计算机系统结构实验Lab06:简单的类 MIPS 多周期流水化处理器实现

张祥东

June 3,2023

本实验的主要内容基于lab05的单周期处理器，实现类MIPS多周期流水线处理器。在实验过程中，我逐步完成了流水线五个阶段的设计，对空指令（NOP）、数据前向传递（Forwarding）、流水线停顿（STALL）以及静态/动态分支跳转预测器等关键技术进行了思考并进行程序设计。

我还实现了指令扩展并完成了Cache的编写，最后进行了仿真验证并利用FPGA板进行上板验证。通过仿真验证和上板验证，我验证了我的多周期流水线处理器能够正确地执行汇编程序，且能够得到了正确的结果。

摘要

目录

1 实验目的 **2**

2 原理分析 **2**

2.1 MIPS多周期流水线处理器部件 2
2.2 MIPS多周期流水线处理器运行逻辑 3
2.3 整体设计思路 4
2.3.1 流水线的五个阶段设计 5
2.3.2 空指令（NOP）原理分析及设计 5
2.3.3 数据前向传递（Forwarding）原理分析及设计 5
2.3.4 流水线停顿（STALL）原理分析及设计 6
2.3.5 静态（predict-not-taken）/动态分支跳转预测器原理分析及设计 6
2.4 Cache原理分析 6
2.5 时序逻辑 7

3 功能实现 **8**

3.1 命名规则 8
3.2 手绘流水线处理器原理图 8
3.3 各部件实现 8
3.4 顶层模块实现 10
3.4.1 流水线的五个阶段实现 10
3.4.2 Jump指令与Branch指令的特殊实现 12
3.4.3 Forwarding实现 13
3.4.4 STALL实现 14
3.4.5 NOP实现 17
3.4.6 静态（predict-not-taken）/动态分支跳转预测器实现 18

3.5 指令扩展实现 19
3.6 Cache编写 21
3.7 激励文件编写 22
3.8 上板验证代码编写 23
3.9 整体调试 24

4 结果验证 **24**
4.1 仿真验证 24
4.2 上板验证 25

5 总结与反思 **25**

6 致谢 **26**

1 实验目的

1. 理解CPU Pipeline、流水线冒险(hazard)及相关性，在lab5基础上设计简单流水线CPU
2. 在1.的基础上设计支持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/竞争、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，减少因数据竞争带来的流水线停顿延时，提高流水线处理器性能
4. 在3. 的基础上，通过predict-not-taken（或者动态分支预测）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
5. 在4.的基础上，将CPU 支持的指令数量从9条或16条扩充为31条，使处理器功能更加丰富
6. 应用 Cache 原理，设计 Cache Line 并进行仿真验证

2 原理分析

2.1 MIPS多周期流水线处理器部件

相比于lab05 MIPS 单周期处理器的部件，MIPS多周期流水线处理器的部件大部分与单周期处理器相同。相同部件如下：

- 指令存储器（Instruction Memory）：用于存储程序的指令。指令存储器是一个存储器单元，可以根据指令的地址读取对应的指令内容。在流水线处理器中，指令存储器通常采用随机存取存储器（RAM）来实现。
- 数据存储器（Data Memory）：用于存储数据。数据存储器也是一个存储器单元，可以根据数据的地址读取或写入对应的数据内容。在流水线处理器中，数据存储器通常采用随机存取存储器（RAM）来实现。
- 寄存器堆（Registers）：用于存储和读取寄存器中的数据。寄存器堆由多个寄存器组成，每个寄存器可以存储一个数据值。在MIPS处理器中，寄存器堆包含了32个通用寄存器，可以通过寄存器编号进行访问。

- 控制单元（Control Unit）：包括Ctr与ALUCtr模块，功能是控制整个处理器的操作，根据指令的类型产生相应的控制信号。控制单元接收指令的译码结果，根据指令的操作类型确定执行的操作，并生成相应的控制信号，如ALU的操作类型、数据通路的选择等。

- 算术逻辑单元（ALU）：执行算术和逻辑运算。ALU接收来自控制单元的操作类型和数据，根据指令的要求进行相应的运算，如加法、减法、逻辑与、逻辑或等。

- 数据扩展单元（Sign extend）：需要根据指令需求对立即数进行有符号/无符号扩展。

但是由于 pipeline 的影响，在MIPS多周期流水线处理器中会同时执行不同指令的不同阶段，所以需要提供不同阶段之间的信息缓存部件，称为buffer，是一种寄存器组，用于进行信息存储与下传。我们需要添加四种buffer，分别是：

- IF_ID_buffer（指令译码缓冲区）：该缓冲区用于将指令从指令存储器（Instruction Memory）传递到指令译码阶段（ID阶段）。它包含了从指令存储器读取的指令内容以及其他必要的控制信号，如PC（程序计数器）的值。

- ID_EX_buffer（译码执行缓冲区）：该缓冲区用于将译码阶段（ID阶段）的结果传递到执行阶段（EX阶段）。它包含了从指令译码阶段得到的操作数、寄存器编号等信息，以及其他必要的控制信号，如ALU的操作类型和源操作数选择等。

- EX_MEM_buffer（执行访存缓冲区）：该缓冲区用于将执行阶段（EX阶段）的结果传递到访存阶段（MEM阶段）。它包含了执行阶段得到的ALU运算结果以及其他必要的控制信号，如访存指令的类型和操作数等。

- MEM_WB_buffer（访存写回缓冲区）：该缓冲区用于将访存阶段（MEM阶段）的结果传递到写回阶段（WB阶段）。它包含了访存阶段得到的数据内容以及其他必要的控制信号，如写回的目标寄存器编号等。

这些缓冲区的作用是在流水线的不同阶段之间传递数据和控制信号，确保数据的正确流动和处理器的正确操作。它们通过保存上一阶段的结果，使得当前阶段可以读取和使用这些结果，并在下一个时钟周期传递给下一阶段。这样，不同阶段的处理可以并行进行，提高了处理器的效率和性能。

2.2 MIPS多周期流水线处理器运行逻辑

MIPS多周期流水线处理器的工作逻辑主要建立在将指令拆分为五个阶段–IF（取指译码），ID（解码），EX（执行），MEM（内存访问），WB（寄存器写回）的基础上，由于五个部分的硬件之间是可以并行的，所以我们可以做到同时执行不同指令的不同阶段，并最终完成这些指令的执行。这就是流水线的思想。其工作原理图如下：

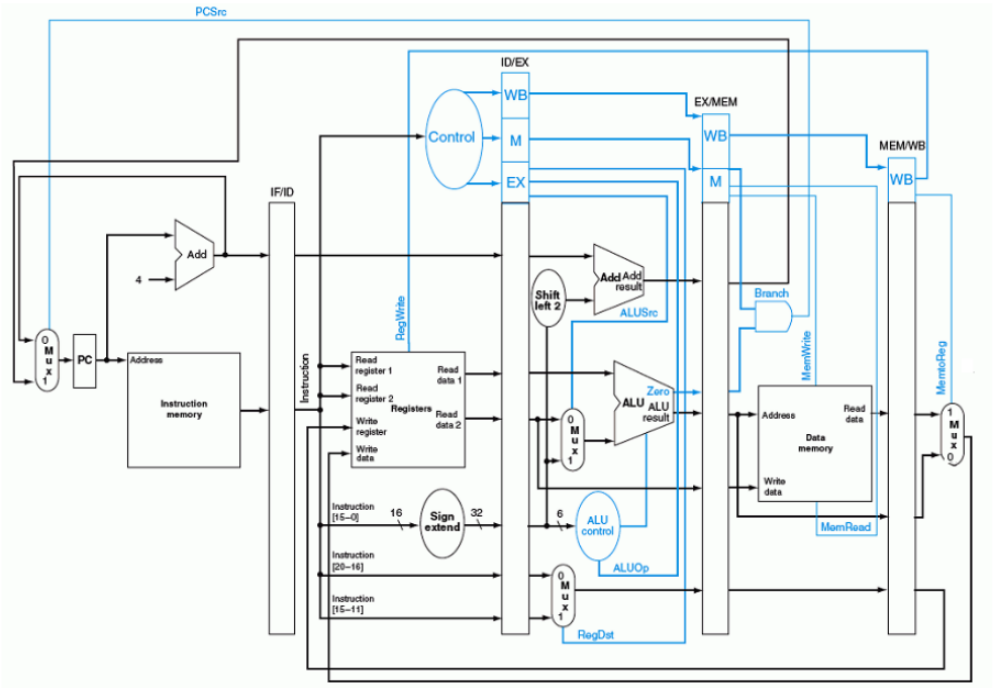

Figure 1: MIPS多周期流水线工作原理图

具体来说：

- 指令的提取（Instruction Fetch）：在该阶段，指令存储器（Instruction Memory）从指令存储器中读取指令，并将其送入指令译码缓冲区（IF_ID_buffer）。
- 指令的译码（Instruction Decode）：在该阶段，指令译码器对从指令译码缓冲区获取的指令进行解析，并提取操作数、确定指令类型和执行需要的控制信号。这些信息被送入译码执行缓冲区（ID_EX_buffer）。
- 执行阶段（Execute）：在该阶段，执行单元（ALU）根据控制信号执行算术或逻辑操作，并产生结果。这个结果和其他必要的控制信号被存储在执行访存缓冲区（EX_MEM_buffer）。
- 存储访问阶段（Memory Access）：在该阶段，访存单元（Data Memory）根据控制信号从数据存储器中读取数据或将数据写入存储器。这个阶段的结果和相关控制信号被存储在访存写回缓冲区（MEM_WB_buffer）。
- 写回阶段（Write Back）：在该阶段，将结果写回到寄存器堆（Register File）。写回的数据和目标寄存器的编号来自访存写回缓冲区（MEM_WB_buffer）。

这样，通过将流水线划分为多个阶段，并在各个阶段之间引入缓冲区（buffer）来传递数据和控制信号，MIPS多周期流水线处理器实现了指令的并行处理，提高了处理器的执行效率和性能。

2.3 整体设计思路

首先对基本的MIPS多周期流水线进行设计，同时针对流水线处理器的三种冒险（control hazard,data hazard,structure hazard），设计STALL,NOP,Forwarding来避免这几种冒险（hazard）

对于流水线的影响。由于本实验中将inst memory与data memory分开实现，所以不存在structure hazard，所以我只需要针对control hazard和data hazard进行处理即可。

其中control hazard是由分支跳转指令引起的，当指令跳转，而PC无法即时变为正确值，则会执行错误的指令，此时需要将这个指令的执行中途丢弃（discard）以保证流水线执行结果正确，针对control hazard，我们利用NOP，predictor处理这个问题。

data hazard是由前后的数据依赖引起的，比如`use after load` 以及 `read after write`，对于这种冒险，我使用forwarding和STALL处理。

2.3.1 流水线的五个阶段设计

前文已经介绍过流水线五个阶段中每个阶段的原理以及具体任务，在此不再赘述。针对流水线五阶段的设计，将指令执行过程分为五个阶段：取指（IF）、译码（ID）、执行（EX）、访存（MEM）和写回（WB）。每个阶段负责不同的操作，并通过流水线寄存器（buffer）连接起来，实现指令的流水执行即可。

2.3.2 空指令（NOP）原理分析及设计

空指令（NOP）是一种特殊的指令，不进行任何操作。在流水线中，当遇到需要停顿或等待的情况时，可以插入空指令来保持流水线的平衡。关于该指令的设计，我通过添加控制信号`NOP`实现这一指令，其值为1时即需要进行向后传递NOP指令的操作。

2.3.3 数据前向传递（Forwarding）原理分析及设计

数据前向传递（Forwarding）是一种解决数据冒险问题的技术。当指令之间存在数据依赖性时，可以通过将计算结果直接传递给后续指令来避免停顿周期，提高流水线的效率。具体来说，Forwarding包括两种类型的传递：

- ID_EX_buffer到EX阶段的数据传递
- EX_MEM_buffer到EX阶段的数据传递

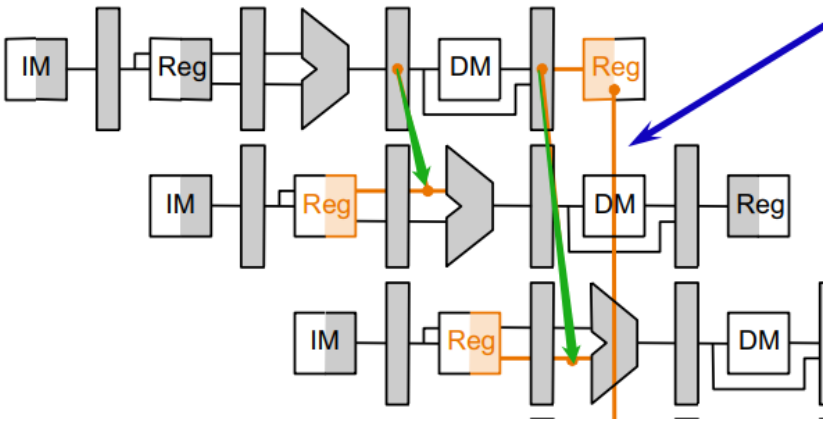

Figure 2: Forwarding示意图

具体如图 figure 2，通过这样的提前数据传递，我们可以利用计算的中间结果，有效减少流水线STALL的周期，提高流水线的执行效率。

2.3.4 流水线停顿（STALL）原理分析及设计

流水线停顿（STALL）是一种机制，用于解决数据冒险（data hazards）的问题。数据冒险指的是在流水线中相邻的指令之间存在数据依赖性，而且后一条指令需要使用前一条指令的结果。由于流水线中的指令是并行执行的，可能会导致数据相关的问题，如数据冲突或数据错误。

为了解决数据冒险的问题，当检测到数据相关时，流水线需要进行停顿操作。停顿操作会在流水线中插入一个或多个空操作周期，使得后续的指令等待前一条指令的结果准备就绪。这样可以确保数据的正确性。

在具体设计时，通过添加控制信号`STALL`，当其值为1时，便进行STALL操作即可。

2.3.5 静态（predict-not-taken）/动态分支跳转预测器原理分析及设计

由于分支跳转指令（Branch相关）的存在，我们可以使用简单的流水线指令丢弃（discard，也称为流水线冲刷（flushing）），终止错误指令的执行，但这样在会造成流水线的吞吐量下降，所以针对跳转指令，我们可以设计分支跳转预测器来进行下一条指令的地址，如果预测正确，则不用进行指令的丢弃。

分支跳转预测器可以分为两种：

- 静态分支跳转预测器。静态分支跳转预测器基于一种固定的预测策略，通常是"predict-not-taken"（预测不跳转，认为下一条指令的地址为PC+4）。它假设所有的分支指令都不会发生跳转，即默认情况下分支指令的下一条指令地址是顺序执行的下一条指令地址。这种预测策略在某些情况下是有效的，特别是当分支跳转不频繁且不容易预测时。然而，当分支跳转频繁发生或者有较高的预测准确性要求时，静态分支跳转预测器的效果会显著降低。

- 动态分支跳转预测器。动态分支跳转预测器基于历史的分支跳转记录和预测算法来预测分支指令的下一条指令地址。它使用一种状态机或者表格来记录过去的分支行为，并根据这些记录进行预测。常见的动态分支跳转预测器包括两位饱和计数器（Two-bit Saturating Counter）、全局历史分支跳转缓冲器（Global History Branch Predictor）和局部历史分支跳转缓冲器（Local History Branch Predictor）等。这些预测器根据分支指令的历史行为来进行预测，以提高预测的准确性和流水线的效率。

通过分支跳转预测器的使用，在预测正确率可观时，流水线处理器的吞吐效率也会大大提升。在本实验中，我既实现了静态（predict-not-taken）预测器又实现了基于二位饱和计数器的动态预测器（Two-bit Saturating Counter）。具体内容在**功能实现**板块进行展示。

2.4 Cache原理分析

缓存（Cache）是一种高速临时存储器，用于存储最常用的数据和指令，以提高数据访问的速度。在流水线处理器中，可以设计和实现缓存来减少对主存的访问次数。

缓存的建议在局部性原理的基础上，即程序在执行过程中会倾向于访问相邻的内存地址。其将存储器分为若干块（Block），每个块包含多个字（Word）。当需要访问数据时，首先检查缓存

是否已经包含该数据，如果包含则直接从缓存中读取，否则需要从主存中读取，并将数据存储到缓存中以备后续访问。

通过缓存的使用，可以显著提高数据的访问速度，减少对主存的访问延迟。缓存的设计和替换策略会影响到缓存的性能，需要根据具体的需求进行优化和调整。

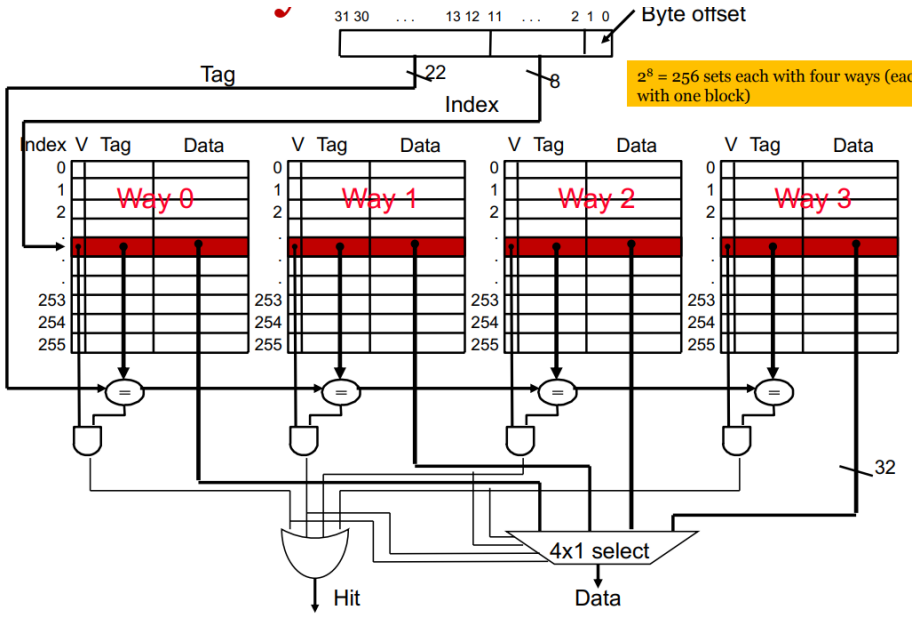

Figure 3: Cache示意图

实现Cache是本实验的拓展部份。具体来说，在本实验中，加入Cache后，在data memory的读写时，我们首先对Cache进行访问，Cache hit则直接在Cache上进行读写操作，反之则从内存中将相应数据读取到Cache再进行读写操作。图片figure 3 便是一个四路Cache（set-associativity Cache）的示意图。

2.5 时序逻辑

在MIPS多周期流水线处理器中，时序逻辑是非常重要的。时序逻辑是指处理器中各个部件和信号之间的时间关系和顺序。它确保了各个阶段的操作按照正确的顺序进行，并保证数据的一致性和正确性。流水线中的时序逻辑由时钟信号来驱动。时钟信号提供了一个统一的时间基准，使得各个部件在不同的时钟周期内执行特定的操作。每个时钟周期被划分为若干个阶段，每个阶段对应着处理器中的一个部件或操作。通过合理地设计和控制时钟信号的传递和触发，可以确保流水线中的各个阶段按照正确的顺序和时间间隔执行。

时序逻辑的正确性对于处理器的正常运行至关重要。任何时序逻辑的错误都可能导致数据错误、冲突或不一致的情况发生。因此，在设计和实现流水线处理器时，需要特别关注时序逻辑的准确性和稳定性，并进行充分的验证和测试，以确保流水线处理器的正确运行。

经过仔细考量，在我的MIPS多周期流水线处理器设计中，主体时序逻辑如下：

- 时钟上升沿：更新PC，更新IF_ID_/ID_EX/EX_MEM/MEM_WB_buffer。
- 时钟下降沿：写寄存器，写data memory。

其余操作均为组合逻辑。

3 功能实现

3.1 命名规则

由于MIPS多周期流水线处理器涉及到的变量非常多，所以我在开始写代码之前制定了一套简洁的命名规则。规则如下：

- IF,ID,EX,MEM,WB,PC,ALU特殊名词大写
- 模块名称首字母小写，单词之间用大写字母分隔
- 模块的I/O变量名称首字母小写，单词之间用大写字母分隔
- TOP模块内部的变量名首字母大写，单词之间用大写字母分隔
- 模块实例化对象名称均小写，单词间用_分隔
- 涉及到buffer时，使用的形式均为buffer前缀加上变量名，利用下划线分隔，形如IF_ID_xx，ID_EX_xx，EX_MEM_xx，MEM_WB_xx。

3.2 手绘流水线处理器原理图

在实现MIPS多周期流水线处理器之前，我通过理解整体工作逻辑，自己画了一张MIPS多周期流水线处理器的工作原理图，加深自己对lab06的理解同时便于后续查阅。如下图：

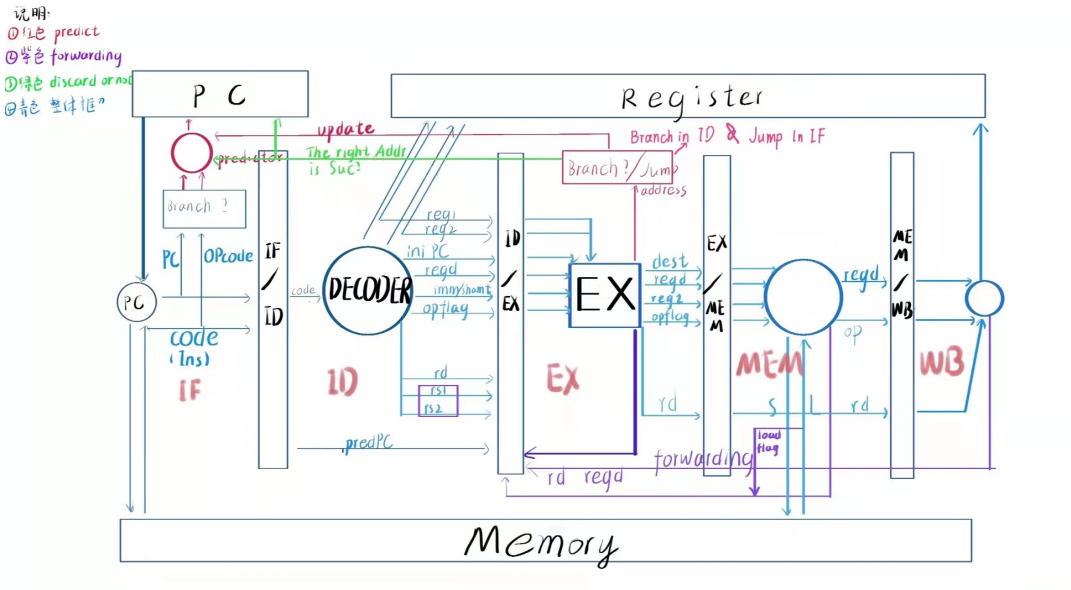

Figure 4: 手绘流水线原理图

3.3 各部件实现

根据原理分析，实现MIPS单周期处理器所需的模块包括：Ctr，ALUCtr，ALU，Data memory，registers，sign extend，Inst memory，forwarding，4个buffer。其中大部分模块在lab01-lab05中已经实现，具体内容在此不再赘述。此处仅介绍之前未实现的4个buffer，即IF_ID_/ID_EX/EX_MEM/MEM_WB_buffer。

针对buffer，我们需要根据原理图，确定每个buffer中需要存储的数据以及控制信号，在Top模块中进行这些变量的定义，并按照预先设计好的时序逻辑更新信息。仔细阅读实验指导书，并分析流水线运行原理图 figure 1 后，实现buffer如下：

- IF_ID_buffer：在这个buffer中，我们需要接受IF阶段的指令以及PC。

```
        reg [31:0] IF_ID_PC,IF_ID_Inst;
```

更新逻辑如下：

```
        IF_ID_PC <= PC;
        IF_ID_Inst <= Inst;
```

即在时钟上升沿的always块中更新变量即可。

- ID_EX_buffer：在这个buffer中我们需要存储ID阶段读取的数据以及产生的控制信号。变量定义如下：

```
        //data
        reg [4:0] ID_EX_Rt,ID_EX_Rd,ID_EX_ReadReg1,ID_EX_ReadReg2;
        reg [31:0] ID_EX_Ext,ID_EX_RegReadData1,
                ID_EX_RegReadData2,ID_EX_PC,ID_EX_Inst;
        //ctr
        reg ID_EX_RegDst,ID_EX_ALUSrc,ID_EX_RegWrite,ID_EX_MemRead,
            ID_EX_MemWrite,ID_EX_Branch,ID_EX_JalFlag,ID_EX_MemToReg;
        reg [3:0] ID_EX_ALUOp;
```

更新逻辑如下（展示代码有省略）：

```
        //data
        ID_EX_Rt <= Rt;
        ID_EX_Rd <= Rd;
        ID_EX_RegReadData1 <= RegReadData1;
        ID_EX_RegReadData2 <= RegReadData2;//...
        //control sig
        ID_EX_MemToReg <= MemToReg;
        ID_EX_RegDst <= RegDst;
        ID_EX_ALUSrc <= ALUSrc;//...
```

即在时钟上升沿的always块中更新变量即可。

- EX_MEM_buffer：在这个buffer中我们需要存储EX阶段产生的ALU计算结果，存储从ID_EX_buffer传过来的部分控制信号以及数据。变量定义如下：

```
        //data
        reg [4:0] EX_MEM_WriteReg;
        reg [31:0] EX_MEM_RegReadData2,EX_MEM_ALURes;
        reg EX_MEM_Zero;
        //ctr
        reg EX_MEM_RegWrite,EX_MEM_MemWrite,EX_MEM_MemRead,
            EX_MEM_Branch,EX_MEM_JalFlag,EX_MEM_MemToReg,EX_MEM_JrFlag;
```

更新操作即在时钟上升沿的always块中更新变量即可。

- MEM_WB_buffer：在这个buffer我们需要存储内存读取的数据以及必要的控制信号。变量定义如下：

```
        //data
        reg [31:0] MEM_WB_MemReadData,MEM_WB_ALURes;
        reg [4:0] MEM_WB_WriteReg;
        //ctr
        reg MEM_WB_MemToReg,MEM_WB_RegWrite,MEM_WB_JrFlag,MEM_WB_JalFlag;
```

更新操作即在时钟上升沿的always块中更新变量即可。

3.4 顶层模块实现

细致的原理分析为我实现顶层模块打下了基础，顶层模块需要实现的内容有：

- 编写基础流水线处理器的功能
- 结合NOP实现branch指令
- 针对数据冒险实现Forwarding
- 针对数据冒险实现STALL
- 针对控制冒险实现分支预测器

3.4.1 流水线的五个阶段实现

1. **IF（Instruction Fetch）取指阶段。**

   这个阶段比较简单，实例化Inst memory模块，将PC传入，读取指令即可。

```
    reg [31:0] PC;
    wire [31:0] Inst;
    instMemory inst_memory(
        .PC_Clk(Clk),
        .address(PC),
        .readData(Inst)
    );
```

2. **ID（Instruction Decode）指令译码**

   在ID阶段，对指令进行解析，并进行寄存器文件的读取，同时生成控制信号（Ctr），此外还需要对立即数进行有/无符号扩展。

```
    //ID stage
    assign ReadReg1 = IF_ID_Inst[25:21];
    assign ReadReg2 = IF_ID_Inst[20:16];
    Registers registers(
        .reset(Reset),
        .Clk(Clk),
```

```
        .readReg1(ReadReg1),
        .readReg2(ReadReg2),
        .writeReg(MEM_WB_WriteReg),
        .writeData(RegWriteData),
        .regWrite(MEM_WB_RegWrite ^ MEM_WB_JrFlag),
        .jalFlag(MEM_WB_JalFlag),
        .readData1(RegReadData1),
        .readData2(RegReadData2)
    );
    Ctr ctr (
        .opCode(IF_ID_Inst[31:26]),
        .regDst(RegDst),
        .aluSrc(ALUSrc),
        .memToReg(MemToReg),
        .regWrite(RegWrite),
        .memRead(MemRead),
        .memWrite(MemWrite),
        .branch(Branch),
        .aluOp(ALUOp),
        .signExtFlag(SignExtFlag),
        .jalFlag(JalFlag)
    );
    signext sign_ext(
        .inst(IF_ID_Inst[15:0]),
        .signExtFlag(SignExtFlag),
        .data(Ext)
    );
```

3. **EX（Execute）执行**

在EX阶段主要是ALUCtr生成ALU控制信号，ALU根据控制信号对两个输入数据进行相应运算并输出运算结果。

```
    ALUCtr alu_ctr(
        .ALUOp(ID_EX_ALUOp),
        .Funct(ID_EX_Ext[5:0]),
        .jrFlag(JrFlag),
        .ALUCtrOut(ALUCtrOut)
    );
    ALU alu(
        .input1(ALUSrc1),
        .input2(ALUSrc2),
        .aluCtr(ALUCtrOut),
        .shamt(Shamt),
        .zero(Zero),
        .aluRes(ALURes)
    );
```

4. **MEM（Memory Access）内存访问**

在MEM阶段，对内存写指令，给内存写入相应数据，对内存读指令，读取相应地址的数据并输出即可。

```
    assign MemAddress = EX_MEM_ALURes;
    assign MemWriteData = EX_MEM_RegReadData2;
    dataMemory data_memory(
        .reset(Reset),
        .Clk(Clk),
        .address(MemAddress),
        .writeData(MemWriteData),
        .memRead(EX_MEM_MemRead),
        .memWrite(EX_MEM_MemWrite),
        .Op1(Op1),
        .Op2(Op2),
        .Out(Out),
        .readData(MemReadData)
    );
```

5. **WB（Write Back）写回**

在WB阶段，由于Registers模块已经在ID阶段定义，所以此处只需更改写入数据，则寄存器模块会根据此时控制信号自动写入。故这一部分的实现较为简单。

```
    assign RegWriteData = MEM_WB_MemToReg ? MEM_WB_MemReadData :
       MEM_WB_ALURes;
```

3.4.2 Jump指令与Branch指令的特殊实现

在实现完流水线的基本功能后，我对Jump指令和Branch指令进行了特殊实现。经过黄正翔助教的讲解，我了解到，为了提高流水线的效率，我需要额外新增模块处理这两条指令。

针对Jump指令，在IF阶段新增一个独立模块，对Jump指令进行解析，在下一个时钟周期直接将PC跳转到正确的地址，相比于将Jump放在EX阶段，这样可以消除Jump带来的流水线停顿（STALL）。代码如下：

```
    assign Jump = (Inst[31:26] == 6'b000010) ?  1'b1 : 1'b0;
    assign JumpAddress = {PC[31:28],(Inst[25:0] << 2)};
```

针对Beq指令，在ID阶段新增一个独立模块，对Beq指令进行出来，额外增加计算单元，在ID阶段就将PC是否跳转确定，相比于在MEM阶段执行Beq指令，这样可以最大程度上减少流水线停顿（STALL）。

```
    wire BneFlag;
    assign BneFlag = (IF_ID_Inst[31:26] == 6'b000101) ? 1 : 0;
    assign BranchPC = IF_ID_iniPC + 4 + (Ext << 2);
    assign BeqSrc1 = RegReadData1;
    assign BeqSrc2 = RegReadData2;
    assign ID_Zero = (BeqSrc1 == BeqSrc2) ? 1 : 0;
```

虽然这两条指令的特殊实现会使代码的逻辑比原来更加复杂，但从提高流水线处理器表现的角度来看，这些工作能够提高流水线处理器的性能，是十分有意义的。

3.4.3 Forwarding实现

为了减少Data hazard带来的流水线停顿，我将Forwarding加入了我的流水线，使得当结果未被写回内存或者寄存器时就能够被后续指令使用，提高流水线运行效率。和原理分析中提到的一样，我的Forwarding包括两种：ID_EX_buffer到EX阶段的数据传递以及EX_MEM_buffer到EX阶段的数据传递，对此，我额外设计了一个forwarding模块，让EX阶段方便地决定操作数，将forwarding功能封装在forwarding模块内部，使用时只需要将IO参数正确赋值即可得到正确输出。

我的forwarding模块实现如下：

```
module forwardingUnit(
    ...
);
assign ALUSrc1 = EX_MEM_RegWrite && (EX_MEM_WriteReg == ID_EX_ReadReg1)
    ? EX_MEM_RegWriteData
            : MEM_WB_RegWrite && MEM_WB_WriteReg == ID_EX_ReadReg1 ?
                MEM_WB_RegWriteData
            : ID_EX_RegReadData1;
assign ALUSrc2 = EX_MEM_RegWrite && (EX_MEM_WriteReg == ID_EX_ReadReg2)
    ? EX_MEM_RegWriteData
                : MEM_WB_RegWrite && MEM_WB_WriteReg == ID_EX_ReadReg2
                    ? MEM_WB_RegWriteData
                : (ID_EX_ALUSrc && ID_OR_EX) ? ID_EX_Ext :
                    ID_EX_RegReadData2;
```

想要在EX阶段加入forwarding，只需要定义一个forwarding模块的实例化对象即可，对ALU的两个输入ALUSrc1 ALUSrc2进行更新，如果满足forwarding条件，则更新它们的值，实例化代码如下：

```
forwardingUnit forwarding_unit_in_EX(
    .ID_EX_RegReadData1(ID_EX_RegReadData1),
    .ID_EX_RegReadData2(ID_EX_RegReadData2),
    .ID_EX_Ext(ID_EX_Ext),
    .ID_EX_ALUSrc(ID_EX_ALUSrc),
    .ID_EX_ReadReg1(ID_EX_ReadReg1),
    .ID_EX_ReadReg2(ID_EX_ReadReg2),
    .EX_MEM_WriteReg(EX_MEM_WriteReg),
    .EX_MEM_RegWrite(EX_MEM_RegWrite),
    .EX_MEM_RegWriteData(EX_MEM_ALURes),
    .MEM_WB_WriteReg(MEM_WB_WriteReg),
    .MEM_WB_RegWrite(MEM_WB_RegWrite),
    .MEM_WB_RegWriteData(MEM_WB_RegWriteData),
    .ALUSrc1(ALUSrc1),
    .ALUSrc2(ALUSrc2)
);
```

如此ALUSrc1 ALUSrc2便能够变为正确的值，输入给ALU进行运算。

除此之外需要注意的是，Beq指令提前到了ID阶段，而它用的也是寄存器的值，所以需要对ID阶段的Beq指令也进行forwarding操作，同样只需要合理实例化forwarding模块即可。代码与EX阶段的forwarding相差不大。

```
    forwardingUnit forwarding_unit_in_ID(
        ...
        .ALUSrc1(BeqSrc1),
        .ALUSrc2(BeqSrc2)
    );
```

3.4.4 STALL实现

加入forwarding后，由于data hazard导致的流水线停顿情况大大减少，经过细致分析，可能导致流水线停顿的情况仅有以下几种：

1. use after load。当上一条指令是load到寄存器 $m，而下一条指令紧接着需要使用该寄存器的值，此时流水线需要暂停一个周期，将EX，ID，IF阶段暂停，不更新对应的buffer，等待下一个时钟周期可以进行forwarding后则流水线可以继续正常工作。从示意图 figure 5中可以更加清晰地看到这种情况下STALL的必要性。

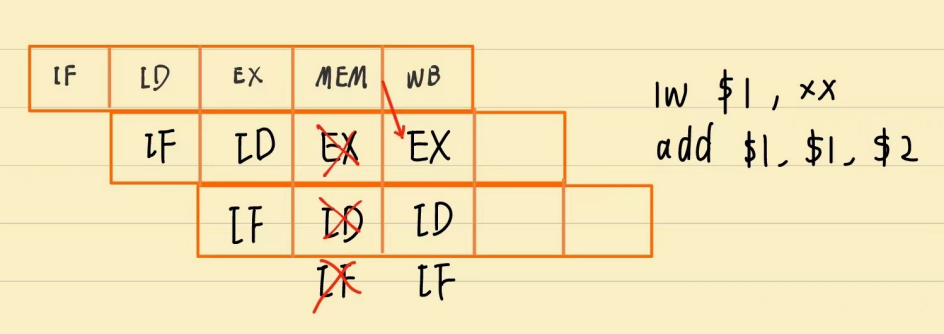

Figure 5: STALL示意图1

2. branch指令提前到ID阶段带来的STALL。如果Branch分支跳转留在EX阶段，则只会有一种STALL，即use after load。但是由于Branch提前到ID，所以此处可以导致两种STALL，本质上相近，都是因为Beq指令的两个操作数在先前的指令被更新，所以流水线需要停顿，等待forwarding相应结果，Branch的STALL与前一种STALL略有不同，其只需要暂停IF，ID的执行即可。

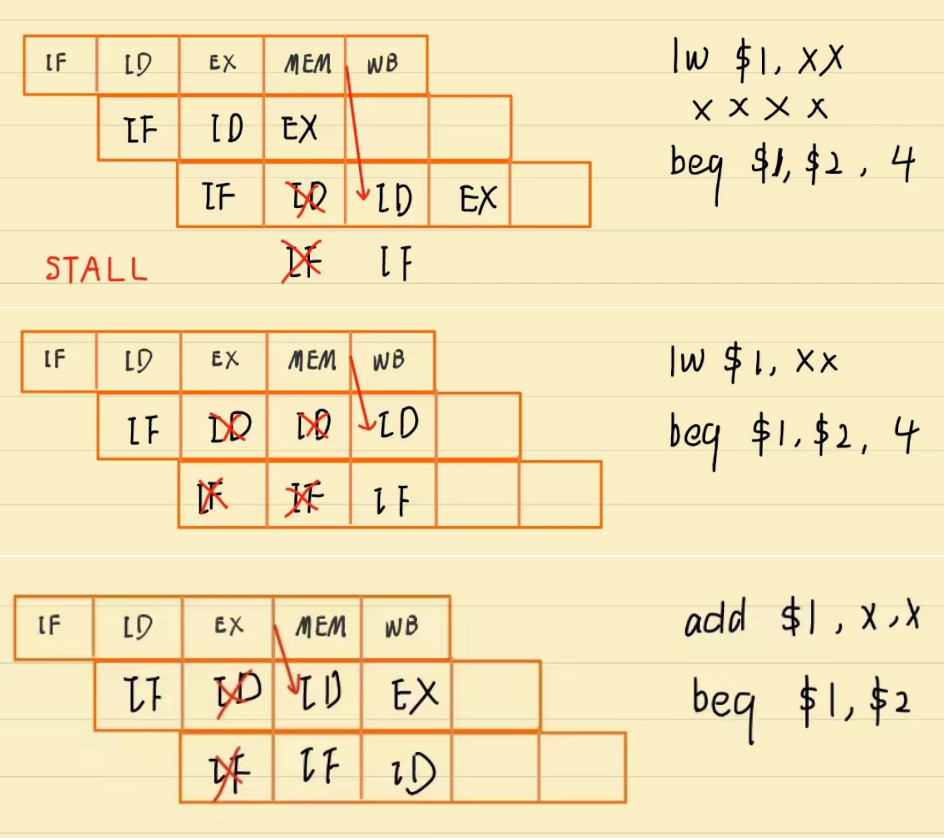

Figure 6: STALL示意图2-Branch带来的STALL

针对以上两种STALL，我设计了两个变量来区分它们，以便执行正确操作，分别为`STALL` 与`BranchSTALL`，对于它们的实现，在考虑清楚之后，发现只需要对`always @` 块进行特判改变即可，通过在某些情况下不更新某些buffer以及类似的操作实现。

首先进行STALL的检测，代码如下：

```
    if (ID_EX_MemToReg == 1 && ID_EX_RegWrite == 1 && (WriteReg == ReadReg1
        || WriteReg == ReadReg2))
        begin
            STALL <= 1;//next Clk STALL
        end
    if (Branch == 1 && ID_EX_RegWrite == 1 && (WriteReg == ReadReg1 ||
        WriteReg == ReadReg2))
        begin
            BranchSTALL = 1;//this Clk BranchSTALL
        end
    if (Branch == 1 && EX_MEM_MemToReg == 1 && EX_MEM_RegWrite == 1 &&
        (EX_MEM_WriteReg == ReadReg1 || EX_MEM_WriteReg == ReadReg2))
        begin
            BranchSTALL = 1;//this Clk BranchSTALL
        end
```

检测完成后，根据STALL与BranchSTALL的值进行buffer以及一些相关变量的更新

```
    // STALL == 1 || BranchSTALL == 1 , do not update PC
    ...
    // STALL == 1 || BranchSTALL == 1 , do not update IF_ID_buffer
    ...
    // STALL == 1 do not update ID_EX_buffer
    ...
    // BranchSTALL == 1,we do not update ID_EX_buffer and we need to
        transfer write disable signals to the buffer to prevent error
        occurring.
    if (!BranchSTALL)
    begin
        update ID_EX_buffer...
    end
    else
    begin
        ID_EX_MemToReg <= 0;
        ID_EX_RegWrite <= 0;
        ID_EX_MemRead <= 0;
        ID_EX_MemWrite <= 0;
        ID_EX_Branch <= 0;
        ID_EX_JalFlag <= 0;
    end
    // STALL == 1 do not update EX_MEM_buffer and we need to transfer write
        disable signals to the buffer to prevent error from occurring.
    if(!Reset && !STALL)
    begin
        update EX_MEM_buffer...
    end
    else
    begin
        EX_MEM_RegWrite <= 0;
        EX_MEM_MemWrite <= 0;
        EX_MEM_Branch <= 0;
        EX_MEM_JalFlag <= 0;
        EX_MEM_JrFlag <= 0;
    end
```

以上是两种STALL信号会影响到的部分，其余部分无论STALL信号为何值，均正常更新即可。

对于两个STALL信号的复位，代码如下：

```
    if(STALL == 1)
        begin
            STALL <= 0;
        end
    if(BranchSTALL)
        begin
            BranchSTALL = 0;
        end
```

注意，此处展示的代码顺序与我在具体实现时略有不同，具体实现时，我根据更新先后顺序进行了精心设计，区别使用了¡=与=两种赋值。

3.4.5 NOP实现

Branch在ID阶段给出分支跳转结果，执行结束后，需要检查PC的值是否正确，如不正确，需要将错误装载进流水线的指令丢弃（或者说冲刷（flush））掉，见如下示意图：

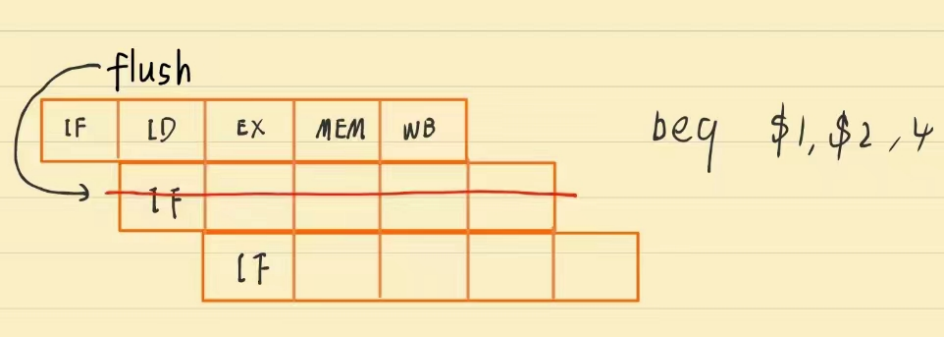

Figure 7: NOP示意图

在我的MIPS多周期流水线处理器中，NOP指令就是专门为其定义的，为了flush掉这条指令我们只需要在更新buffer时，不将错误指令的内容更新到IF_ID_buffer中，而是将NOP指令信息传给它，那么这条指令会变成空指令，不会影响流水线指令执行结果。

NOP同样定义一个名为NOP的变量作为标记，在always @块中更改添加相应代码即可，主体相关代码如下：

首先是NOP信号的赋值，当读取指令的PC与正确的PC不相同时，赋值NOP为1。

```
if (PCSrc == 1 && PC != BranchPC)
    begin
        if(!STALL && !BranchSTALL)
        begin
            PC = BranchPC;
            NOP = 1;
        end
    end
    else if (Branch == 1 && PCSrc == 0 && PC != IF_ID_iniPC + 4)
    begin
        if(!STALL && !BranchSTALL)
        begin
            Tmp = IF_ID_iniPC;
            PC = IF_ID_iniPC + 4;
            NOP = 1;
        end
    end
```

NOP的处理仅仅针对IF_ID_buffer，当NOP信号为高电平时，给Inst赋值NOP的指令，代码如下：

```
    if (!Reset && !BranchSTALL && !STALL && NOP == 1) //NOP for branch
            begin
                IF_ID_PC <= PC;
                IF_ID_Inst <= 32'b111111_00000_00000_00000_00000_000000;//NOP
                NOP <= 0;//reset NOP
            end
        else if(!Reset && !BranchSTALL && !STALL)
            begin
                ...
            end
```

对于NOP的指令定义，在我的流水线处理器中我将其定义为32'b1111110000...，我在Ctr中额外加入了NOP的opCode，当识别为NOP时，所有的写控制信号都为0，则这个信号不会影响流水线处理器运行结果。注意到，我在代码中也对NOP进行了复位，至此，NOP实现完毕。

3.4.6 静态（predict-not-taken）/动态分支跳转预测器实现

这部分的实现与其他部分的实现相对独立，由于我在设计流水线时，在IF阶段添加了predictor模块实例化对象，为predictor模块的设计留出了接口，所以在这部分只需要对predictor.v进行编写即可。

接下来分别进行静态（predict-not-taken）分支跳转预测器和基于二位饱和计数器的动态分支跳转预测器的实现。

1. 静态（predict-not-taken）分支跳转预测器

   这部分比较好实现，一直预测不跳转即可，对PC直接加4输出。

```
    module predictor(
        input [31:0] iniPC,
        output [31:0] predBranchPC
    );
        assign predBranchPC = iniPC + 4;
```

2. 基于二位饱和计数器的动态分支跳转预测器

   首先简单对二位饱和计数器进行介绍，基于二位饱和计数器的动态分支跳转预测器使用一种状态机来记录分支指令的历史行为，并根据历史行为进行预测。

   要实现该动态分支跳转预测器，我们需要定义相关变量。在predictor.v文件中，可以使用以下代码实现二位饱和计数器：

```
    reg [31:0] BTB[63:0];      //Branch target buffer
    reg [1:0] BHT[4095:0];     //Branch History Table
    wire [31:0] BTB_VALUE;//6bits index
    wire [1:0] BHT_VALUE;  //1024 * 2 * 2 12 bits
    reg [31:0] tmp;
    integer i;
    assign BTB_VALUE = BTB[iniPC & 6'b111111];
    assign BHT_VALUE = BHT[iniPC & 12'b111111111111];

```

```
    //predict
    assign predBranchPC = (BHT_VALUE & 2'b10) ? BTB_VALUE : iniPC + 4;
    //upd in ID
    always @ (iniPC or iniPCForUpd or branchTaken or branchPC)
    begin
        if(branch)
        begin
            tmp = branchPC;
            if (branchTaken == 1)
            begin
                if(BHT_VALUE < 2'b11) BHT[iniPCForUpd &
                    12'b111111111111] = BHT[iniPCForUpd &
                    12'b111111111111] + 1;
                BTB[iniPCForUpd & 6'b111111] = branchPC;
            end
            else
            begin
                if(BHT_VALUE > 2'b00) BHT[iniPCForUpd &
                    12'b111111111111] = BHT[iniPCForUpd &
                    12'b111111111111] - 1;
            end
        end
    end
```

其中， BTB-Branch target buffer，记录历史跳转地址，便于进行跳转目的地预测。

BHT-Branch History Table，记录跳转信息，决定是否跳转预测结果，每一个单元由两位组成，是一个状态机， 11和10表示这条分支会跳转；01和00表示分支不会跳转。在update逻辑中进行BHT与BTB的更新即可。

至此，我便成功实现了两种分支跳转预测器，在实际应用中，可以根据情况选择不同的预测器类型。

3.5 指令扩展实现

在完成流水线处理器的基本内容后，我进行了指令扩展，将我的流水线处理器从支持16条指令扩展到能够支持31条指令。首先我们需要明确需要添加哪些指令，再每次针对一类指令进行统一添加即可。需要添加的指令有：addu,subu,sltu,sllv,lrlv,srav,sra,srav,addiu,andi,lui,xori,ori,bne,slti,sltiu

由于这个指令部分需要添加的指令较多，修改的代码过于长，我仅取其中一些例子进行讲解。

- 对R type的扩展，比如sra,sllv,sltu等等，我们需要更改的模块有ALU，ALUCtr，将相应的操作加入到ALU中，将对应的ALU控制信号输出添加到ALUCtr模块中，我事先自己进行编码设计后将它们成功添加到了流水线处理器中。

```
        //aluctr.v
        10'b0010_101011://sltu
```

```
        begin
            ALUCtrOut = 5'b01101;
            jrFlag = 0;
        end
        10'b0010_000011://sra
        begin
            ALUCtrOut = 5'b01110;
            jrFlag = 0;
        end
        10'b0010_000100://sllv
        begin
            ALUCtrOut = 5'b01111;
            jrFlag = 0;
        end
        ...
        //ALU.v
        5'b01101://sltu
        begin
            if ($unsigned(input1) < $unsigned(input2))
                aluRes = 1;
            else
                aluRes = 0;
        end
        5'b01110://sra
        begin
            aluRes = input2 >>> shamt;
        end
        5'b01111://sllv
        begin
            aluRes = input2 << input1;//rt << rs
        end
        ...
```

- 对几个特殊指令的扩展（BNE，Jr）。

  由于这些特殊指令涉及到计算与跳转，我将它们的实现与Beq的实现结合起来，都放在ID阶段，这样他们可以共享forwarding,STALL操作，可以减少额外的代码量。具体来说，我在Top模块中添加了一些变量，将它们与Beq阶段的变量巧妙结合起来利用。代码实现如下：

```
    assign Beq = (Inst[31:26] == 6'b000100 || Inst[31:26] == 6'b000101)
        ? 1'b1 : 1'b0;//BNE & Beq
    ...
    wire BneFlag;
    wire ID_JrFlag;
    assign ID_JrFlag = (IF_ID_Inst[31:26] == 6'b000000 &&
        IF_ID_Inst[5:0] == 6'b001000) ? 1 : 0;
    assign BneFlag = (IF_ID_Inst[31:26] == 6'b000101) ? 1 : 0;
    assign BranchPC = ID_JrFlag ? BeqSrc1 : IF_ID_iniPC + 4 + (Ext << 2);
    ...
```

```
    assign ID_Zero = (BeqSrc1 == BeqSrc2) ? 1 : 0;
    assign PCSrc = ID_JrFlag ? 1 : BneFlag ? ~(Branch & ID_Zero) :
        (Branch & ID_Zero);
```

经过仔细地思考与分析，我成功将我的MIPS流水线处理器扩展到了支持31条指令。

3.6 Cache编写

为了实现方便与代码简洁，我将Cache放在了data memory中编写，不过在实际应用中，Cache应被认为是一个与内存硬件上分开实现的模块。

我的Cache是set-associativity的，有32个Cache set，每个set有2个Cache line。

部分实现代码如下：

```
    ...
    reg [31:0] memFile[0:255];
    reg [24:0] tag[0:31][0:1];      //cache
    reg [31:0] cache[0:31][0:1];
    reg [63:0] timeStamp[0:31][0:1];
    reg valid[0:31][0:1];
    reg [4:0] setNumber;    //read Set Number
    reg [4:0] writeSetNumber;
    reg dirtyBit[0:31][0:1];

    if (memRead == 1)
            begin
                //access cache
                setNumber = ((address >> 2) & 5'b11111);   //get set index
                if (valid[setNumber][0] && tag[setNumber][0] == (address >>
                    7))
                begin
                    //hit
                    if(dirtyBit[setNumber][0] == 1)
                    begin
                        dirtyBit[setNumber][0] = 0;
                        cache[setNumber][0] = memFile[address >> 2];
                    end
                    readData = cache[setNumber][0];
                end
                else if (valid[setNumber][1] && tag[setNumber][1] ==
                    (address >> 7))
                    //hit ...
                else
                begin
                    //miss
                    if (valid[setNumber][0] == 0)
                    begin
                        dirtyBit[setNumber][0] = 0;
                        valid[setNumber][0] = 1;
```

```
                        tag[setNumber][0] = address >> 7;
                        cache[setNumber][0] = memFile[address >> 2];
                        timeStamp[setNumber][0] = curTime;
                        readData = cache[setNumber][0];
                    end
                    else if(valid[setNumber][1] == 0)
                        //miss ...
                    else //find a eviction with LRU
                    begin
                        if (timeStamp[setNumber][0] <
                            timeStamp[setNumber][1])
                        begin //evict 0
                            dirtyBit[setNumber][0] = 0;
                            valid[setNumber][0] = 1;
                            tag[setNumber][0] = address >> 7;
                            cache[setNumber][0] = memFile[address >> 2];
                            timeStamp[setNumber][0] = curTime;
                            readData = cache[setNumber][0];
                        end
                        else
                        //...
                    end
                end
            end
        end
```

在访问数据时，我通过计算地址得到Set的索引，并检查Tag值来判断是否命中Cache。

如果命中，我会检查dirtyBit，如果内存被写过，则需刷新Cache对应内容，并返回Cache中的数据。如果未命中，如果有空闲Cache line，则直接使用，反之依据LRU策略选择要替换的Cache Line，并将数据加载到Cache中。

以上是我的Cache实现，经过测试，发现我的Cache可以正常工作。

3.7 激励文件编写

在完成MIPS多周期流水线处理器代码的编写后，为了进行后续的仿真测试，我编写了如下激励文件。

```
module Top_tb(

    );
    reg Clk;
    reg Reset;
    always #10 Clk = ~Clk;
    Top cpu(
        .Clk(Clk),
        .Reset(Reset)
    );

```

```
    initial begin
        Clk = 0;
        Reset = 1;
        # 25;
        Reset = 0;
    end
endmodule
```

3.8 上板验证代码编写

经过lab05的上板验证，我对上板已经比较熟悉，我回顾了上板的操作，计划利用8个switch作为输入（4个switch作为输入1，另外4个作为输入2），运行指令最终将输入1与输入2相乘的结果输出在七段数码管上。我将两个输入分别赋值给memFile[0],memFile[1]，并将memFile[2]处的值赋给输出，这样便能够进行动态输入计算。主体代码修改部分如下

```
module Top(
    input clk_p,
    input clk_n,
    input [3:0] a,
    input [3:0] b,
    input Reset,

    output led_clk,
    output led_do,
    output led_en,
    output seg_clk,
    output seg_en,
    output seg_do
);
wire [31:0] Op1;
assign Op1 = {28'b0,a};
wire [31:0] Op2;
assign Op2 = {28'b0,b};
wire [31:0] Out;
```

其余代码为利用差分时钟对输入时钟进行调频，使其适合于我的流水线处理器，同时定义display模块。display定义如下：

```
display DISPLAY(
    .clk(Clk_25M),
    .rst(1'b0),
    .en(8'b11111111),
    .data({Op1,4'b0 ,Op2, 4'b0,Out[15:0]}),
    .dot(8'b00000000),
    .led(~Out[15:0]),
    .led_clk(led_clk),
    .led_en(led_en),
    .led_do(led_do),
```

```
    .seg_clk(seg_clk),
    .seg_en(seg_en),
    .seg_do(seg_do)
);
```

可以看到我将两个输入以及运算结果作为display模块的输入，显示在了七段数码管上，方便后续上板验证。此外我们还需要编写管脚约束文件，这个部分与lab05的内容一致，代码过于长在此暂且不展示，简单来说就是对`module Top`的输入输出端口进行约束连线，写在名为`lab06_xdc.xdc`文件中。

3.9 整体调试

完成所有工作后，我开始进行MIPS多周期流水线处理器的调试工作。由于前期工作比较完备，且编写时仔细小心，并未出现大的程序漏洞，经过1-2小时的调试，bug均被解决，我的程序能够正常运行且输出正确结果。

4 结果验证

4.1 仿真验证

完成设计源文件和激励文件编写后，我进行了仿真验证。通过点击 run behavior simulation 运行仿真。

在Top模块的initial块中，我使用系统任务`$readmemb`和`$readmemh`对指令内存和数据内存进行了初始化。

```
initial begin
    PC = 0;
    $readmemb("E:/Archlab/lab06/lab06.srcs/instruction.txt",
                                        inst_memory.memFile);
    $readmemh("E:/Archlab/lab06/lab06.srcs/data.txt",data_memory.memFile);
end
```

我使用了课程组提供的测试用例，该测试用例与lab05中的测试用例相同，是一个简单的乘法循环程序。初始化时，所有寄存器被置为0，memFile的第一个字是9，第二个字是13，乘法结果存储在第三个字中，预期最终的结果是117。得到如下波形图：

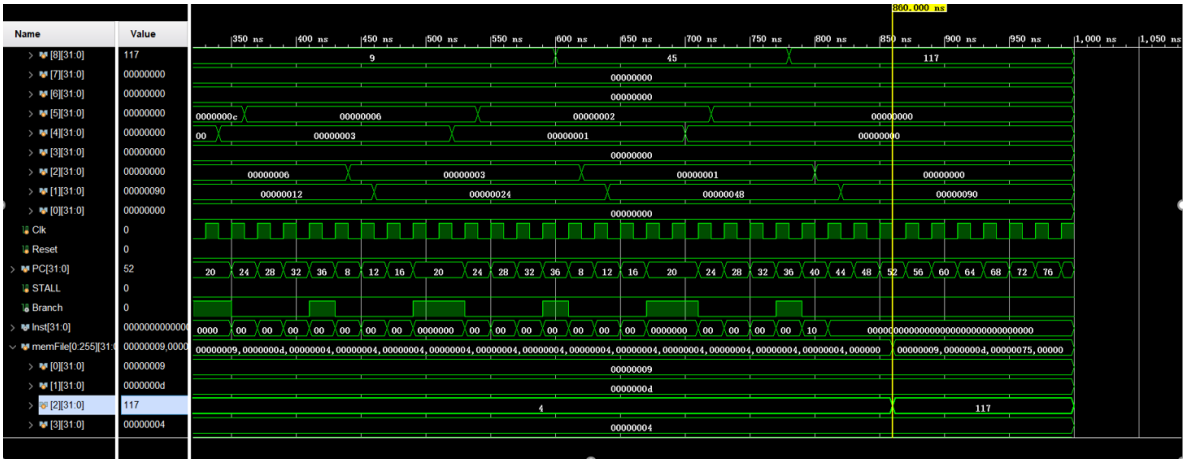

Figure 8: 仿真验证波形图

通过观察仿真波形图（如图8所示），可以看到，在经过若干个时钟周期后，我的MIPS多周期流水线处理器将运算结果117正确地存储到了memFile[2]中。这符合预期，表明我的MIPS多周期流水线处理器能够正常运行并得到正确的结果。

4.2 上板验证

在生成比特流文件后，我将其烧写到FPGA开发板上进行验证。经过验证，在输入为9和13的情况下，我的MIPS多周期流水线处理器成功执行了乘法程序，并在七段数码管上显示结果117。

通过上板验证，我测试了我的MIPS多周期流水线处理器在执行实际有意义的汇编程序时能够与硬件良好结合，并得到正确的结果。这进一步证明了我的MIPS多周期流水线处理器的正确性。

5 总结与反思

本次实验的总体内容是在实现了lab05的单周期处理器的基础上，进一步实现类MIPS多周期流水线处理器，在实现难度上有了不小的提升。

起初，我对具体如何实现这样一个流水线处理器并没有太多头绪。然而，通过仔细阅读实验指导书、积极上网查阅资料，并积极询问老师与助教，我逐渐了解了如何开始本次实验。在实验过程中，我首先在lab05的代码基础上，将各个阶段分开，实现了简单的流水线处理器。接下来，我开始处理forwarding、STALL和NOP这几个相对较难实现的环节。这几个环节的难点并不在于编写代码，而在于必须要将它们的实现逻辑想清楚，这样才能够保证它们能够正常工作。否则，一旦初始逻辑错误，将很难对程序进行调试，因为其基础逻辑就是错误的。

所以，我花了大量时间在思考流水线的运行逻辑上，具体到每个时钟周期，每种情况，最终，我想清楚了所有的情况，并将我的想法转化成代码写进我的MIPS多周期流水线处理器中。这样的思考虽然花费时间久，复杂程度高，但我认为这是值得的，因为在我在思考出来的那一刻，我感觉到我对流水线的结构有了更加深刻的了解。

在本次实验中，我还学习到了保持耐心的重要性。从基本逻辑的思考、框架实现到扩展部分的实现，我总共花费了十几个小时的时间。在这个过程中，我有时会感到烦躁，产生过质疑，会焦虑如果逻辑错误，或者程序无法正常运行的时候怎么办。然而，我并没有被这种焦虑所打败，

而是始终保持耐心。无论是对基本逻辑的思考还是对代码的编写和调试，我都耐心地一步一个脚印地向前推进。最终，这份耐心成功地引领我走向了成功。

虽然这次实验难度不小，但我觉得在完成的过程中，我并没有被这种困难吓倒，而是将更多的时间投入到思考中。在这样的过程中，我体会到了思考的乐趣，也体会到了本次实验的乐趣。

最后，我想说，经过此次实验，我进行Vivado工程开发的能力大大提高，对处理器的理解也变得更加深刻，这为我未来进行工程开发打下良好的基础，也为我未来在计算机系统结构领域的进一步学习打下坚实的基础。

6 致谢

由衷感谢王老师和黄老师的答疑与耐心指导。

由衷感谢雷帅助教和黄正翔助教的答疑与耐心指导。

由衷感谢孙雨林，张明宇，秦啸涵同学对我的帮助。