



《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 11 月 21 日

成绩：

单周期CPU设计与实现

一. 实验目的

- 1、 掌握单周期CPU的设计思路和方法；
- 2、 通过设计单周期CPU熟悉CPU各个部件工作的原理以及数据通路传输数据的原理。

二. 实验内容

利用Vivado软件设计一个MIPS单周期CPU，能够支持精简的MIPS指令集，并在CPU上运行冒泡排序程序（升序）。完成仿真，并在实验板上利用开关和按钮依次输入十个数，拨动转换开关后在数码管上依次显示排好序的十个数。

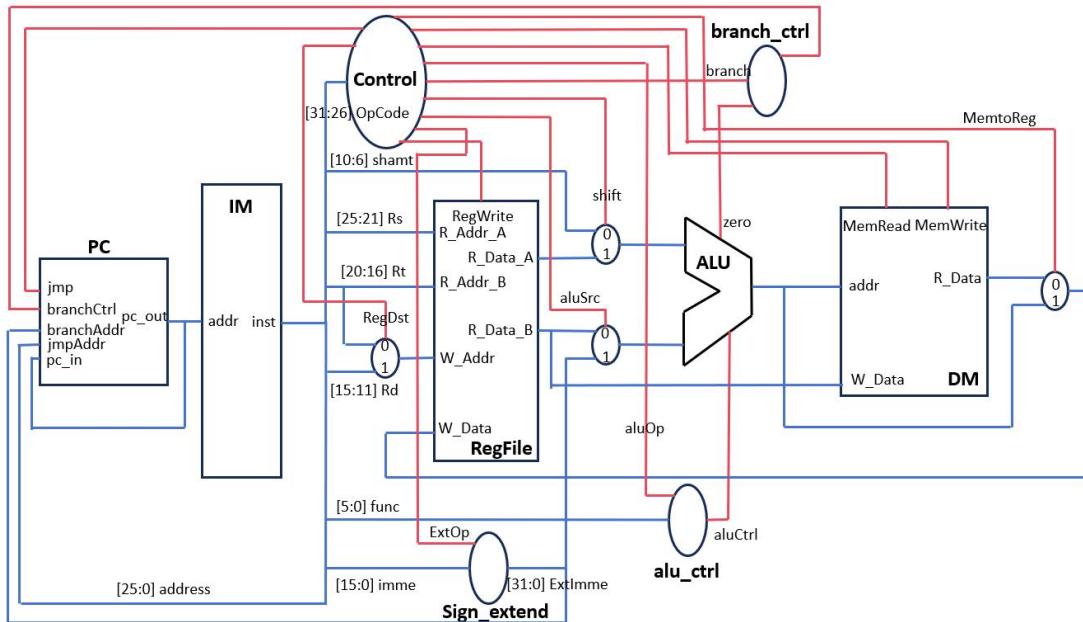
三. 实验原理

在单周期CPU中，每条指令的执行时间都是一个时钟周期，并且一个时钟周期只会执行一条指令。一条指令在单周期CPU中执行的基本流程为：程序计数器（PC）取指令——指令译码确定执行操作——进行逻辑或算术运算——影响存储单元和程序计数器。

单周期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、控制单元和多路选择器。这些部件的工作原理在“实验流程”板块再讲述。

设计单周期CPU的流程，大致分为三步：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，并烧录到实验板中。

数据通路图如下。（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）



四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

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” 板块提供了设计单周期CPU的流程，以下分为这三步来讲述。

(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

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

J型

op	address
6 bits	26 bits

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	$Rd = \sim(Rs Rt)$
000000	R	101010	slt	$Rd = (Rs < Rt) ? 1 : 0$
000000	R	000000	sll	$Rd = Rt \ll shamt$
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	$\text{if}(Rs == Rt) PC = PC + 4 + SignExtImme \ll 2$
000101	I	000101	bne	$\text{if}(Rs != Rt) PC = PC + 4 + SignExtImme \ll 2$
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	$Rt = Rs \wedge ZeroExtImme$
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) 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。

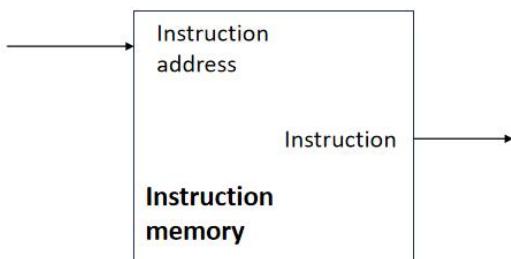
① 程序计数器 (PC)

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单周期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。

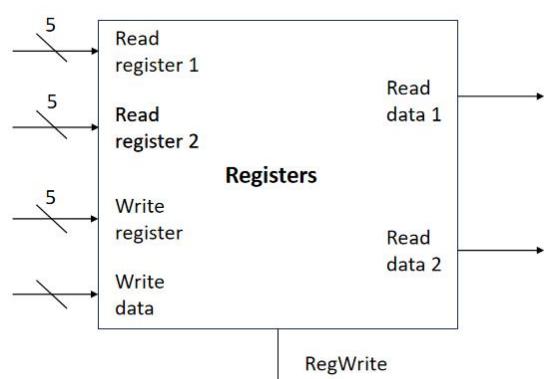
② 指令存储器

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。

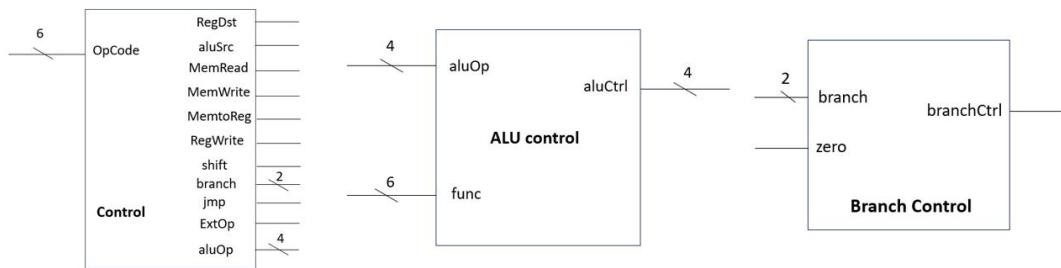


④ 控制单元

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，

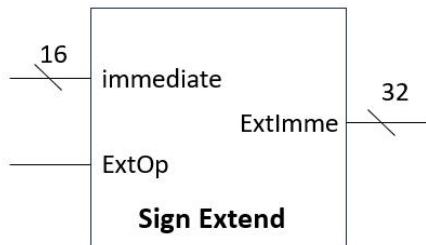
让其他部件正确执行该指令。ALU控制单元接收aluOp和func，输出aluCtrl给ALU。

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。



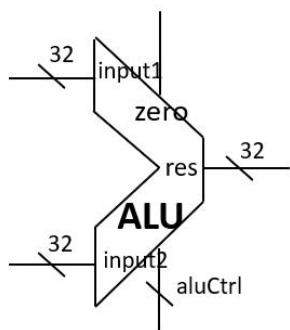
⑤ 符号扩展单元

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。



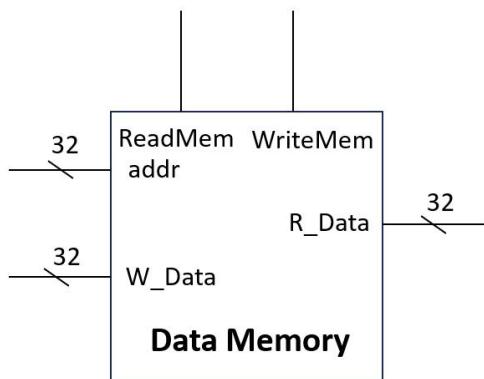
⑥ 算术逻辑单元 (ALU)

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。



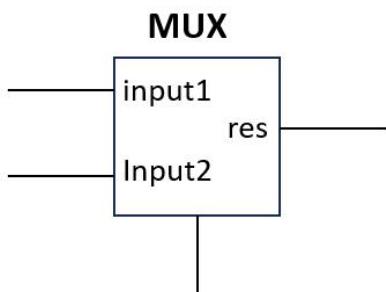
⑦ 数据存储器

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。



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。

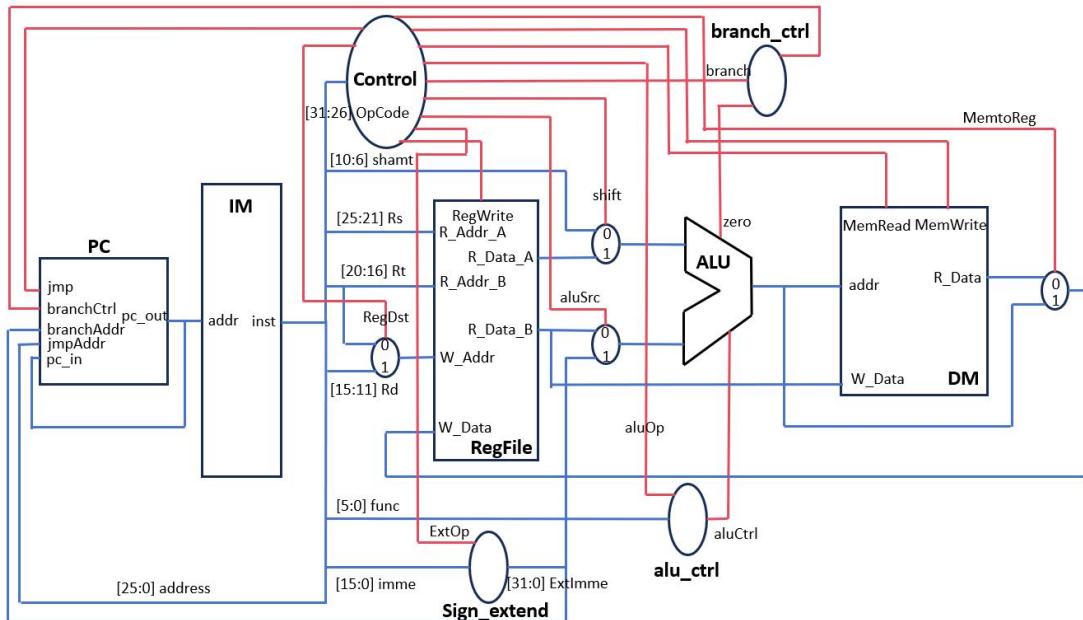
首先是R型指令。指令存储器将程序计数器对应的指令取出，指令译码后从寄存器堆读出Rs和Rt的数据，同时控制单元发送控制信号。然后Rs/shamt和Rt被送入ALU进行运算，运算结果写回Rd寄存器。

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。这里再次展示数据通路图。



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
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/slta	0	0	x	2' b00
sll/srl/sra	1	0	x	2' b00

续表

指令名称	shift	jmp	ExtOp	[1:0]branch
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(bne)	0	0
2' b10(bne)	1	0
2' b10(bne)	0	1

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/sl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PC模块的输出又被传进PC模块造成混乱的情况，在“实验感想”板块会进一步说明。U0—U11是设计CPU相关的，U12—U15是物理实现相关的。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'd113)
        pc <= pc;
    else
        pc <= pc_plus;
end
```

U0符号扩展模块：根据ExtOp的内容进行零扩展或最高位扩展。

U1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。

U2ALU控制单元模块：综合aluOp和func，输出aluCtrl。根据ALU功能表进行编写。

U3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。写的时候用时钟信号进行同步。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];
```

U4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。

U5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。

U6ALU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。

U7ALU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。

U8ALU：通过aluCtrl信号对两个输入的数进行运算，并输出零标志位。

U9分支控制单元：接收branch和zero，输出branchCtrl。

U10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。写的时候用时钟信号进行同步。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单周期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];
```

U11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
);
```

U12时钟分频器一：用来将时钟分频到1s，便于在数码管上依次输出数据。

U13按钮去抖动器：用来给按钮去抖动，防止按钮出现不稳定信号。

U14显示模块：数码管显示数据。

U15时钟分频器二：用来给时钟分频，让频率低一些。实验板的时钟信号频率过高，会导致程序执行失败。

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控制信号、排好序的10个数据等，方便仿真时查看。物理实现时，只需要输出数码管显示、LED灯即可。LED灯的实现可以利用一个计数器从1开始计数，每输入一个数计数器加一，同时LED灯左移一位，用来表示现在输入的数据在存储器的位置。LED灯的实现可以参考跑马灯实验的原理。同时也可以用这个计数器作为地址向数据存储器输入数据，在数据存储器添加data_in, btn, read和addr_in的输入，按下按钮就将数据写入addr_in的位置中。在数据存储器模块添加以下代码：

```
always @(posedge clk) begin
    if(read == 1 && btn == 1) begin
        memory[addr_in-1'b1] <= data_in;
```

(3) 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单周期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。

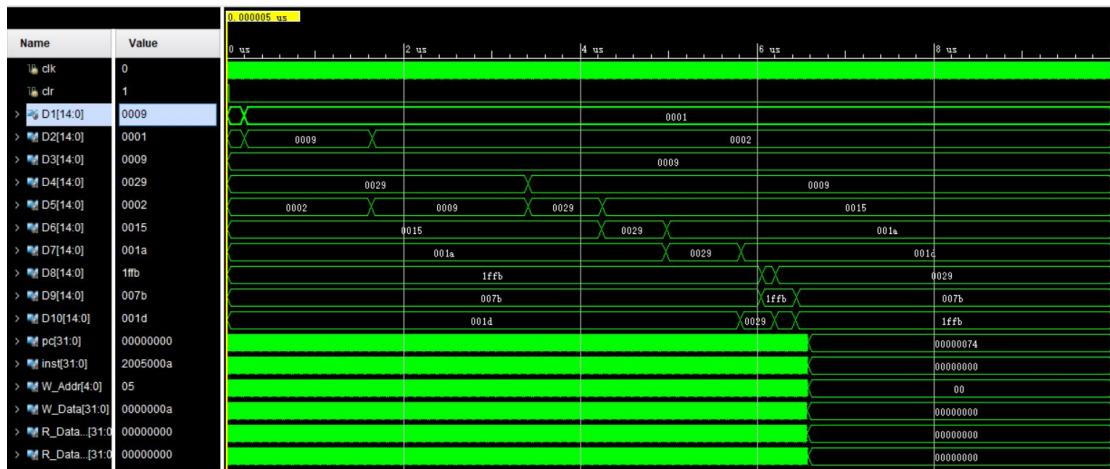
然后我们在数据存储器模块中对第0到第9共10个存储单元进行初始化，随意地给这十个数赋值。之后编写仿真文件（仿真文件里对顶层文件进行实例化）运行仿真，查看结果。

最后添加物理实现的代码（U12—U15），将程序烧录到实验板上，查看结果。

(三) 实验结果

(1) 仿真结果及分析

运行仿真，得到波形图，其全貌如下。



通过比较最开始和最后的D1—D10，我们可以发现原来的十个数已经按升序的顺序排列好了，说明设计的CPU和冒泡排序程序在结果上看是没问题的。

Name	Value	Name	Value
clk	0	clk	0
clr	1	clr	0
D1[14:0]	0009	> D1[14:0]	0001
D2[14:0]	0001	> D2[14:0]	0002
D3[14:0]	0009	> D3[14:0]	0009
D4[14:0]	0029	> D4[14:0]	0009
D5[14:0]	0002	> D5[14:0]	0015
D6[14:0]	0015	> D6[14:0]	001a
D7[14:0]	001a	> D7[14:0]	001d
D8[14:0]	1ffb	> D8[14:0]	0029
D9[14:0]	007b	> D9[14:0]	007b
D10[14:0]	001d	> D10[14:0]	1ffb

接下来我们需要验证指令执行过程的正确性，考察一些具有代表性的指令的控制信号和部分数据通路。下面对于部分指令的波形图进行分析。

1、addi \$t4, \$a1, -1

这是一条I型指令。机器代码为001000_00101_01100_1111111111111111。ALU的两个操作数是Rs的值（10）和最高位扩展后的立即数（-1），ALU做加法运算，结果为9，将ALU结果送到Rt寄存器（\$12）中。该指令正确运行。同理可以推出与这条指令类似的I型指令（andi/ori等）也是正确运行的。

Name	Value	46 ns	48 ns	50 ns	52 ns
> inst[31:0]	20acffff		20acffff		
> W_Addr[4:0]	0c		0c		
> W_Data[31:0]	00000009		00000009		
> R_Data...[31:0]	0000000a		0000000a		
> R_Data...[31:0]	00000000		00000000		
> alu_res[31:0]	00000009		00000009		
> R_Mem...31:0	00000000				
RegDst	0				
aluSrc	1				
MemWrite	0				
MemRead	0				
RegWrite	1				
MemtoReg	1				
shift	0				
> branch[1:0]	0				
branchCtrl	0				
jmp	0				
ExtOp	1				
> aluOp[3:0]	0		0		
> aluCtrl[3:0]	2		2		
> alu1[31:0]	0000000a		0000000a		
> alu2[31:0]	ffffffff		ffffffff		
zero	0				

2、sll \$t0, \$s0, 2

这是一条R型指令，机器代码为000000_00000_10000_01000_00010_000000。

ALU的两个操作数为shamt (2) 和Rt的值 (0) ， ALU做左移运算，结果为0，将ALU结果送到Rd寄存器 (\$8) 中。该指令正确运行。同理可以推出其他R型指令也是正确运行的。

Name	Value	68 ns	88 ns	90 ns	92 ns	94 ns
> inst[31:0]	00104080		00104080			
> W_Addr[4:0]	08					
> W_Data[31:0]	00000000					
> R_Data...[31:0]	00000000					
> R_Data...[31:0]	00000000					
> alu_res[31:0]	00000000					
> R_Mem...31:0	00000000					
RegDst	1					
aluSrc	0					
MemWrite	0					
MemRead	0					
RegWrite	1					
MemtoReg	1					
shift	1					
> branch[1:0]	0					
branchCtrl	0					
jmp	0					
ExtOp	0					
> aluOp[3:0]	f					
> aluCtrl[3:0]	4		4			
> alu1[31:0]	00000002		00000002			
> alu2[31:0]	00000000					
zero	1					

3、lw \$s1, 0(\$t0)

这是一条I型指令，机器代码为100011_01000_10001_0000000000000000。ALU的两个操作数为Rs的值（0）和最高位扩展后的立即数（0），ALU做加法运算，结果为0，ALU结果将被送入数据寄存器，读取地址为0的数据（9），该数据写回Rt寄存器。该指令正确运行。

Name	Value	106 ns	108 ns	110 ns	112 ns
> inst[31:0]	8d110000		8d110000		
> W_Addr[4:0]	11		11		
> W_Data[31:0]	00000009		00000009		
> R_Data...[31:0]	00000000				
> R_Data...[31:0]	00000000				
> alu_res[31:0]	00000000		00000000		
> R_Mem...31:0	00000009		00000009		
RegDst	0				
aluSrc	1				
MemWrite	0				
MemRead	1				
RegWrite	1				
MemtoReg	0				
shift	0				
> branch[1:0]	0				
branchCtrl	0				
jmp	0				
ExtOp	1				
> aluOp[3:0]	0				
> aluCtrl[3:0]	2				
> alu1[31:0]	00000000				
> alu2[31:0]	00000000			00000000	
zero	1				

4、sw \$s1, 0(\$t2)

这是一条I型指令，机器代码为101011_01010_10001_0000000000000000。ALU的两个操作数为Rs的值（4）和最高位扩展后的立即数（0），ALU做加法运算，结果为0，ALU结果将被送进数据存储器，把Rt的值（9）写到地址为4的存储单元。该指令正确执行。

Name	Value
> inst[31:0]	ad510000
> W_Addr[4:0]	11
> W_Data[31:0]	00000000
> R_Data...[31:0]	00000004
> R_Data...[31:0]	00000009
> alu_res[31:0]	00000004
> R_Mem...31:0	00000000
RegDst	0
aluSrc	1
MemWrite	1
MemRead	0
RegWrite	0
MemtoReg	0
shift	0
> branch[1:0]	0
branchCtrl	0
jmp	0
ExtOp	1
> aluOp[3:0]	0
> aluCtrl[3:0]	2
> alu1[31:0]	00000004
> alu2[31:0]	00000000
zero	0

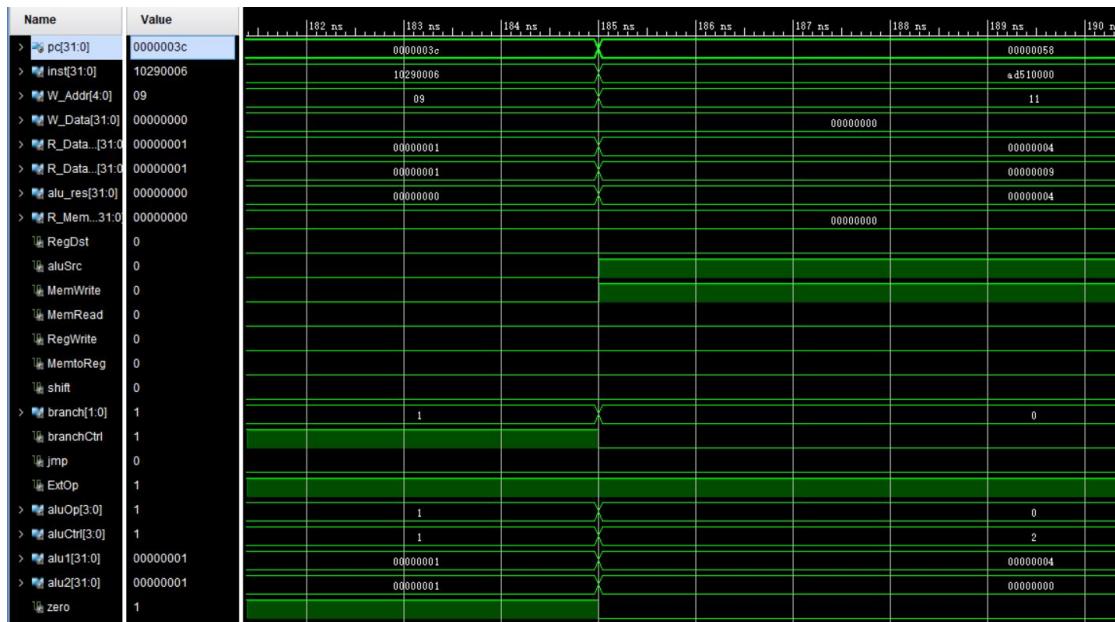
可以从下图看到，两条sw指令执行完之后，D1和D2的数值进行了交换。

Name	Value
clk	1
clr	0
> D1[14:0]	0001
> D2[14:0]	0009

5、beq \$t1, 1, swap

这条指令是I型指令，机器代码为000100_00001_01001_0000000000000110。ALU的两个操作数为Rs的值（1）和Rt的值（1，这个数是数据），ALU做减法运算，结果为0，零标志位为1，分支控制单元发出分支信号，程序计数器被修改到 $3c+4+6*4 = 58$ 的位置。

可以看到下一个的程序计数器为58。该指令正确运行。可以推出bne指令也是正确执行的。

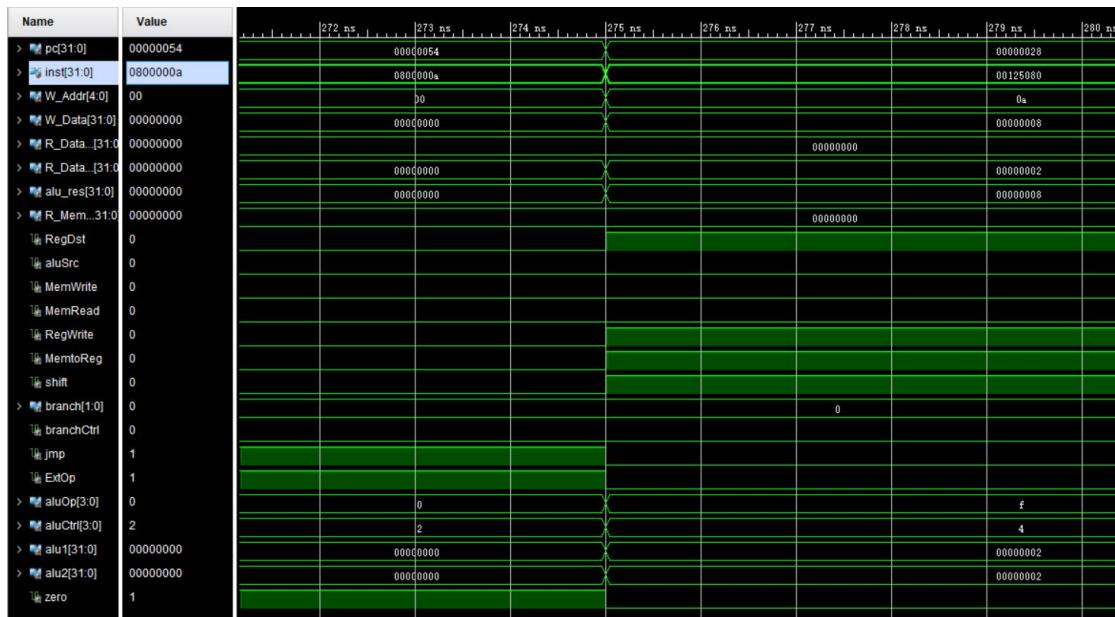


值得一提的是，beq和bne指令是分两条指令执行的。以上面的指令为例，它被分解为下面的两条指令：

```
addi $at, $zero, 1
beq $at, $t1, swap
```

6、j in_loop

这条指令是J型指令，机器代码为000010_00000000000000000000000000001010。主控制单元发出jmp信号，程序计数器直接修改为{PC+4[31:28], jmpAddr, 2' b00} = 0000_00000000000000000000000000001010_00即28。可以看到下一个程序计数器的值为28。该指令正确执行。

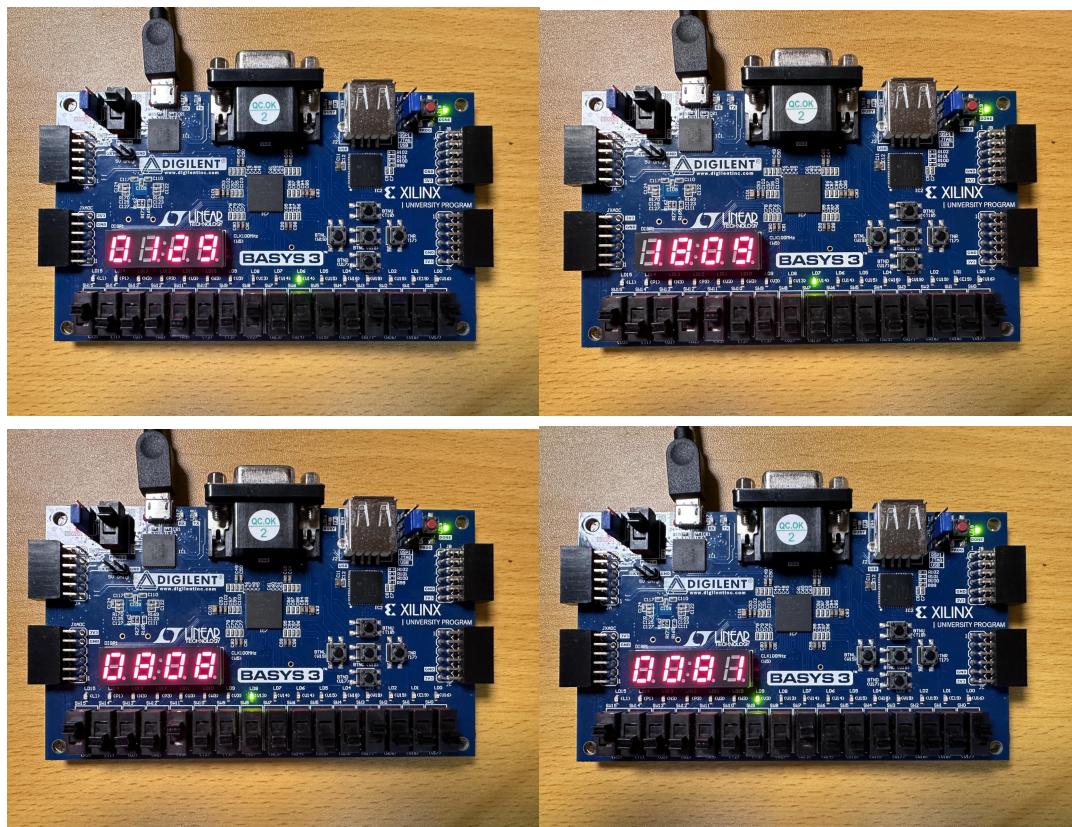


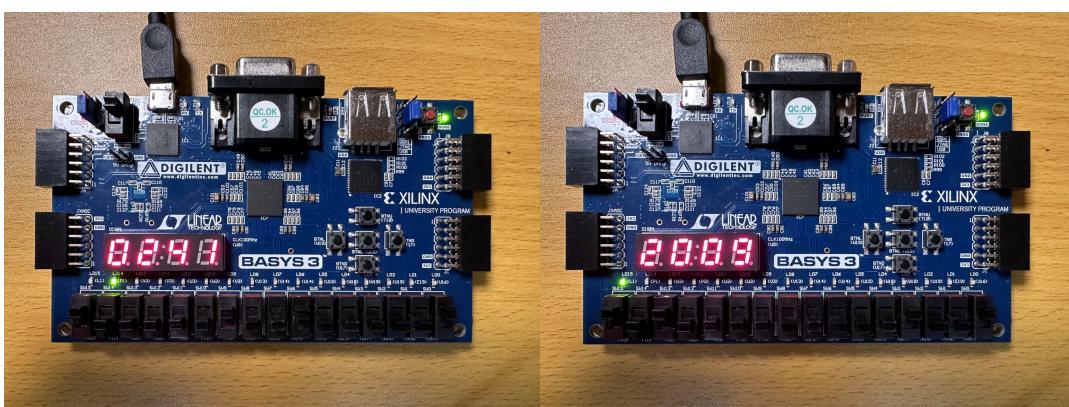
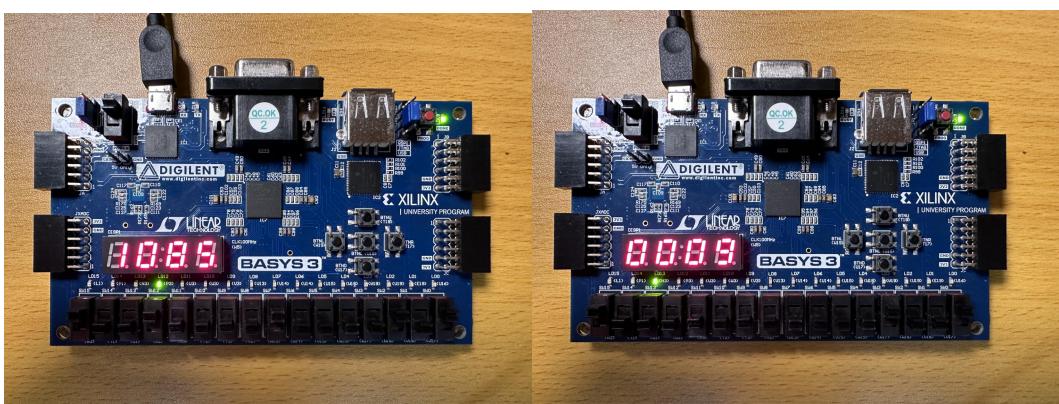
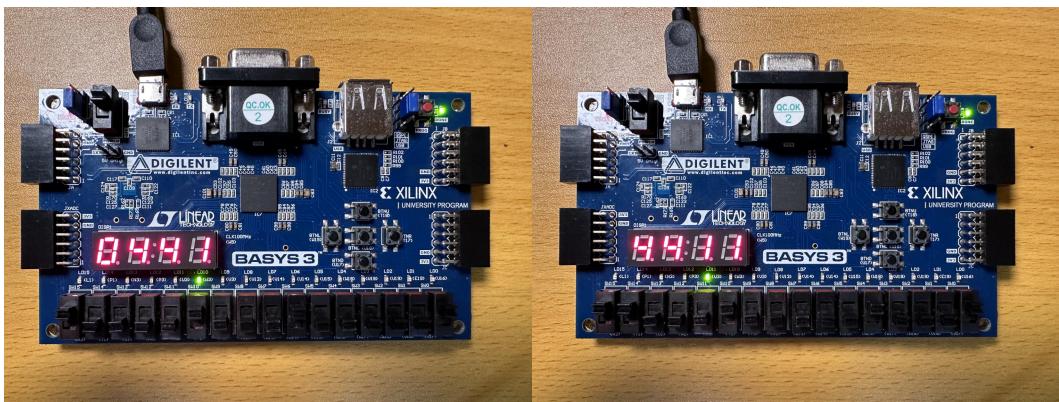
综上，我们从仿真出来的波形图可以验证设计的CPU的正确性。

(2) 物理实现结果

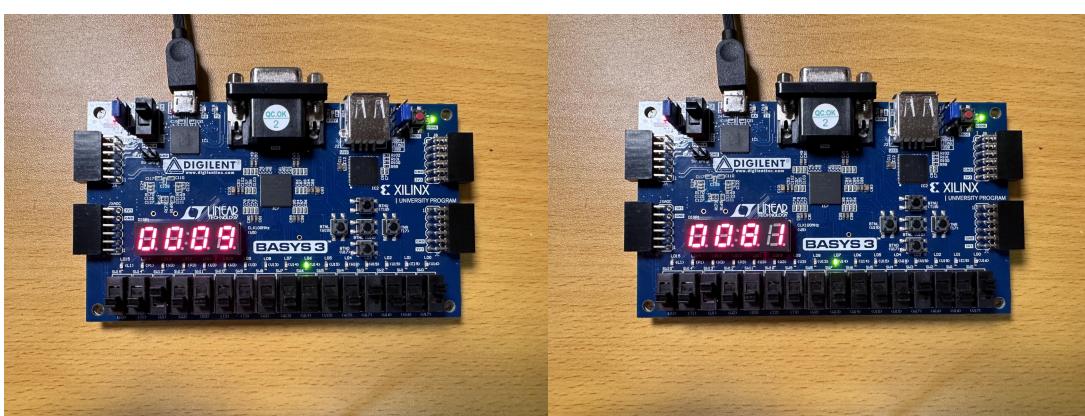
加上模块U12—U15，并在顶层文件加上显示数据和显示LED灯的代码，将项目烧录到实验板上查看结果。具体的操作如下：先将最左边的开关（R2）置1，表示现在正在输入数字。利用剩下的15个开关输入数字，输入的内容可以在数码管上看到，同时LED灯也会显示当前数据会被存放在哪个位置（最左边是10，最右边是1）。长按按钮，等LED灯左移了一位就代表此时的数据已经存进去了，就可以松开按钮（此处需要长按按钮的原因是需要等待时钟上升沿到来）。等10个数全部输入完毕后，将R2开关置0，就可以看到升序排序的数字依次并循环显示。输入和输出的图片如下。（顺序均为从左到右，从上到下）

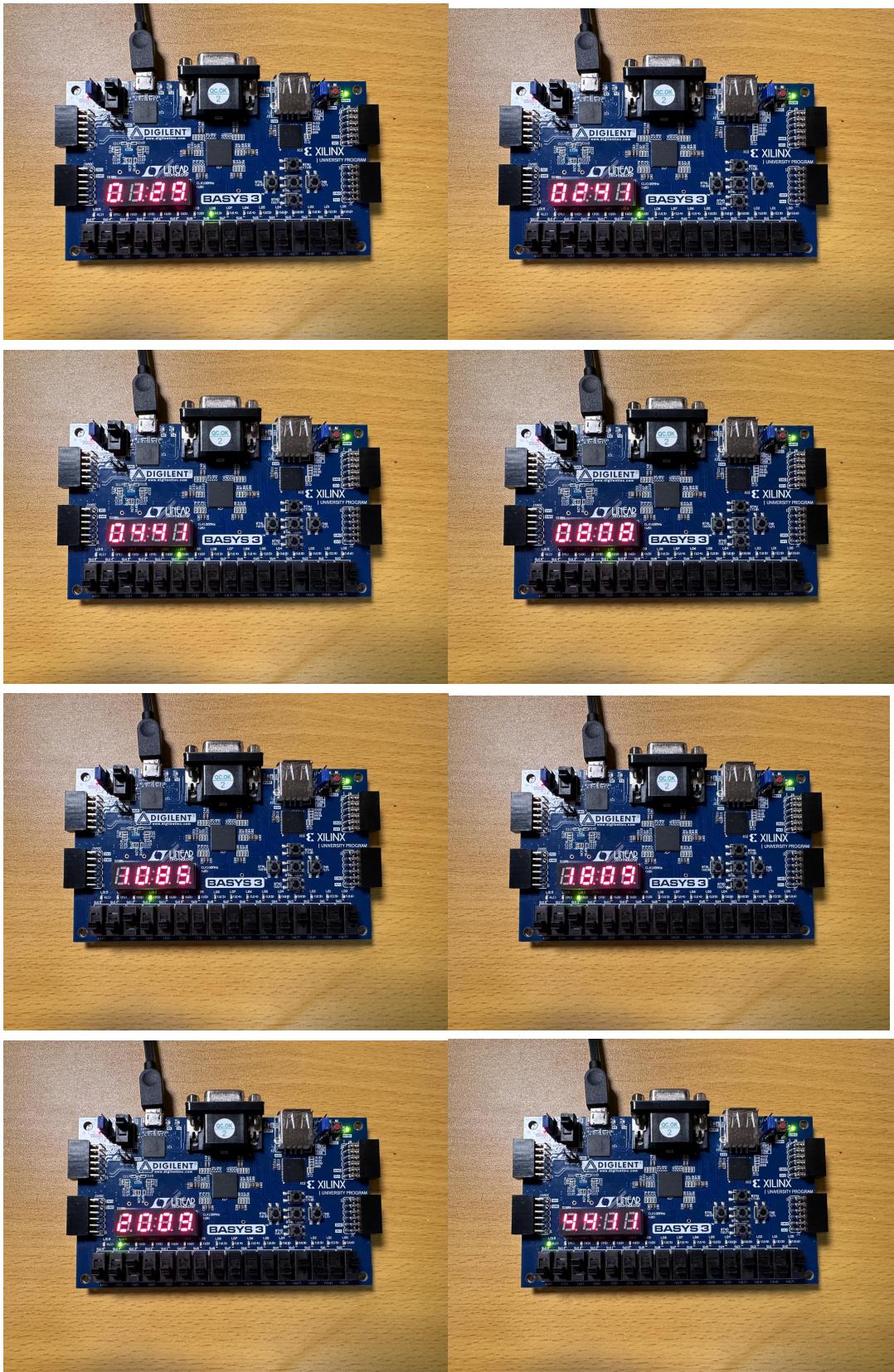
输入：





输出：





至此，MIPS单周期CPU设计完毕，实验完成。

六. 实验心得

本次实验大概用了30个小时，其中编写代码用了2个小时，后面的时间都在纠错。回顾整个实验的过程，我总结了一下本次实验所遇到的困难以及解决措施。

1、没有仿真就开始编写物理实现的代码进行物理实现

我在写完每个模块之后就直接在顶层文件编写输入和输出的代码，好不容易能够让LED灯正确运行并能够将数据输入到存储器中，却发现数据根本没有排序。于是我才意识到仿真的重要性，开始运行仿真并分析。

2、PC运行的错误

一开始我是将PC单独写成一个模块，在顶层文件让PC模块的输出赋值给PC模块的输入。但是这样出来的仿真图像就会出现一个时钟周期内有输出，但下一个时钟周期内的输出是X（未知量）。反复调整无果后，我打算直接把PC模块写进顶层文件中，这样输出和输出就是同一个变量了，于是问题解决，每个时钟周期都是正确的PC值。

3、指令的取出和PC的值不同步

调整完PC后我发现仿真波形图中指令对应的地址和PC不同步，指令总是比PC慢一个时钟周期。比如，第二条指令应是PC为4时取出，可是波形图显示取出第二条指令的PC为8。这样就导致分支语句和跳转语句无法正确运行。当分支语句后面接跳转语句的时候，分支语句执行完且需要跳转，但指令的取出会比PC慢一个时钟周期，那么下一条指令也就是跳转指令会先被取出，然后再执行分支后的指令，但是接着便会执行跳转后的指令，造成混乱。究其原因，我发现是我使用的IP核Block Memory Generator本身会有一个时钟周期的延迟，如下图所示。

Information

```

Memory Type: Single Port Memory
Block RAM resource(s) (18K BRAMs): 1
Block RAM resource(s) (36K BRAMs): 0
Total Port A Read Latency : 1 Clock Cycle(s)
Address Width A: 4

```

上网搜索之后，我发现有一个ROM IP核是没有延迟的，就是Distributed Memory Generator，于是我就使用这个IP核，解决了这个问题。

4、时钟频率过高

通过仿真验证正确后，我开始进行物理实现。但是当我输入10个数之后，输出的数

虽然是正确排序的，但是有些数是凭空出现的，我在输入的时候根本没有输入这些数。其原因是实验板上的时钟频率太高，可能是指令的执行速度较慢，跟不上时钟，于是我便添加了时钟分频模块，最终输出正确。

当然，这次实验遇到的不止这些问题和困难，还有一些小问题，比如如何执行位移指令，如何对beq和bne指令进行区分等等。对于前者，我添加了shift信号来控制ALU的其中一个操作数，对于后者，我额外添加了分支控制单元以给出最终的分支信号。

关于这次实验的不足之处，其一是在实验板上输入数据的时候需要长按按钮等待时钟上升沿来临才能够完成存储，更好的方式是按一下就可以完成存储。其二是本次实验一共写了四个不同的多选器，但是应该可以想办法用一个通用的多选器来实现。

这次实验其实就是一个不断发现错误、纠正错误的过程。实验原理就是MIPS单周期CPU的运行原理，但是要真正利用代码去实现并没有想象中容易。在上手之前我参考了学长们编写的实验报告，确定了整个实验流程和模块化的思想，这为我节省了不少时间和精力，在此对于他们表示感谢。整个实验最难的部分我认为是PC部分的编写，因为只有正确取出指令才能够让接下来执行指令是正确的。在做实验的过程中，很多次我认为我卡在某一点，无法再进行突破了，因为根本没有解决问题的思路。但当我第二天继续思考和查找，会突然想到和找到解决问题的方法，或者是可以进行创新的地方。所以，不要低估自己，不要轻易放弃。