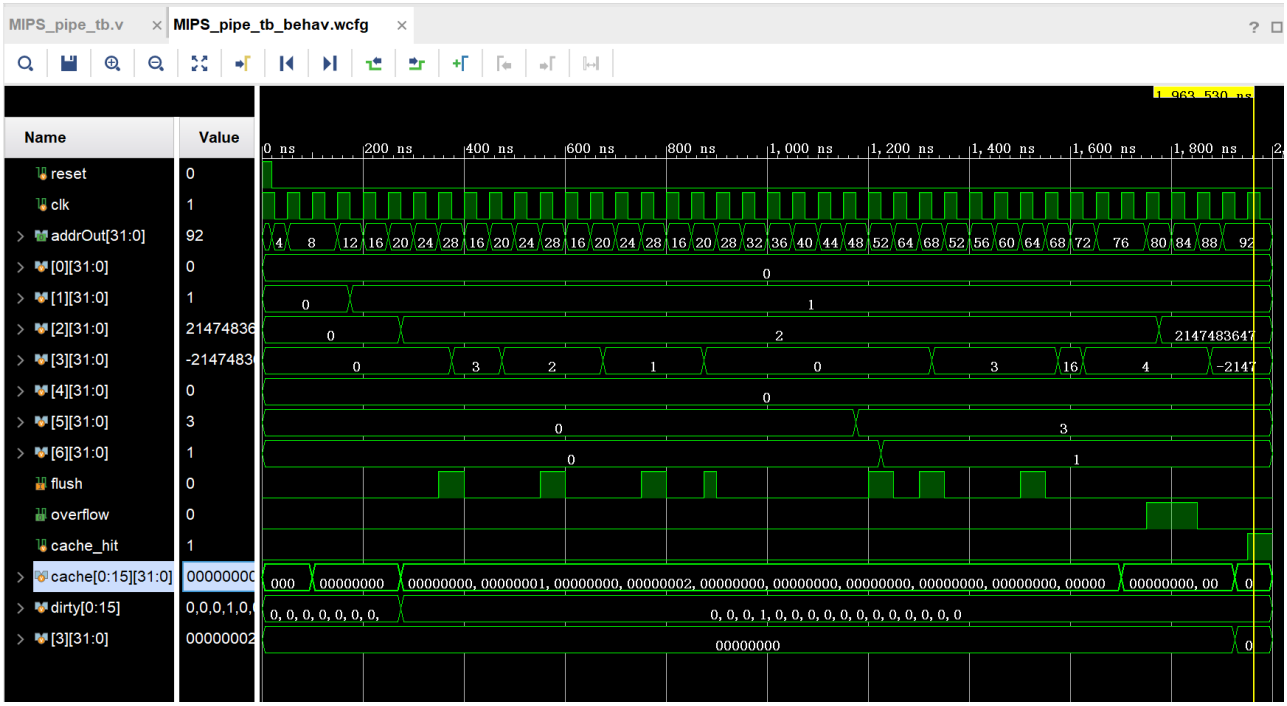

图 4 类MIPS流水线处理器的基本功能验证

第一是遇到“读存储器-使用冒险”时的停顿。从**图5**中可以看出，由于0语句和4指令之间存在“读存储器-使用冒险”，因此在这两条指令之间处理器停顿了一周期以解决数据冒险。

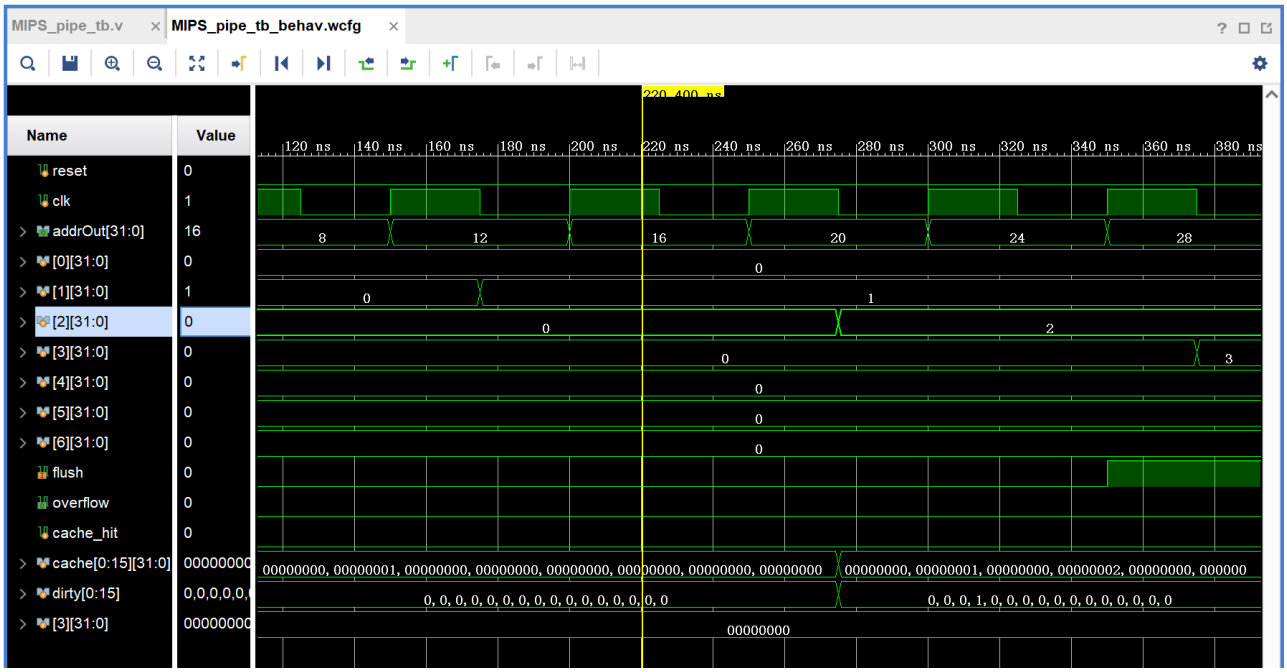

图 5 类MIPS流水线处理器的停顿功能验证

第二是遇到“写后读数据冒险”时采取前向通路（Forwarding）。 从**图6**中可以看出：通过对于48与4c指令之间的数据冒险，处理器并没有停顿，而是通过前向通途解决了数据冒险。

第三是对有符号运算遇到溢出时产生overflow信号。同样从**图6**中可以看出：由于48指令的7FFFFFFFH+1H发生了溢出，所以overflow变成高电平；类似的，由于4c指令的80000000H+80000000H发生了溢出，所以overflow也为高电平。

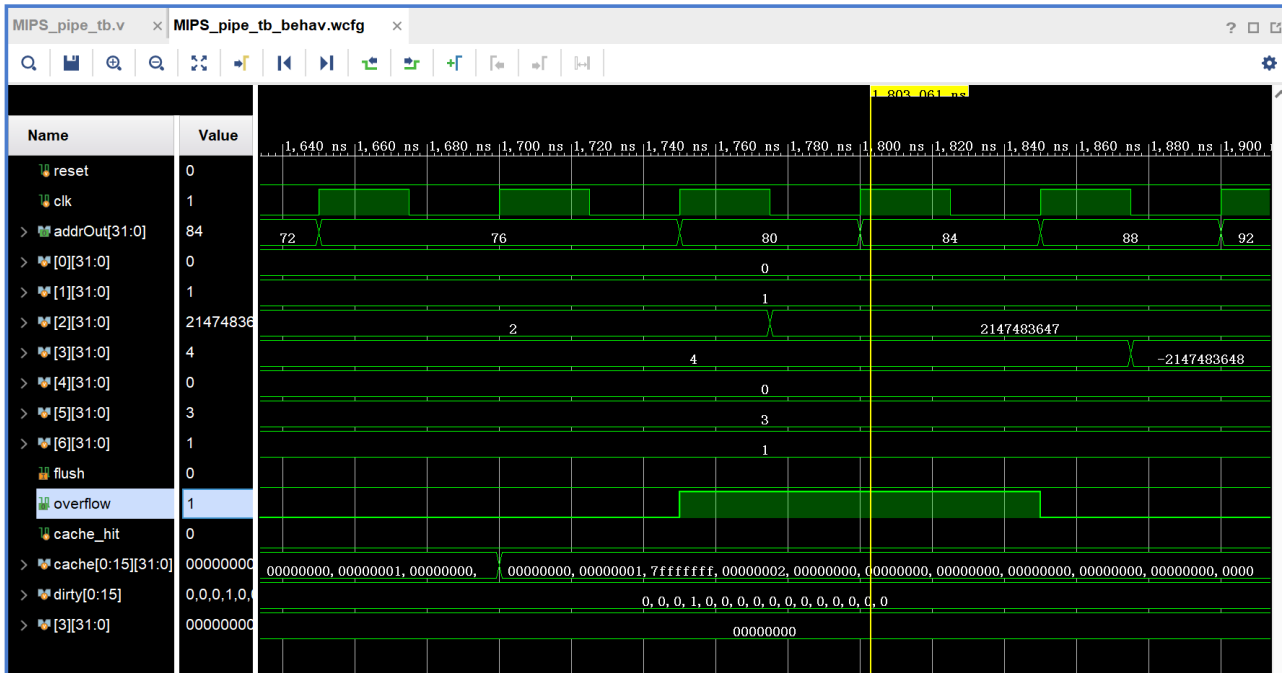

图 6 类MIPS流水线处理器的前向通路功能及溢出监测功能验证

第四，是预测不转移（predict-not-taken）策略和分支指令时的刷新（Flush）。从**图7**中可以看出：对于10到18的这三条指令，当预测正确时，处理器没有产生额外的停顿；当需要刷新时，flush信号为高电平。

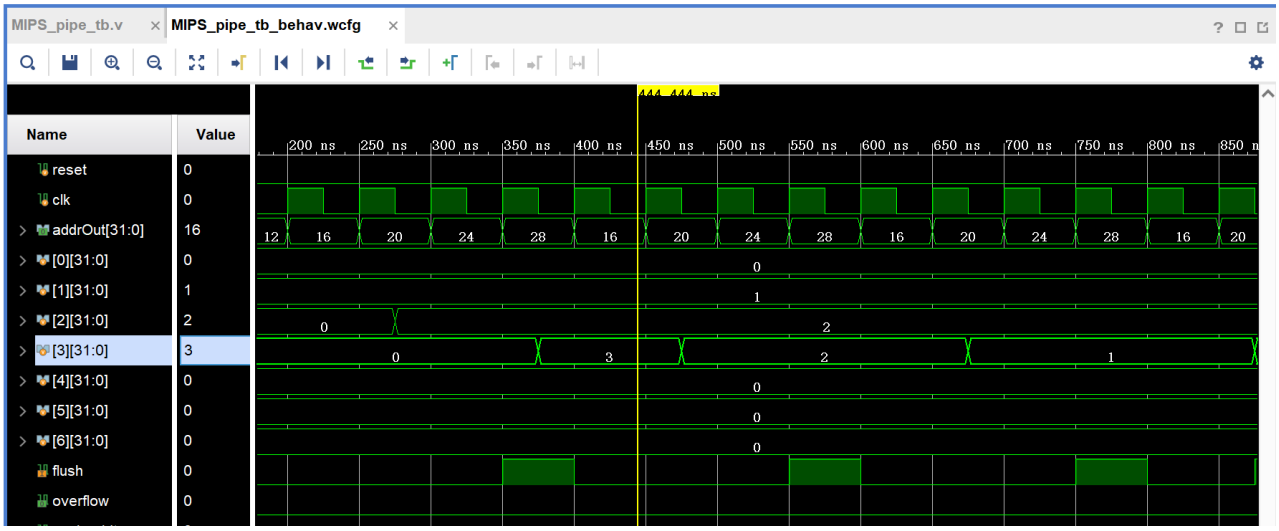

图 7 类MIPS流水线处理器的预测不转移功能及刷新功能验证

第五，是cache miss时对Cache的操作和Cache的Write-back策略。从**图8**中可以看出：对于写操作时的cache miss，我们所设计的处理器找到了需要被替换的数据，确认脏位后将其写入了对于的数据内存单元（3号）；对于读操作时的cache hit，将直接从cache中获取对应的数据。

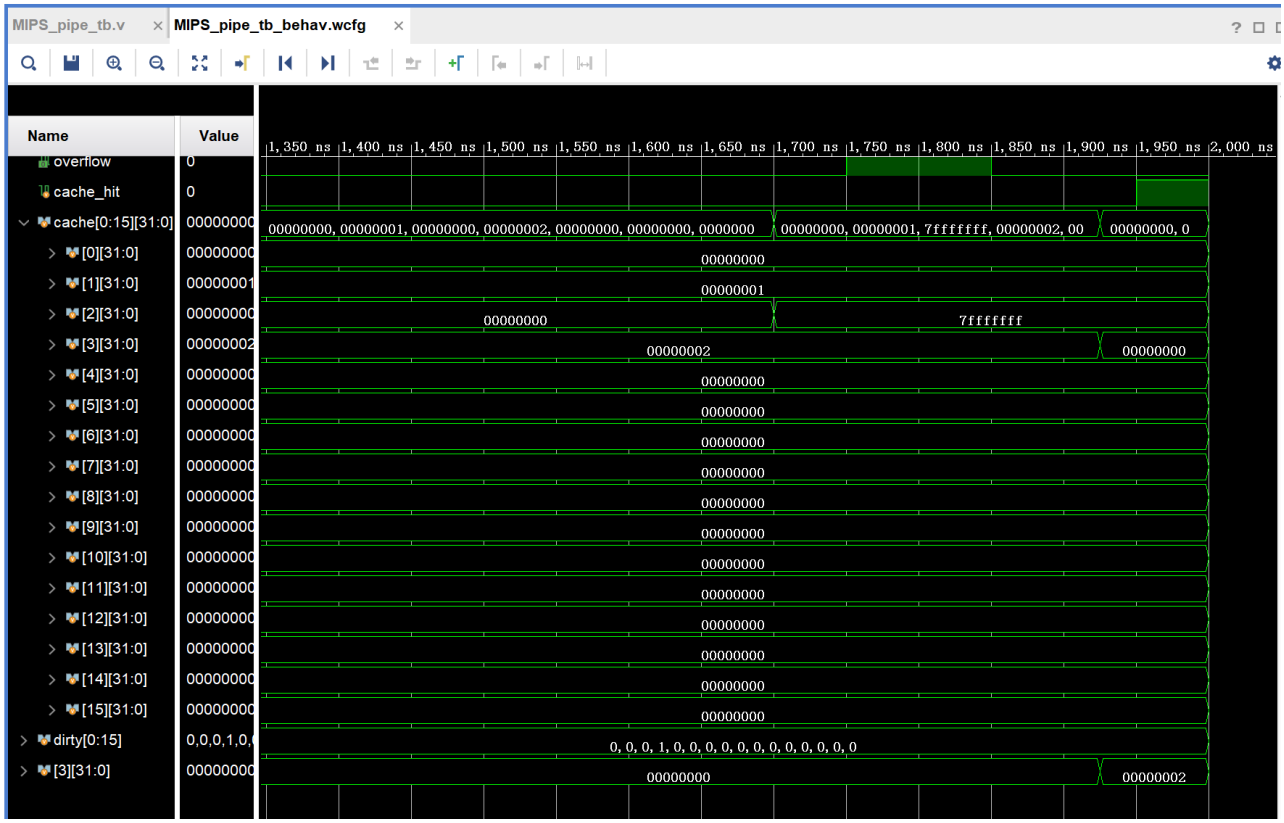

图 8 类MIPS流水线处理器的缓存功能验证

其他语句基本为运算指令和访存指令，通过对比寄存器和内存单元中的值可以发现，我们所设计的处理器正确地按照我们的指令执行了运算或者是访存操作。可见，通过这一系列测试指令，我们全面地测试了处理器所支持的各项功能的正确性，类MIPS流水线处理器的基本功能验证通过。

4.2 指令完备性验证

此部分将通过设计汇编代码来验证在4.1中未验证的指令的正确性：

汇编代码：

```
0: lw $1 2($0)
4: lw $2 1($0)
8: addu $3 $1 $2
c: addiu $4 $1 2
10: xor $3 $1 $2
14: sltu $4 $1 $2
18: xori $3 $1 1
1c: lui $4 0xffff
20: slti $3 $1 1
24: sltiu $4 $1 0x7fff
28: bne $1 $0 0x1
2c: nop
30: srl $3 $1 16
34: sra $4 $1 16
38: lw $30 0($0)
3c: sllv $5 $1 $30
40: srlv $6 $1 $30
41: srav $3 $1 $30
```

二进制代码：

```
10001100000000010000000000000010
10001100000000010000000000000001
00000000001000100001100000100001
00100100001001000000000000000010
00000000001000100001100000100110
00000000001000100010000000101011
00111000001000110000000000000001
00111100000001001111111111111111
00101000001000110000000000000001
00101100001001000111111111111111
00010100001000000000000000000001
00000000000000000000000000000000
00000000000000001000111000000010
00000000000000001001001000000011
10001100000111100000000000000000
00000011110000010010100000000100
00000011110000010011000000000110
00000011110000010001100000000111
```

对于前三个内存单元，我们使用采用如下数据进行填充：

```
00000001
7FFFFFFF
FFFFFFFF
```

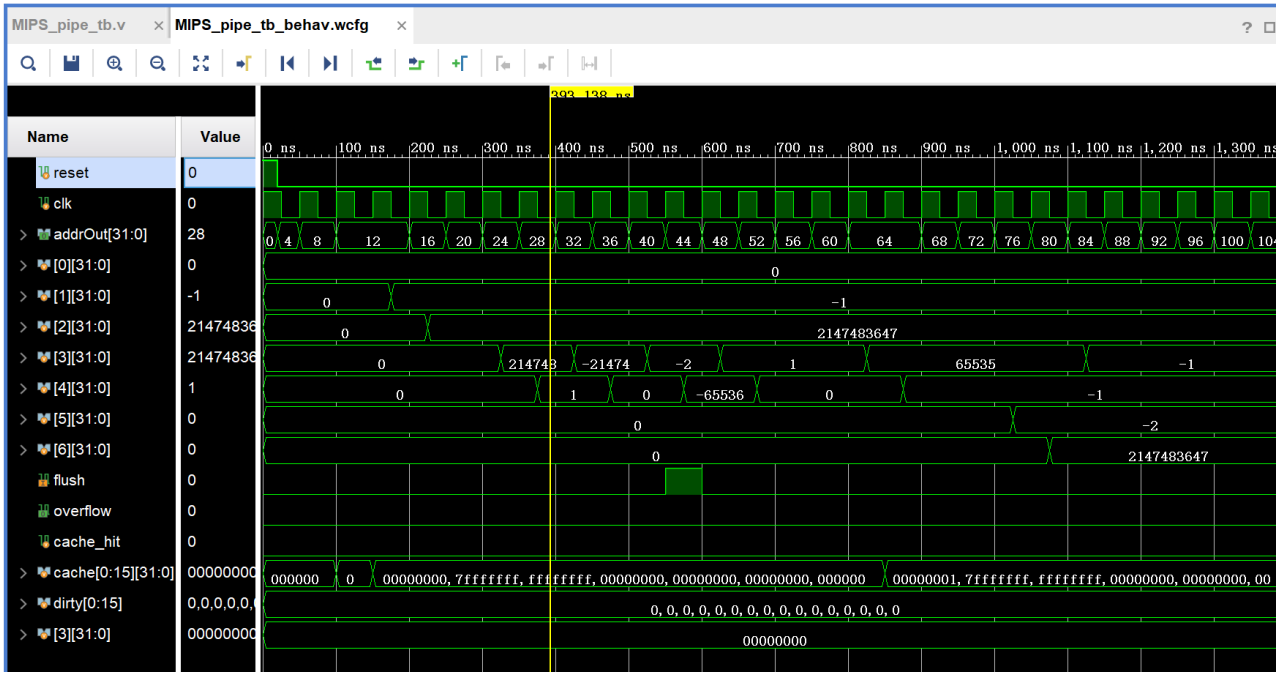

图 9 类MIPS流水线处理器的指令完备性验证

测试结果如**图9**。通过对比寄存器和内存单元中的值可以发现，我们所设计的处理器正确地按照我们的指令执行了运算，并且很好地区分了相似指令（srl与sra）以及有符号无符号（slti与sltiu）之间的区别。类MIPS流水线处理器的指令完备性验证通过。

5 总结与反思

本次实验中，我们在前三个实验的基础上通过增加新模块、设计流水线结构、分析解决各类冒险的方式，实现了一个简单的类MIPS多周期流水线CPU。本次实验，相比于实验5，在拥有更大的自由度的同时，也对我们的能力提出了新的要求。我认为，经过本次实验，我在三个方面的能力得到了提升。

首先，是知识层面的提升。流水线CPU的运行机制更加复杂，同一时刻，每个阶段的各个元件在为不同的指令工作，因此我们需要设计段寄存器来保存、传递状态。而需要保存哪些状态，则需要我们对各个阶段的工作内容和整个顶层设计中的数据通路有着全面且准确的了解。此外，当我们增加了分支提前预测、前向通路等功能后，整个处理器会变得更加复杂，设计难度也大大提高。可以说，经过这次亲手设计CPU后，我对流水线CPU有了一个更加高维度的认识，对MIPS指令集的各个指令也更加了解。

其次，是工程能力的提升。当我们了解如何设计之后，更重要的是要具备动手实现的能力。在本次实验中，我依旧采用模块化设计的思想，并且按照整体-部分-整体的流程进行模块设计和代码书写。经过这次高难度、极具挑战的工程训练之后，我的较大工程项目的设计和实现能力得到了极大地提升，代码习惯和工程素养也得以提高。

最后，是自主思考能力和创新能力的提升。在本次实验中，我们没有现成的图纸，更多的只是理论课上所教授的理论知识。如何高效地将理论转化为具体的设计，需要我们的自主思考。而除了已教授的知识，我们如何进一步拓展，举一反三，则需要我们的创新能力。在本次实验中，除了必做内容，我通过自主思考为处理器增加了缓存功能，使其更接近真实的处理器。这便是创新能力的一次良好培养。

6 致谢

感谢计算机系的黄小平老师在本次实验中给予的细致指导。

感谢助教学长的辛勤工作。

感谢计算机系统结构实验的课程负责老师编写了详尽的实验指导材料，帮助我们能够没有障碍地完成实验。

感谢电子信息与电气工程学院提供实验的场地与器材。

感谢计算机系的张鼎言、叶航宇同学与我讨论、交流经验。

参考文献:

[1] 2020计算机系统结构实验指导书-LAB06.
[2] Verilog HDL入门教程.
[3] 1.1 Verilog 教程 | 菜鸟教程 (runoob.com)