第五次实验——单周期CPU

一、实验目的

1.掌握单周期CPU的设计方法。

2.加深对CPU内各模块单元、数据通路的理解。

3.使用设计出的CPU实现输入十个数后进行冒泡排序。

二、实验内容

1.指令

在这个实验中，我设计的CPU将支持下述精简的MIPS指令集：

(1).6个R-type运算指令

(1) `add tar, src1, src2`

(2) `sub tar, src1, src2`

(3) `and tar, src1, src2`

(4) `or tar, src1, src2`

(5) `slt tar, src1, src2`

(6) `sll tar, src1, imm`

(2).3个I-type运算指令

由于 `sub slt sll` 指令没有相应的 `I` 型形式，故 `I-type` 运算指令只有以下三个：

(7) `addi tar, src1, imm`

(8) `andi tar, src1, imm`

(9) `ori tar, src1, imm`

(3).2个I-type存取数指令

(10) `lw tar, imm(src1)`

(11) `sw src2, imm(src1)`

(4).2个I-type分支指令

(12) `beq src1, src2, imm`

(13) `bgtz src1, imm`

(5).1个J-type跳转指令

(14) `j imm`

(6).1个J-type停止指令

(15) `halt`

与原生MIPS指令集稍有不同的是，这里不使用 `rd rs rt` 助记符，相应以 `src1 src2` 表示源寄存器， `tar` 表示目标寄存器。这是为了更加清楚表示助记符中寄存器的性质。 `src1 src2` 都是只读的， `tar` 是只写的，这在之后列出的 `I-type` 机器码中会清晰体现。

在这样的规定下，可以减少一个多选器(`WrReg Mux` ，唯一一个五位多选器)，并且信号 `RegDst` 只有“是否限制第二个源寄存器读取”的功能，这会在之后体现。

为此，给出上述指令的机器码形式：

R-type	opcode[31:26]	src1[25:21]	src2[20:16]	tar[15:11]	shamt[10:6]	funct[5:0]
add	000000	.....	.....	.....	00000	100000
sub	000000	.....	.....	.....	00000	100010
and	000000	.....	.....	.....	00000	100100
or	000000	.....	.....	.....	00000	100101
slt	000000	.....	.....	.....	00000	101010
``	opcode[31:26]	src2[25:21]	src1[20:16]	tar[15:11]	shamt[10:6]	funct[5:0]
sll	000000	00000	.....	.....	.....	000000

可以看到 `R_type` 运算指令的 `opcode` 都是 `0` ， 因此区分的重点在于 `funct` 。

I_type	opcode[31:26]	src1[25:21]	tar[20:16]	imm[15:0]
addi	001000	.....	.....	xxx[15:0]
andi	001100	.....	.....	xxx[15:0]
ori	001101	.....	.....	xxx[15:0]
lw	100011	.....	.....	xxx[15:0]
	opcode[31:26]	src1[25:21]	src2[20:16]	imm[15:0]
beq	000100	.....	.....	xxx[15:0]

I_type	opcode[31:26]	src1[25:21]	tar[20:16]	imm[15:0]
bgtz	000111	.....	00000	xxx[15:0]
sw	101011	.....	.....	xxx[15:0]

在 src1 src2 tar 规则助记下， I_type 被分为两类。这会清楚地表明指令中是否存在要被写入的寄存器及其编号。这会帮助控制单元的简化，见稍下处。

J-type	opcode[31:26]	imm[25:0]
j	000010	xxx[25:0]
halt	111111	111[25:0]

2.控制单元Control与ALUControl码

在这个实验中，与原生的MIPS CPU不同，我将控制单元 Control 在原来的基础上赋予了更多功能，比如直接由它输出源寄存器和目标寄存器编号，同时也由它根据 opcode funct 字段直接生成四位 ALUControl 信号、而非生成两位 ALUop 信号。

与原生的 ALUControl 信号表不同，我将左移信号替代了或非信号，这是因为原生CPU中有至少两处左移硬件：

Operation	^	|	+	-	<	<<
ALUControl	0000	0001	0010	0110	0111	1100

在此基础上，给出 ALUControl 信号生成表：

ALUControl	Symbol	Opcode	Funct
0000,'^'	and	000000	100100
	andi	001100	NULL
0001,'|'	or	000000	100101
	ori	001101	NULL
0010,'+'	add	000000	100000
	addi	001000	NULL
	lw	100011	NULL
	sw	101011	NULL
	j	000010	NULL
0110,'-'	sub	000000	100010
	bgtz	000111	NULL
	beq	000100	NULL
0111,'<'	slt	000000	101010

ALUControl	Symbol	Opcode	Funct
1100,'<<'	sll	000000	000000

由于直接由控制单元指定源寄存器与目标寄存器，所以相比于原生MIPS CPU，少了一个 `WrReg MUX` 多选器。而原来的 `RegDst` 信号相应会直接集成到 `RegFile` 寄存器堆中。这样，剩下的所有多选器都是32位的二路多选，只需要编写一个 `MUX` 再生成四个不同名的实例即可。

相比于原生CPU，由于多了一条 `halt` 停止指令，所以控制单元还多了一个 `Halt_en` 信号。这个信号会直接送入程序计数器中。

3.冒泡排序代码

冒泡排序的C语言代码如下：

```
int A[10] = {4, 5, 2, 1, 3, 6, 9, 7, 8, 10}; //测试数据
void BubbleSort(int A[])
{
    for (int j = 0; j < 9; j++) {
        for (int i = 0; i < 9 - j; i++) {
            if (A[i] < A[i + 1]) {
                int temp = A[i];
                A[i] = A[i + 1];
                A[i + 1] = temp;
            }
        }
    }
}
```

编译为MIPS汇编如下：

```
main:
    addi $a0, $zero, 0
    addi $t0, $zero, 9

BubbleSort:
    addi $t1, $zero, 0
    addi $t2, $zero, 0

outer_loop:
    beq $t2, $t0, done_outer_loop
    add $a1, $a0, $zero
    addi $t1, $zero, 0

inner_loop:
    sub $t7, $t0, $t2
    beq $t1, $t7, done_inner_loop
    lw $t3, 0($a1)
    lw $t4, 4($a1)
    slt $t8, $t3, $t4
    beq $t8, $zero, no_swap
    sw $t4, 0($a1)
    sw $t3, 4($a1)

no_swap:
    addi $a1, $a1, 4
    addi $t1, $t1, 1
    j inner_loop

done_inner_loop:
    addi $t2, $t2, 1
    j outer_loop

done_outer_loop:
    halt
```

这段代码是直接针对本次实验的CPU编写的。生成如下二进制码（修改了无条件跳转的低26位）：

```
00100000 00000100 00000000 00000000
00100000 00001000 00000000 00001001
00100000 00001001 00000000 00000000
00100000 00001010 00000000 00000000
00010001 01001000 00000000 00001111
00000000 10000000 00101000 00100000
00100000 00001001 00000000 00000000
00000001 00001010 01111000 00100010
00010001 00101111 00000000 00001001
10001100 10101011 00000000 00000000
10001100 10101100 00000000 00000100
00000001 01101100 11000000 00101010
00010011 00000000 00000000 00000010
10101100 10101100 00000000 00000000
10101100 10101011 00000000 00000100
00100000 10100101 00000000 00000100
00100001 00101001 00000000 00000001
00001000 00000000 00000000 00000111
00100001 01001010 00000000 00000001
00001000 00000000 00000000 00000100
11111111 11111111 11111111 11111111
```

三、实验过程(CPU部分)

接下来我会按照指令的执行过程展示CPU各模块。

1.程序计数器 `ProgramCounter`

代码如下：

```
module ProgramCounter(
    input CLK_in,
    input Start_en,
    input Halt_en,
    input [31 : 0] Address_in,
    output [31 : 0] Address_out
    );

    reg [31 : 0] Address;

    initial Address <= 0;

    always@(posedge CLK_in) begin
        if (Start_en == 0) Address <= 0;
        else begin
            if (Halt_en == 1) Address = Address;
            else Address = Address_in;
        end
    end

    assign Address_out = Address;

endmodule
```

PC是时序的，每个时钟周期更新一次。只有在允许开始执行的时候PC才会开始增长。当遇到停止指令（信号）时，PC就会直接停止在停止指令处。

2.指令存储器 `InstructMem`

代码如下：

```
module InstructMem(
    input [31 : 0] InstructAddress,
    output [31 : 0] Instruction
    );

    reg [7 : 0] Instruct [0 : 200];

    initial begin
        $readmemb("H:\ViVado\Vivado Projects\SingleCPU\Bubble.txt", Instruct);
    end

    assign Instruction[31 : 24] = Instruct[InstructAddress + 0];
    assign Instruction[23 : 16] = Instruct[InstructAddress + 1];
    assign Instruction[15 : 8]  = Instruct[InstructAddress + 2];
    assign Instruction[7 : 0]   = Instruct[InstructAddress + 3];

endmodule
```

这个模块会从已生成的二进制文本中读取装载指令。这些指令已经在上面给出。我的MIPS代码有25行，因此至少需要100个8位单元。基于冗余的思想，这里多设置了一倍的单元。

3.控制单元 `Control`

本实验中设计的控制单元与原生CPU有一些不同，具体可见上述*实验内容*中的阐述。代码如下：

```
module Control(
    input [31 : 0] Instruct,

    output reg [3 : 0] ALUControl,

    output reg [4 : 0] RegRead1,
    output reg [4 : 0] RegRead2,
    output reg [4 : 0] RegWrite,

    output reg ALUSrc,

    output reg RegDst,
    output reg RegWrite_en,

    output reg MemRead_en,
    output reg MemWrite_en,
    output reg MemtoReg_en,

    output reg Jump_en,
    output reg Branch_en,
    output reg Halt_en
    );

    always@(Instruct) begin
        { ALUSrc, RegDst, RegWrite_en, MemRead_en, MemWrite_en, MemtoReg_en, Jump_en, Branch_en, Halt_en } = 9'b00000
        case (Instruct[31 : 26])
            6'b000000: begin     //R_type
                RegRead1 <= Instruct[25 : 21];
                RegRead2 <= Instruct[20 : 16];
                RegWrite <= Instruct[15 : 11];
                case (Instruct[5 : 0])
                    6'b100000: begin ALUControl <= 4'b0010; { ALUSrc, RegWrite_en } <= 2'b11; end     //add
                    6'b100010: begin ALUControl <= 4'b0110; { ALUSrc, RegWrite_en } <= 2'b11; end     //sub
                    6'b100100: begin ALUControl <= 4'b0000; { ALUSrc, RegWrite_en } <= 2'b11; end     //and
                    6'b100101: begin ALUControl <= 4'b0001; { ALUSrc, RegWrite_en } <= 2'b11; end     //or
                    6'b101010: begin ALUControl <= 4'b0111; { ALUSrc, RegWrite_en } <= 2'b11; end     //slt
                    6'b000000: begin     //sll
                        ALUControl <= 4'b1100;
                        RegRead1 <= Instruct[20 : 16];
                        { RegDst, RegWrite_en } <= 2'b11;
                    end
                endcase
            end

            6'b001000: begin      //addi
                RegRead1 <= Instruct[25 : 21];
                RegWrite <= Instruct[20 : 16];
                ALUControl <= 4'b0010;
                { RegDst, RegWrite_en } <= 2'b11;
            end
            6'b001100: begin      //andi
                RegRead1 <= Instruct[25 : 21];
```

```
                RegWrite <= Instruct[20 : 16];
                ALUControl <= 4'b0000;
                { RegDst, RegWrite_en } <= 2'b11;
            end
            6'b001101: begin      //ori
                RegRead1 <= Instruct[25 : 21];
                RegWrite <= Instruct[20 : 16];
                ALUControl <= 4'b0001;
                { RegDst, RegWrite_en } <= 2'b11;
            end

            6'b100011: begin      //lw
                RegRead1 <= Instruct[25 : 21];
                RegWrite <= Instruct[20 : 16];
                ALUControl <= 4'b0010;
                { RegDst, RegWrite_en, MemRead_en, MemtoReg_en } <= 4'b1111;
            end
            6'b101011: begin      //sw
                RegRead1 <= Instruct[25 : 21];
                RegRead2 <= Instruct[20 : 16];
                ALUControl <= 4'b0010;
                { MemWrite_en } <= 1'b1;
            end

            6'b000100: begin      //beq
                RegRead1 <= Instruct[25 : 21];
                RegRead2 <= Instruct[20 : 16];
                ALUControl <= 4'b0110;
                { ALUSrc, Branch_en } <= 2'b11;
            end
            6'b000111: begin      //bgtz
                RegRead1 <= Instruct[25 : 21];
                RegRead2 <= 5'b00000;
                ALUControl <= 4'b0110;
                { ALUSrc, Branch_en } <= 2'b11;
            end

            6'b000010: begin Jump_en <= 1; end    //j
            6'b111111: begin Halt_en <= 1; end    //halt
        endcase
    end

endmodule
```

控制单元是组合逻辑的，只要有指令输入就会相应。

4.寄存器堆 `RegisterFile`

寄存器堆是组合逻辑的。但是当有数据需要写入时，只有在时钟下降沿的同时写使能信号为 `1` 时才写入，这部分是时序逻辑的。信号 `RegDst` 直接集成在其中，这个信号决定是否有第二个源寄存器的读取。在这里还需要注意 `0` 号寄存器的特殊性。

代码如下：

```
module RegisterFile(
    input CLK_in,
    input [4 : 0] RegRead1,
    input [4 : 0] RegRead2,
    input [4 : 0] RegWrite,
    input RegWrite_en,
    input RegDst,
    input [31 : 0] RegWriteData,
    output [31 : 0] ReadData1,
    output [31 : 0] ReadData2
    );

    reg [31 : 0] Reg [1 : 31];
    integer i;

    initial begin
        for (i = 0; i <= 31; i = i + 1) Reg[i] <= 0;
    end

    assign ReadData1 = (RegRead1 == 0) ? 0 : Reg[RegRead1];
    assign ReadData2 = (RegDst == 1) ? 1'bz : ((RegRead2 == 0) ? 0 : Reg[RegRead2]);

    always@(negedge CLK_in) begin
        if (RegWrite_en == 1 && RegWrite != 0) Reg[RegWrite] = RegWriteData;
    end

endmodule
```

5.符号扩展 SignExtend

符号扩展单元总要接受指令低16位的输入并输出扩展后的32位数，只是在之后的 ALUMux 或者地址计算中才决定是否使用。由于本实验中寻址范围较小，简便起见无需区别零扩展或是符号扩展；兼容起见，默认符号扩展。代码如下：

```
module SignExtend(
    input [15 : 0] InstructLow16bits,
    output reg [31 : 0] Data32bits
    );

    always@(*) begin
        Data32bits[15 : 0] <= InstructLow16bits;
        if (InstructLow16bits[15] == 0) Data32bits[31 : 16] <= 0;
        else Data32bits[31 : 16] <= 16'hFFFF;
    end

endmodule
```

6.算术逻辑单元 ALU

本实验中的 ALUControl 直接由控制单元生成，具体见上述。在将第六个功能由或非替换为左移之后，简便起见，由于程序中只有左移2位的操作，无需再从控制单元引出一条传送位移量 shamt 的线， ALU 中的左移操作固定只能移2位。这样，也可以再

避免重复写 ShiftLeft2 模块的工作，直接由 ALU 实例化即可。

代码如下：

```
module ALU(
    input [3 : 0] ALUControl,
    input [31 : 0] Src1,
    input [31 : 0] Src2,
    output reg [31 : 0] Result,
    output ZF,
    output SF
    );

    always@(*) begin
        case (ALUControl)
            4'b0000: Result = Src1 & Src2;
            4'b0001: Result = Src1 | Src2;
            4'b0010: Result = Src1 + Src2;
            4'b0110: Result = Src1 - Src2;
            4'b0111: Result = (Src1 < Src2) ? 1 : 0;
            4'b1100: Result = Src1 << 2;
        endcase
    end

    assign ZF = (Src1 == Src2) ? 1 : 0;
    assign SF = Result[0];

endmodule
```

7.数据存储器 DataMem

数据存储器是组合逻辑的，在读使能为 1 的情况下允许读出数据。但要写入时是时序逻辑的，只有在时钟下降沿与写使能为 1 的时候允许写入。代码如下：

```
module DataMem(
    input CLK_in,
    input [31 : 0] DataAddress,
    input [31 : 0] Data_in,
    input MemRead_en,
    input MemWrite_en,
    output [31 : 0] Data_out
    );

    reg [7 : 0] Data [0 : 64];

    assign Data_out[7 : 0]   = (MemRead_en == 1) ? Data[DataAddress + 3] : 8'bz;
    assign Data_out[15 : 8]  = (MemRead_en == 1) ? Data[DataAddress + 2] : 8'bz;
    assign Data_out[23 : 16] = (MemRead_en == 1) ? Data[DataAddress + 1] : 8'bz;
    assign Data_out[31 : 24] = (MemRead_en == 1) ? Data[DataAddress + 0] : 8'bz;

    always@(negedge CLK_in) begin
        if (MemWrite_en == 1) begin
            Data[DataAddress + 0] <= Data_in[31:24];
            Data[DataAddress + 1] <= Data_in[23:16];
            Data[DataAddress + 2] <= Data_in[15:8];
            Data[DataAddress + 3] <= Data_in[7:0];
        end
    end

endmodule
```

8.跳转地址生成单元 JumpAddGenerator

当有跳转指令时，跳转地址是由 PC+4 高4位与指令低26位左移2位拼接产生的。这部分不适合在 top 文件中实现，因此专门为其设计一个地址生成单元。代码如下：

```
module JumpAddGenerator(
    input [31 : 0] PC4High4bits,
    input [31 : 0] SL2Low28bits,
    output [31 : 0] Address_out
    );

    assign Address_out = {PC4High4bits[31 : 28], SL2Low28bits[27 : 0]};

endmodule
```

在这里，指令低26位已经左移2位。

9.二路32位多选器 Mux

这种多选器很简单，不多赘述，代码如下：

```
module Mux(
    input Condition,
    input [31 : 0] Data1,
    input [31 : 0] Data2,
    output [31 : 0] Result
    );

    assign Result = (Condition == 0) ? Data1 : Data2;

endmodule
```

在本实验中，由于去掉了 `WrReg Mux` ，并且不对分支、跳转另编指令，因此无需四路多选器，二路32位多选器可以囊括所有需求。将要生成的四个多选器实例是：

`ALU Mux` ， `Mem/ALU Mux` ， `Branch Mux` ， `Jump Mux`

10.输入输出接口 `IOPort`

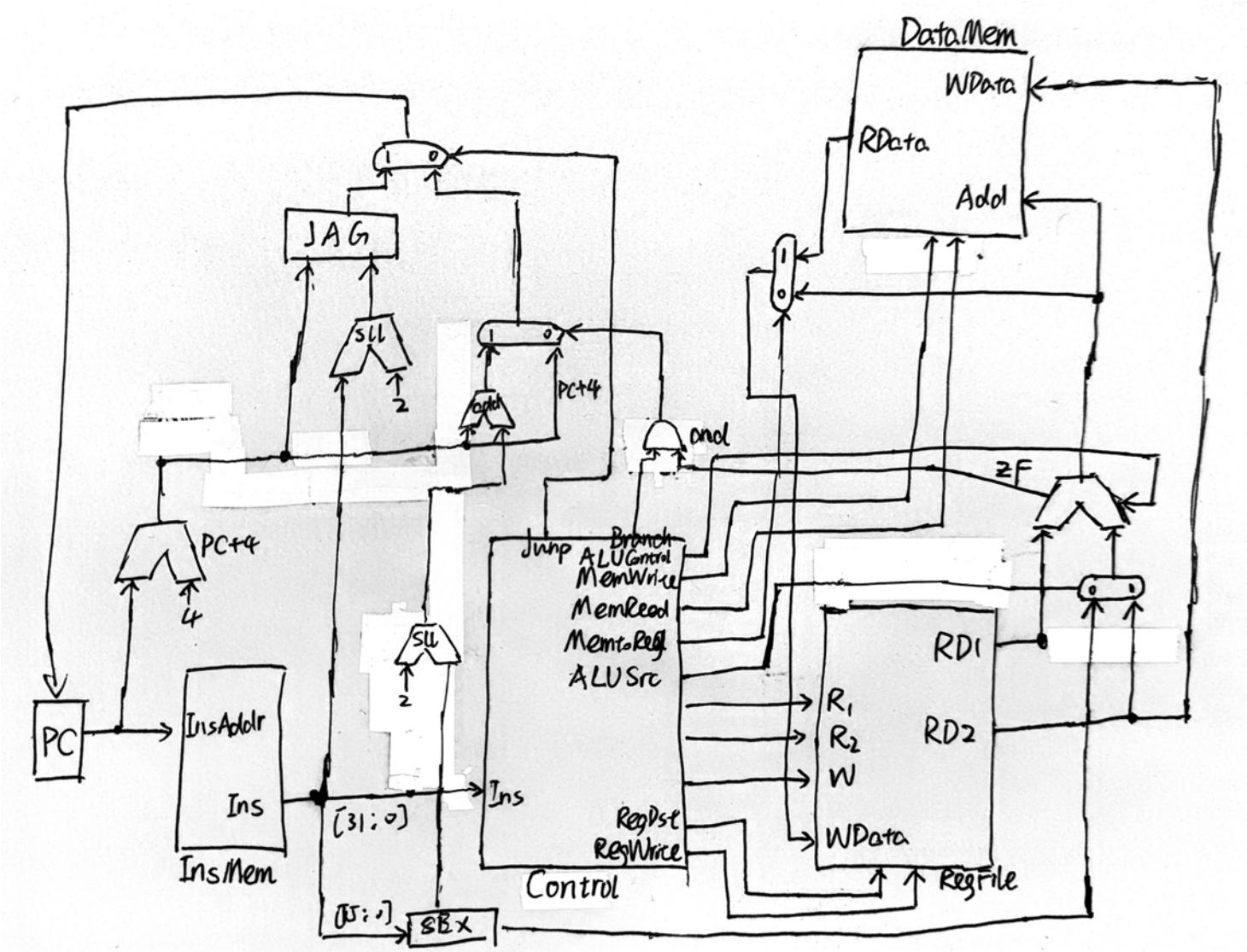
由于实验要求从外部输入数据、并显示数据，上述的模块都不具备对外界联系的功能。因此在这里为CPU提供一个输入输出接口模块，这个模块拥有修改与读取内存(实际是指数据存储器)的权限，并且仅对内存生效。这一部分的代码因为要与负责处理外界信息的模块部分联动，所以放到之后展示。

11.CPU部分TOP文件 `SingleCPU`

CPU部分的TOP文件将各模块串联起来。由于代码非常冗长，这里只展示 `wire` 变量定义部分：

```
module SingleCPU(
    input CLK,
    ......
    );

    wire [31 : 0] PCIn_wire;
    wire [31 : 0] PCOri_wire;
    wire [31 : 0] PCPlus4_wire;
    wire [31 : 0] Instruct_wire;
    wire [31 : 0] ADD_wire;
    wire [31 : 0] SL2toJAG_wire;
    wire [31 : 0] SL2toADD_wire;
    wire [31 : 0] JAG_wire;
    wire [31 : 0] SignEx_wire;
    wire [31 : 0] ReadData1_wire;
    wire [31 : 0] ReadData2_wire;
    wire [31 : 0] RegWriteData_wire;
    wire [31 : 0] ALU_Mux_wire;
    wire [31 : 0] Branch_Mux_wire;
    wire [31 : 0] ALUGeneral_wire;
    wire [31 : 0] DataMemRead_wire;
    wire [31 : 0] MemWriteData_wire;
    wire [31 : 0] MemAdd_wire;
    wire [31 : 0] AddtoMem_wire;
    wire [31 : 0] NumtoMem_wire;

    wire [3 : 0] ALUControl_wire;

    wire [4 : 0] RegRead1_wire;
    wire [4 : 0] RegRead2_wire;
    wire [4 : 0] RegWrite_wire;

    wire ALUSrc_wire;
    wire RegDst_wire;
    wire RegWrite_en_wire;
    wire MemRead_en_wire;
    wire MemWrite_en_wire;
    wire IOPMemRead_en_wire;
    wire IOPMemWrite_en_wire;
    wire MemtoReg_en_wire;
    wire Jump_en_wire;
    wire Branch_en_wire;
    wire Halt_en_wire;
    wire ZF_wire;

    ......(instantiations)

endmodule
```

Vivado生成的 `Elaborated Design` 非常庞大巨细，这里不好展示。一幅手绘的CPU连接图如下(不含 `IOPort`):



在这里可以看到，所有的四个多选器都是二路32位多选器，除与门外所有算数逻辑操作都由 ALU 完成，以至共有5个 ALU ，极大地提高了模块的复用性。

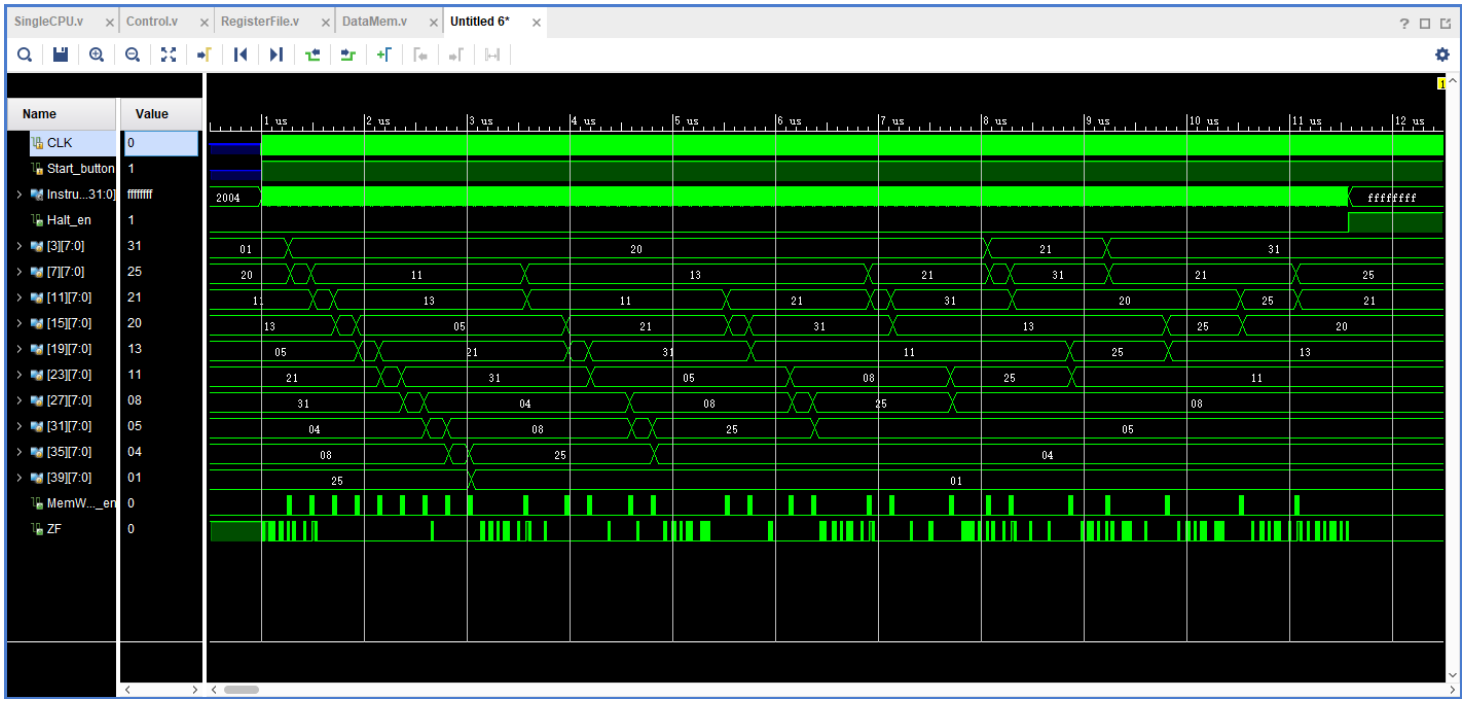
12.运行验证

(1)整体运行验证

首先将整个冒泡排序程序运行一遍。如果结果正确，基本可以断定设计是没有问题的。为此，在数据存储器 DataMem 中添加下述初始语句块：

```
module DataMem(
    ......
    );

    ......
    integer i;
    initial begin
        for (i = 0; i <= 63; i = i + 1) Data[i] <= 0;
        Data[3] <= 8'h01;
        Data[7] <= 8'h20;
        Data[11] <= 8'h11;
        Data[15] <= 8'h13;
        Data[19] <= 8'h05;
        Data[23] <= 8'h21;
        Data[27] <= 8'h31;
        Data[31] <= 8'h04;
        Data[35] <= 8'h08;
        Data[39] <= 8'h25;
    end
    ......

endmodule
```

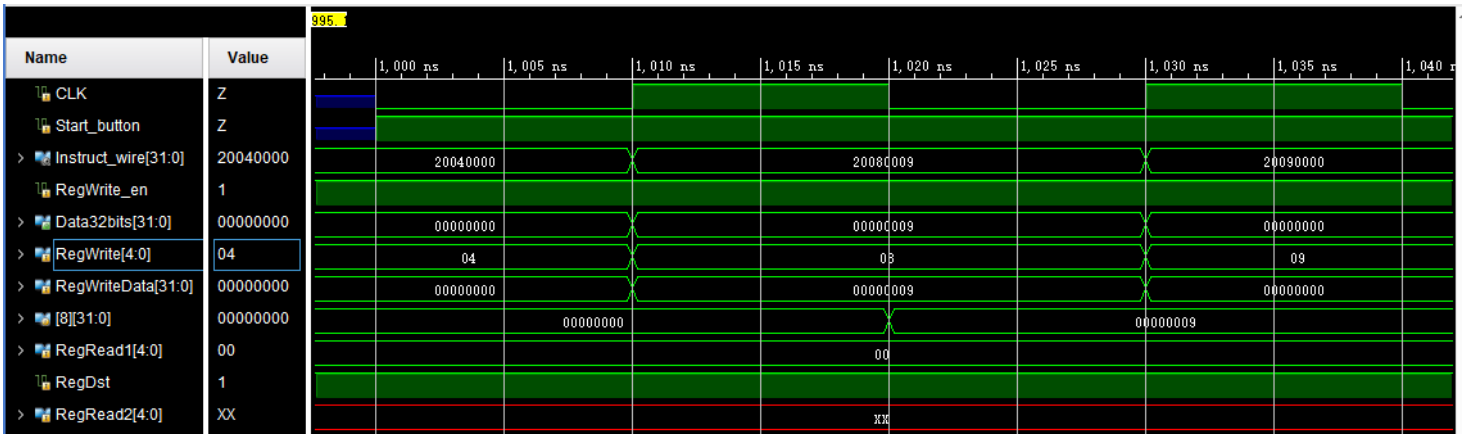
仿真运行后查看整体波形图：



可以看到，内存中的十个乱序数，在经过排序后由大到小排列。并且，在 `halt_en` 为 `1` ，即运行到 `halt` 指令后， `PC` 中输出的 `Instruct_wire` 被锁定在了这条指令上，说明 `halt` 指令是被正确设置了的。

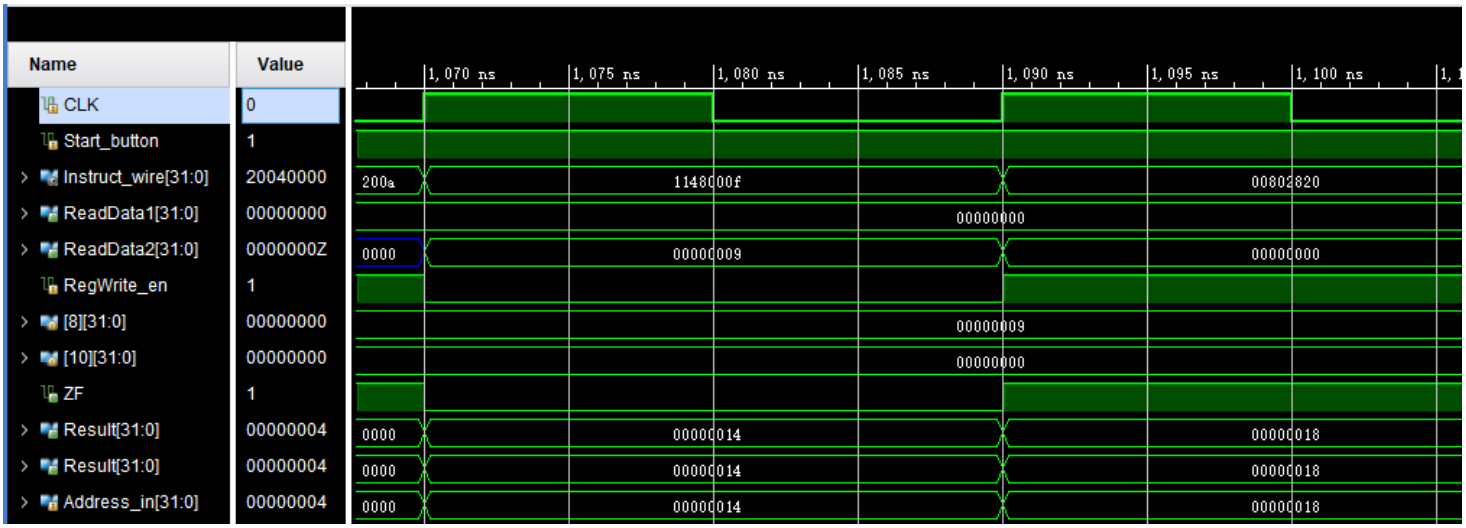
(2)具体运行验证

在这段程序中，前四条指令都是 `addi` 指令。不妨查看第二条指令 `addi $t0, $zero, 9` ，机器码为 `20080009` 。波形如下：



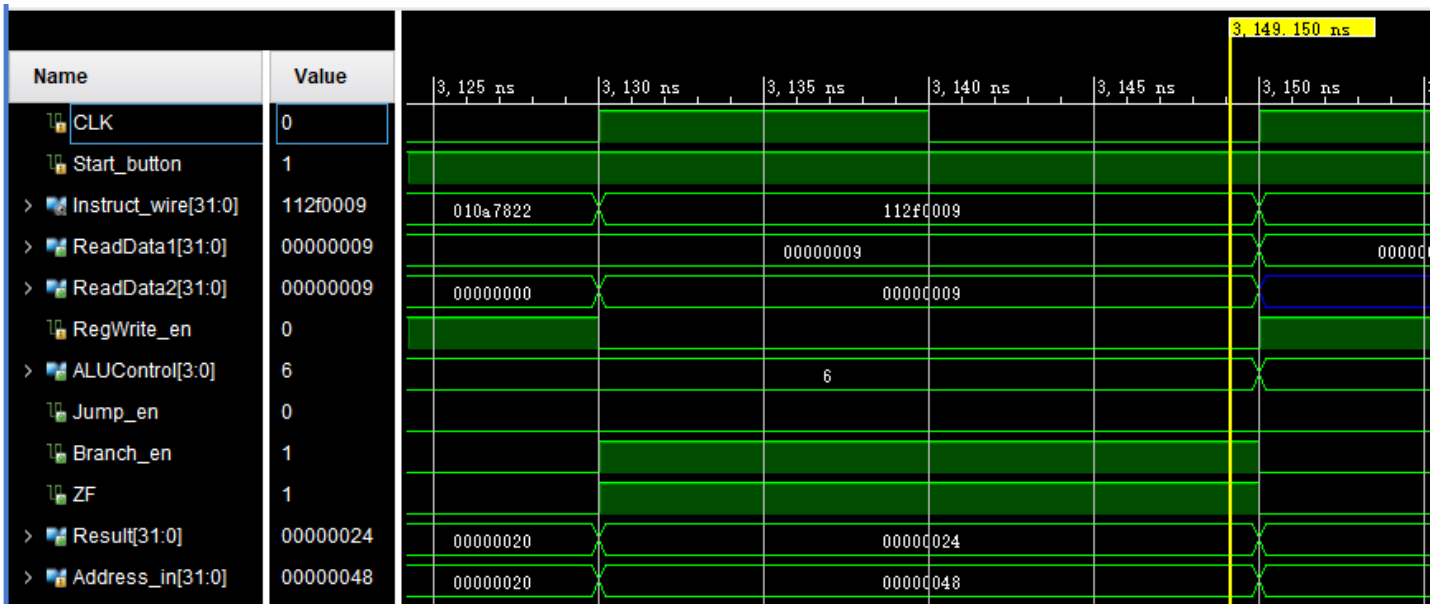
在第二个指令周期中，寄存器写使能信号恒为 `1` 。同时，立即数符号扩展也是从指令低四位 `0009` 扩展而来。 `RegWrite` 指明要写入的寄存器编号，这里确实是八号寄存器 `$t0` 。写入的数据应是 `0+9=9` ， 这里 `RegWriteData` 也确实如此。同时，由于是 `I-type` 指令， `RegDst` 为 `1` ， 限制第二个寄存器的读取，这在波形图上也是切实反映的。

第五条指令是 `beq $t2, $t0, done_outer_loop` ， 机器码 `1148000f` 。在第一次判断中是不等的情况，波形如下：



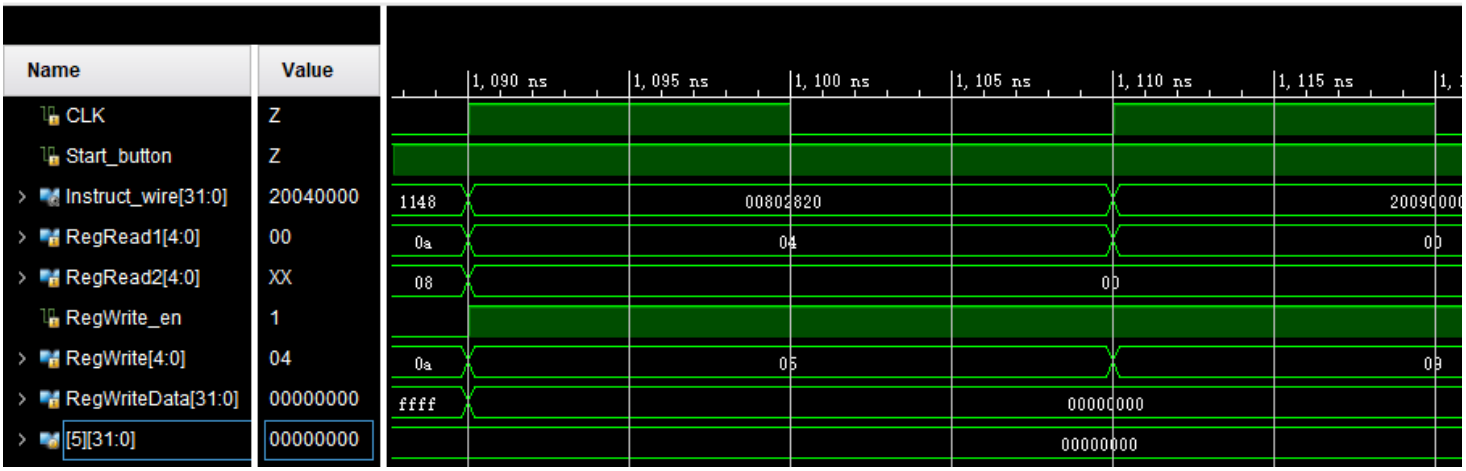
在这里，从寄存器中读出的两个数据分别是 `0` 和 `9` ， 这确实来源于下面看到的八号寄存器与十号寄存器。写使能信号为 `0` 。由于这两个数不相等， `ALU` 的 `ZF` 信号为 `0` ， 这与 `Branch_en` 信号与操作后为 `0` 。这样，最后 `PC` 的下一地址输入 `Address_in` 就是原来的 `PC+4` ， 即 `14` 。

在之后的运行过程中，截取了一条判断相等则分支的情况，波形如下：



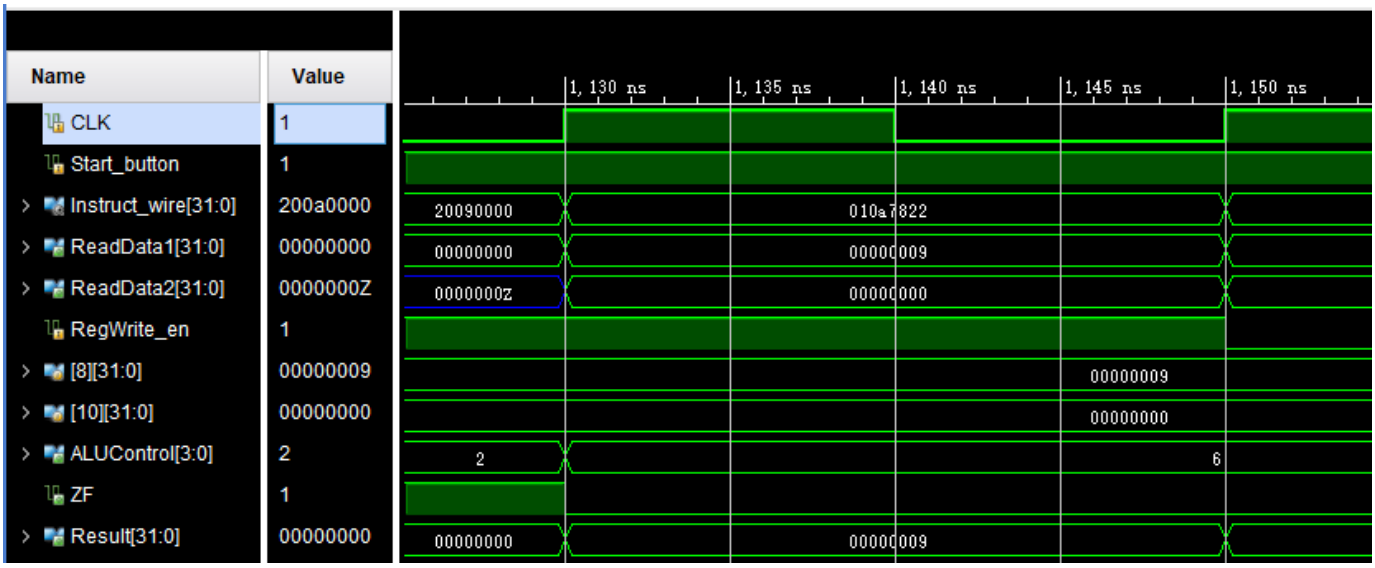
这条指令是 `beq $t1, $t7, done_inner_loop` ，机器码 `112f0009` 。在这个指令中， `Branch_en` `ZF` 信号同时为 `1` ，说明分支发生。 `Jump_en` 为 `0` ，说明使用 `Branch_Mux` 产生的结果。在这里， `Result` 的结果是 `PC+4` ，可以看到最终进入 `PC` 的地址不是 `PC+4` 而是分支地址。

第六条指令是 `add $a1, $a0, $zero` ，机器码 `00802820` 。波形如下：

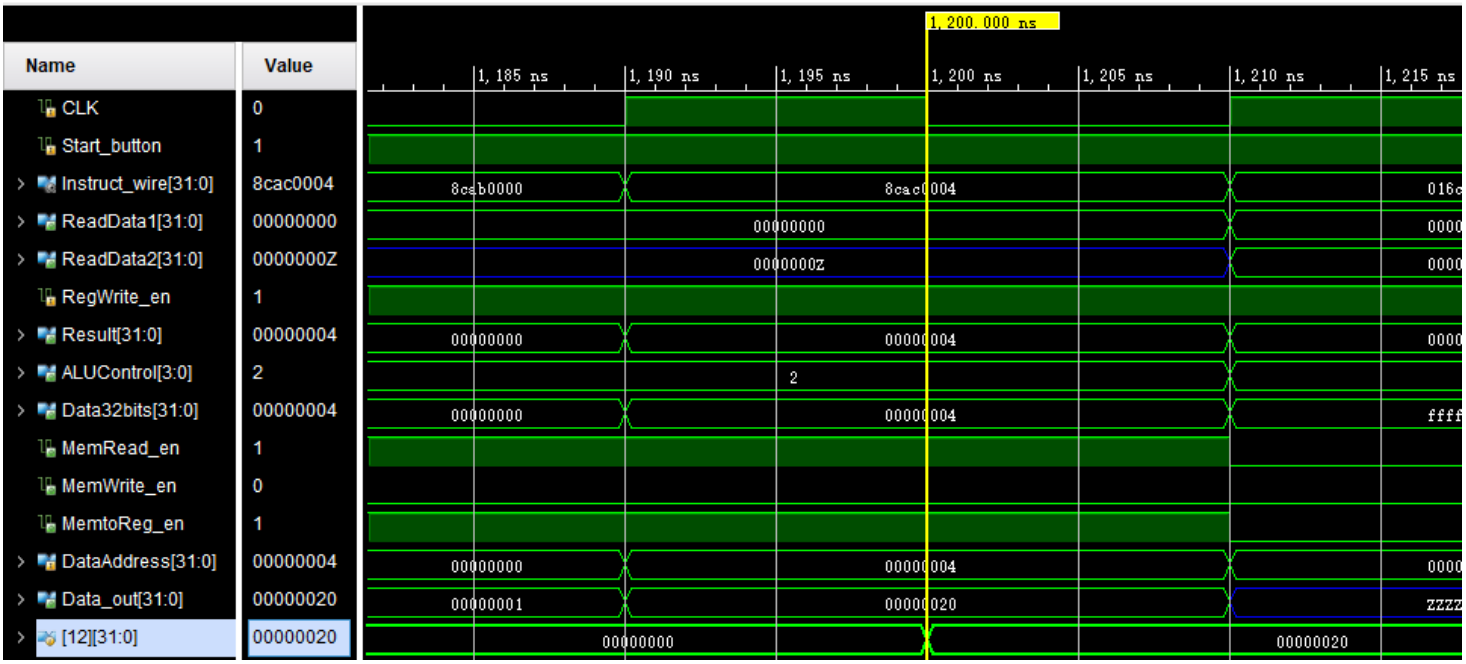


在这里，指定了五号寄存器 `a1` 为目标寄存器。在这个实验中， `$a0` 恒指定存储数据数组的基地址，亦即 `0` 。因此在这条指令中 `$a1` 得到的值也是 `0` ，显得没有变化。其它信号在以上已有过解释，不多赘述。

指令 `sub $t7, $t0, $t2` ，机器码 `010a7822` 。这条指令在整体上与 `add` 指令的唯一不同，就是 `ALUControl` 不同。波形如下：

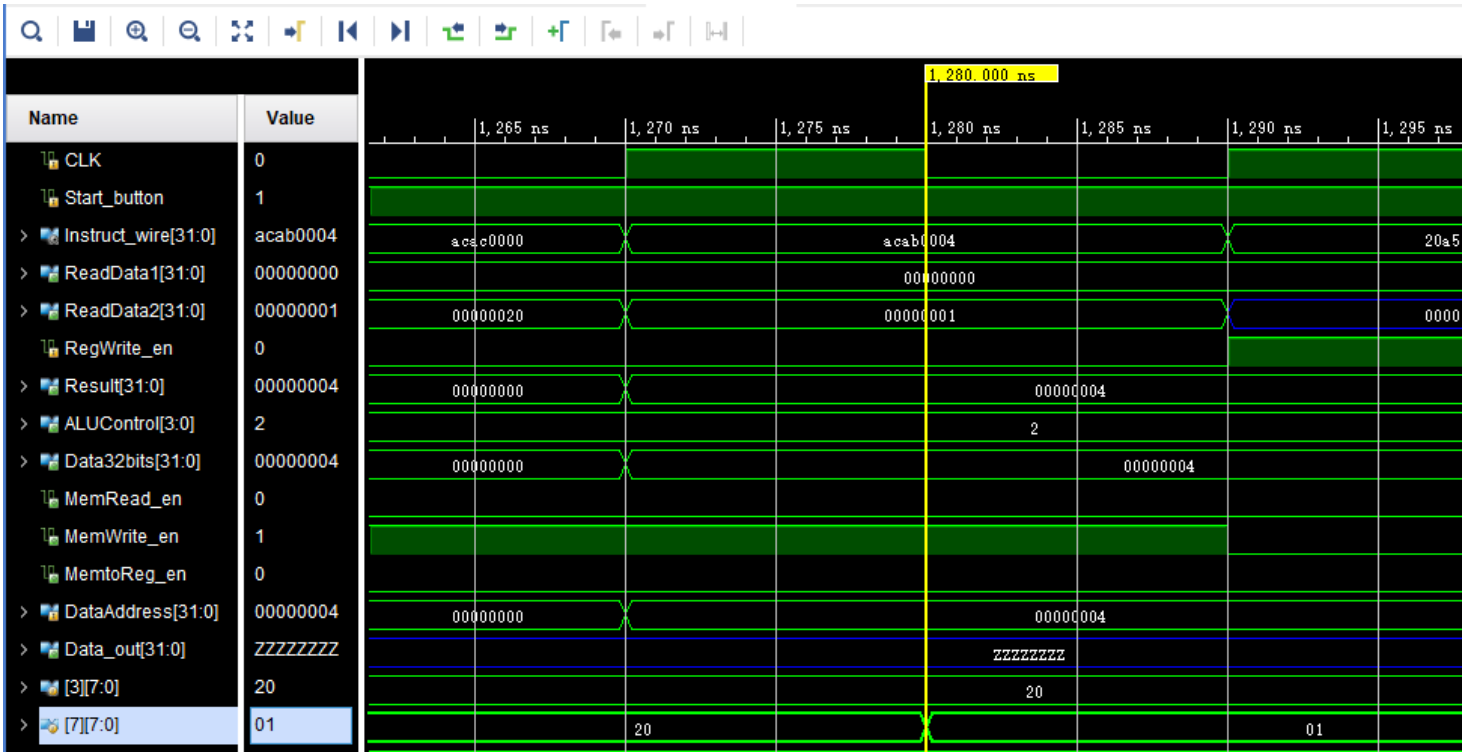


接下来两条分别是取数与存数指令。指令 `lw $t4, 4($a1)`，机器码 `8cac0004`，波形如下：



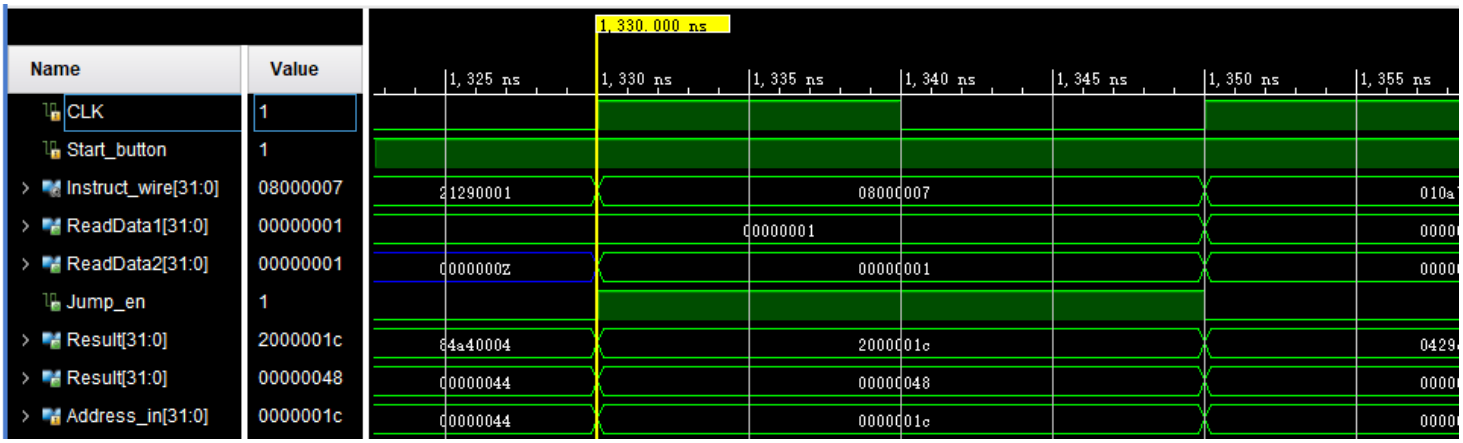
此时，限制第二个源寄存器的读取，并允许写入寄存器。 `Result` 是经过 `ALU` 计算的 `$a1+(SignExtend)imm` 的值，为 `4`，与 `DataAddress` 是相等的。 `(SignExtend)imm` 显示在 `Data32bits` 中。内存读使能为 `1`、相应地写使能为 `0`，从中读出的数是 `20`，可以看到在时钟下降沿写入了寄存器中。

指令 `sw $t3, 4($a1)`，机器码 `acab0004`，波形如下：



与上类似，但寄存器不被写入，内存写使能为 `1` 、相应地读使能为 `0` ，内存不送出数据。可以看到在时钟下降沿时，寄存器中读出的数字被送入指定内存地址中。

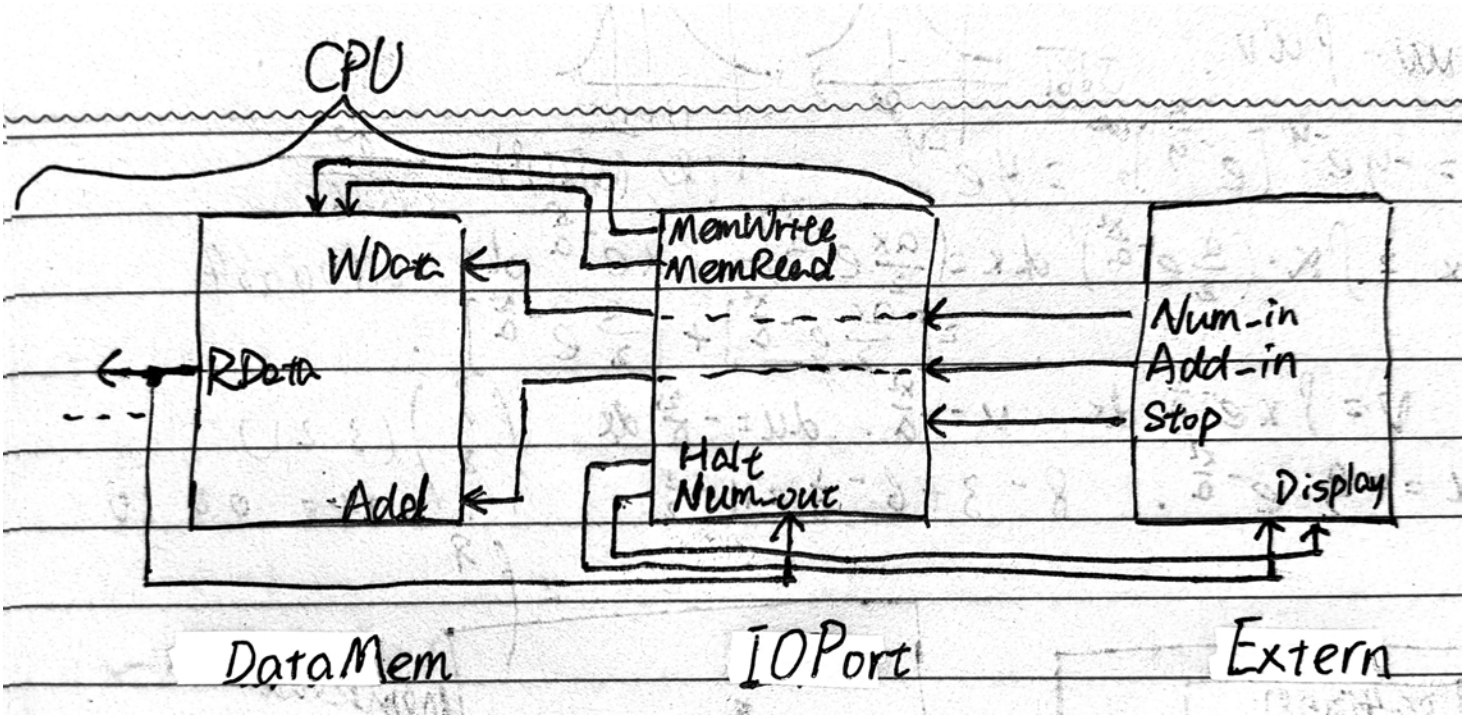
示例无条件跳转指令 `j inner_loop` ，机器码 `08000007` ，波形图如下：



在这个指令中， `Jump_en` 为 `1` ，说明直接选择 `JumpAddGenerator` 生成的地址。第一个 `Result` 是原指令低26位左移2位的结果，在这里实际上为了复用 `ALU` 是直接将原指令左移2位。第二个 `Result` 是 `PC+4` 。可以看到送入 `PC` 的地址就是第一个 `Result` 低28位与第二个 `Result` 高4位的拼接。

四、实验内容(外部输入输出部分)

在这里，我将*外部整体*命名为 `Extern` 。外部的输入输出不能和CPU内的核心组件产生直接联系，因此 `Extern` 必须通过 `IOPort` 来修改内存。这里我大致手绘了一幅 `Extern` 与CPU间的数据通路：



其中， Extern 中的 Stop 信号用来通知 IOPort 停止输入。 IOPort 中的 Halt 用于通知 Extern 排序完成。

以下部分是 IOPort 与 Extern 中的具体模块。

1.时钟分频 CLK_div

为了七段码显示，刷新速度不能太快。因此引入时钟分频模块，代码如下：

```
module CLK_div #(parameter N = 99999)(
    input CLK_in,
    output CLK_out
    );

    reg [31 : 0] counter = 0;
    reg out = 0;

    always@(posedge CLK_in) begin
        if (counter == N - 1) counter <= 0;
        else counter <= counter + 1;
    end

    always@(posedge CLK_in) begin
        if (counter == N - 1) out <= !out;
    end

    assign CLK_out = out;

endmodule
```

2.显示模块 Display

这一部分直接复用之前写过的4位数加减，代码如下：

```
module Display(
    input CLK_in,
    input [15 : 0] Data,
    output reg [3 : 0] segment,
    output reg [6 : 0] position
    );

    reg [1 : 0] temp;
    reg [3 : 0] x;

    always@(posedge CLK_in) begin
        case (temp)
            0: begin x = Data[15 : 12]; segment = 4'b0111; end
            1: begin x = Data[11 : 8]; segment = 4'b1011; end
            2: begin x = Data[7 : 4]; segment = 4'b1101; end
            3: begin x = Data[3 : 0]; segment = 4'b1110; end
        endcase
        temp = temp + 1;
        if (temp % 4 == 0) temp = 0;
    end

    always@(*) begin
        case (x)
            4'h0: position = 7'b0000001;
            4'h1: position = 7'b1001111;
            4'h2: position = 7'b0010010;
            4'h3: position = 7'b0000110;
            4'h4: position = 7'b1001100;
            4'h5: position = 7'b0100100;
            4'h6: position = 7'b0100000;
            4'h7: position = 7'b0001111;
            4'h8: position = 7'b0000000;
            4'h9: position = 7'b0000100;
            4'hA: position = 7'b0001000;
            4'hB: position = 7'b0010000;
            4'hC: position = 7'b0110001;
            4'hD: position = 7'b0010001;
            4'hE: position = 7'b0110000;
            4'hF: position = 7'b0111000;
        endcase
    end

endmodule
```

为了让B和8，D和0区分开来，特意让这两个字母的第三管脚不亮。

3.地址生成模块 `AddGenerator`

这个地址生成器与CPU中的*跳转地址生成单元*不同，本模块生成的地址会指向内存。代码如下：

```
module AddGenerator(
    input CLK_in,
    input Next,
    input Pre,
    input Verify,
    input Reset,
    input Stop,
    output reg [31 : 0] Address,
    output [3 : 0] Count
    );

    (* KEEP = "TRUE" *)  reg [31 : 0] counter = 32'b0;

    always@(posedge CLK_in) begin
        if (Next == 1) begin
            counter = counter + 4;
            if (counter == 40) counter = 0;
            if (Stop == 1) Address = counter;
        end
        else if (Pre == 1) begin
            if (counter == 0) counter = 40;
            counter = counter - 4;
            if (Stop == 1) Address = counter;
        end
        else if (Reset == 1) begin
            counter = 0;
            Address = 0;
        end
        else if (Stop == 0 & Verify == 1) Address = counter;
    end

    assign Count = counter / 4;

endmodule
```

这个模块的编写与调试拖慢了整个实验的进度。具体可见最后的实验总结部分。

4.CPU部分输入输出接口 IOPort

设置 IOPort 的原因如上所述，代码如下：

```
module IOPort(
    input [31 : 0] AddfromEx,
    output [31 : 0] AddtoMem,
    input [31 : 0] NumfromEx,
    output [31 : 0] NumtoMem,
    input [31 : 0] NumfromMem,
    output [31 : 0] NumtoEx,

    input Stop_en,
    input Verify_en,
    output MemWrite_en,
    output MemRead_en
    );

    assign MemWrite_en = (Stop_en == 0 & Verify_en == 1) ? 1 : 0;
    assign MemRead_en = (Stop_en == 0) ? 0 : 1;

    assign NumtoMem = (Stop_en == 0 & Verify_en == 1) ? NumfromEx : 32'bz;
    assign NumtoEx = (Stop_en == 1) ? NumfromMem : 32'bz;
    assign AddtoMem = AddfromEx;

endmodule
```

这个模块是单纯的组合逻辑，只需要简单地复制赋值就行，相当于一个线路汇总站。其中， `Stop_en` 用来控制数据的输送方向。

5.按键消抖 `ButtonDebounce`

由于机械按钮的机械属性，必须保证按一次按钮只触发一次响应。因此引入按键消抖模块，代码如下：

```
module ButtonDebounce(
    input CLK_in,
    input Button_in,
    output Button_out
    );

    reg [2 : 0] Button = 0;

    always@(posedge CLK_in) begin
        Button[0] <= Button_in;
        Button[1] <= Button[0];
        Button[2] <= Button[1];
    end

    assign Button_out = (Button[2] & Button[1] & Button[0]) | (~Button[2] & Button[1] & Button[0]);

endmodule
```

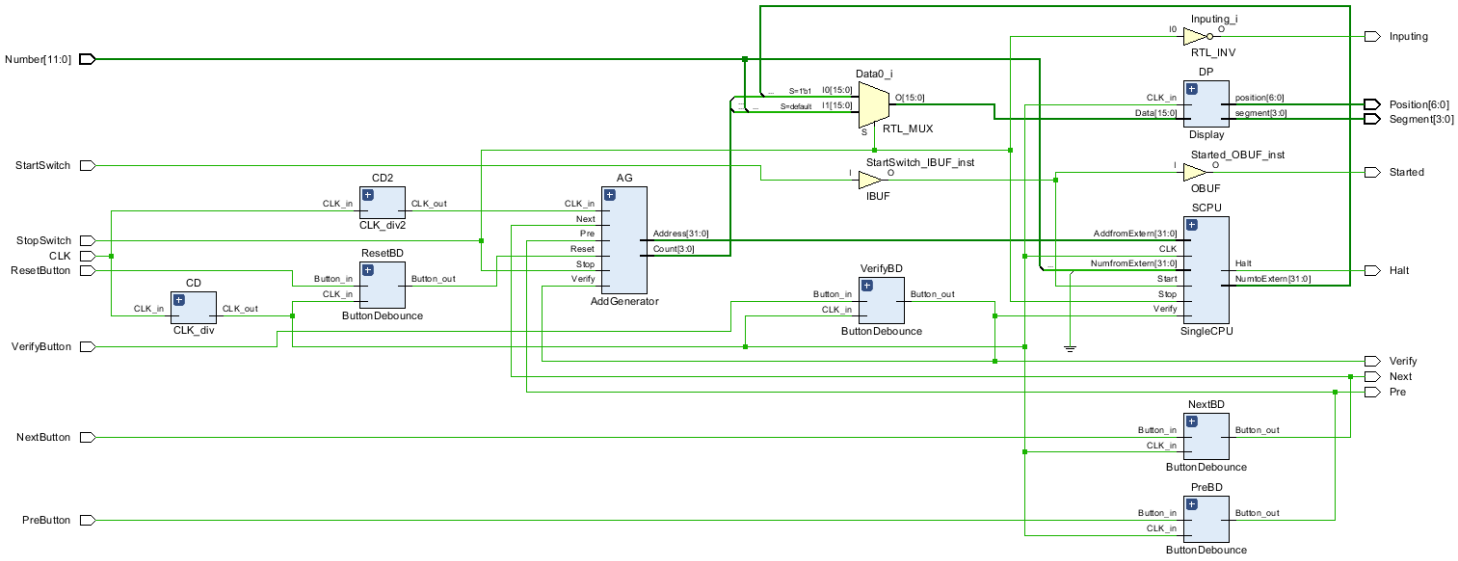
6.外部顶层文件 `Extern`

在这个模块中，直接将单周期CPU封装进来。相当于直接在CPU上加装一个输入与显示器件。代码如下：

```
module External(
    input CLK,
    input NextButton,
    input PreButton,
    input ResetButton,
    input VerifyButton,
    input StartSwitch,
    input StopSwitch,
    input [11 : 0] Number,
    output Halt,
    output Inputing,
    output Started,
    output Next,
    output Pre,
    output Verify,
    output [3 : 0] Segment,
    output [6 : 0] Position
    );

    wire CLK_div_wire;
    wire CLK_div2_wire;
    wire NextBD_wire;
    wire PreBD_wire;
    wire ResetBD_wire;
    wire VerifyBD_wire;
    (* KEEP = "TRUE" *) wire [3 : 0] Count_wire;
    (* KEEP = "TRUE" *) wire [31 : 0] Add_wire;
    (* KEEP = "TRUE" *) wire [31 : 0] NumfromCPU_wire;

    CLK_div CD (
        .CLK_in( CLK ),
        .CLK_out( CLK_div_wire )
    );

    CLK_div2 CD2 (
        .CLK_in( CLK ),
        .CLK_out( CLK_div2_wire )
    );

    (* DONT_TOUCH = "1" *) ButtonDebounce NextBD (
        .CLK_in( CLK_div_wire ),
        .Button_in( NextButton ),
        .Button_out( NextBD_wire )
    );

    (* DONT_TOUCH = "1" *) ButtonDebounce PreBD (
        .CLK_in( CLK_div_wire ),
        .Button_in( PreButton ),
        .Button_out( PreBD_wire )
    );

    (* DONT_TOUCH = "1" *) ButtonDebounce ResetBD (
```

```
        .CLK_in( CLK_div_wire ),
        .Button_in( ResetButton ),
        .Button_out( ResetBD_wire )
    );

    (* DONT_TOUCH = "1" *) ButtonDebounce VerifyBD (
        .CLK_in( CLK_div_wire ),
        .Button_in( VerifyButton ),
        .Button_out( VerifyBD_wire )
    );

    (* DONT_TOUCH = "1" *) Display DP (
        .CLK_in( CLK_div_wire ),
        .Data( (StopSwitch == 1) ? { Count_wire, NumfromCPU_wire[11 : 0] } : { Count_wire, Number } ),
        .segment( Segment ),
        .position( Position )
    );

    (* DONT_TOUCH = "1" *) AddGenerator AG (
        .CLK_in( CLK_div2_wire ),
        .Next( NextBD_wire ),
        .Pre( PreBD_wire ),
        .Verify( VerifyBD_wire ),
        .Reset( ResetBD_wire ),
        .Stop( StopSwitch ),
        .Address( Add_wire ),
        .Count( Count_wire )
    );

    (* DONT_TOUCH = "1" *) SingleCPU SCPU (
        .CLK( CLK_div_wire ),
        .Start( StartSwitch ),
        .Stop( StopSwitch ),
        .Verify( VerifyBD_wire ),
        .AddfromExtern( Add_wire ),
        .NumfromExtern( { 20'h00000, Number } ),
        .Halt( Halt ),
        .NumtoExtern( NumfromCPU_wire )
    );

    assign Inputing = !StopSwitch;
    assign Started = StartSwitch;
    assign Next = NextBD_wire;
    assign Pre = PreBD_wire;
    assign Verify = VerifyBD_wire;

endmodule
```

到此为止，带输入与显示的MIPS CPU已经完成。下图是Vivado生成的 `Elaborated Design`：

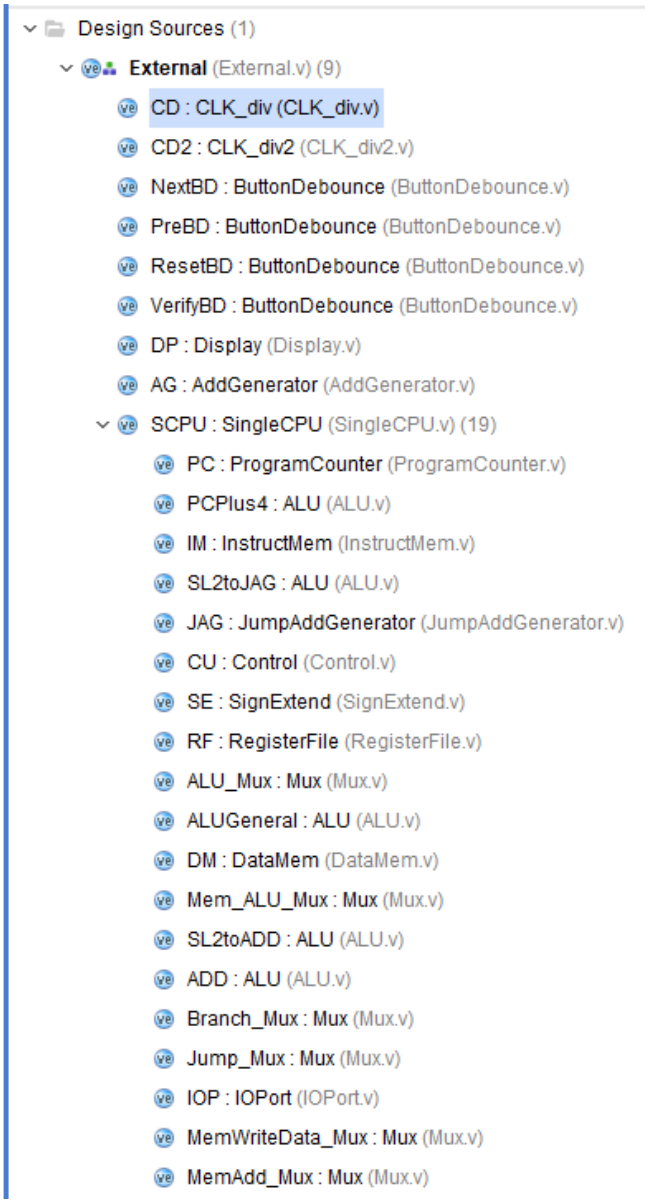


在这里值得注意的是，虽然对于CPU来说一个数字是32位，但由于开发板上只有16个拨板，因此能够一次性输入的数字只有16位；另一方面，我对程序的运行状态做出一些约束，占用了两个拨板开关，因此本实验中允许输入的数的范围被我限制在了$0 \sim 2^{11} - 1$，即只使用12位二进制数。但实际这个范围已经非常大了：单次显示只能显示$4 \times 4 = 16$位二进制数，因此上述范围是完全够用的。

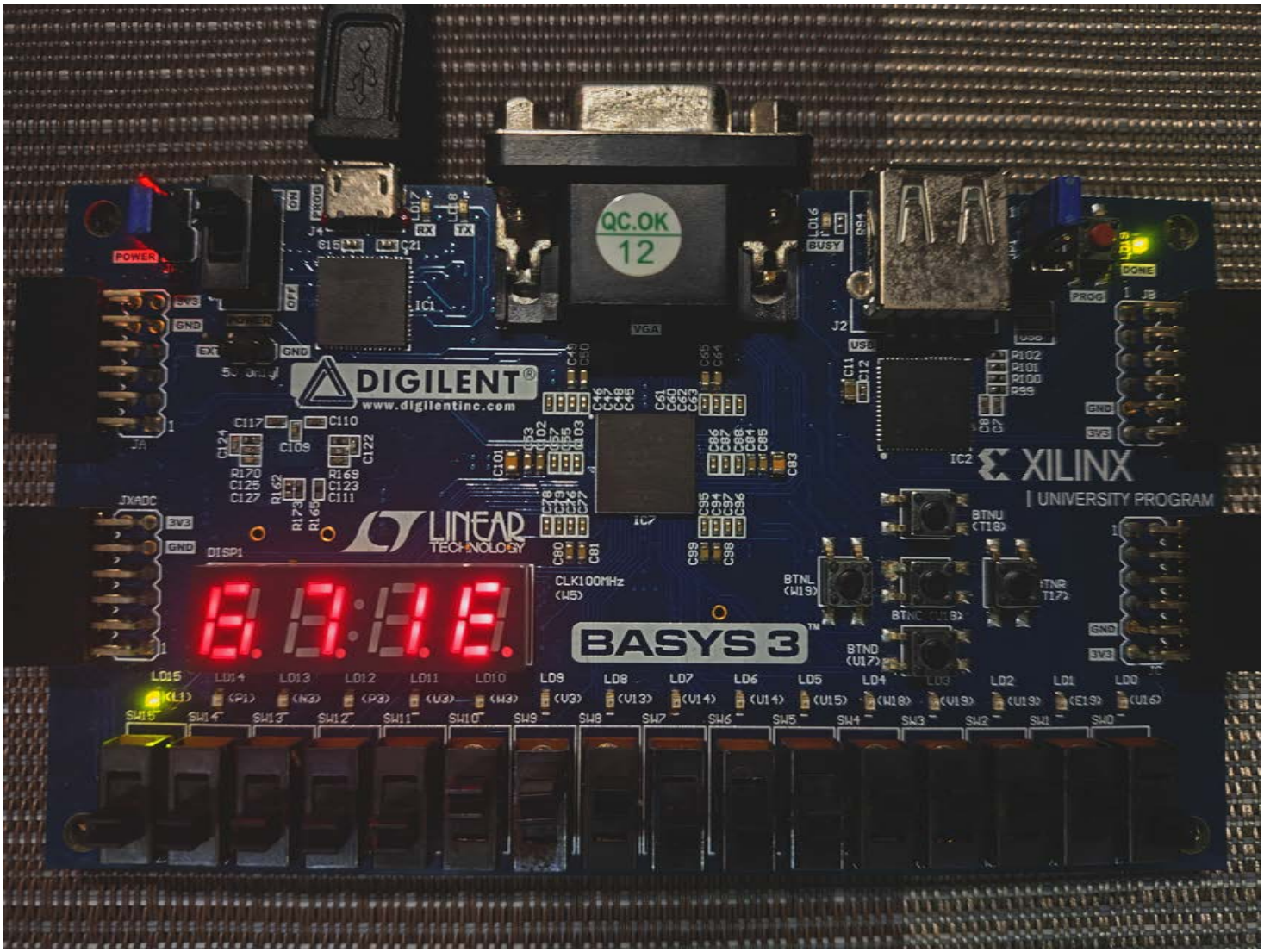
对于剩下的高四位二进制数——对应一位十六进制数——我让它表示目前正在显示的三位十六进制数是内存中的第几个数。这样可以算是物尽其用了。

五、实验结果

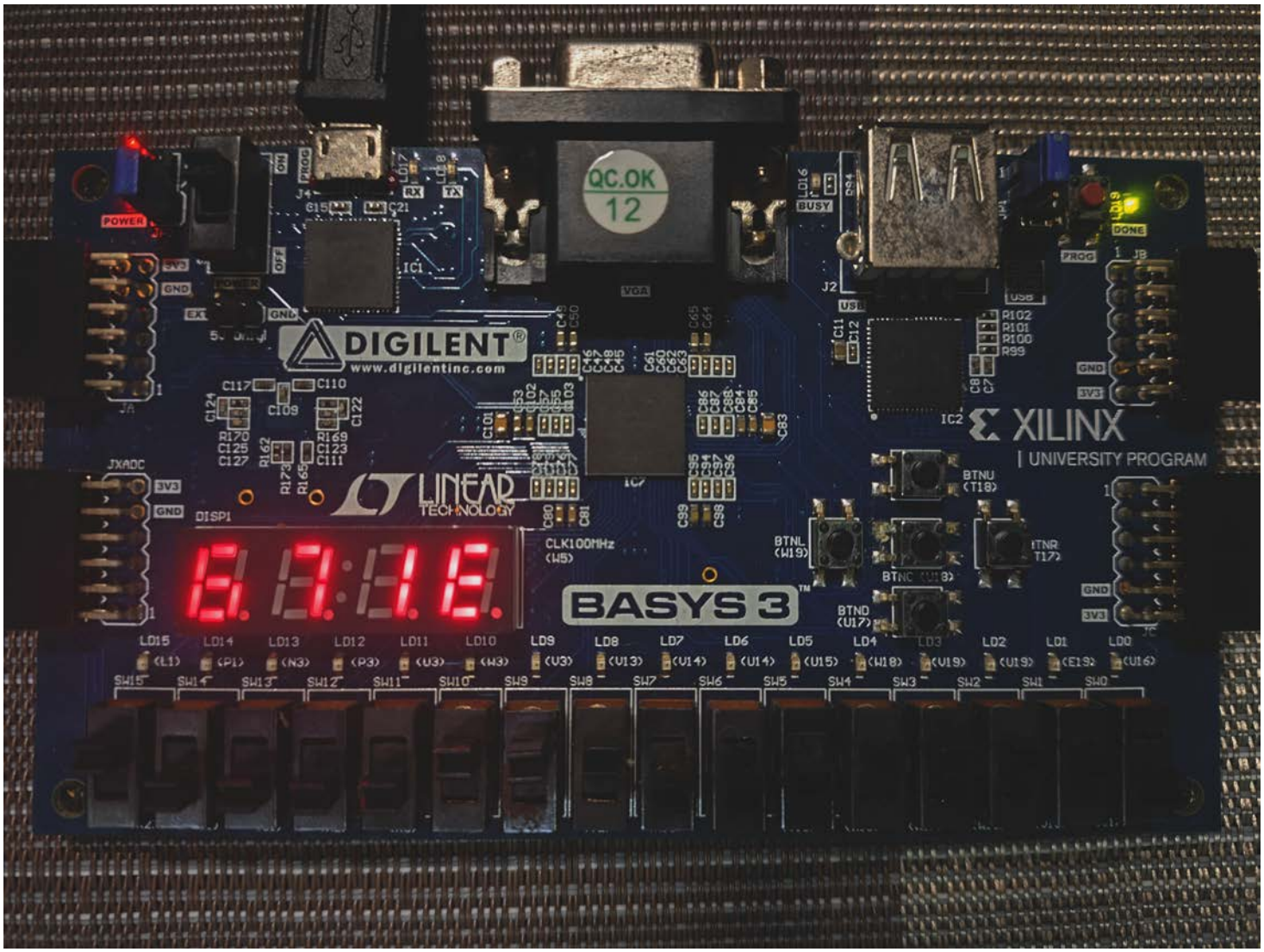
最终的整个实验文件结构如图：



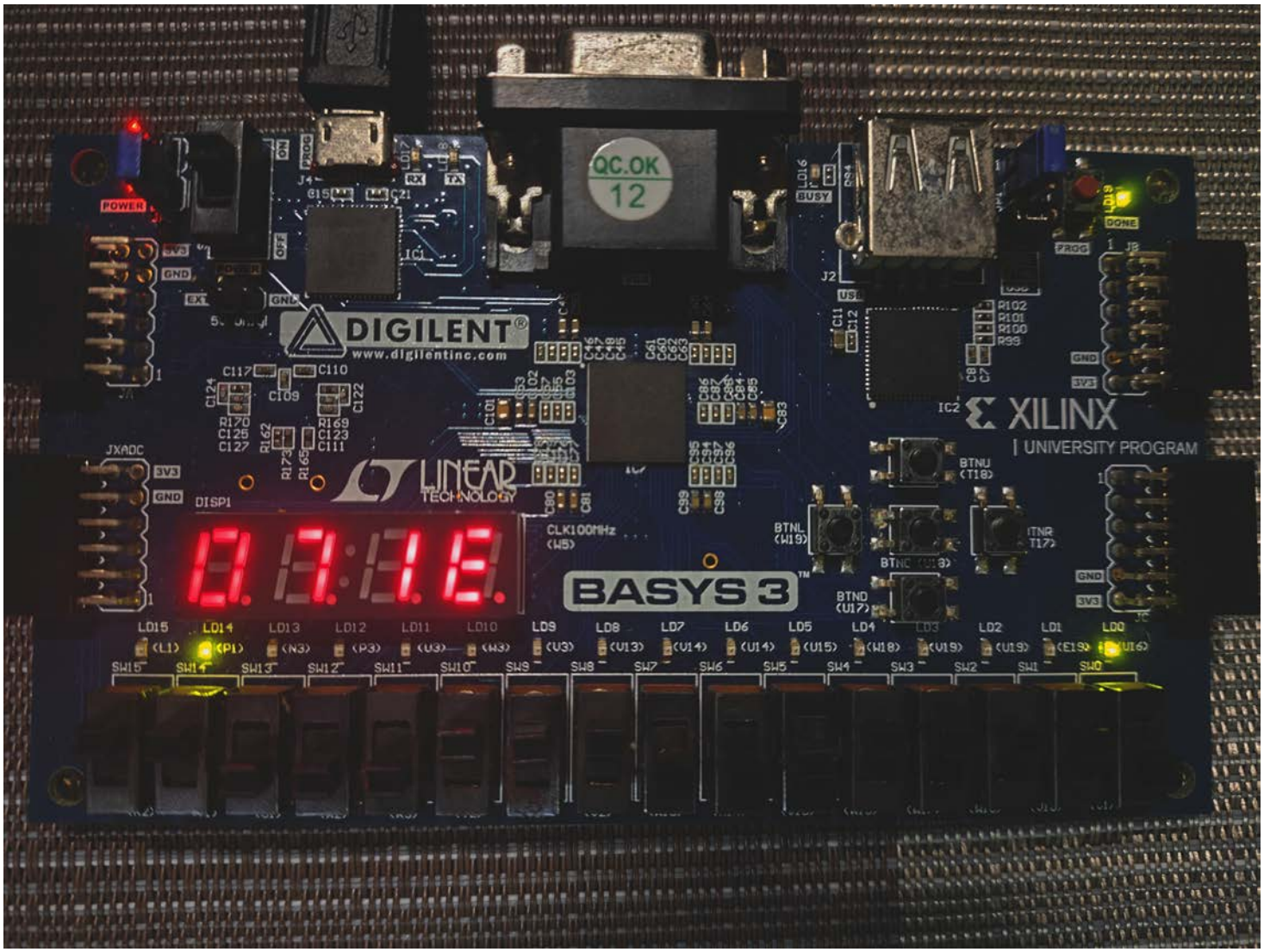
由于文本无法完全展现成果，这里仅放一些示例图片：



此时，最左的灯亮起，表示正在输入阶段，对内存中第$6+1=7$个机器字赋值 `71E` ，这是输入的数中最大的数。



最左边拨板调至 1， 表示结束输入。此时直接从内存读取数据，可以看到内存中第七个机器字是 71E， 说明已经成功写入内存中。



将第二个拨板调至 `1` ，左边第二个灯亮起，说明进入排序阶段。一段时间后，最左边的灯亮起，表示排序完成。此时，最大的数即 `71E` 处于第一个位置，对于机器而言是 `0` 位置。这表明了排序是正常执行完成了的。

六、实验总结

1.变量命名

在这样一个非常浩大的工程中，我深刻意识到变量命名的重要性。好的变量命名必须能够清晰反映这个变量的性质。在这次实验中，我对于模块都使用驼峰命名法，并且直接使用原生CPU的命名。其中的变量也在原生CPU的基础上使用驼峰命名法，但对于表示“是非”信号的变量，都在其后添加 `_en` ，即 `enable` 表示。

对于模块中存在的输入或输出变量，在可能的情况下会添加 `_in` 或 `_out` 。

在顶层文件中，对于一切 `wire` 型用来连接模块的变量，都在其对应的模块内变量命的基础上，直接在后面添加 `_wire` 表示。

在模块实例化时，具有特定性质的模块命名尽可能简短，比如 `InstructMem` 实例化为 `IM` 。而具有通用性质的模块命名在尽可能简短的条件下尽可能具体，比如四个多选器中有一个实例化名称为 `Mem_ALU_Mux` 。

2.仿真与实际效果不一致

在行为仿真的时候，我设想中的操作全部都完美实现。但是将程序烧写到开发板上后，我发现无论如何操作，开发板都不会对我发出的信号做出回应。具体来说，就是 `Verify` 确认输入按钮不会将输入送到内存； `Next` 下一个地址按钮不会改变地址； `StartSwith` 调到 `1` 后本来应该开始冒泡排序、并在至多1秒内完成、使得 `Halt` 灯亮起，但实际似乎根本就没运行冒泡排序。总结就是——仿真结果与实际效果不一致。

根据网络检索到的信息，首先猜测是有些信号在综合时被优化掉了，这就需要从 `warning` 中寻找报警原因。其中有一些报警如下：

```
[Synth 8-3332] Sequential element (...) is unused and will be removed from modul ...
```

这是在说把一些寄存器优化掉了！在此之前秉持从编写高级语言代码中继承来的习惯， `warning` 并不重要；但由于现在需要直接与硬件打交道，有必要将 `warning` 减少到最低。基于上述的不希望的优化，在一些变量定义前加 `(* KEEP = "TRUE" *)` ，将所有被自动优化掉的变量都禁止优化。

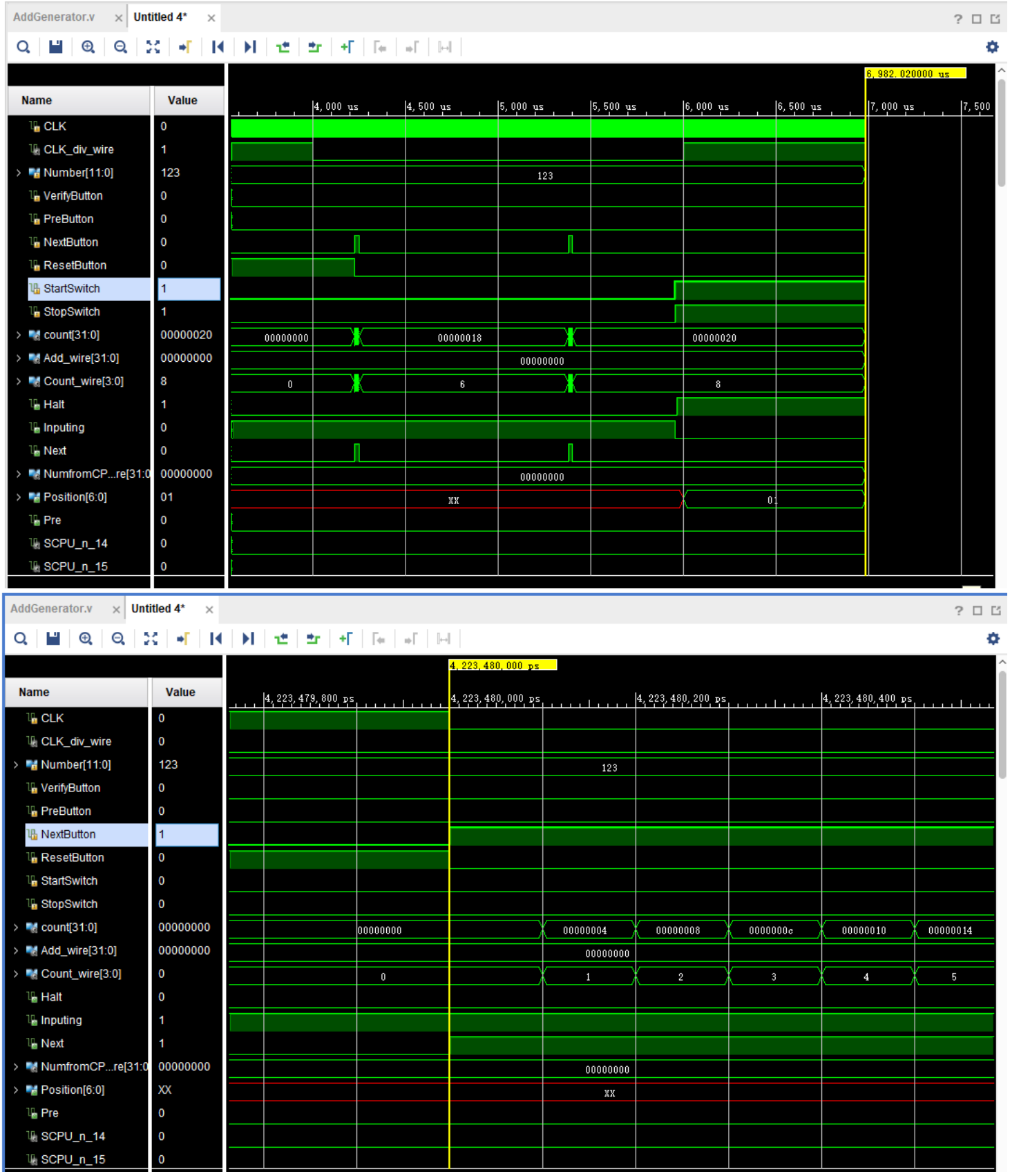
然而这样依旧不行！我尝试运行了综合后功能仿真，发现从此处就已经开始“断连”的效果，也就是外部的按钮不会影响内部任何内容！查看综合后功能仿真的 `Scope` ，发现有些模块直接被优化掉了，为此我在所有模块例化前又加了限制： `(* DONT_TOUCH = "1" *)`

这一次所有预想中的模块都被保留下来了，但是通过一次非常耗费时间的综合后功能仿真后，我发现 `Next` 信号还是无法触发 `count` 的自增！在这之后，我对 `AddressGenerator` 模块进行了无数次微调，中间接二连三出现非常多问题，比较典型的就是迭代次数过多导致仿真不能继续：

```
Iteration limit 10000 is reached. Possible zero delay oscillation detected where simulation time can not advance......
```

这个通过修改 `always` 块内的敏感变量解决。但是尝试过多种变量触发方式，依旧无法解决从一开始就存在的问题：行为仿真虽然正确，但是综合后仿真、以及实际情况都不正确。

这两张图也展现了典型问题之一：



Next 信号可以触发 count 自增，但是在 Next 的持续时间内， count 以极快的速度自增直到上限、并重新计数！

最后我使用了一种比较万能的方法来解决，即变组合逻辑为时序逻辑。通过引入极慢的时钟信号，在时钟上升沿时才判断是否有按键按下，将上述的“在按键按下期间模块行为不可控”的问题解决了。

3.冒泡排序不停止

在实验中，遇到的另一个比较严重的问题，就是冒泡排序不停止——即 `Halt` 灯不亮起，这也是仿真与实际不符的一部分。这个问题也困扰了我很久，但是我将CPU的时钟信号替换为七段码显示所用的较低频时钟后，这个问题就顺利解决了。

我猜测，开发板自带的 `100MHz` 时钟对于我所编写的CPU来说太快，或者开发板中线路的延迟本就限制了时钟频率不能太快。这使得，使用 `100MHz` 时钟时，指令本身就没被解析完成执行完成，使得冒泡排序程序本身就变得不可控。因此，以后在使用开发板时应当注意时钟频率是否适合。