lab06 report

郑航 520021911347

1 实验目的

- 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
- 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
- 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
- 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
- 在4.的基础上， 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

2 实验原理

2.1 整体原理描述

在lab05里已经完成了单周期的处理器，多周期处理器相比单周期，只有在顶层模块和Ctr方面存在些许差别，同时五周期流水的话还需要增加四个段寄存器。除此之外，针对转发机制的前向通路，实现predict-not-taken 以及冲突检测的机制也需要额外实现一些模块。此外，将16条指令扩充为31条指令，主要增加的是对无符号指令的支持，需要对ALUCtr和ALU进行一定的功能扩充，下面详细描述

2.2 lab05原有模块：

本实验中，用到了lab05中除pc外的所有模块，其中data memory，signext，inst memory，mux，regmux与lab05完全相同，在本报告中不加赘述，而为了支持更多的指令以及前向通路等机制，Ctr，ALUCtr，ALU和registers需要做些许调整和功能增加，放在后半部分2.8中叙述

2.3 段寄存器Segment Registers原理分析

在本实验中，流水线共分为五个阶段，包括取指 (IF)、译码 (ID)、执行 (EX)、内存访问 (MA)和写回 (WB) ，总共需要设计四个段寄存器，分别位于IF到ID，ID到EX，EX到MA，MA到WB阶段之间，分别命名为IFID_REG，IDEX_REG，EXMA_REG和MAWB_REG，每个段寄存器的主要功能就是存储上一阶段产生，下一阶段需要使用的信息，也可以对传入的信息做适当的处理再输出（相当于一个小型的控制器），四个段寄存器分为四个模块分别进行实现

2.3.1 IFID_REG

- 介绍：该段寄存器主要存储的信息就是pc和inst的值。由于需要支持stall功能和beq，jump等指令跳转的branch功能，而一旦需要stall就需要继续维持当前pc和inst，或一旦需要branch就需要在此先将后续的pc和inst都置为零防止误执行错误的指令，因此IFID_REG需要接受stall和branch信号。
- 输入：
  - clock：时钟

    - reset：reset信号
    - stall：停顿一周期
    - branch：条件跳转信号
    - 2位的ctr_signals_In：指令中的无条件跳转信号
    - pcIn：IF阶段的PC值
    - instIn：IF阶段的inst
- 输出：
    - pcOut：ID阶段的PC值
    - instOut：ID阶段的inst

2.3.2 IDEX_REG

- 介绍：该段寄存器主要存储的信息是pc和inst的值，stall和branch信号，ID阶段Registers中读取的两个数据，signext拓展的结果。同时，在数据转发过程中，会将Ctr处理inst所得的控制信号也一并存入并进行一些处理
- 输入：
    - clock：时钟
    - reset：reset信号
    - stall：停顿一周期
    - branch：条件跳转信号
    - pcIn：ID阶段的PC值
    - instIn：ID阶段的inst
    - 32位的dataIn1和dataIn2：Registers读出的数据
    - 32位的extendIn：signext拓展的结果
    - 5位的rdIn：Ctr解析的register destination
    - 4位的aluop_in：aluop
    - 8位的ctr_signal_in：Ctr的控制信号
- 输出：
    - pcOut：EX阶段的PC
    - dataOut1和dataOut2：EX阶段时，ID阶段Registers读出的数据
    - extendOut：EX阶段时，ID阶段signext拓展的结果
    - aluop_out：aluop
    - ctr_signal_out：EX阶段时，ID阶段得到的Ctr的控制信号
    - rdOut，inst0_5Out，inst6_10Out，inst16_20Out，inst21_25Out：对inst解码后所得的rs，rt，rd，funct和shamt值

具体代码类似IFID_REG，出于篇幅考虑放于附录

2.3.3 EXMA_REG

- 介绍：该段寄存器主要存储的信息是alu的计算结果，Registers读入的第二个数据（address），rd以及Ctr的控制信号，注意到此时我们没有存入pc和inst的值，原因是pc有关的跳转等操作已经在EX阶段完成，而inst也已经被充分解析得到了一系列的控制信号。
- 输入：
    - clock：时钟
    - reset：reset信号
    - 32位的aluResIn：alu计算结果
    - 4位的ctr_signals_In：MA及WB阶段需要的控制信号
    - 32位的readData2In：memory access的内存地址
    - 5位的regdestIn：寄存器写的register destination

- 输出：
    - 32位的aluResOut：alu计算结果
    - 4位的ctr_signals_Out：MA及WB阶段需要的控制信号
    - 32位的readData2Out：memory access的内存地址
    - 5位的regdestOut：寄存器写的register destination

具体代码类似IFID_REG，出于篇幅考虑放于附录

2.3.4 MAWB_REG

- 介绍：该段寄存器主要存储的信息是WB阶段待写入的结果，写入寄存器rd以及寄存器写的使能信号
- 输入：
    - clock：时钟
    - reset：reset信号
    - 32位的memDataIn：读出的memory数据or上阶段的alures
    - ctr_signals_In：寄存器写的enable信号
    - 5位的regdestIn：寄存器写的register destination
- 输出：
    - 32位的memDataOut：待写入registers的结果
    - ctr_signals_Out：寄存器写的enable信号
    - 5位的regdestOut：寄存器写的register destination

具体代码类似IFID_REG，出于篇幅考虑放于附录

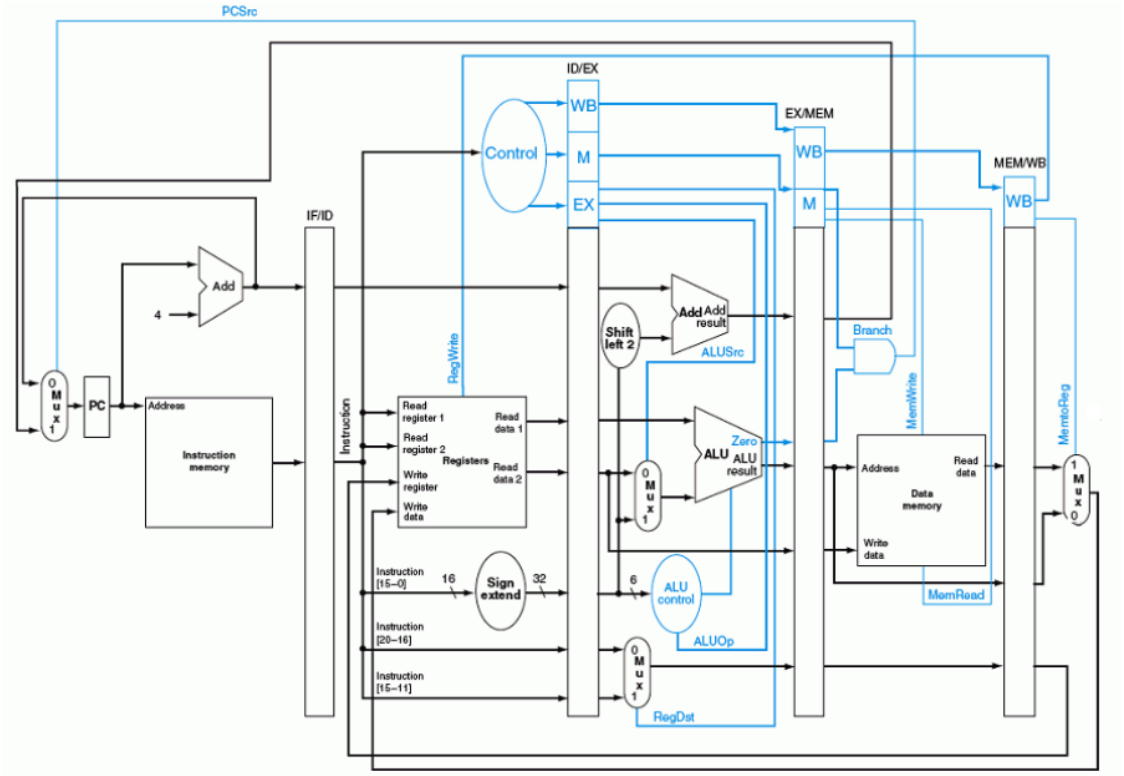
2.4 基础流水线的组装

只需要根据上述的各阶段分析和段寄存器内容，将各主要部件（后续优化的部件暂不包括在内）分别按如下放在各阶段里：

- IF：PC和inst_memory
- ID：Ctr，Registers，Signext
- EX：ALUCtr，ALU
- MA：data_memory
- WB：没有模块

然后在各阶段部件之间加入段寄存器即可，注意每一阶段的输入都是来源于上一阶段的段寄存器的输出

连接方式如下图：



2.5 冒险和stall

2.5.1 冒险的介绍

- 结构冒险：在同一个时刻有多条指令试图访问同一个结构单元则会产生结构冒险，将inst memory和data memory分开后，我们的处理器中就不再会产生结构冒险了
- 数据冒险：分为load-use hazard和read-after-write hazard两种
  - load-use hazard，一条指令从内存中读取某个数据到寄存器中，而下一条或两条指令对该数据进行访问，则由于此时该数据仍未写入寄存器，读到的是错误数据，会产生数据冒险
  - read-after-write hazard，前一条指令将数据写到某个寄存器中，而下一条寄存器就使用该数据进行运算等，也会因为数据还未写入而导致读到错误数据
- 控制冒险：在需要进行指令跳转时，该指令的继续执行会导致控制冒险，因此可以将该指令用nop替代，并刷新流水线

2.5.2 stall

在经过前向通路的优化后，只有在相邻指令发生load-use hazard时才需要使得整个流水线暂停一个周期，故实现比较简单，只需要判断在lw指令时，前一条指令的rd是否和后一条指令的rs或rt相同即可，代码如下：

```
assign STALL = ((ID_EX_INST_RT == IF_ID_INST [25 : 21]) | (ID_EX_INST_RT ==
IF_ID_INST [20 : 16])) &                    ID_EX_CTR[2] ;
```

2.6 前向通路

采用前向通路可以解决2.5.1，数据冒险中的所有read-after-write hazard以及间隔一条指令的load-use hazard（不间隔指令的则需要stall一个周期），故为了提高处理器的效率有必要添加前向通路的设计

在本实验中，我们设计了两条前向通路，两条通路分别将MA阶段读内存所得的数据放入下两条指令的EX阶段前和EX阶段ALU的计算结果放入下一条指令的EX阶段前，我们将前向通路封装成了一个模块Forward

- 输入：两个选路信号select1和select2，第一个选路信号选择MA阶段后的数据来源是memory还是Registers，第二个选路信号选择是要MA阶段的数据还是EX阶段ALU的计算结果，另外三个输入的分别是上述的三个数据
- 输出：一个32位输出out，表示最终作为前向通路结果传到ALU前的数据

我们总共需要两个这样的Forward模块，分别对rt和rs的数据来源进行前向传递。在进入ALU之前，还需要一个选路器选择是否将前向通路结果作为ALU的输入，较为简单，在此不加赘述

2.7 predict-not-taken

针对跳转指令可能存在的控制冒险，我们在本实验中采取预测不转移的方式（即每次都预测指令不进行跳转），并在预测错误时刷新后续流水线，并从正确的新指令地址开始执行，即可以消除控制冒险。

跳转指令分为条件跳转和无条件跳转这两类，针对：①条件跳转，我们每次都预测不转移，由于预测失败的结果必须在EX阶段后才可被发现，因此这种情况会影响两个指令周期；②无条件跳转，由于不需要运算进行判断，我们可以将跳转判断提前到ID阶段前进行，则其停顿为零，不会影响流水线

2.7.1 实现

其实现分为两部分，一是对后续指令的刷新，二是对PC的更新。

①对指令的刷新，可以依靠在IFIF_REG中增加对branch信号的判定，若需要branch，则将后续所有指令和信号等都置为默认的无效状态

②对PC的更新，我们设计了一个新的模块Jump_Ctr，用来根据branch相关信息更新PC，该过程应该在EX阶段任务完成后进行，具体的实现如下：

2.7.2 Jump_Ctr

- Jump_Ctr其实类似一个比较复杂的选路器，根据传入的两个控制信号以及两个branch信号，从四个传入的PC可能取值选择一个作为新的PC值
- 输入
    - 2位的ctr_signals：分别代表是否为jump或jr指令
    - pc_jump_In，pc_In、data和branch_dest是PC的四个可能取值，分别是jump和jal指令的目标地址，下一条指令地址，jr指令目标地址和条件跳转指令的目标地址
    - beq_signal，bne_signal分别代表branch指令需要跳转的两种情况：①beq指令且相等，②bne指令且不相等
- 输出：pcOut，代表更新后的下一个待执行指令的地址

2.8 指令的扩充（16->31）

从lab05中的16条指令扩充为如今的lab06指令，由于扩充的指令主要都是无符号数运算指令以及立即数运算等ALU相关的指令，因此无需增加任何新的模块等，只需要将ALUCtr和ALU的功能进行扩充即可（即ALUCtr中增加一些条件判断和ALU中利用>>>等Verilog自带的运算符），实现起来较为简单，在此不加赘述

2.9 Registers的功能扩充

关于jal指令，在lab05中我们只需在一个周期后将目标位置写入31号寄存器中即可，但在流水线处理器中，我们需要尽早实现跳转，将跳转地址写入31号寄存器，可以减少停顿的周期数，而若是等到WB阶段再写入则会浪费多个周期。因此我设计在Registers模块上增加两个接口：jalSign和jalDest，分别表示jal信号和jal的目标地址，若是jal指令则访问Registers一并将数据写入31号寄存器

3 实验过程

3.1 Segment Registers

原理已经在前边完整说明，实现主要就是进行数据的交换以及少量的信号处理，下面展示IDEX_REG的部分代码，完整代码及其他段寄存器代码较长，此处不加展示，详见附录

```
module IDEX_REG(
    input clock,
    input reset,
    input stall,
    input branch,
    input [31:0] pcIn,
    input [31:0] instIn,
    input [31:0] dataIn1,
    input [31:0] dataIn2,
    input [31:0] extendIn,
    input [4 : 0] rdIn,
    input [3 : 0] aluop_in,
    input [7 : 0] ctr_signal_in,
    output reg [31:0] pcOut,
    output reg [31:0] dataOut1,
    output reg [31:0] dataOut2,
    output reg [31:0] extendOut,
    output reg [3 : 0] aluop_out,
    output reg [7 : 0] ctr_signal_out,
    output reg [4:0] rdOut,
    output reg [4:0] inst16_20Out,
    output reg [4:0] inst21_25Out,
    output reg [5:0] inst0_5Out,
    output reg [4:0] inst6_10Out
    );

always @ (posedge clock)
    begin
       if(stall||branch)
       begin
```

```
            pcOut <= pcIn;
            dataOut1 <= 0;
            dataOut2 <= 0;
            extendOut <= 0;
            aluop_out <= 4'hf;
            ctr_signal_out <= 0;
            rdOut <=0;
            inst16_20Out <= 0;
            inst21_25Out <= 0;
            inst0_5Out <= 0;
            inst6_10Out <= 0;
        end else
        begin
         pcOut = pcIn;
         dataOut1 <= dataIn1;
         dataOut2 <= dataIn2;
         extendOut <= extendIn;
         aluop_out <= aluop_in;
         ctr_signal_out <= ctr_signal_in;
         rdOut <= rdIn;
         inst16_20Out <= instIn[20:16];
         inst21_25Out <= instIn[25:21];
         inst0_5Out <= instIn[5:0];
         inst6_10Out <= instIn[10:6];
         end
    end
```

3.2 流水线组装

如原理部分所述，将各主要部件和段寄存器按顺序连接起来即可，下面展示IFID寄存器和EXMA寄存器，详见Top文件中于附录

```
// Segment Registers IF_ID
wire [31 : 0] IF_ID_INST;
wire [31 : 0] IF_ID_PC;
wire [12 : 0] ID_CTR;
wire [3 : 0] ID_CTR_SIGNAL_ALUOP;
wire BRANCH;
wire STALL;
IFID_REG
ifid(.clock(clk),.reset(reset),.stall(STALL),.branch(BRANCH),.ctr_signals_In
(ID_CTR[12:11]),

.pcIn(PC),.instIn(IF_INST),.pcOut(IF_ID_PC),.instOut(IF_ID_INST));

// Segment Registers EXMA
wire [3 : 0] EX_MA_CTR;
wire [31 : 0] EX_MA_ALU_RES;
wire [31 : 0] EX_MA_REG_READ_DATA_2;
wire [4 : 0] EX_MA_REG_DEST;
EXMA_REG
exma(.clock(clk),.reset(reset),.aluResIn(EX_ALU_RES),.ctr_signals_In(ID_EX_C
TR [3 : 0]),

.readData2In(EX_ALU_INPUT_2_FORWARDING),.regdestIn(ID_EX_REG_DEST),
```

```
18 
   .aluResOut(EX_MA_ALU_RES),.ctr_signals_Out(EX_MA_CTR),.readData2Out(EX_MA_RE
   G_READ_DATA_2),
19                 .regdestOut(EX_MA_REG_DEST));
```

3.2 ALUCtr

ALUCtr的实现与lab03中的实现非常类似，主体也是一个casex块，然后针对所有可能case对ALUCtrOut进行赋值，具体的赋值对应关系在原理分析中已给出，注意做好default情况的处理

代码较长，此处不加展示，完整代码见附录

3.3 STALL

其实现见2.5.2，不再赘述

3.4 Forward

根据原理中的功能描述，Forward模块其实可以视作多个选路器的功能叠加

完整代码如下：

```
1  module Forward(
2             input select1,
3             input select2,
4             input [31:0] data1,
5             input [31:0] data2,
6             input [31:0] alures,
7             output wire [31:0] out
8         );
9  wire [31:0] tmp;
10 begin
11     assign tmp=select1?data1:data2;
12     assign out=select2?alures:tmp;
13 end
14 endmodule
```

3.5 Jump_Ctr

类似Forward，也是实现多信号多数据的选路功能，只是其逻辑比Forward还要更加复杂一些

完整代码如下：

```
1  module Jump_Ctr(
2             input [1:0] ctr_signals,
3             input [31:0] pc_jump_In,
4             input [31:0] pc_In,
5             input [31:0] data,
6             input beq_signal,
7             input bne_signal,
```

```
            input [31:0] branch_dest,
            output wire [31:0] pcOut
        );
wire [31:0] tmp1;
wire [31:0] tmp2;
wire [31:0] tmp3;

begin
    assign tmp1=ctr_signals[1]?pc_jump_In:pc_In;
    assign tmp2=ctr_signals[0]?data:tmp1;
    assign tmp3=beq_signal?branch_dest:tmp2;
    assign pcOut=bne_signal?branch_dest:tmp3;
end

endmodule
```

3.6 Registers

与lab05中的Registers相比，增加了jal指令相关的部分，只需要增加一个条件判断即可，其余部分完全一致，新增代码如下：

```
if(jalSign)
    regFile[31] = jalDest;
```

完整代码见附录：

3.7 Top

top模块的主体是对各模块进行实例化并用数据线进行连接，时序部分代码则是在每个clk下降沿检查stall并更新pc，其余的功能都如上封装在模块中了，只需按照正确的顺序和端口进行连接即可

具体代码较长，在此不加展示，详见附录

4 实验结果

4.1 数据和指令的编写

编写初始数据如下：

地址	数据	地址	数据	地址	数据	地址	数据
0	00000001	8	00001111	16	000000FF	24	00000024
1	00000010	9	00000111	17	00000100	25	00000025
2	00000100	10	00000100	18	00001100	26	00000026
3	00001000	11	00001000	19	00000100	27	00000027
4	00010000	12	00000010	20	00000010	28	00000028
5	00100000	13	00000000	21	00000001	29	00000029
6	01000000	14	0000000F	22	0000000F	30	00000030
7	10000000	15	00000FFF	23	000000FF	31	00000031

设计初始指令如下（含指令含义和指令执行结果）：

指令地址	指令（二进制）	指令	指令结果
0	100011 00000 00001 0000000000000000	lw $1, 0($0)	$1 = Mem[0] = 1
1	100011 00001 00010 0000000000000000	lw $2, 0($1)	$2 = Mem[1] = 16
2	100011 00000 00011 0000000000000010	lw $3, 2($0)	$3 = Mem[2] = 256
3	100011 00001 00100 0000000000001111	lw $4, 15($1)	$4 = Mem[16] = 255
4	000000 00001 00011 00101 00000 100000	add $5, $1, $3	$5 = 1 + 256 = 257
5	000000 00011 00100 00110 00000 100010	sub $6, $3, $4	$6 = 256 –255 = 1
6	000000 00101 00110 00111 00000 100100	and $7, $5, $6	$7 = 257 & 1 = 1
7	100011 00000 01001 0000000000001001	lw $9, 9($0)	$9 = Mem[9] = 273
8	100011 00000 01010 0000000000001010	lw $10, 10($0)	$10 = Mem[10] = 256
9	000000 01001 01010 01000 00000 100101	or $8, $9, $10	$8 = 273 |256 = 273
10	001000 01010 01011 0000000100000000	addi $11, $10, 256	$11 = 256 + 256 = 512
11	100011 00000 01100 0000000000001100	lw $12, 12($0)	$12 = Mem[12] = 16
12	001100 01100 01101 0000000011111111	andi $13, $12, 255	$13 = 16 & 255 = 16
13	100011 00000 10000 0000000000010000	lw $16, 16($0)	$16 = Mem[16] = 255
14	001101 10000 01111 0000000100000000	ori $15, $16, 256	$15 = 255 |256 = 511
15	101011 01110 01111 0000000000000111	sw $15, 7($14)	Mem[7] = 511
16	000000 00000 00110 10100 00100 000000	sll $20, $6, 4	$20 = $6 << 4 = 16
17	000000 00000 01010 10110 00100 000010	srl $22, $10, 4	$22 = $10 >> 4 = 16

指令地址	指令（二进制）	指令	指令结果
18	000000 00001 00011 11000 00000 101010	slt $24, $1, $3	$24 = 1
19	000000 01111 10000 11001 00000 101010	slt $25, $15, $16	$25 = 0
20	001111 00000 00010 1111111111111111	lui $2,65535	$2=-65536
21	000000 00001 00010 00011 00000 100001	addu $3,$1,$2	$3=-65535
22	000100 10100 10110 0000000000000001	beq $20, $22, 1	go to line 22;
23	000000 01111 10000 10010 00000 101010	slt $18, $15, $16	(skipped)
24	000010 00000000000000000000011000	j 24	go to line 24
25	001000 01011 01011 0000000000000010	addi $11, $11, 2	(skipped)
26	000011 00000000000000000000011010	jal 26	go to line 26
27	000100 00101 00110 0000000000000010	beq $5, $6, 2	go to line 2
28	000000 11111 00000 00000 00000 001000	jr $31	go to line 25
29	001000 01011 01011 0000000000000010	addi $11, $11, 2	(skipped)
30	001000 01011 01011 0000000000000010	addi $11, $11, 2	$11 = $11 + 2 = 514

4.2 激励文件编写

4.1.1 激励文件

激励文件中，主要分三个部分：

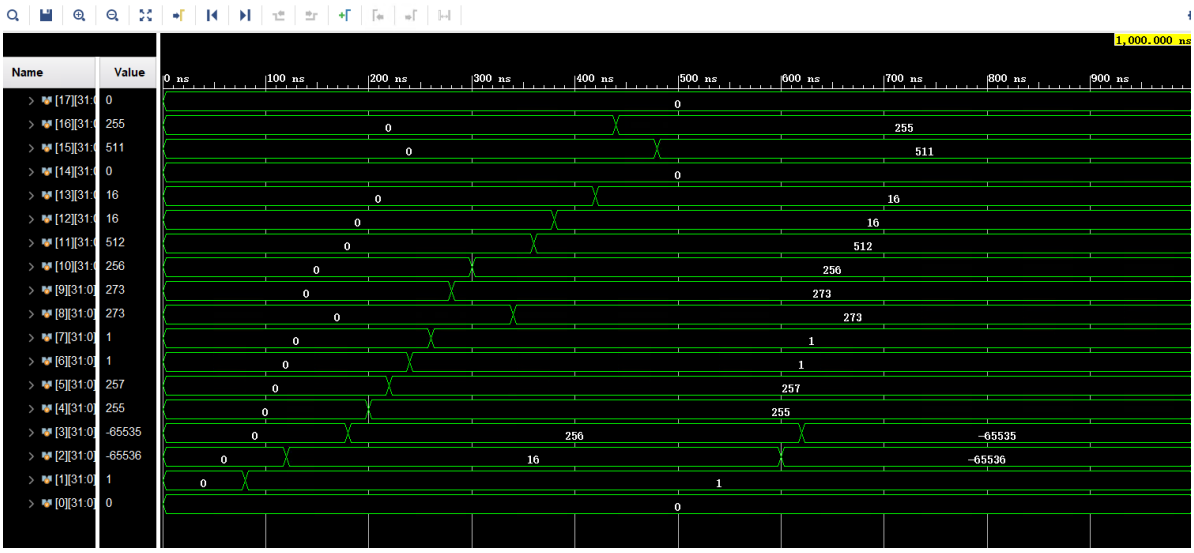
- 实例化Top顶层模块
- 初始化，包括初始化reset和clk信号，利用readmemb和readmemh函数读取外部数据和指令文件，对instFile和memFile进行初始化
- 设置时钟周期

测试代码与lab05基本一致，如下：

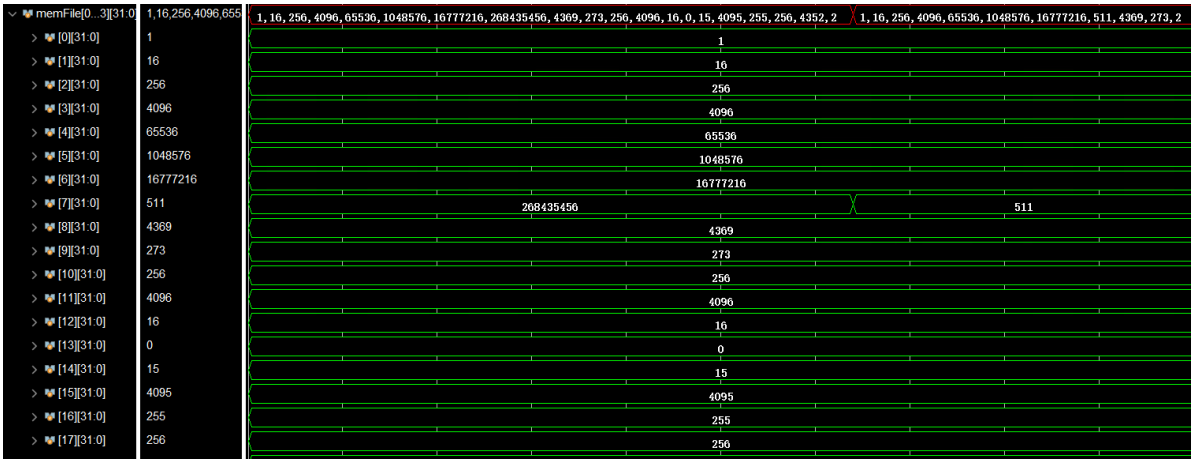
```
1 module top_tb(
2
```

```
    );
    reg clk;
    reg reset;

    Top top(.clk(clk), .reset(reset));

    initial begin
        $readmemb("C:/Archlabs/lab06/mem_inst.txt", top.inst_memory.instFile);
        $readmemh("C:/Archlabs/lab06/mem_data.txt", top.data_memory.memFile);
        reset = 1;
        clk = 0;
    end

    always #10 clk = ~clk;

    initial begin
        #10 reset = 0;
    end
endmodule
```

4.1.2 仿真图像



由图像可知，各寄存器值都符合我们预期，新的指令也可以得到正确的结果



由图像可知，511的值顺利写入了MemFile中的对应位置，sw指令也成功实现

5 反思总结

5.1 difficulty

- 本次实验中，梳理好流水线中各部件的连线和数据依赖关系是比较麻烦且容易出错的一件事，尤其是随着各模块的功能变得更加复杂，接口也变得更多，需要耐心的进行梳理，小心细致地进行连线
- 本实验的目标要求初看起来难度很高，而且没有什么好的切入点，假如同时考虑所有的功能那完成起来会特别费劲，后续我是按照实验指导书给的目标的顺序，由易到难地进行实现，搭好流水线的框架后一切就变得简单很多了

5.2 summary

本实验的难度中等，是在lab05的单周期处理器上进行改进，使其成为一个简单的，支持31条指令的带流水线的处理器，提高处理器的效率。在这个过程中，我通过自己动手实现的实践过程，对系统结构课内所学的冒险处理、前向通路、预测不转移等策略方法有了更深刻的了解，对于流水线的原理也更加理解，对学习系统结构课内知识有非常大的帮助

完成情况：有了lab05的基础之后，连线和设计模块对我来说变得轻松了很多，但由于涉及到的模块较多且连线复杂，我运用在lab05中学到的绘制连线图的方式，才得以较快的完成了任务，充分说明实践才是出真知的最好方式。同时，本次实验的线路较为复杂，且需要考虑到各周期各阶段的时序关系，非常锻炼我耐心和细致的学习习惯和实验素养，使我受益匪浅

6 附录

6.1 Ctr完整代码

```
module Ctr(
           input [5 : 0] opCode,
           input [5 : 0] funct,
           input nop,
           output regDst,
           output aluSrc,
           output regWrite,
           output memToReg,
```

```
            output memRead,
            output memWrite,
            output beqSign,
            output bneSign,
            output luiSign,
            output extSign,
            output jalSign,
            output jrSign,
            output [3 : 0] aluOp,
            output jump
        );
reg RegDst;
reg ALUSrc;
reg MemToReg;
reg RegWrite;
reg MemRead;
reg MemWrite;
reg Branch;
reg [3:0] ALUOp;
reg Jump;
reg ExtSign;
reg JalSign;
reg BeqSign;
reg BneSign;
reg LuiSign;
reg JrSign;

always @(opCode or funct or nop) begin
    if (nop) begin
        RegDst = 0;
        ALUSrc = 0;
        RegWrite = 0;
        MemToReg = 0;
        MemRead = 0;
        MemWrite = 0;
        BeqSign = 0;
        BneSign = 0;
        LuiSign = 0;
        ExtSign = 0;
        JalSign = 0;
        JrSign = 0;
        ALUOp = 4'b1111;
        Jump = 0;
    end
    else begin
        case(opCode)
            6'b000000:      // R Type
            begin
                if (funct == 6'b001000) begin    // jr
                    RegDst = 0;
                    RegWrite = 0;
                    JrSign = 1;
                    ALUOp = 4'b1111;
                end
                else begin
                    RegDst = 1;
                    RegWrite = 1;
                    JrSign = 0;
```

```
                    ALUOp = 4'b1101;
                end
                ALUSrc = 0;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 0;
                JalSign = 0;
                Jump = 0;
            end
            6'b001000:      // addi
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 1;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 1;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b0000;
                Jump = 0;
            end
            6'b001001:      // addiu
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 1;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 0;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b0001;
                Jump = 0;
            end
            6'b001100:      // andi
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 1;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
```

```
                ExtSign = 0;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b0100;
                Jump = 0;
            end
            6'b001101:      // ori
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 1;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 0;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b0101;
                Jump = 0;
            end
            6'b001110:      // xori
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 1;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 0;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b0110;
                Jump = 0;
            end
            6'b001111:      // lui
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 1;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 1;
                ExtSign = 0;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b1010;
                Jump = 0;
            end
            6'b100011:      // lw
```

```
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 1;
                MemToReg = 1;
                MemRead = 1;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 1;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b0001;
                Jump = 0;
            end
            6'b101011:      // sw
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 0;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 1;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 1;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b0001;
                Jump = 0;
            end
            6'b000100:      // beq
            begin
                RegDst = 0;
                ALUSrc = 0;
                RegWrite = 0;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 1;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 1;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b0011;
                Jump = 0;
            end
            6'b000101:      // bne
            begin
                RegDst = 0;
                ALUSrc = 0;
                RegWrite = 0;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
```

```
                BeqSign = 0;
                BneSign = 1;
                LuiSign = 0;
                ExtSign = 1;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b0011;
                Jump = 0;
            end
            6'b001010:      // slti
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 1;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 1;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b1000;
                Jump = 0;
            end
            6'b001011:      // sltiu
            begin
                RegDst = 0;
                ALUSrc = 1;
                RegWrite = 1;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 0;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b1001;
                Jump = 0;
            end
            6'b000010:      // Jump
            begin
                RegDst = 0;
                ALUSrc = 0;
                RegWrite = 0;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 0;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b1111;
```

```
                Jump = 1;
            end
            6'b000011:      // jal
            begin
                RegDst = 0;
                ALUSrc = 0;
                RegWrite = 0;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 0;
                JalSign = 1;
                JrSign = 0;
                ALUOp = 4'b1111;
                Jump = 1;
            end
            default: begin
                RegDst = 0;
                ALUSrc = 0;
                RegWrite = 0;
                MemToReg = 0;
                MemRead = 0;
                MemWrite = 0;
                BeqSign = 0;
                BneSign = 0;
                LuiSign = 0;
                ExtSign = 0;
                JalSign = 0;
                JrSign = 0;
                ALUOp = 4'b1111;
                Jump = 0;
            end
        endcase
    end
end

    assign regDst = RegDst;
    assign aluSrc = ALUSrc;
    assign memToReg = MemToReg;
    assign regWrite = RegWrite;
    assign memRead = MemRead;
    assign memWrite = MemWrite;
    assign aluOp = ALUOp;
    assign jump = Jump;
    assign extSign=ExtSign;
    assign jalSign=JalSign;
    assign bneSign=BneSign;
    assign beqSign=BeqSign;
    assign jrSign=JrSign;
    assign luiSign=LuiSign;

endmodule

```

6.2 ALUCtr完整代码

```
module ALUCtr(
           input [3 : 0] aluOp,
           input [5 : 0] funct,
           output [3 : 0] aluCtrOut,
           output shamtSign
       );
reg [3 : 0] ALUCtrOut;
reg ShamtSign;

always @ (aluOp or funct) begin
    if (aluOp == 4'b1101 || aluOp == 4'b1110) begin
        case (funct)
            6'b100000:      // add
                ALUCtrOut = 4'b0000;
            6'b100001:      // addu
                ALUCtrOut = 4'b0001;
            6'b100010:      // sub
                ALUCtrOut = 4'b0010;
            6'b100011:      // subu
                ALUCtrOut = 4'b0011;
            6'b100100:      // and
                ALUCtrOut = 4'b0100;
            6'b100101:      // or
                ALUCtrOut = 4'b0101;
            6'b100110:      // xor
                ALUCtrOut = 4'b0110;
            6'b100111:      // nor
                ALUCtrOut = 4'b0111;
            6'b101010:      // slt
                ALUCtrOut = 4'b1000;
            6'b101011:      // sltu
                ALUCtrOut = 4'b1001;
            6'b000000:      // sll
                ALUCtrOut = 4'b1010;
            6'b000010:      // srl
                ALUCtrOut = 4'b1011;
            6'b000011:      // sra
                ALUCtrOut = 4'b1100;
            6'b000100:      // sllv
                ALUCtrOut = 4'b1010;
            6'b000110:      // srlv
                ALUCtrOut = 4'b1011;
            6'b000111:      // srav
                ALUCtrOut = 4'b1100;
            6'b001000:      // jr
                ALUCtrOut = 4'b1111;
            default:
                ALUCtrOut = 4'b1111;
        endcase

        if (funct == 6'b000000 || funct == 6'b000010 || funct == 6'b000011)
            ShamtSign = 1;
        else
```

```
            ShamtSign = 0;
    end
    else begin
        ALUCtrOut = aluOp;
        ShamtSign = 0;
    end
end

assign aluCtrOut = ALUCtrOut;
assign shamtSign=ShamtSign;

endmodule

```

6.3 ALU完整代码

```
module ALU(
           input [31 : 0] inputA,
           input [31 : 0] inputB,
           input [3 : 0] aluCtrOut,
           output zero,
           output [31 : 0] aluRes
       );
reg Zero;
reg [31 : 0] ALURes;

always @ (inputA or inputB or aluCtrOut) begin
    casex (aluCtrOut)
        4'b000x:        // add
            ALURes = inputA + inputB;
        4'b001x:        // sub
            ALURes = inputA - inputB;
        4'b0100:        // and
            ALURes = inputA & inputB;
        4'b0101:        // or
            ALURes = inputA | inputB;
        4'b0110:        // xor
            ALURes = inputA ^ inputB;
        4'b0111:        // nor
            ALURes = ~(inputA | inputB);
        4'b1000:        // slt
            ALURes = ($signed(inputA) < $signed(inputB));
        4'b1001:        // slt (unsigned)
            ALURes = (inputA < inputB);
        4'b1010:        // left-shift
            ALURes = (inputB << inputA);
        4'b1011:        // right-shift
            ALURes = (inputB >> inputA);
        4'b1100:        // right-shift (arithmetic)
            ALURes = ($signed(inputB) >>> inputA);
        default:        // default
            ALURes = 0;
    endcase
    if (ALURes == 0)
        Zero = 1;
```

```
    else
        Zero = 0;
end
assign zero = Zero;
assign aluRes = ALURes;
endmodule
```

6.4 Registers完整代码

```
module Registers(
           input [4 : 0] readReg1,
           input [4 : 0] readReg2,
           input [4 : 0] writeReg,
           input [31 : 0] writeData,
           input regWrite,
           input reset,
           input clk,
           input jalSign,
           input [31 : 0] jalDest,
           output [31 : 0] readData1,
           output [31 : 0] readData2
       );

reg [31 : 0] regFile [31 : 0];
integer i;

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;
        if(jalSign)
            regFile[31] = jalDest;
    end
end
endmodule
```

6.5 IFID_REG完整代码

```
module IFID_REG(
    input clock,
    input reset,
    input stall,
    input branch,
    input [1:0] ctr_signals_In,
    input [31:0] pcIn,
```

```
    input [31:0] instIn,
    output reg [31:0] pcOut,
    output reg [31:0] instOut
    );
    initial begin
        pcOut = 0;
        instOut = 0;
    end
    always@ (reset)
    begin
       if(reset)
       begin
           pcOut=0;
           instOut=0;
        end
    end
    always @ (posedge clock)
    begin
        if(branch || ctr_signals_In[1] ||ctr_signals_In[0])
        begin
          pcOut<=0;
          instOut<=0;
        end else if(!stall)
        begin
          instOut=instIn;
          pcOut = pcIn;
        end
    end
endmodule
```

6.6 IDEX_REG完整代码

```
module IDEX_REG(
    input clock,
    input reset,
    input stall,
    input branch,
    input [31:0] pcIn,
    input [31:0] instIn,
    input [31:0] dataIn1,
    input [31:0] dataIn2,
    input [31:0] extendIn,
    input [4 : 0] rdIn,
    input [3 : 0] aluop_in,
    input [7 : 0] ctr_signal_in,
    output reg [31:0] pcOut,
    output reg [31:0] dataOut1,
    output reg [31:0] dataOut2,
    output reg [31:0] extendOut,
    output reg [3 : 0] aluop_out,
    output reg [7 : 0] ctr_signal_out,
    output reg [4:0] rdOut,
    output reg [4:0] inst16_20Out,
    output reg [4:0] inst21_25Out,
    output reg [5:0] inst0_5Out,
```

```
    output reg [4:0] inst6_10Out
    );

    initial begin
        pcOut <= 0;
        dataOut1 <= 0;
        dataOut2 <= 0;
        extendOut <= 0;
        aluop_out <= 0;
        ctr_signal_out <= 0;
        rdOut <=0;
        inst16_20Out <= 0;
        inst21_25Out <= 0;
        inst0_5Out <= 0;
        inst6_10Out <= 0;
    end

    always@ (reset)
    begin
       if(reset)
       begin
           pcOut <= pcIn;
           dataOut1 <= 0;
           dataOut2 <= 0;
           extendOut <= 0;
           aluop_out <= 0;
           ctr_signal_out <= 0;
           rdOut <=0;
           inst16_20Out <= 0;
           inst21_25Out <= 0;
           inst0_5Out <= 0;
           inst6_10Out <= 0;
        end
    end

    always @ (posedge clock)
    begin
       if(stall||branch)
       begin
           pcOut <= pcIn;
           dataOut1 <= 0;
           dataOut2 <= 0;
           extendOut <= 0;
           aluop_out <= 4'hf;
           ctr_signal_out <= 0;
           rdOut <=0;
           inst16_20Out <= 0;
           inst21_25Out <= 0;
           inst0_5Out <= 0;
           inst6_10Out <= 0;
       end else
       begin
        pcOut = pcIn;
        dataOut1 <= dataIn1;
        dataOut2 <= dataIn2;
        extendOut <= extendIn;
        aluop_out <= aluop_in;
        ctr_signal_out <= ctr_signal_in;
```

```
        rdOut <= rdIn;
        inst16_20Out <= instIn[20:16];
        inst21_25Out <= instIn[25:21];
        inst0_5Out <= instIn[5:0];
        inst6_10Out <= instIn[10:6];
        end
    end
endmodule
```

6.7 EXMA_REG完整代码

```
module EXMA_REG(
    input clock,
    input reset,
    input [31:0] aluResIn,
    input [3:0] ctr_signals_In,
    input [31:0] readData2In,
    input [4:0] regdestIn,
    output reg [31:0] aluResOut,
    output reg [3:0] ctr_signals_Out,
    output reg [31:0] readData2Out,
    output reg [4:0] regdestOut
    );

    initial begin
        aluResOut <= 0;
        ctr_signals_Out=0;
        readData2Out<=0;
        regdestOut <= 0;
    end

    always@ (reset)
    begin
       if(reset)
       begin
        aluResOut <= 0;
        ctr_signals_Out=0;
        readData2Out<=0;
        regdestOut <= 0;
        end
    end

    always @ (posedge clock)
    begin
        aluResOut <= aluResIn;
        ctr_signals_Out=ctr_signals_In;
        readData2Out<=readData2In;
        regdestOut <= regdestIn;
    end
endmodule
```

6.8 MAWB_REG完整代码

```
module MAWB_REG(
    input clock,
    input reset,
    input [31:0] memDataIn,
    input ctr_signals_In,
    input [4:0] regdestIn,
    output reg [31:0] memDataOut,
    output reg ctr_signals_Out,
    output reg [4:0] regdestOut
    );

    initial begin
        memDataOut <= 0;
        ctr_signals_Out <= 0;
        regdestOut <= 0;
    end

    always@ (reset)
    begin
       if(reset)
       begin
       memDataOut <= 0;
       ctr_signals_Out <= 0;
       regdestOut <= 0;
       end
    end
    always @ (posedge clock)
    begin
        memDataOut <= memDataIn;
        ctr_signals_Out <= ctr_signals_In;
        regdestOut <= regdestIn;
    end
endmodule
```

6.9 Top完整代码

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

// IF
reg [31 : 0] PC;
wire [31 : 0] IF_INST;

InstMemory inst_memory(.address(PC), .inst(IF_INST));

// Segment Registers IF_ID
wire [31 : 0] IF_ID_INST;
wire [31 : 0] IF_ID_PC;
wire [12 : 0] ID_CTR;
wire [3 : 0] ID_CTR_SIGNAL_ALUOP;
```

```
wire BRANCH;
wire STALL;
IFID_REG
ifid(.clock(clk),.reset(reset),.stall(STALL),.branch(BRANCH),.ctr_signals_I
n(ID_CTR[12:11]),

.pcIn(PC),.instIn(IF_INST),.pcOut(IF_ID_PC),.instOut(IF_ID_INST));

// ID
Ctr main_controller(.opCode(IF_ID_INST[31 : 26]), .funct(IF_ID_INST[5 :
0]), .nop(IF_ID_INST == 0),
                    .jump(ID_CTR[12]), .jrSign(ID_CTR[11]),
.extSign(ID_CTR[10]),
                    .regDst(ID_CTR[9]), .jalSign(ID_CTR[8]),
.aluOp(ID_CTR_SIGNAL_ALUOP),
                    .aluSrc(ID_CTR[7]), .luiSign(ID_CTR[6]),
.beqSign(ID_CTR[5]),
                    .bneSign(ID_CTR[4]), .memWrite(ID_CTR[3]),
.memRead(ID_CTR[2]),
                    .memToReg(ID_CTR[1]), .regWrite(ID_CTR[0]));

wire [31 : 0] ID_REG_READ_DATA_1;
wire [31 : 0] ID_REG_READ_DATA_2;
wire [4 : 0] WB_WRITE_REG;
wire [31 : 0] WB_REG_WRITE_DATA;
wire WB_CTR_SIGNAL_REG_WRITE;
Registers registers(.readReg1(IF_ID_INST[25 : 21]), .readReg2(IF_ID_INST[20
: 16]),
                    .writeReg(WB_WRITE_REG), .writeData(WB_REG_WRITE_DATA),
                    .regWrite(WB_CTR_SIGNAL_REG_WRITE), .clk(clk),
.reset(reset),
                    .jalSign(ID_CTR[8]), .jalDest(IF_ID_PC + 4),
                    .readData1(ID_REG_READ_DATA_1),
.readData2(ID_REG_READ_DATA_2));

wire [31 : 0] ID_EXT_RES;
signext SignExt(.extSign(ID_CTR[10]), .inst(IF_ID_INST[15 : 0]),
.data(ID_EXT_RES));

wire [4 : 0] ID_REG_DEST;
RegMux rt_rd_mux(.select(ID_CTR[9]),
                 .input0(IF_ID_INST[15 : 11]),
                 .input1(IF_ID_INST[20 : 16]),
                 .out(ID_REG_DEST));

// Segment Registers ID_EX
wire [31 : 0] ID_EX_PC;
wire [31 : 0] ID_EX_REG_READ_DATA_1;
wire [31 : 0] ID_EX_REG_READ_DATA_2;
wire [31 : 0] ID_EX_EXT_RES;
wire [3 : 0] ID_EX_CTR_SIGNAL_ALUOP;
wire [7 : 0] ID_EX_CTR;
wire [4 : 0] ID_EX_REG_DEST;
wire [4 : 0] ID_EX_INST_RT;
wire [4 : 0] ID_EX_INST_RS;
wire [5 : 0] ID_EX_INST_FUNCT;
wire [4 : 0] ID_EX_INST_SHAMT;
```

```
IDEX_REG idex(.clock(clk),.reset(reset),.stall(STALL),.branch(BRANCH),.pcIn(IF_ID_PC),.instIn(IF_ID_INST),

.dataIn1(ID_REG_READ_DATA_1),.dataIn2(ID_REG_READ_DATA_2),.extendIn(ID_EXT_RES),

.rdIn(ID_REG_DEST),.aluop_in(ID_CTR_SIGNAL_ALUOP),.ctr_signal_in(ID_CTR),.pcOut(ID_EX_PC),

.dataOut1(ID_EX_REG_READ_DATA_1),.dataOut2(ID_EX_REG_READ_DATA_2),
              .extendOut(ID_EX_EXT_RES),.aluop_out(ID_EX_CTR_SIGNAL_ALUOP),
              .ctr_signal_out(ID_EX_CTR),.rdOut(ID_EX_REG_DEST),
              .inst16_20Out(ID_EX_INST_RT),.inst21_25Out(ID_EX_INST_RS),
              .inst0_5Out(ID_EX_INST_FUNCT),.inst6_10Out(ID_EX_INST_SHAMT)
             );

// stall
assign STALL = ((ID_EX_INST_RT == IF_ID_INST [25 : 21]) | (ID_EX_INST_RT == IF_ID_INST [20 : 16])) & ID_EX_CTR[2] ;

// Ex
wire [3 : 0] EX_ALU_CTR_OUT;
wire EX_SHAMT_SIGNAL;
ALUCtr alu_controller(.aluOp(ID_EX_CTR_SIGNAL_ALUOP),
.funct(ID_EX_INST_FUNCT),
                      .aluCtrOut(EX_ALU_CTR_OUT),
.shamtSign(EX_SHAMT_SIGNAL));

//Forwarding wires
wire [31 : 0] EX_ALU_INPUT_1_FORWARDING;
wire [31 : 0] EX_ALU_INPUT_2_FORWARDING;

wire [31 : 0] EX_ALU_INPUT_1;
wire [31 : 0] EX_ALU_INPUT_1_TEMP;
wire [31 : 0] EX_ALU_INPUT_2;
Mux rs_forward_mux(.select(EX_SHAMT_SIGNAL),
                   .input0({27'b000000000000000000000000000,
ID_EX_INST_SHAMT}),
                   .input1(EX_ALU_INPUT_1_FORWARDING),
                   .out(EX_ALU_INPUT_1_TEMP));

Mux rs_lui_mux(.select(ID_EX_CTR[6]),
               .input0(32'h00000010),
               .input1(EX_ALU_INPUT_1_TEMP),
               .out(EX_ALU_INPUT_1));

Mux rt_forward_mux(.select(ID_EX_CTR[7]),
                   .input0(ID_EX_EXT_RES),
                   .input1(EX_ALU_INPUT_2_FORWARDING),
                   .out(EX_ALU_INPUT_2));

wire EX_ZERO;
wire [31 : 0] EX_ALU_RES;
ALU alu(.inputA(EX_ALU_INPUT_1), .inputB(EX_ALU_INPUT_2),
        .aluCtrOut(EX_ALU_CTR_OUT), .zero(EX_ZERO),
        .aluRes(EX_ALU_RES));

```

```
wire [31 : 0] BRANCH_DEST = ID_EX_PC + 4 + (ID_EX_EXT_RES * 4);

// predict-not-taken
wire BRANCH_CON_1 = ID_EX_CTR[5] & EX_ZERO;
wire BRANCH_CON_2 = ID_EX_CTR[4] & (~ EX_ZERO);
assign BRANCH = BRANCH_CON_1 | BRANCH_CON_2;
wire [31 : 0] PC_NEW;
Jump_Ctr jump_ctr(.ctr_signals(ID_CTR[12:11]),.pc_jump_In(((IF_ID_PC + 4) & 32'hf0000000) + (IF_ID_INST [25 : 0] * 4)),
                  .pc_In(PC + 4),.data(ID_REG_READ_DATA_1),.beq_signal(BRANCH_CON_1),.bne_signal(BRANCH_CON_2),
                  .branch_dest(BRANCH_DEST),.pcOut(PC_NEW));

// Segment Registers EXMA
wire [3 : 0] EX_MA_CTR;
wire [31 : 0] EX_MA_ALU_RES;
wire [31 : 0] EX_MA_REG_READ_DATA_2;
wire [4 : 0] EX_MA_REG_DEST;
EXMA_REG exma(.clock(clk),.reset(reset),.aluResIn(EX_ALU_RES),.ctr_signals_In(ID_EX_CTR [3 : 0]),

.readData2In(EX_ALU_INPUT_2_FORWARDING),.regdestIn(ID_EX_REG_DEST),

.aluResOut(EX_MA_ALU_RES),.ctr_signals_Out(EX_MA_CTR),.readData2Out(EX_MA_REG_READ_DATA_2),
              .regdestOut(EX_MA_REG_DEST));

// MA
wire [31 : 0] MA_MEM_READ_DATA;
dataMemory data_memory(.clk(clk), .address(EX_MA_ALU_RES), .writeData(EX_MA_REG_READ_DATA_2),
                       .memWrite(EX_MA_CTR[3]), .memRead(EX_MA_CTR[2]),
                       .readData(MA_MEM_READ_DATA));

wire [31 : 0] MA_DATA;
Mux reg_mem_mux(.select(EX_MA_CTR[1]),
                .input0(MA_MEM_READ_DATA),
                .input1(EX_MA_ALU_RES),
                .out(MA_DATA));

// Segment Registers MA_WB
wire [31 : 0] MA_WB_DATA;
wire MA_WB_CTR;
wire [4 : 0] MA_WB_REG_DEST;
MAWB_REG mawb(.clock(clk),.reset(reset),.memDataIn(MA_DATA),.ctr_signals_In(EX_MA_CTR[0]),
              .regdestIn(EX_MA_REG_DEST),.memDataOut(MA_WB_DATA),
              .ctr_signals_Out(MA_WB_CTR),.regdestOut(MA_WB_REG_DEST));

// forwarding mux
Forward forward_1(.select1(MA_WB_CTR & (MA_WB_REG_DEST == ID_EX_INST_RS)),.select2(EX_MA_CTR[0] & (EX_MA_REG_DEST == ID_EX_INST_RS)),

.data1(MA_WB_DATA),.data2(ID_EX_REG_READ_DATA_1),.alures(EX_MA_ALU_RES),.out(EX_ALU_INPUT_1_FORWARDING));
```

```
153
154 Forward forward_2(.select1(MA_WB_CTR & (MA_WB_REG_DEST ==
    ID_EX_INST_RT)),.select2(EX_MA_CTR[0] & (EX_MA_REG_DEST == ID_EX_INST_RT)),
155
    .data1(MA_WB_DATA),.data2(ID_EX_REG_READ_DATA_2),.alures(EX_MA_ALU_RES),.ou
    t(EX_ALU_INPUT_2_FORWARDING));
156
157 // WB
158 assign WB_WRITE_REG = MA_WB_REG_DEST;
159 assign WB_REG_WRITE_DATA = MA_WB_DATA;
160 assign WB_CTR_SIGNAL_REG_WRITE = MA_WB_CTR;
161
162
163 always @(posedge clk) begin
164     if (!STALL)
165         PC <= PC_NEW;
166 end
167
168 initial
169     PC = 0;
170 endmodule
```