《计算机组成原理实验》

实验报告

学 院 名 称：	计算机学院
专业（班级）：	信息与计算科学
学 生 姓 名：	张天泽
学　　号：	23336308
时　　间：	2024 年 12 月 21 日

成 绩：

简单流水线CPU设计与实现

一. 实验目的

1、设计一个具有旁路（Forwarding）功能的简单的MIPS流水线CPU；

2、熟悉流水线CPU的运行原理，知道三种冒险的内容和处理方法，重点了解旁路是如何实现的。

二. 实验内容

实现一个具有旁路功能的简单MIPS流水线CPU，完成10个乱序16位数的升序冒泡排序。实现旁路功能即可，可以外部手动实现阻塞。

三. 实验原理

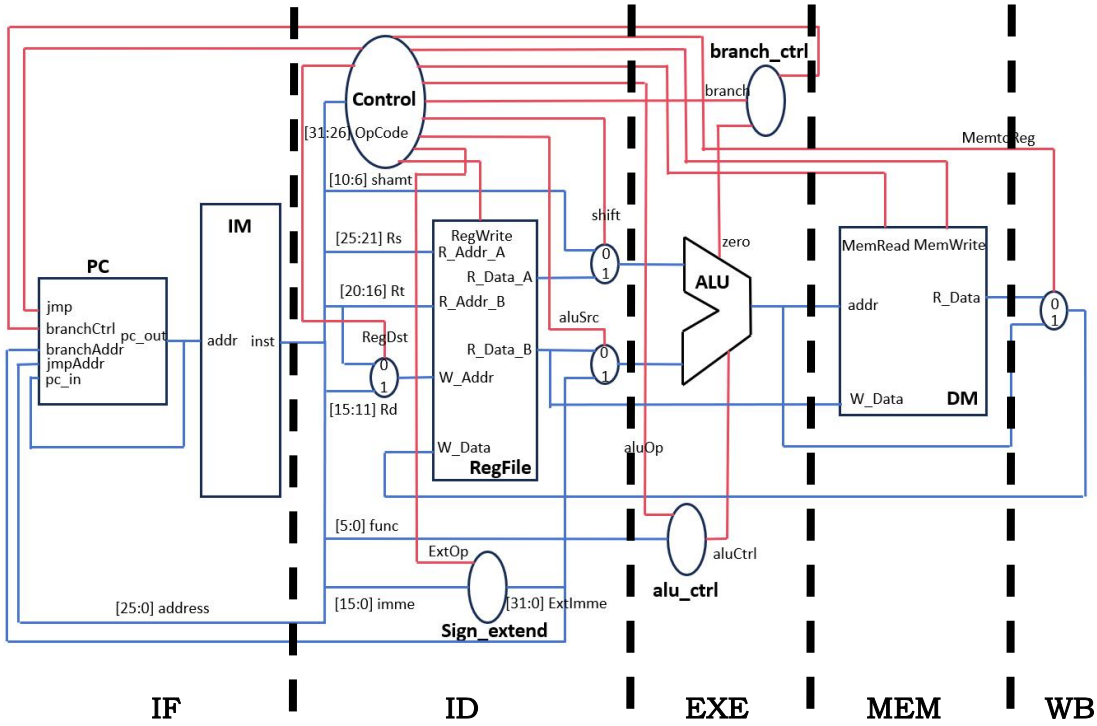
流水线CPU的执行分为五个阶段：取指（IF）、译码（ID）、执行（EXE）、访存（MEM）、写回（WB）。每个阶段对应一个时钟周期。在设计多周期CPU时我们知道，每个阶段执行完后就会空着，等待其他阶段执行完之后才再执行。而流水线CPU就是利用上了这段空窗期，让每个阶段一直执行，可以有5条指令同时执行。比如，IF阶段在取完该指令后，下一个时钟周期会取下一条指令，而上一个指令就被送到ID阶段执行。因此，流水线CPU提升效率的方法就是提高吞吐率，虽然每条指令仍要经过5个时钟周期，但是当流水线运作起来时，每一个时钟周期就可以完成一条指令，大大地提升了效率。

流水线CPU要用到的基本部件为：程序计数器（PC）、指令存储器、寄存器堆、算术逻辑单元（ALU）、数据存储器、控制单元、多路选择器、流水线寄存器和旁路单元。这

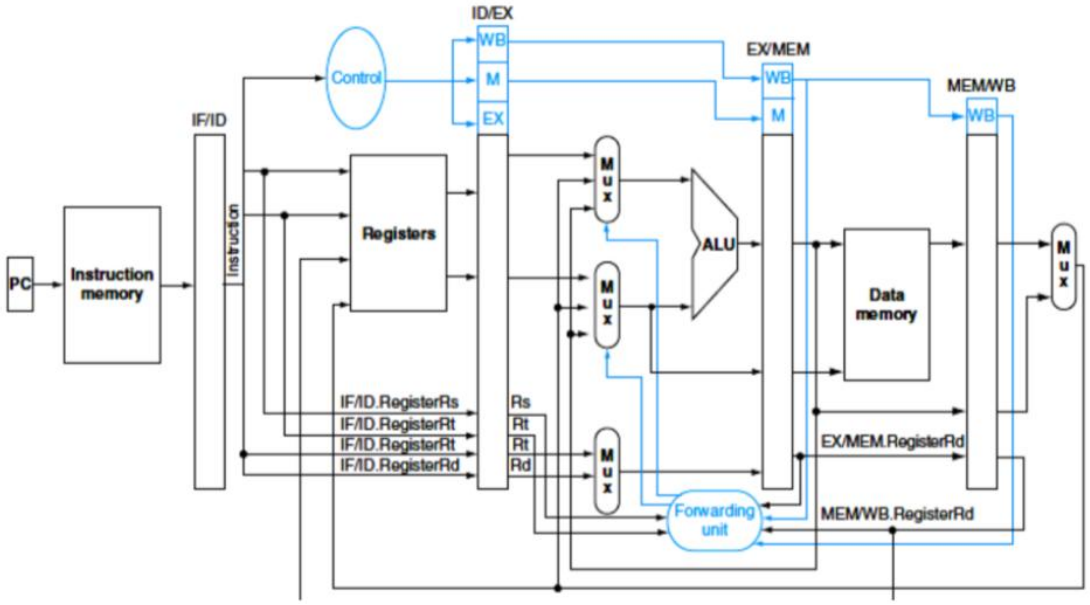
些部件的工作原理在“实验流程”板块再讲述。

设计流水线CPU的流程，大致分为四步：首先，要编写指令集、确定控制信号内容、确定ALU功能等，这些内容在“实验流程”板块再提供。其次，要处理三种冒险（实现旁路）。第三，设计数据通路，来完成不同部件之间的数据传递。最后，编写指令运行仿真。

流水线CPU的数据通路是在单周期CPU的基础上增加了流水线寄存器和旁路单元。篇幅限制，先给出单周期CPU的数据通路图，再给出流水线寄存器的数据通路。单周期CPU的数据通路图如下，在下面标注出了5个阶段。（注：为了区分，图中红色线条表示信号线，蓝色线条表示数据线）



下面是带有旁路和流水线寄存器的数据通路。



四. 实验器材

电脑一台，Xilinx Vivado 软件一套。

五. 实验过程与结果

（一） 实验设计思想

在本次实验中，我确定好了数据通路后，对每个部件进行单独编写，然后在顶层文件中把每个部件进行连接。这里运用了模块化的思想，将整个CPU拆解成不同模块再合并。

（二） 实验过程

“实验原理”板块提供了设计流水线CPU的流程，以下分为这四步来讲述。

（1） 指令集和控制信号

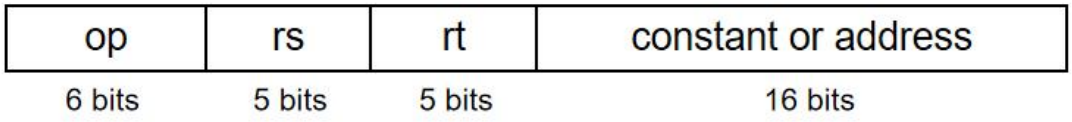
1、指令集

MIPS指令集分为三类：R型、I型和J型。每类指令的格式如下。

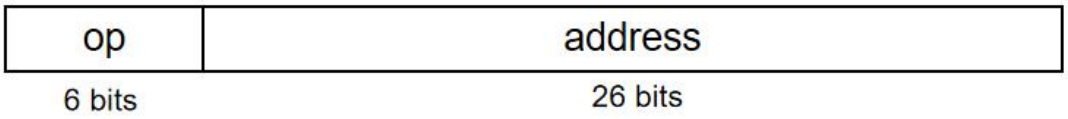
R型

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I型



J型



本次实验所用到的指令集是精简的MIPS指令集，方便将汇编语言转化成机器语言。

设计的CPU支持的指令及其具体功能如下：

OpCode	类型	func	指令	功能
000000	R	100000	add	Rd = Rs + Rt
000000	R	100010	sub	Rd = Rs - Rt
000000	R	100100	and	Rd = Rs & Rt
000000	R	100101	or	Rd = Rs | Rt
000000	R	100110	xor	Rd = Rs ^ Rt
000000	R	100111	nor	Rd = ~(Rs | Rt)
000000	R	101010	slt	Rd = (Rs < Rt) ? 1 : 0
000000	R	000000	sll	Rd = Rt << shamt
000000	R	000010	srl	Rd = Rt >> shamt
000000	R	000011	sra	Rd = Rt >>> shamt
100011	I	100011	lw	Rt = M[Rs + SignExtImme]
101011	I	101011	sw	M[Rs + SignExtImme] = Rt
000100	I	000100	beq	if(Rs == Rt) PC = PC + 4 + SignExtImme << 2
000101	I	000101	bne	if(Rs != Rt) PC = PC + 4 + SignExtImme << 2
001000	I	001000	addi	Rt = Rs + SignExtImme
001100	I	001100	andi	Rt = Rs & ZeroExtImme
001101	I	001101	ori	Rt = Rs | ZeroExtImme

续表

OpCode	类型	func	指令	功能
001110	I	001110	xori	Rt = Rs ^ ZeroExtImme
001010	I	001010	slti	Rt = (Rs < SignExtImme) ? 1 : 0
000010	J	000010	j	PC = {(PC+4)[31:28], address, 2’b00}

2、控制信号

CPU控制单元生成的控制信号及其内容如下。

控制信号名称	0	1
RegDst	数据写回Rt	数据写回Rd
aluSrc	将Rt数据送到ALU	将扩展的立即数送到ALU
MemWrite	不写数据存储器	写数据存储器
MemRead	不读出数据存储器的数据	读出数据存储器的数据
RegWrite	不写回寄存器	写回寄存器
MemtoReg	将数据存储器的数据写回寄存器	将ALU结果写回寄存器
shift	将Rs数据送到ALU	将位移量shamt送到ALU
jmp	PC不跳转	PC跳转
ExtOp	立即数扩展0	立即数扩展最高位

续表

控制信号名称	内容
[1:0] branch	2’b01：beq；2’b10：bne，送到分支控制单元
[3:0] aluOp	对应不同的ALU操作，送到ALU控制单元生成ALU控制信号

每条指令对应的控制信号内容在构建数据通路的时候再讲述。

3、ALU功能

控制单元提供的aluOp大致地提供了ALU要进行的操作，如加法（lw, sw）、减法（beq, bne）或需要根据func字段进行进一步判断（R型指令）。因此对于I型指令，直接就可以生

成ALU控制信号，而对于R型指令，还需要通过func字段进一步确定ALU的操作。

各条指令生成的aluCtrl信号如下。

指令	指令类型	aluOp	func	ALU操作	aluCtrl
lw	I	0000	xxxxxx	add	0010
sw	I	0000	xxxxxx	add	0010
beq	I	0001	xxxxxx	sub	0001
bne	I	0001	xxxxxx	sub	0001
addi	I	0000	xxxxxx	add	0010
andi	I	1000	xxxxxx	and	1111
ori	I	1001	xxxxxx	or	1110
xori	I	1010	xxxxxx	xor	1101
slti	I	0110	xxxxxx	slt	0011
add	R	1111	100000	add	0010
sub	R	1111	100010	sub	0001
and	R	1111	100100	and	1111
or	R	1111	100101	or	1110
xor	R	1111	100110	xor	1101
nor	R	1111	100111	nor	1100
slt	R	1111	101010	slt	0011
sll	R	1111	000000	sll	0100
srl	R	1111	000010	srl	0101
sra	R	1111	000011	sra	0110

注：我们不关心I型指令的func字段的内容，故I型指令的func字段为xxxxxx。由于J型指令不会使用ALU，因此不关心J型指令的aluCtrl信号。

（2） 处理冒险

在流水线CPU中会有冒险，即流水线会因为一些因素被阻塞或错误执行。因此处理冒险是设计流水线CPU中最重要的部分。

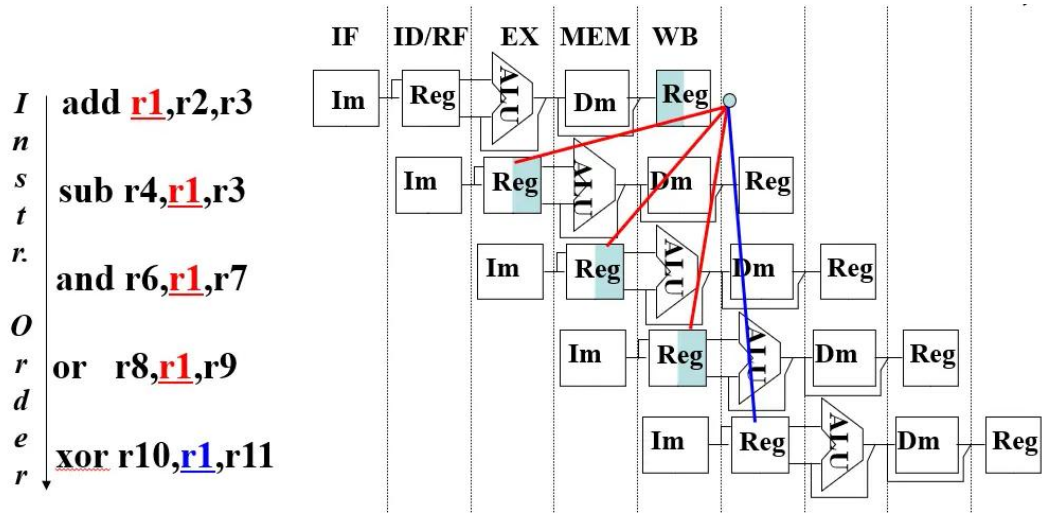
流水线CPU一共有三种冒险。

1、结构冒险

结构冒险指的是资源的冲突，即同一个部件被不同指令同时使用，比如CPU只有一个存储器部件，就会导致结构冒险。

2、数据冒险

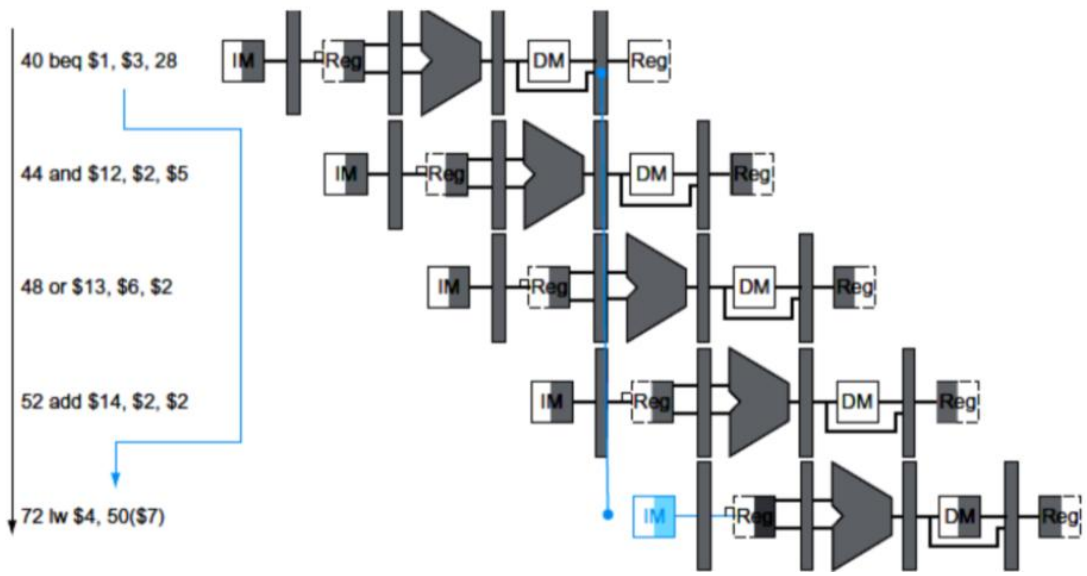
数据冒险指的是读后写冒险，就是前一条指令还没有将新数据写回寄存器堆的时候，后面的指令就读取该数据了，因此后面的指令读的是旧数据，造成错误。



第一条指令要将r2+r3的结果写回r1，在第五个时钟周期后才完成写回，但是第二条指令在第三个时钟周期就需要用r1，所以此时第二条指令读到的是旧数据。同理，第三和第四条指令也存在数据冒险。第五条指令在第六个时钟周期读r1，但是此时r1已经完成写回了，不存在数据冒险。

3、控制冒险

控制冒险也叫分支冒险，是遇到分支指令（beq, bne）时，由于分支信号的发出是在EXE阶段，PC的计算需要用到MEM/WB寄存器，而此时下面已经有三条指令进入流水线了，如果要分支，则这三条指令可能不是接下来需要执行的指令，造成错误。



对于以上三种冒险，分别给出解决措施。

1、结构冒险

结构冒险主要出现在数据和指令共用一个存储器的情况，因此我们可以将数据和指令分别存放在两个不同的存储器中。

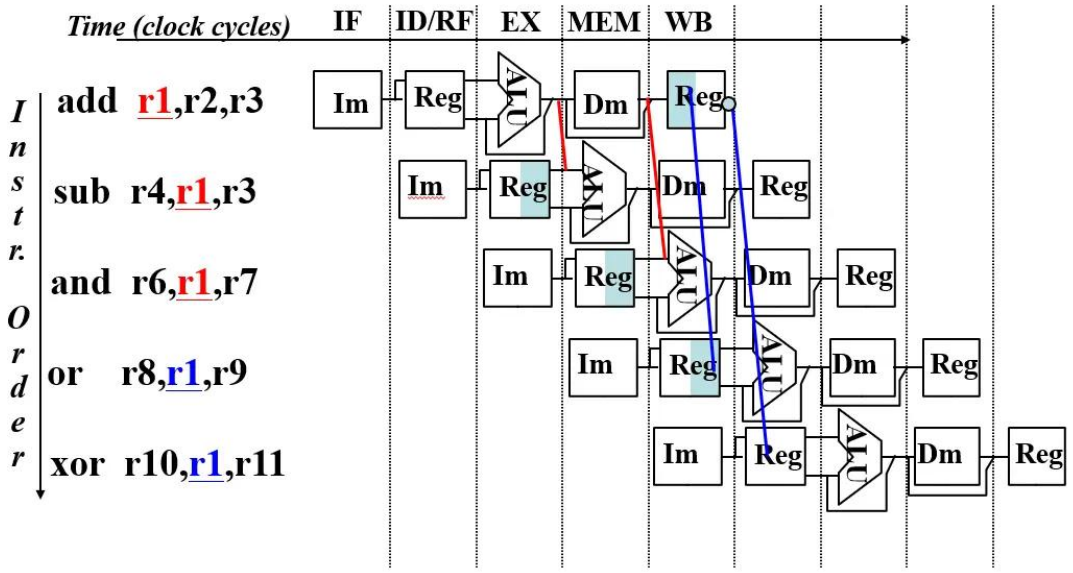
2、数据冒险

解决数据冒险的方法主要有以下两种。

① 添加阻塞。添加阻塞是解决数据冒险和后面的控制冒险的通用方法，但是如果添加过多阻塞会导致流水线效率降低。添加阻塞就是让下面一条指令等待若干个时钟周期再执行。对于数据冒险，我们可以添加阻塞，等前面的指令把数据写回后在开始执行后续指令。

指令	时钟周期								
	1	1	2	3	4	5	6	8	9
add	IF	ID	EX	MEM	WB				
sub		阻塞	阻塞	阻塞	IF	ID	EX	MEM	WB

② 旁路（Forwarding）。运用旁路，后续指令不再等前面的指令写回，也不从寄存器堆读数据，而是直接读保存在流水线寄存器中的数据。

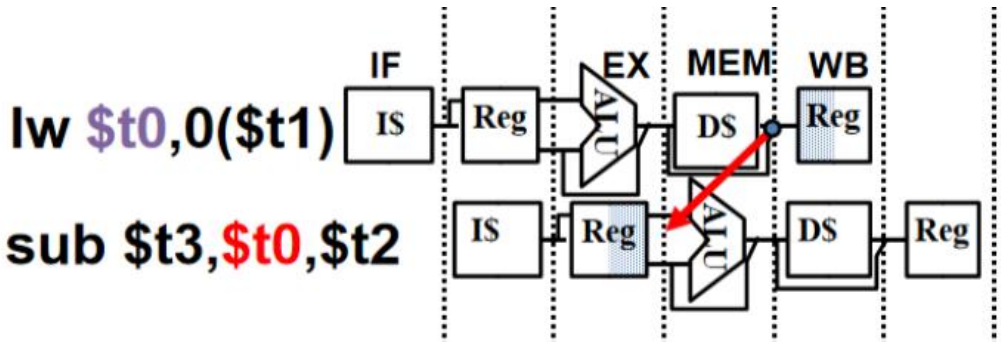


图中红色线是旁路。可以看到，第二条指令直接取EXE/MEM寄存器的值作为ALU的其中一个数，第三条指令取MEM/WB作为ALU的其中一个数。

另外，在本次实验中，图中的第四条指令也存在数据冒险。因为图中是按照MIPS CPU来描述的，MIPS CPU中读寄存器和写寄存器的时钟周期都是其他阶段时钟周期的一半，所以可以实现在一个时钟周期内先写回寄存器堆再读出新的数据的操作，因此可以避免一次数据冒险。而在本次实验设计的CPU中，所有阶段的时钟周期都是一样的，因此不能实现先写后读，所以这里也需要一个旁路。

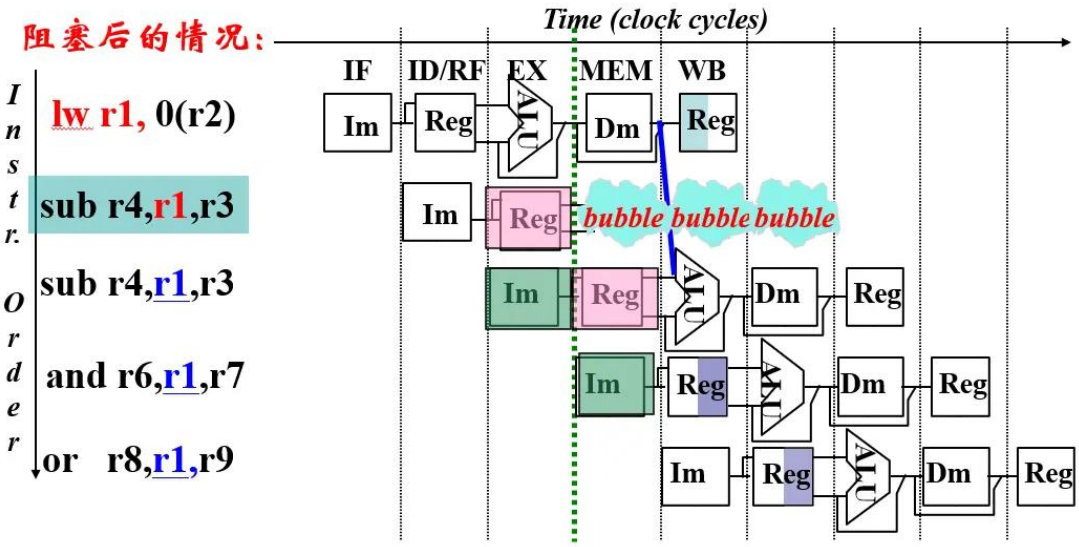
所以我们可以总结出旁路的三种情况：EXE/MEM寄存器传到EXE阶段的rs/rt；MEM/WB寄存器传到EXE阶段的rs/rt；MEM/WB寄存器传到ID阶段的rs/rt。并且RegWrite要为1。

并不是所有的数据冒险都可以使用旁路来解决，比如load-use数据冒险，即上一条指令是load类型，下一条指令就要读load的寄存器的值。



由于load发生在MEM阶段，因此下一条指令需要用到该数据的时候还没有完成

load操作，无法使用旁路。解决办法就是先阻塞一个时钟周期，再使用旁路。



阻塞了一个时钟周期之后，就可以从MEM/WB寄存器用旁路读取数据了。

3、控制冒险

解决控制冒险的方法主要有以下两种。

① 添加阻塞。可以添加两个时钟周期的阻塞，等到EXE阶段执行完毕发出branch信号后再计算PC，从而执行下一条指令。

② 分支预测。分支预测分为静态预测和动态预测。静态预测就是固定假设分支发生或不发生，如果假设错误，就要冲刷掉之前错误执行的指令；动态预测就是根据之前分支的情况预测是否分支，会比静态预测的出错率小。

分支预测的方法本次实验不涉及，因此本次实验通过添加阻塞的方式来解决控制冒险。

（3） 数据通路和各个部件模块化

1、构建数据通路

首先，我们先确定CPU工作所需要的部件及其功能。

① 程序计数器（PC）

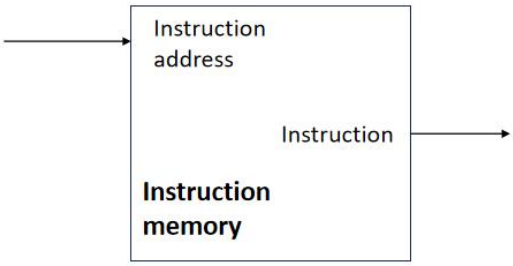
程序计数器用来标记指令的位置，从而可以取出存放在指令存储器中的指令。在取出一条指令之后，程序计数器会指向下一条指令的位置。如果有跳转、分支信号，程序计数器就修改到目标位置。

值得指出的是，在教材的MIPS CPU中，程序计数器并不是一个部件。通过在数据

通路中添加的加法器和多选器来修改程序计数器的内容，然后直接输出。考虑到这样做略为复杂，本次实验把程序计数器设计为一个部件，接收跳转和分支信号，以及跳转地址和分支地址，计算出下一条指令的位置。

② 指令存储器

指令存储器用来存储指令，根据程序计数器的值输出相应的指令。

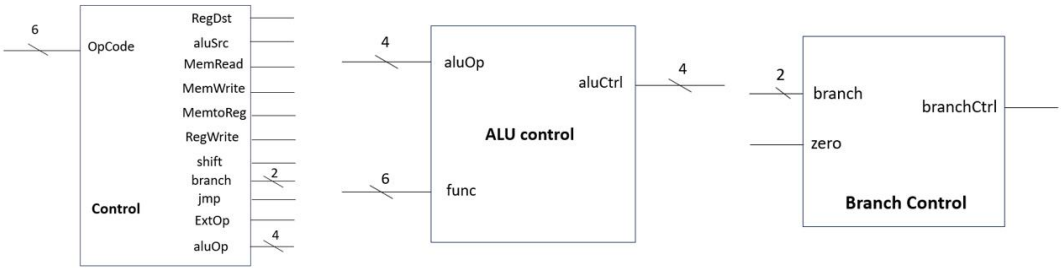


③ 寄存器堆

在设计的CPU中，寄存器堆是由32个32位的存储单元组成。从指令译码得到Rs, Rt, Rd的值，输出Rs和Rt对应的存储单元的值。如果有寄存器写RegWrite信号就修改Rt/Rd的值。

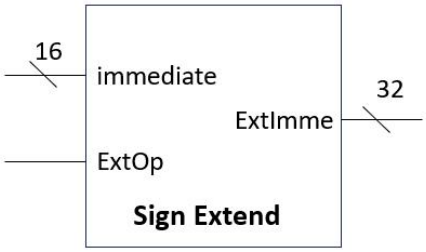
④ 控制单元

控制单元分为主控制单元、ALU控制单元和分支控制单元。主控制单元接收指令的OpCode和func字段来对指令进行译码，判断是什么指令，从而生成不同的控制信号，让其他部件正确执行该指令。ALU控制单元接收aluOp和func，输出aluCtrl给ALU。分支控制单元接收分支信号判断是beq或bne，再结合ALU传送的零标志位，判断是否执行分支。



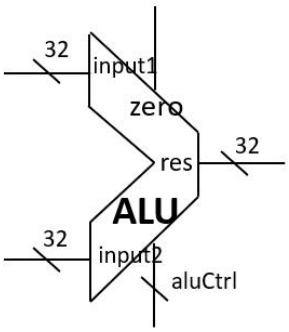
⑤ 符号扩展单元

符号扩展单元对I型指令的16位立即数字段进行符号扩展，扩展方法分为0扩展和最高位扩展，由ExtOp信号决定。



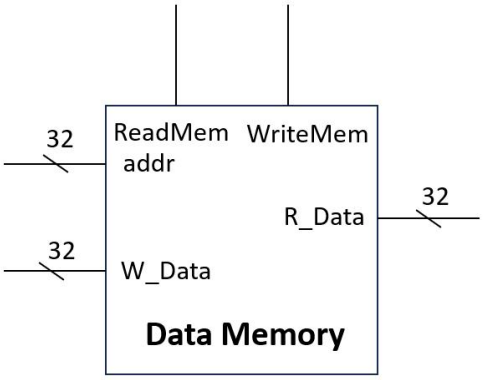
⑥ 算术逻辑单元（ALU）

算术逻辑单元对输入的两个数进行算术运算或逻辑运算，具体的操作由aluCtrl信号决定。同时要输出零标志位zero，传送到分支控制，目的是判断是否应该分支。



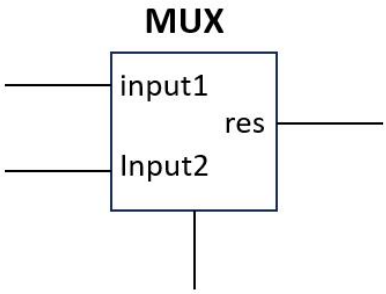
⑦ 数据存储器

数据存储器用来存储数据，通过MemRead和MemWrite信号可以写入或读出数据。



⑧ 多路选择器（MUX）

多路选择器接收一个选择信号（0或1）和两个数据，输出选择信号对应位置的数据。

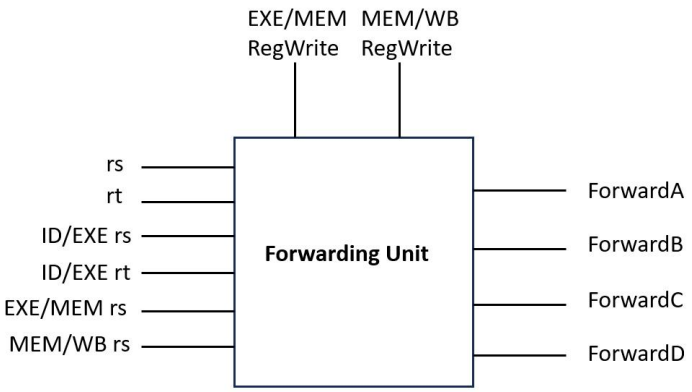


⑨ 流水线寄存器

流水线CPU中一共有4个流水线寄存器，分别为IF/ID，ID/EXE，EXE/MEM，MEM/WB，用来存储该指令后面的阶段需要用到的数据和控制信号。每个阶段执行时使用上一个流水线寄存器的数据和控制信号，并将新数据写入下一个流水线寄存器。控制信号继承自上一个流水线寄存器。

⑩ 旁路单元

旁路单元接收译码出来的rs，rt，和ID/EXE寄存器的rs，rt，以及EXE/MEM和MEM/WB寄存器的rd，RegWrite信号，输出四个Forward信号，用来表示是否需要旁路。



确定了设计的CPU所需要的部件及其功能后，我们便可以分析各个代码的运行机制，从而构建数据通路。下面对各类指令进行分析。

首先是R型指令。指令存储器将程序计数器对应的指令取出，指令译码后从寄存器堆读出Rs和Rt的数据，同时控制单元发送控制信号。然后Rs/shamt和Rt被送入ALU进行运算，运算结果写回Rd寄存器。

其次是I型指令。I型指令分为非分支指令和分支指令。第一步也是取出指令。对于非分支指令，控制单元发送的控制信号将Rs和扩展后的立即数送到ALU进行运算。对于addi/andi此类指令，运算结果写回Rt；对于lw指令，读取数据存储器地址为ALU运算结果的数据并写回Rt；对于sw指令，Rt的数据将被写入数据存储器地址为ALU运算结果的位置。对于分支指令，Rs和Rt送入ALU进行减法运算，得到零标志位，然后分支控制单元发出分支信号，如果要分支，则程序计数器修改为分支地址；如果不分支，程序计数器为下一条指令的地址。

最后是J型指令，设计的CPU只支持一条J型指令：j。第一步也是取指令。控制单元发出跳转信号，如果要跳转，程序计数器修改为目标地址；如果不跳转，程序计数器为下一条指令的地址。

分析完每条指令的工作机制和数据通路，我们就可以综合所有指令，从而构建CPU的数据通路。

最后，综合分析每条指令的功能和上方构建的数据通路，给出每条指令对应的控制信号表。

指令名	RegDst	aluSrc	MemWrite	MemRead	RegWrite	MemtoReg
R型	1	0	0	0	1	1
lw	0	1	0	1	1	0
sw	x	1	1	0	0	x

续表

指令名	RegDst	aluSrc	MemWrite	MemRead	RegWrite	MemtoReg
beq/bne	x	0	0	0	0	x
addi/andi/ ori/xori/slti	0	1	0	0	1	1
j	x	x	0	0	0	x

续表

指令名称	shift	jmp	ExtOp	[1:0]branch
add/sub/and/ or/xor/nor/slt	0	0	x	2’b00
sll/srl/sra	1	0	x	2’b00
lw/sw/addi/slti	0	0	1	2’b00
andi/ori/xori	0	0	0	2’b00
beq	0	0	x	2’b01
bne	0	0	x	2’b10
j	x	1	x	2’b00

aluCtrl信号表在“ALU功能”板块已给出，下面给出branchCtrl信号表。

[1:0]branch	zero	branchCtrl
2’b00	x	0
2’b01(beq)	1	1
2’b01(beq)	0	0
2’b10(bne)	1	0
2’b10(bne)	0	1

下面是每个Forward信号的含义。

ForwardA

数值	含义
2’b00/2’b11	无旁路
2’b10	将EXE/MEM的rd旁路到EXE的rs
2’b01	将MEM/WB的rd旁路到EXE的rs

ForwardB

数值	含义
2’b00/2’b11	无旁路
2’b10	将EXE/MEM的rd旁路到EXE的rt
2’b01	将MEM/WB的rd旁路到EXE的rt

ForwardC

数值	含义
0	无旁路
1	将MEM/WB的rd旁路到ID的rs

ForwardD

数值	含义
0	无旁路
1	将MEM/WB的rd旁路到ID的rt

可以看到，上面两个表格中有一些控制信号的值为x，这代表我们不关心它们的值，这些信号不管值是什么，都不会对指令的运行产生影响。下面对于控制信号为x的指令进行解释。

第一，sw的RegDst和MemtoReg。由于sw指令的RegWrite信号为0，即不写寄存器，所以写回哪个寄存器和写回的内容对于这条指令来说没有意义，因为不管这些数据是什么都不会对寄存器堆产生影响。beq/bne同理。第二，J型指令只有MemRead, MemWrite, RegWrite, jmp信号有效，因为J型指令不会修改寄存器堆和数据存储器的值，不会利用到ALU和符号扩展。第三，add/sub/and/or/xor/nor/slt的ExtOp。这些指令把Rt送进ALU，而且R型指令没有立即数字段，所以对立即数进行怎样的扩展没有意义。beq/bne同理。第四，branch为2’b00时的zero。当branch为2’b00时，代表不会进行分支，所以不管零标志位是多少，branchCtrl都为0。

2、模块化部件

下面对每个部件进行模块化，在Vivado中编写代码。下图展示了所编写的模块。

其中，PC模块直接写在了顶层文件中，为了防止模块的输出又被传进PC模块造成混乱的情况。下面依次讲述每个模块的具体实现。

PC模块：考虑三种情况——分支、跳转和一般情况。最后对边界情况进行判断，并用时钟信号进行同步。核心代码如下：

```
always @(posedge clk or posedge clr) begin
if(clr)
    pc <= 32'b0;
else if(jmp == 1 && branchCtrl == 0)
    pc <= {pc_plus[31:28], jmpAddr, 2'b00};
else if(jmp == 0 && branchCtrl == 1)
    pc <= pc_plus + (ExtImme << 2);
else if(pc >= 32'd170)
    pc <= pc;
else
    pc <= pc_plus;
end
```

U0主控制单元模块：对OpCode和func进行译码，输出各个控制信号。根据控制信号表进行编写。控制信号表中为x的信号默认为0。

U1数据存储器：我创建了256个32位的存储单元，接收地址和数据，还有读信号和写信号。要注意的是，这里传进来的地址应该除以4（右移2位），因为MIPS CPU中，32位数据是分为4个字节来存储的，而这里直接把32位存到一个存储单元里。在此基础上，又增加了10个输出，依次为存储器第0到第9个数据，便于查看排好序的数据。核心代码如下：

```
reg [31:0] memory [0:255];
    always @(posedge clk) begin
        if (MemWrite) begin
            memory[addr >> 2] <= write_data;
        end
    end
assign read_data = (MemRead == 1) ? memory[addr >> 2] : 32'b0;
assign D1 = memory[0][14:0];
assign D2 = memory[1][14:0];
```

U2EXE/MEM寄存器：存储MEM和WB阶段需要的数据和控制信号。为避免覆盖，在时钟上升沿读出数据，在时钟下降沿写入数据。存储内容为：

```
reg MemWrite;
reg MemRead;
reg RegWrite;
reg MemtoReg;
reg [4:0] rd;
reg [31:0] write_data;
reg [31:0] alu_res;
```

U3、U4EXE阶段的旁路多选器：根据旁路信号ForwardA和ForwardB选择rs/rt，EXE/MEM寄存器的写回内容或MEM/WB寄存器的写回内容。

U5ID/EXE寄存器：存储EXE、MEM和WB阶段需要的数据和控制信号。为避免覆盖，在时钟上升沿读出数据，在时钟下降沿写入数据。存储内容为：

```
reg RegDst;
reg aluSrc;
reg MemWrite;
reg MemRead;
reg RegWrite;
reg MemtoReg;
reg shift;
reg ExtOp;
reg [1:0] branch;
reg [3:0] aluOp;
```

```
reg [4:0] rs;
reg [4:0] rt;
reg [4:0] rd;
reg [4:0] shamt;
reg [5:0] func;
reg [31:0] R_Data_A;
reg [31:0] R_Data_B;
reg [15:0] imme;
```

U6IF/ID寄存器：存储ID、EXE、MEM和WB阶段需要的数据和控制信号（也就是存储当前指令）。为避免覆盖，在时钟上升沿读出数据，在时钟下降沿写入数据。存储内容为：

```
reg [31:0] inst;
```

U7MEM/WB寄存器：存储WB阶段需要的数据和控制信号。为避免覆盖，在时钟上升沿读出数据，在时钟下降沿写入数据。存储内容为：

```
reg RegWrite;
reg MemtoReg;
reg [4:0] rd;
reg [31:0] mem_data;
reg [31:0] alu_res;
```

U8写回寄存器多选器：用来确定写回Rt还是Rd，用RegDst作为选择信号，Rt和Rd作为输入。

U9寄存器堆：构造32个32位的存储单元，如果要写寄存器就修改对应存储单元的值，读寄存器就用赋值语句输出。核心代码如下：

```
reg [31:0]REG_Files[0:31];
always@(posedge Clk) begin
    if(Write_Reg)
        REG_Files[W_Addr] <= W_Data;
end
assign R_Data_A = REG_Files[R_Addr_A];
assign R_Data_B = REG_Files[R_Addr_B];
```

U10写回数据多选器：用来确定写回寄存器的数据是ALU结果还是数据存储器数据。用MemtoReg作为选择信号，ALU结果和读存储器结果作为输入。

U11ALU：通过aluCtrl信号对两个输入的数进行运算，并输出零标志位。

U12旁路单元：根据不同的旁路情况发出不同的旁路信号。EXE/MEM到EXE的优先度比MEM/WB到EXE的优先度更高。核心代码如下：

```
    if(MEM_WB_RegWrite && MEM_WB_rd != 0 && (MEM_WB_rd == ID_EX_rs || MEM_WB_rd == ID_EX_rt))begin
        if(MEM_WB_rd == ID_EX_rs) ForwardA <= 2'b01;
        if(MEM_WB_rd == ID_EX_rt) ForwardB <= 2'b01;
    end
    if(EX_MEM_RegWrite && EX_MEM_rd != 0 && (EX_MEM_rd == ID_EX_rs || EX_MEM_rd == ID_EX_rt))begin
        if(EX_MEM_rd == ID_EX_rs) ForwardA <= 2'b10;
        if(EX_MEM_rd == ID_EX_rt) ForwardB <= 2'b10;
    end
    if(MEM_WB_RegWrite && MEM_WB_rd != 0 && (MEM_WB_rd == rs || MEM_WB_rd == rt))begin
        if(MEM_WB_rd == rs) ForwardC <= 1;
        if(MEM_WB_rd == rt) ForwardD <= 1;
    end
```

U13ALU多选器其一：用来确定将Rs还是shamt送进ALU，用shift作为选择信号，Rs和shamt作为输入。

U14ALU多选器其二：用来确定将Rt还是扩展后的立即数送进ALU，用aluSrc作为选择信号，Rt和ExtImme作为输入。

U15ALU控制单元模块：综合aluOp和func，输出aluCtrl。根据ALU功能表进行编写。

U16分支控制单元：接收branch和zero，输出branchCtrl。

U17符号扩展模块：根据ExtOp的内容进行零扩展或最高位扩展。

U18指令存储器：我直接使用了Vivado中的名为Distributed Memory Generator的IP核作为指令存储器，将汇编语言转换成机器代码写入.coe文件中。下面是该IP核的实例化模版：

```
    Inst your_instance_name (
    .a(a),      // input wire [7 : 0] a
    .spo(spo)  // output wire [31 : 0] spo
);
```

U19、U20ID阶段的旁路多选器：根据旁路信号ForwardC和ForwardD选择rs/rt或MEM/WB的rd。

顶层文件：根据数据通路，在各个模块之间添加导线，并实例化各个模块，同时输出一些内容如流水线寄存器的内容、ALU相关数据、排好序的10个数据等，方便仿真时查看。

（4） 编写指令，运行仿真并烧录到实验板上

编写冒泡排序的汇编代码，如下：

```
addi $a1, $zero, 10
addi $s0, $zero, 0
out_loop:
addi $t4, $a1, -1
slt $t1, $s0, $t4
bne $t1, 1, Exit_out
sll $t0, $s0, 2
add $t0, $t0, $a2
lw $s1, 0($t0)
addi $s2, $s0, 1
in_loop:
sll $t2, $s2, 2
add $t2, $t2, $a2
lw $s3, 0($t2)
slt $t1, $s3, $s1
beq $t1, 1, swap
j after_swap
after_swap:
addi $s2, $s2, 1
slt $t1, $s2, $a1
bne $t1, 1, Exit_in
j in_loop
swap:
sw $s1, 0($t2)
sw $s3, 0($t0)
lw $s1, 0($t0)
j after_swap
Exit_in:
addi $s0, $s0, 1
j out_loop
Exit_out: addi $s0, $zero, 0
```

在MARS 4.5软件里将上面的代码转成机器码写入指令存储器中。需要注意的是，MIPS

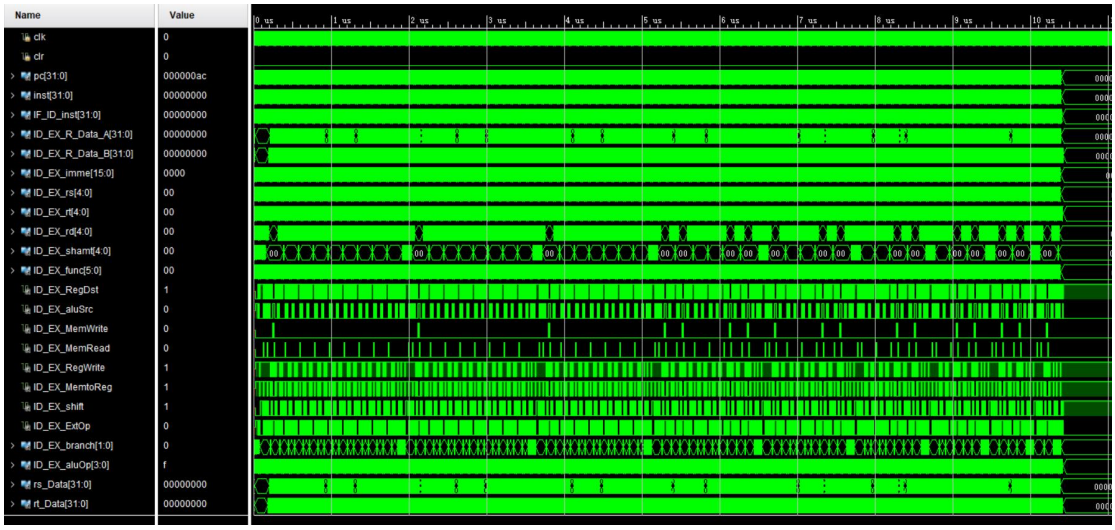
CPU的指令是分为4个字节进行存储的，所以下一条指令的地址为PC+4，而我们设置的指令存储器是以32位为单位进行存储的（也可以修改IP核设置，让数据宽度为8位，再把一条指令分为4个8位存储，这样就和MIPS CPU中一样了），下一条指令的地址应为PC+1。为了保持一致性，我们需要在每条指令之间添加3条无关指令（随便添加内容即可，因为PC加4之后会直接跳过这些无关指令），这里默认添加32位0。

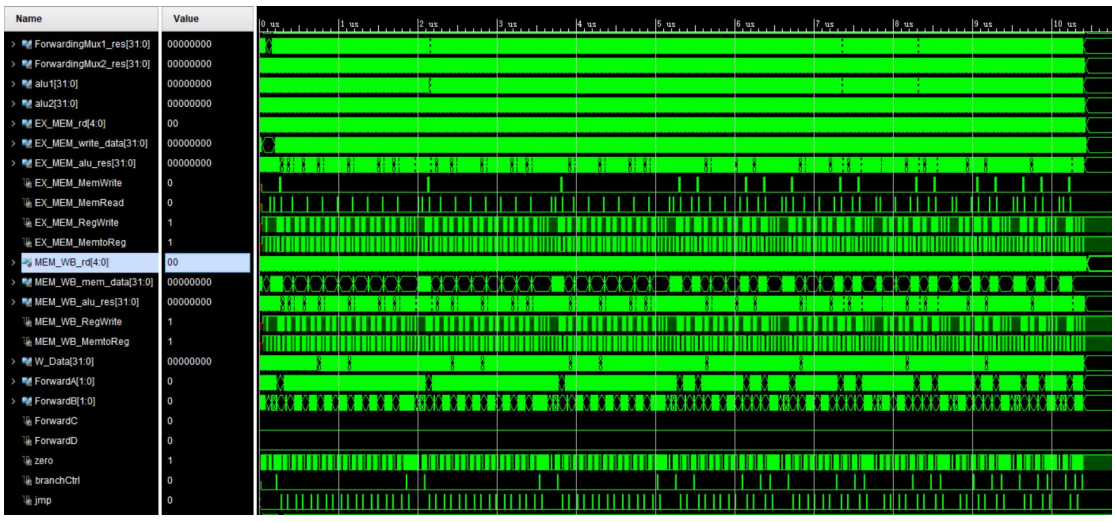
另外，我们添加阻塞的方法是添加一条全0的指令`sll $zero, $zero, 0`，这条指令不会做任何影响CPU的操作，而且可以对流水线寄存器进行清零的操作。所以，我们在beq/bne后面添加两个阻塞，在j和lw后面添加一个阻塞。

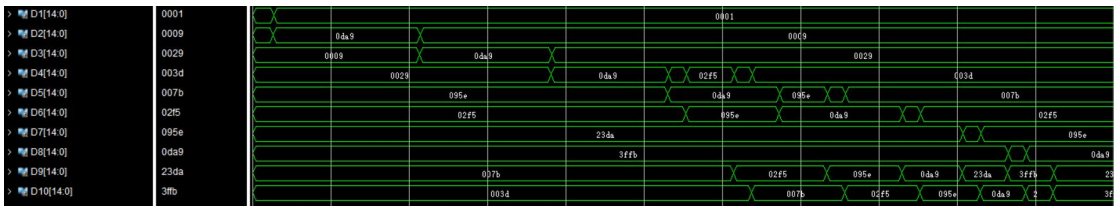
然后我们在数据存储器模块中对第0到第9共10个存储单元进行初始化，随意地给这十个数赋值。之后编写仿真文件（仿真文件里对顶层文件进行实例化）运行仿真，查看结果。

（三） 实验结果

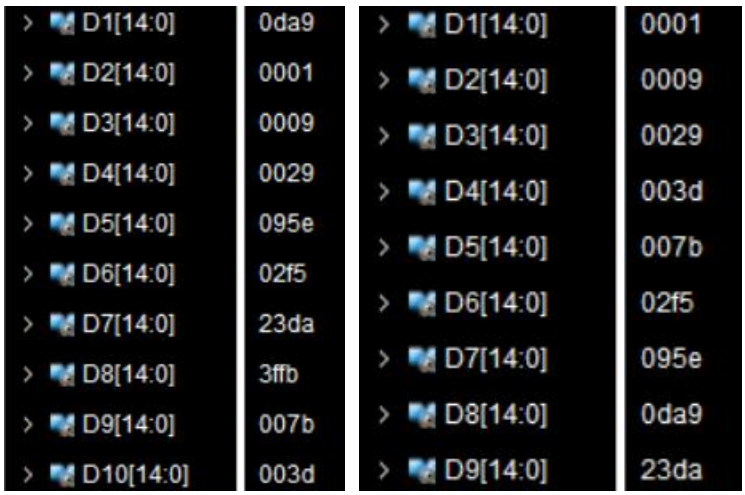
运行仿真，整体波形图如下：







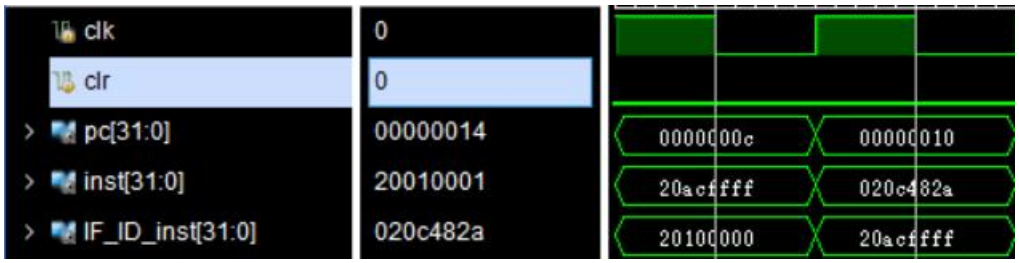
可以看到，一开始的乱序的10个数在程序运行完后按升序排序。



下面来分析一些指令的运行情况。

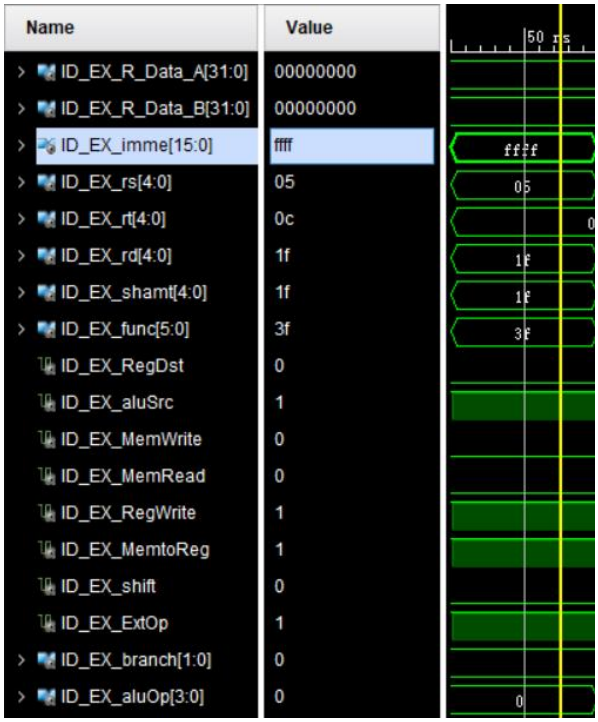
1、addi $t4, $a1, -1

IF：获取指令，并将指令存入IF/ID存储器。



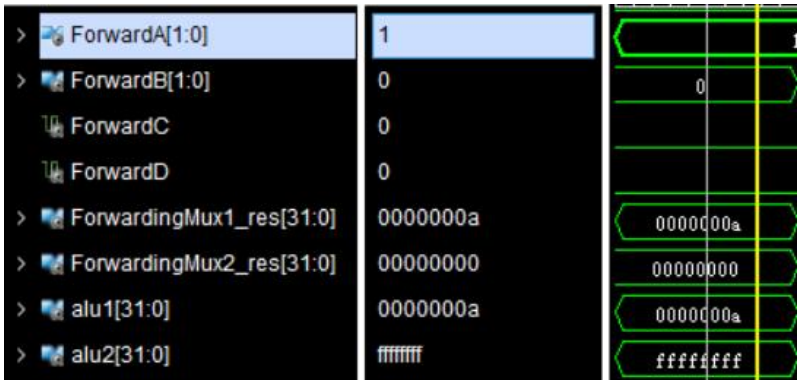
inst是当前获得的指令，从第二个周期的IF_ID_inst可以看到，该指令已经被存入IF/ID中了。

ID：指令译码，从寄存器取出对应的值，同时控制单元输出控制信号，并存入ID/EXE寄存器。

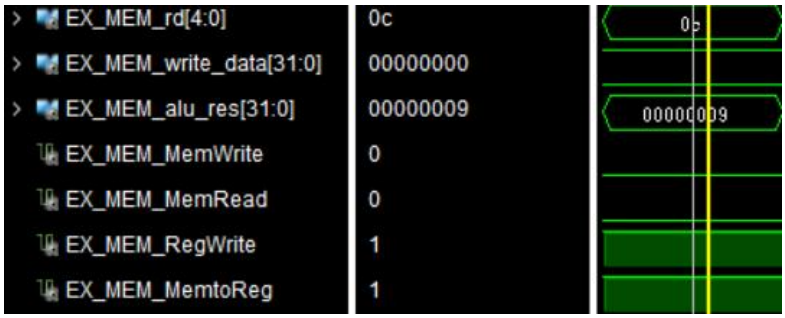


从第三个时钟周期ID/EXE寄存器的值可以看到ID阶段的译码结果。rs和rt分别为5和12，读两个寄存器的值都为0（$a1的值还没有写回，因此为旧值0），立即数为-1，还有其余的控制信号都正确被存入。

EXE：执行指令，将两个操作数送入ALU进行加法计算，输出结果，将控制信号和结果存入EXE/MEM寄存器中。其中一个操作数要通过旁路从MEM/WB寄存器传送过来（$a1），旁路单元应输出旁路信号。

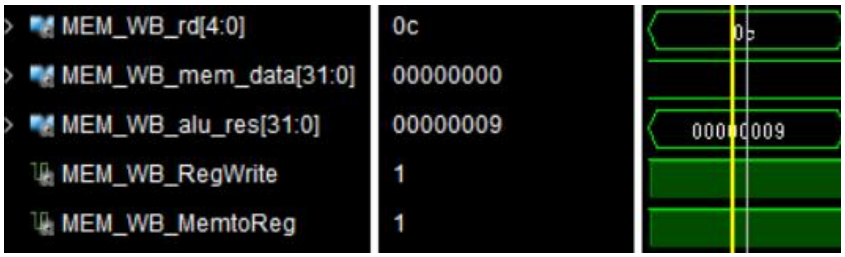


从第三个时钟周期可以看到ALU的运算内容，ForwardA信号为01，表示将MEM/WB寄存器的写回数据送到EXE阶段的rs。所以ALU的两个操作数分别为10和-1。



从第四个时钟周期可以看到EXE/MEM寄存器的值，rd是12，alu_res是9，还有四个MEM和WB需要用到的控制信号。

MEM：访问存储器，该指令是R型指令，不会访存，所以会把数据和控制信号存入MEM/WB寄存器。



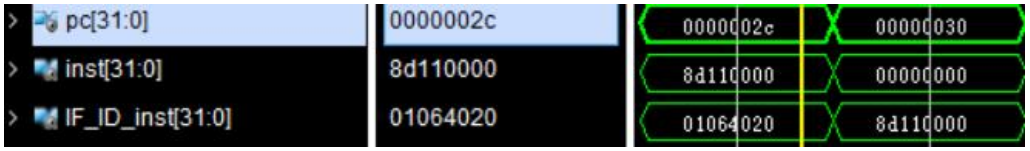
从第五个时钟周期可以看到MEM/WB寄存器的值，rd为12，alu_res为9。

WB：写回数据，将ALU结果写回寄存器。

在第五个时钟周期的W_Data就是要写回的数据，rd就是MEM/WB的rd即12。

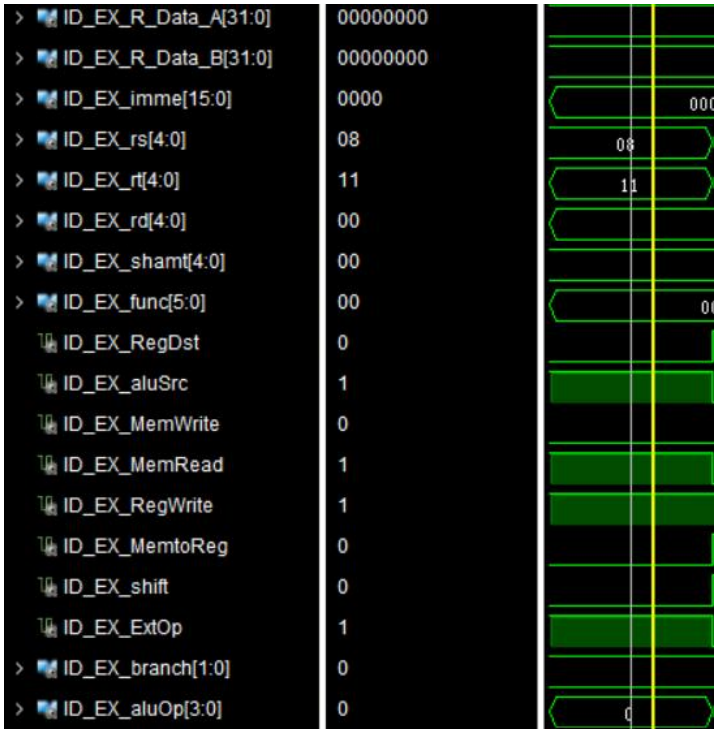
2、lw $s1, 0($t0)

IF：获取指令，并将指令存入IF/ID存储器。



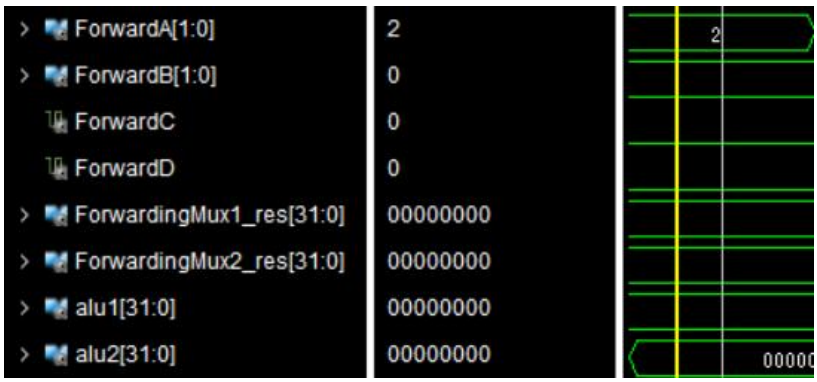
从第一个时钟周期可以看到当前读取的指令，从第二个时钟周期可以看到指令被存入IF/ID存储器。同时，由于这是一条load指令，后面应该有一个阻塞，所以第二个时钟周期读取的指令是0指令。

ID：指令译码，从寄存器取出对应的值，同时控制单元输出控制信号，并存入ID/EXE寄存器。

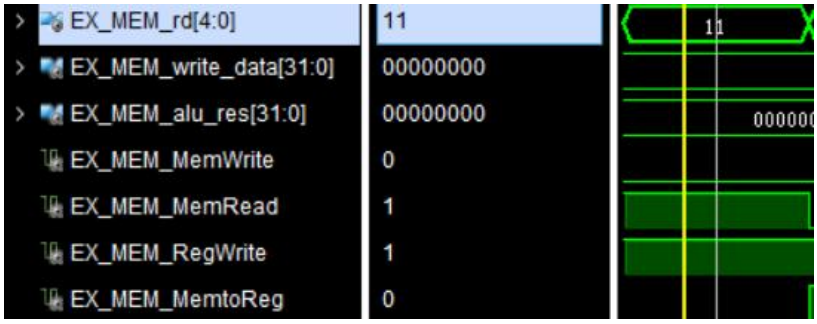


从第三个时钟周期可以看到ID/EXE寄存器的内容，其中rs和rt分别为8和17，读两个寄存器的值都是0（$t0的值还没有写回，因此为旧值0），立即数为0，还有其他的控制信号。

EXE：执行指令，将两个操作数送入ALU进行加法运算，输出结果，将结果和控制信号存入EXE/MEM寄存器。其中一个操作数要通过旁路从EXE/MEM寄存器传送过来（$t0），旁路单元生成旁路信号。

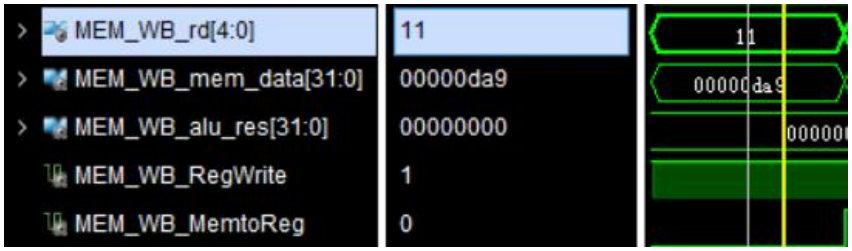


从第三个时钟周期可以看到ALU运算的内容，其中ForwardA为10，表示把EXE/MEM寄存器的写回值（alu_res）送到EXE阶段的rs。ALU的两个操作数均为0，结果为0。



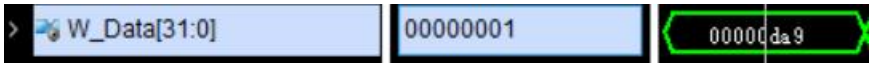
从第四个时钟周期可以看到EXE/MEM寄存器的内容，rd为17，alu_res为0，同时要读存储器。

MEM：访问存储器，读出存储器中地址为alu_res的数据，并存入MEM/WB寄存器。



从第五个时钟周期可以看到MEM/WB寄存器的内容，rd为17，mem_data即从存储器读取的数据为da9，还有两个WB阶段需要的控制信号。

WB：写回数据，将mem_data写回寄存器。



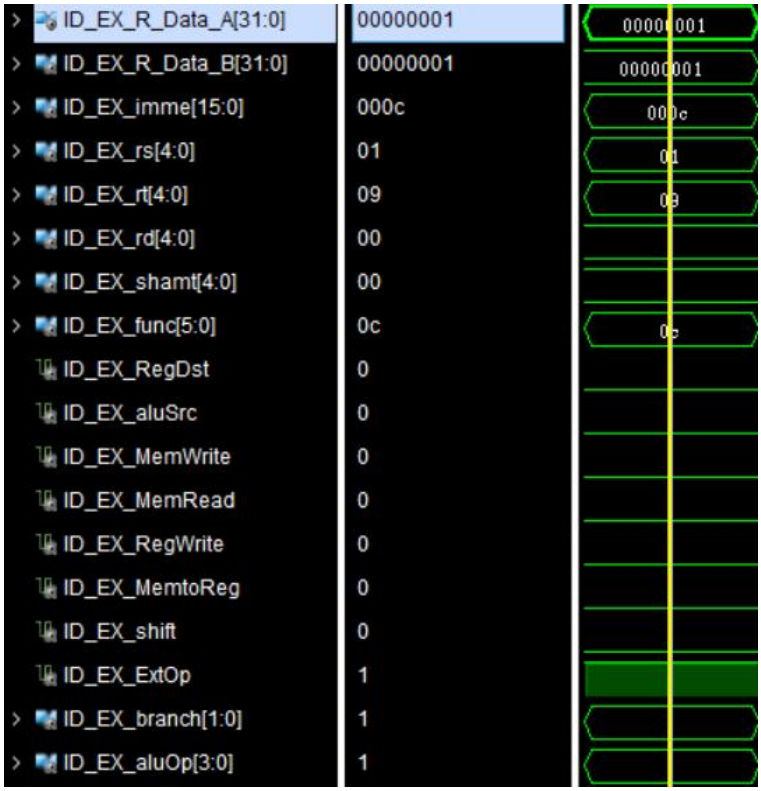
从第五个时钟周期可以看到写回的数据为da9，写回的寄存器为MEM/WB寄存器的rd即17。

3、beq $t1, 1, swap

IF：获取指令，并将指令存入IF/ID存储器。

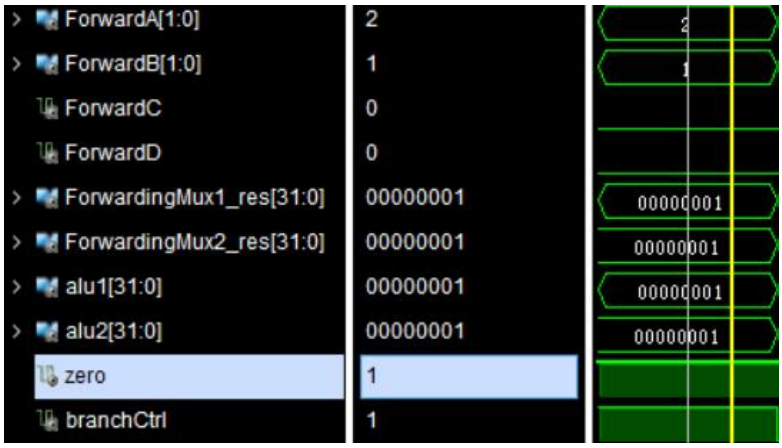
从第一个时钟周期可以看到当前指令，从第二个时钟周期可以看到IF/ID的内容，发现该指令已经被存进去了。由于这是一条beq指令，所以后面有两个时钟周期的阻塞。

ID：指令译码，从寄存器取出对应的值，同时控制单元输出控制信号，并存入ID/EXE寄存器。



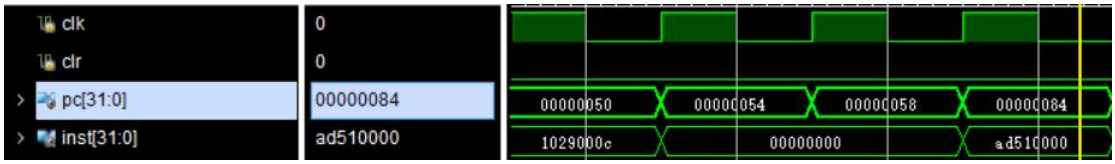
从第三个时钟周期可以看到ID/EXE寄存器的内容，rs为1，rt为9。因为该beq指令是分两条指令执行的：`addi $at, $zero, 1`和`beq $at, $t1, swap`，所以没有立即数。读两个寄存器的值都为1（实际上这两个寄存器都要通过旁路得到，这里只是巧合）。控制信号也被正确存入寄存器中。

EXE：执行指令，将两个操作数送到ALU进行减法运算，生成零标志位并将其送入分支单元生成分支信号。其中两个操作数都要使用到旁路，一个是从EXE/MEM寄存器传送过来（$at），另一个是从MEM/WB寄存器传送过来（$t1）。



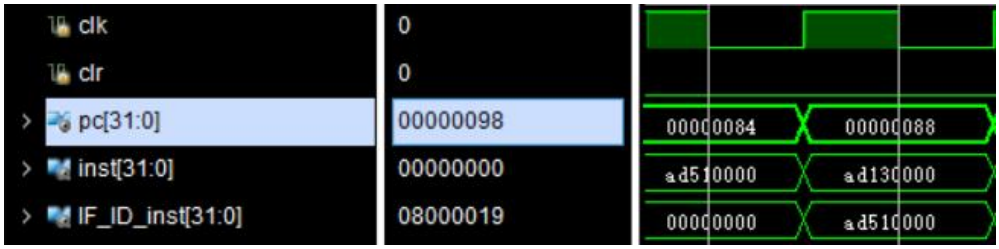
从第三个时钟周期可以看到ALU的运算情况。其中ForwardA为10，表示将EXE/MEM寄存器的写回数据（alu_res）传送到EXE阶段的rs；ForwardB为01，表示将MEM/WB寄存器的写回数据传送到EXE阶段的rt。ALU的结果为0，零标志位为1，所以分支信号为1，表示需要分支。

beq指令不会经过MEM和WB阶段，因此这里就不再展示这两个阶段的流水线寄存器的情况了。可以看到，在第四个时钟周期，PC分支到了正确的地址。



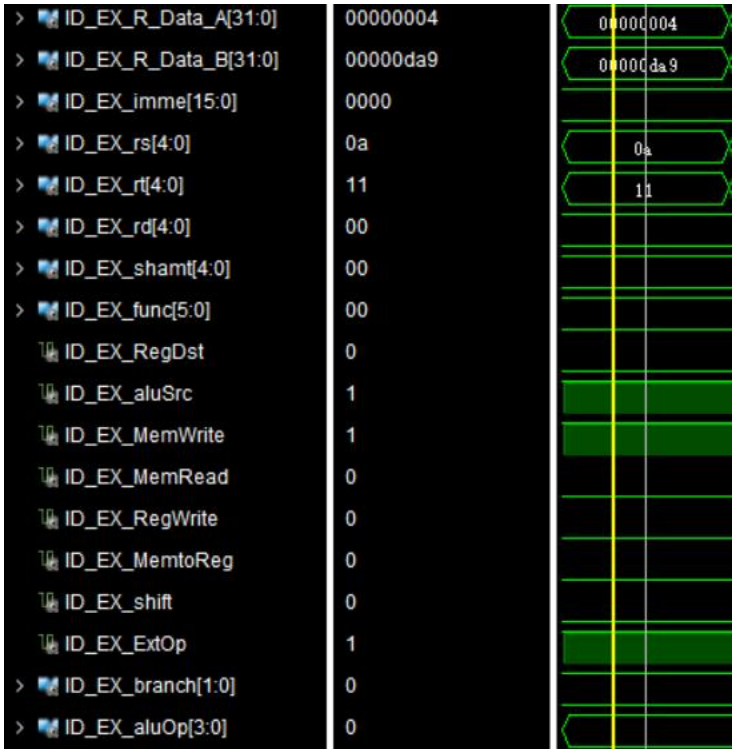
4、sw $s1, 0($t2)

IF：获取指令，并将指令存入IF/ID存储器。



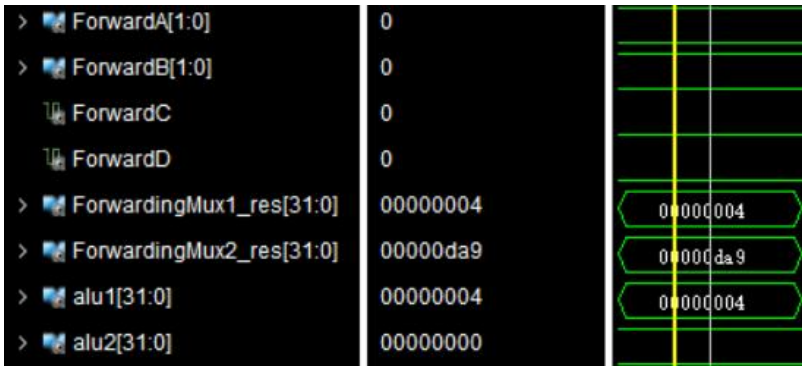
从第一个时钟周期可以看到当前指令，从第二个时钟周期可以看到IF/ID中存放了该指令。

ID：指令译码，从寄存器取出对应的值，同时控制单元输出控制信号，并存入ID/EXE寄存器。

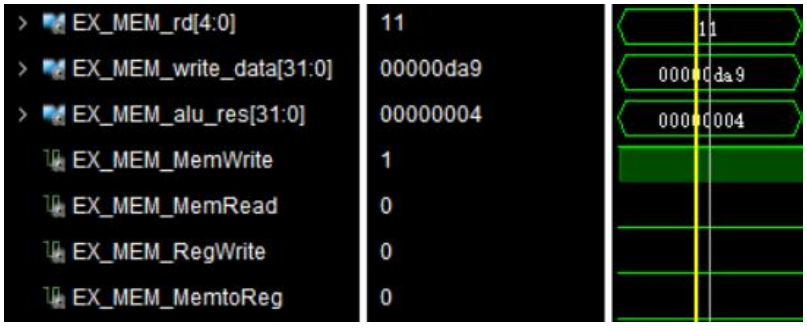


从第三个时钟周期可以看到ID/EXE寄存器的内容，rs和rt分别为10和17，读出的寄存器的数据为4和da9，立即数为0，控制信号也被正确存入。

EXE：将两个操作数送到ALU进行加法运算，将结果和控制信号存入EXE/MEM寄存器。



在第三个时钟周期可以看到ALU运算的内容，不需要旁路，ALU的两个操作数为4和0。

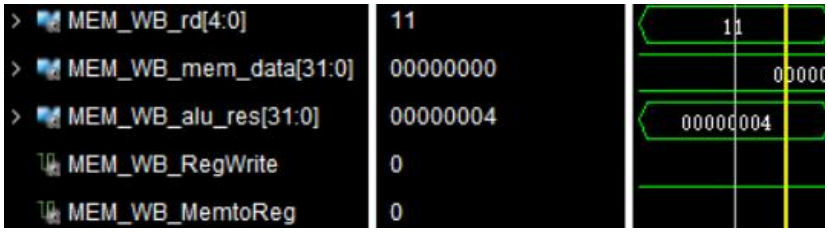


在第四个时钟周期可以看到EXE/MEM寄存器的内容，write_data即写入存储器的数据为da9，alu_res即写入存储器的地址为4，MemWrite为1，表示要写存储器。

MEM：访问存储器，将数据写入存储器对应的地址。

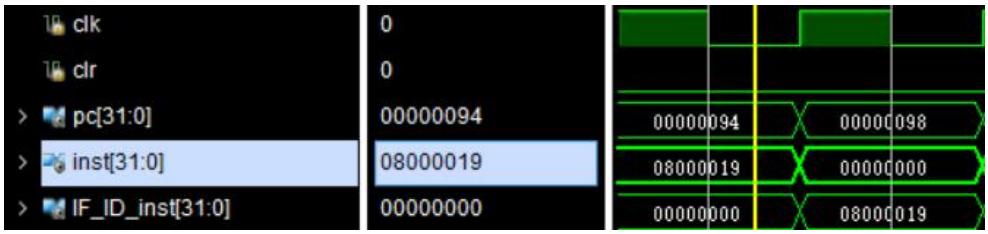
在第四个时钟周期结束后，0da9被存到D2的位置。

WB：sw指令不需要写回寄存器，所以RegWrite应为0。



5、j after_swap

IF：获取指令，并将指令存入IF/ID存储器。



第一个时钟周期读取了当前指令，从第二个时钟周期的IF/ID寄存器的内容可以看到该指令被正确存入。同时，由于这是一条j指令，所以后面有一个时钟周期的阻塞。

ID：指令译码，输出jmp信号，PC跳转到目标地址。

从第二个时钟周期可以看到jmp信号为1，表示需要跳转。

可以看到，在第三个时钟周期，PC跳转到了正确的地址。

j指令不会经过EXE、MEM和WB阶段，因此这里就不再展示后面的流水线寄存器的内容了。

至此，MIPS简单流水线CPU设计完毕，实验完成。

六. 实验心得

下面先讲述一下设计流水线CPU时遇到的问题、困难以及解决措施。

（1） 流水线寄存器的实现

首先需要确定每个流水线寄存器需要存储什么内容，就是根据该流水线后面的阶段所需要的数据和控制信号，比如WB阶段需要RegWrite和MemtoReg控制信号和rd，alu_res，mem_data等，所以MEM/WB就需要存储这些数据。然后要实现流水线寄存器的读出和写入的逻辑，可以利用时钟信号，把时钟信号分为两部分，在时钟上升沿来临时把数据读出，时钟下降沿来临时写入数据，这样在一个时钟周期内就可以完成读写的操作，而且数据不会被覆盖。在编写流水线寄存器的时候要仔细，不要遗漏了内容。

（2） 旁路的实现

我先是通过书上对于旁路的描述设置了ForwardA和ForwardB两个旁路信号，但是发现不能解决读后写的问题。如果继续用ForwardA和ForwardB来解决MEM/WB寄存器到ID阶段的旁路，会出现旁路信号被覆盖的问题。所以我新增了ForwardC和ForwardD信号用来实现MEM/WB寄存器到ID阶段的旁路。

在本次实验的过程中，我认识到合理命名变量的重要性。由于流水线寄存器的数据和控制信号繁杂，每个流水线寄存器都有自己的数据和控制信号，每个时钟周期又要读和写，正确合理地命名可以使整个CPU的数据通路变得非常清晰。同时，也要对每个时钟周期做什么进行剖析，比如EXE阶段，先读出ID/EXE的数据和控制信号，然后通过两个多选器进入ALU，输出ALU结果和零标志位，最后把相关MEM和WB需要用到的数据和控制信号写入EXE/MEM寄存器，这才是一个完整的时钟周期的操作。

这次实验的不足之处在于没有阻塞检测模块，无法自动插入阻塞，而是需要手动加入阻塞。同时，对于分支指令，没有实现分支预测的模块。因为时间和精力有限，本次实验只是实现了一个基本的流水线CPU，具有旁路功能，可以通过人为加入阻塞来运行指令。