第六次实验——多周期CPU

一、实验目的

1.掌握多周期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.加深理解多周期CPU在单周期CPU基础上的改进。

4.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]
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(一)——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

ALUControl	Symbol	Opcode	Funct
	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
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、实验原理

1.多周期的五阶段

与单周期CPU不同，单周期CPU是每条指令都用一个时钟周期完成，使得时钟周期必须与用时最长的指令(即 `lw`)一致，造成了时间上的极大浪费。

对于多周期CPU而言，将指令的执行划分为五个阶段，只有耗时最长的指令需要完整经历这五个阶段，其余大部分指令只需要将必要的阶段完成即可。这样，时钟周期就由耗时最长的阶段决定，从理论上而言应当比单周期CPU有更高的时间利用率。

这五个阶段分别是：

(1)取指令(`Instruction Fetch` ，简称 `IF`)：根据程序计数器 `PC` 所指向地址，从指令存储器中取出指令；

(2)指令译码(`Instruction Decode` ，简称 `ID`)：将取出的指令进行分析译码，产生各控制信号，指导之后阶段指令的运行；

(3)指令执行(`Execution` ，简称 `EXE`)：根据控制信号具体地执行指令；

(4)存储器访问(`Memory Access` ，简称 `MEM`)：根据送入的地址，向存储器中写入数据或是取出数据；

(5)结果写回(Write Back ，简称 WB)：将运算结果或者从存储器中取到的数据写回到寄存器中。

每一个指令需要经历的阶段不同，因此这里给出各指令的阶段表：

	IF	ID	EXE	MEM	WB
add	√	√	√		√
sub	√	√	√		√
and	√	√	√		√
or	√	√	√		√
slt	√	√	√		√
sll	√	√	√		√
addi	√	√	√		√
andi	√	√	√		√
ori	√	√	√		√
beq	√	√	√		
bgtz	√	√	√		
lw	√	√	√	√	√
sw	√	√	√	√	
j	√	√			
halt	√	√			

2.控制单元Control(二)——仿微指令

由上表，我们发现所有的指令都需要经过 IF 和 ID 阶段。因此完全可以在指令译码阶段，决定接下来需要经过哪些阶段。这便是控制单元 Control 的职责。经过单周期CPU的设计，已经知道以下控制信号是可能必要的：

```
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
    );
......
```

以上控制单元模块总共需要输出 28 位二进制数，而标识有哪些阶段需要执行只需要 3 位数！似乎这些信号都能存储在一条 32 位的指令中。因此在这次多周期CPU的实验中，我遵照微指令的思想，将指令译指改造为仿微指令的生成。仿微指令各位的意义如下：

MicroInstruct	Significance
[0]	ALUSrc
[1]	RegDst
[2]	RegWrite_en
[3]	MemRead_en
[4]	MemWrite_en
[5]	MemtoReg_en
[6]	Jump_en
[7]	Branch_en
[8]	Halt_en

MicroInstruct	Significance
[9]	ZeroFlag
[14 : 10]	RegWrite
[19 : 15]	RegRead2
[24 : 20]	RegRead1
[28 : 25]	ALUControl
[31 : 29]	StageControl

其中三位的 StageControl ，从第 0 位到第 2 位分别代表 EXE ， MEM ， WB 阶段是否发生。在具体的处理过程中，会追加设计一个 Stage 模块，将 StageControl 信号在前面扩展两位的 1 之后送入此模块，再根据时钟决定接下来执行什么阶段。

上面文字叙述的信号总共只需要 31 数，对于 32 位仿微指令来说还剩 1 位数。这里巧妙地将 ALU 产生的 ZF 给放了进来，作为仿微指令的第 9 位数。这样算是对这个 32 位指令物尽其用了（当然，后来发现 ALU 的 ZF 信号是没法直接赋值到非输入的 wire 型变量上的某一位的，所以仿微指令的第 9 位数实际还是没有作用）。这样，原来的 Control 现在编写起来非常简洁，只需要一个输出即可：

```
module Control(
    input [31 : 0] Instruct,
    output reg [31 : 0] MicroInstruct
    );
......
```

四、实验过程(CPU部分)

1.阶段控制器 Stage

在这次实验中，与原生多周期CPU不同，我单独设计了一个控制全局的阶段控制器 Stage 。这个模块接收来自仿微指令 MicroInstruct 的 StageControl 字段的内容，根据它输出随时钟变化的阶段独热码 Stage 。

例如，如果 StageControl=101 ，这是指现在执行的指令会经过 EXE 阶段、然后跳过 MEM 阶段直接开始 WB 阶段，那么模块就应该接收一个高位扩展为五位的 StageControl=11101 ，即无论什么指令都应该经过 IF 和 ID 阶段。之后，阶段独热码从 10000 开始，然后每过一个时钟周期就分别变为 01000 、 00100 、 00001 ，表示现在要执行什么阶段。这个模块在一定程度上代替了时钟。

完整代码如下：

```
module Stage(
    input CLK_in,
    input Halt_en,
    input [4 : 0] StageControl,
    output [4 : 0] Stage
    );

    reg [4 : 0] counter;
    initial counter <= 5'b10000;

    always@(posedge CLK_in) begin
        case (counter[4 : 0])
            5'b10000: begin counter = 5'b01000; end
            5'b01000: begin
                if (StageControl[2] == 1) begin counter = 5'b00100; end
                else if (StageControl[1] == 1) begin counter = 5'b00010; end
                else if (StageControl[0] == 1) begin counter = 5'b00001; end
                else begin counter = 5'b10000; end
            end
            5'b00100: begin
                if (StageControl[1] == 1) begin counter = 5'b00010; end
                else if (StageControl[0] == 1) begin counter = 5'b00001; end
                else begin counter = 5'b10000; end
            end
            5'b00010: begin
                if (StageControl[0] == 1) begin counter = 5'b00001; end
                else begin counter = 5'b10000; end
            end
            5'b00001: begin counter = 5'b10000; end
        endcase
    end

    assign Stage = (Halt_en == 1) ? 5'b10000 : counter;

endmodule
```

2.程序计数器 ProgramCounter

与单周期时稍有不同，每条指令执行时有不同的时长，因此指令不应该胡乱变化，在一条指令的执行期间应当恒指向当前指令。为此，规定只有在阶段独热码为 `10000` ，即开始取指令 `IF` 阶段时，才允许输出的地址变化。因此，将输出 `Address_out` 由原来的线型改为了寄存器型。

完整代码如下：

```
module ProgramCounter(
    input CLK_in,
    input Start_en,
    input Halt_en,
    input Write_en,
    input [31 : 0] Address_in,
    output reg [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

    always@(Write_en) begin
        if (Write_en == 1) Address_out = Address;
    end

endmodule
```

3.PC写入模块 PCWriter

与单周期CPU时相比，本次实验中还加入了一个独创的模块 PCWriter 。这实际相当于一个 32 位 3 路多选器，但是由于多周期的特殊性，必须把当前CPU整体所处的阶段给纳入考虑。只有在符合条件的情况下才在第二阶段 ID 、第三阶段 EXE 才会将跳转地址或是分支地址写入 PC 中的组合逻辑寄存器中。

同时，这个模块吸收了之前单周期时跳转地址生成单元 JumpAddGenerator 的功能，减少了一个冗余模块。

完整代码如下：

```
module PCWriter(
    input ZF,
    input [4 : 0] Stage,
    input [31 : 0] MicroInstruct,
    input [31 : 0] PCPlus4,
    input [31 : 0] JumpPC,
    input [31 : 0] BranchPC,
    output [31 : 0] NextPC
    );

    reg [31 : 0] counter;

    initial counter = 32'h00000000;

    always@(Stage) begin
        case(Stage)
            5'b10000: counter = PCPlus4;
            5'b01000: counter = (MicroInstruct[6]) ? JumpPC : PCPlus4;
            5'b00100: counter = (MicroInstruct[7] && ZF) ? BranchPC : PCPlus4;
            default: counter = PCPlus4;
        endcase
    end

    assign NextPC = counter;

endmodule
```

4.指令寄存器 InstructMem

指令寄存器基本可以看做是与 PC 联动的。由于 PC 会一直持续到下一个 Stage=10000 状态时才发生变动，从组合逻辑的指令寄存器中取出的指令也自然会一直持续到下一个 Stage=10000 状态。

完整代码如下：

```
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
```

在这里，因为都使用冒泡排序测试，所以使用的还是 `SingleCPU` 中的冒泡排序二进制码。

5.控制单元 `Control`

正如之前所述，在本次实验中，遵照微指令的思想，现在控制单元会将指令译码为一条 `32` 位的仿微指令 `MicroInstruct` 以输出，各位的含义已经在*实验原理*中给出。大致结构与单周期时相似，但是添加了每条指令对 `StageControl` 字段的写入。

完整代码如下：

```
module Control(
    input [31 : 0] Instruct,
    output reg [31 : 0] MicroInstruct
    );

    always@(Instruct) begin
        MicroInstruct[31 : 0] = 32'h00000000;
        case (Instruct[31 : 26])
            6'b000000: begin     //R_type
                MicroInstruct[31 : 29] <= 3'b101;
                MicroInstruct[24 : 10] <= Instruct[25 : 11];
                case (Instruct[5 : 0])
                    6'b100000: begin MicroInstruct[28 : 25] <= 4'b0010; { MicroInstruct[0], MicroInstruc
                    6'b100010: begin MicroInstruct[28 : 25] <= 4'b0110; { MicroInstruct[0], MicroInstruc
                    6'b100100: begin MicroInstruct[28 : 25] <= 4'b0000; { MicroInstruct[0], MicroInstruc
                    6'b100101: begin MicroInstruct[28 : 25] <= 4'b0001; { MicroInstruct[0], MicroInstruc
                    6'b101010: begin MicroInstruct[28 : 25] <= 4'b0111; { MicroInstruct[0], MicroInstruc
                    6'b000000: begin     //sll
                        MicroInstruct[28 : 20] <= { 4'b1100, Instruct[20 : 16] };
                        MicroInstruct[2 : 1] <= 2'b11;
                    end
                endcase
            end

            6'b001000: begin      //addi
                MicroInstruct[31 : 20] <= { 3'b101, 4'b0010, Instruct[25 : 21] };
                MicroInstruct[14 : 10] <= Instruct[20 : 16];
                MicroInstruct[2 : 1] <= 2'b11;
            end
            6'b001100: begin      //andi
                MicroInstruct[31 : 20] <= { 3'b101, 4'b0000, Instruct[25 : 21] };
                MicroInstruct[14 : 10] <= Instruct[20 : 16];
                MicroInstruct[2 : 1] <= 2'b11;
            end
            6'b001101: begin      //ori
                MicroInstruct[31 : 20] <= { 3'b101, 4'b0001, Instruct[25 : 21] };
                MicroInstruct[14 : 10] <= Instruct[20 : 16];
                MicroInstruct[2 : 1] <= 2'b11;
            end

            6'b100011: begin      //lw
                MicroInstruct[31 : 20] <= { 3'b111, 4'b0010, Instruct[25 : 21] };
                MicroInstruct[14 : 10] <= Instruct[20 : 16];
                { MicroInstruct[3 : 1], MicroInstruct[5] } <= 4'b1111;
            end
            6'b101011: begin      //sw
```

```
                MicroInstruct[31 : 15] <= { 3'b110, 4'b0010, Instruct[25 : 21], Instruct[20 : 16] };
                MicroInstruct[4] <= 1'b1;
            end

            6'b000100: begin      //beq
                MicroInstruct[31 : 15] <= { 3'b100, 4'b0110, Instruct[25 : 21], Instruct[20 : 16] };
                { MicroInstruct[0], MicroInstruct[7] } <= 2'b11;
            end
            6'b000111: begin      //bgtz
                MicroInstruct[31 : 15] <= { 3'b100, 4'b0110, Instruct[25 : 21], 5'b00000 };
                { MicroInstruct[0], MicroInstruct[7] } <= 2'b11;
            end

            6'b000010: begin MicroInstruct[6] <= 1; end    //j
            6'b111111: begin MicroInstruct[8] <= 1; end    //halt
        endcase
    end

endmodule
```

6.寄存器堆 RegisterFile

寄存器堆的代码与单周期情形是一样的。只有在允许写入的情况下，才会在时钟下降沿写入数据。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。

完整代码如下：

```
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
```

7.符号扩展 SignExtend

这是一个基本单元，实现起来并不复杂。只是单纯根据输入的 16 位数据的最高位进行符号扩展而已。

完整代码如下：

```
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
```

8.算术逻辑单元 ALU

与单周期时相同，本实验中的 ALUControl 直接由控制单元生成，具体见上述。原生的第六个功能由或非替换为左移。对于组件中最重要最通用的一个算术单元 ALUGeneral ，注意到左移时第二个操作数是由指令符号扩展而来，所以在单周期实验中“简便起见左移固定只移两位”是无稽之谈，在本次实验中已改正。

完整代码如下：

```
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 << Src2[10 : 6];
            default: Result = 32'bz;
        endcase
    end

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

endmodule
```

9.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。

完整代码如下：

```
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 : 63];

    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
```

10.二路32位多选器 Mux

这种多选器很简单，与之前类似，不多赘述。完整代码如下：

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

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

endmodule
```

在本次实验中，去掉了 `WrReg Mux` ， `Branch Mux` ， `Jump Mux` ，二路32位多选器可以囊括所有需求。将要生成的两个多选器实例是：

ALU Mux ， Mem/ALU Mux

11.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。

12.CPU部分TOP文件 MultiCPU

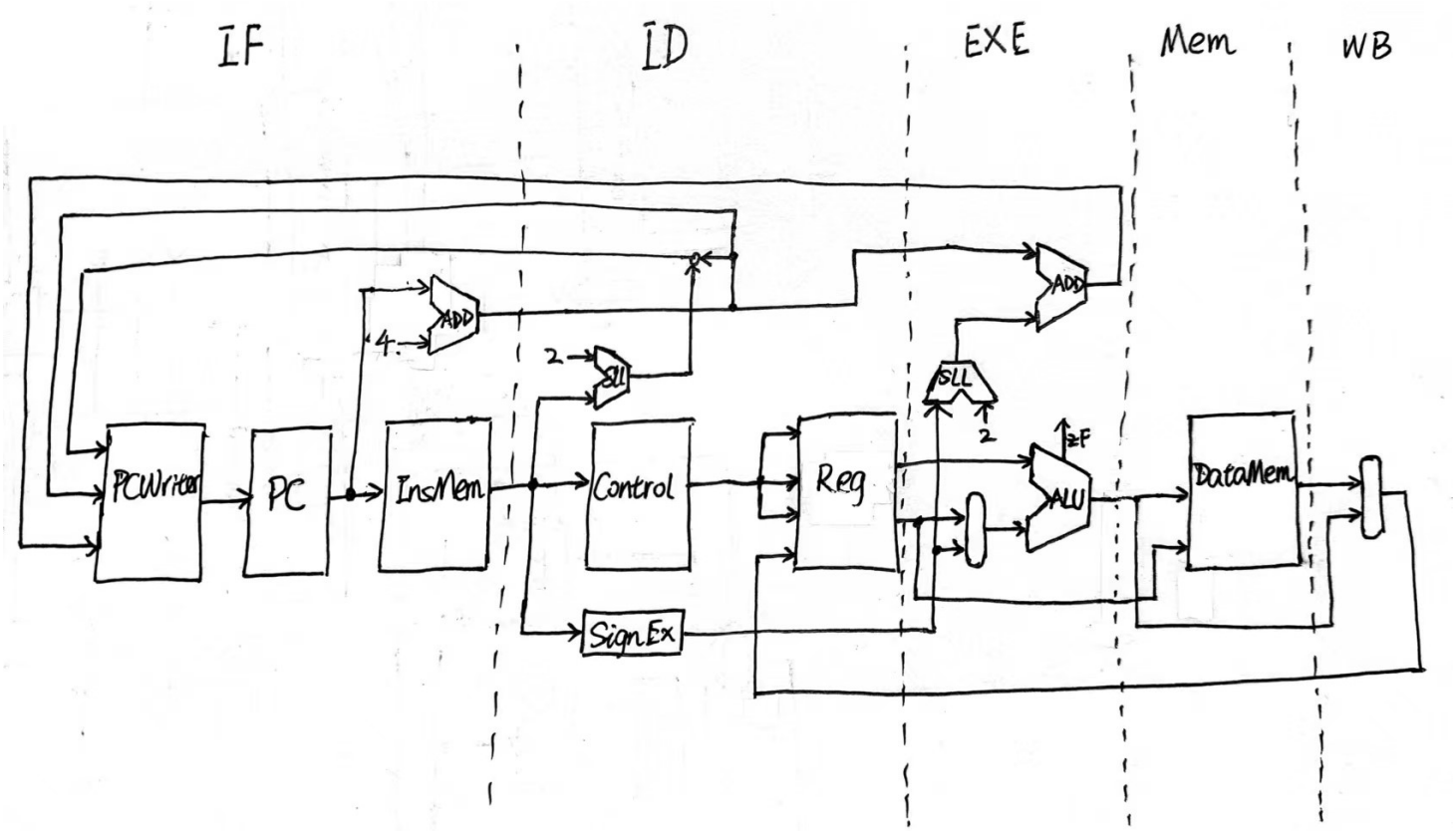
CPU部分的TOP文件将各模块串联起来。由于实例化内容很多、代码冗长，这里只展示 wire 变量定义部分：

```
module MultiCPU(
    input CLK,
    input Start,
    ......
    );

    wire ZF_wire;
    wire [4 : 0] Stage_wire;

    wire [31 : 0] InstructAddress_wire;
    wire [31 : 0] PCPlus4_wire;
    wire [31 : 0] JumpPC_wire;
    wire [31 : 0] BranchPC_wire;
    wire [31 : 0] SL2toPW_wire;
    wire [31 : 0] NextPC_wire;
    wire [31 : 0] Instruction_wire;
    wire [31 : 0] MicroInstruct_wire;
    wire [31 : 0] SignEx_wire;
    wire [31 : 0] ReadData1_wire;
    wire [31 : 0] ReadData2_wire;
    wire [31 : 0] ALU_Mux_wire;
    wire [31 : 0] ALUGeneral_wire;
    wire [31 : 0] SL2toBranch_wire;
    wire [31 : 0] DataMemRead_wire;
    wire [31 : 0] RegWriteData_wire;
    ......
```

当然，由于有之前设计单周期CPU的经验，并且使用了仿微指令，实际顶层文件是比单周期时简单易读的。多周期CPU例化之后的结构如图所示：

- MultiCPU (MultiCPU.v) (15)
  - PC : ProgramCounter (ProgramCounter.v)
  - PCPlus4 : ALU (ALU.v)
  - PW : PCWriter (PCWriter.v)
  - IM : InstructMem (InstructMem.v)
  - SE : SignExtend (SignExtend.v)
  - CU : Control (Control.v)
  - SC : Stage (Stage.v)
  - SL2toPW : ALU (ALU.v)
  - RF : RegisterFile (RegisterFile.v)
  - ALU_Mux : Mux (Mux.v)
  - ALUGeneral : ALU (ALU.v)
  - SL2toBranch : ALU (ALU.v)
  - ADD : ALU (ALU.v)
  - DM : DataMem (DataMem.v)
  - Mem_ALU_Mux : Mux (Mux.v)

Vivado生成的 `Elaborated Design` 原理图不能很好地体现多周期CPU的结构化、阶段化，所以我还是以一幅手绘的原理图来呈现顶层文件的连接效果：



大致的架构与单周期CPU是相同的，但是特化处理了有关“阶段”的模块与内容。

五、实验内容(外部输入输出部分)

与单周期时相似，由于需要接收外部的输入、也需要将数据输出到外部，需要一个外部输入输出模块实现。由于大致结构与单周期时是相同的，并没有将指令存储器与数据存储器合并，可以直接复用单周期时的代码。这分为以下一些部分：

1.时钟分频 CLK_div

时钟分频非常基本，不多赘述：

```
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

经过多次实验，数字显示也是非常基本的内容，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.按键消抖 ButtonDebounce

由于机械按钮的机械属性，必须引入按键消抖模块，这也算是非常基本的内容之一。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
```

4.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直接指向数据存储器。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
    );

    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
```

5.CPU部分输入输出接口 IOPort

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。当 `Stop_en=0` ，表示正在接收外部输入的数据；反之，由数据存储器向外部输出数据。

```
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
```

与单周期时相同，我规定只用12个拨板开关输入数字，虽然没有完全用到开发板上的16个拨板开关，但能输入的数值已有$0 \sim 2^{11} - 1$的大小，已是相当足够的了。开发板上只能同时显示四个数字，低三位用于显示数值的16进制形式，最高位用于显示这个数字在内存中的位置。

六、实验验证

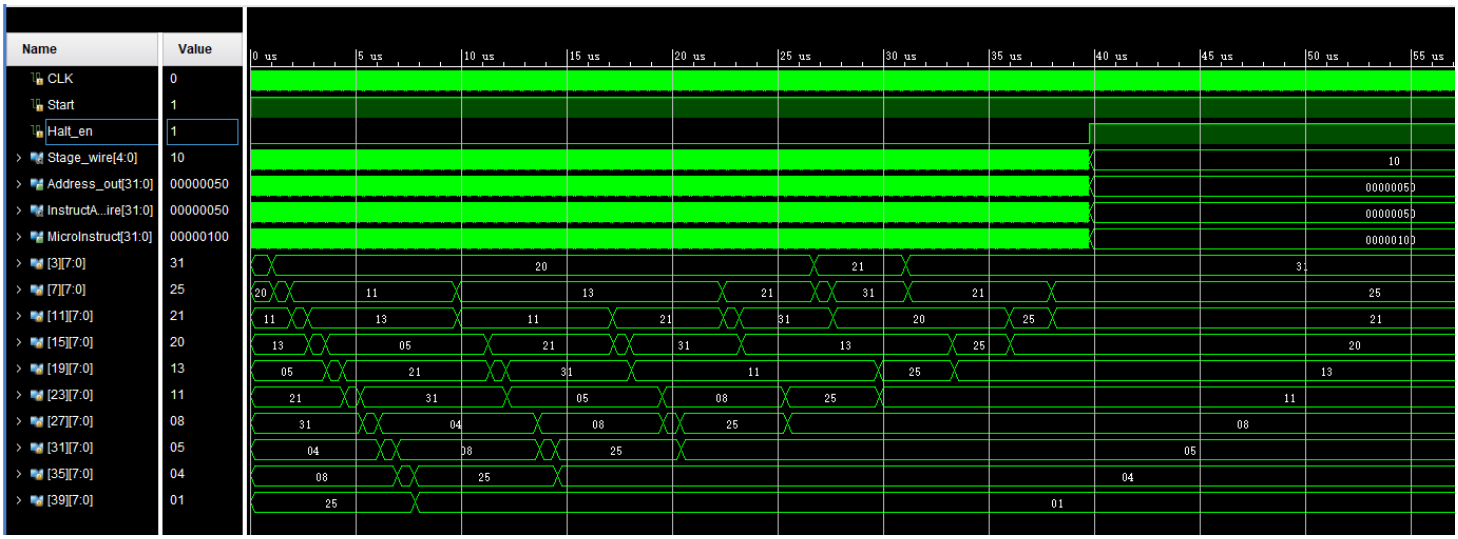
接下来单独对CPU部分进行模拟仿真。

1.整体验证

首先还是整体验证，即对预置入内存中的十个数运行冒泡排序。如果最后结果正确，即可基本断定没有问题。向数据存储器 `DataMem` 模块中添加一个 `initial` 块：

```
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
```

之后仿真运行：

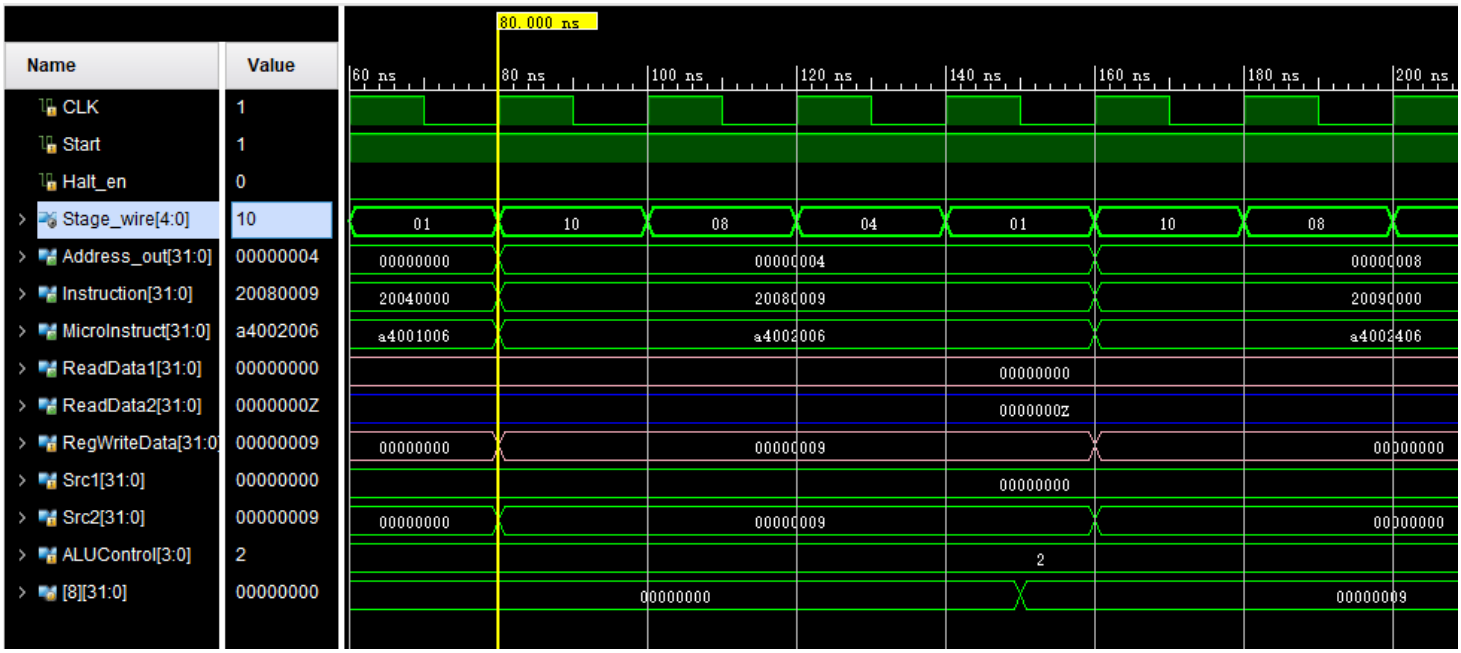


可以看到原本内存中预存的乱序的数，运行冒泡排序之后变得有序。这首先说明了冒泡排序程序的正确性。

另外，在程序运行到第 `50` (十六进制)地址处，此时是 `Halt` 指令， `Halt_en` 变为高电平，整个CPU停止运行。

2. `addi` 与 `add`

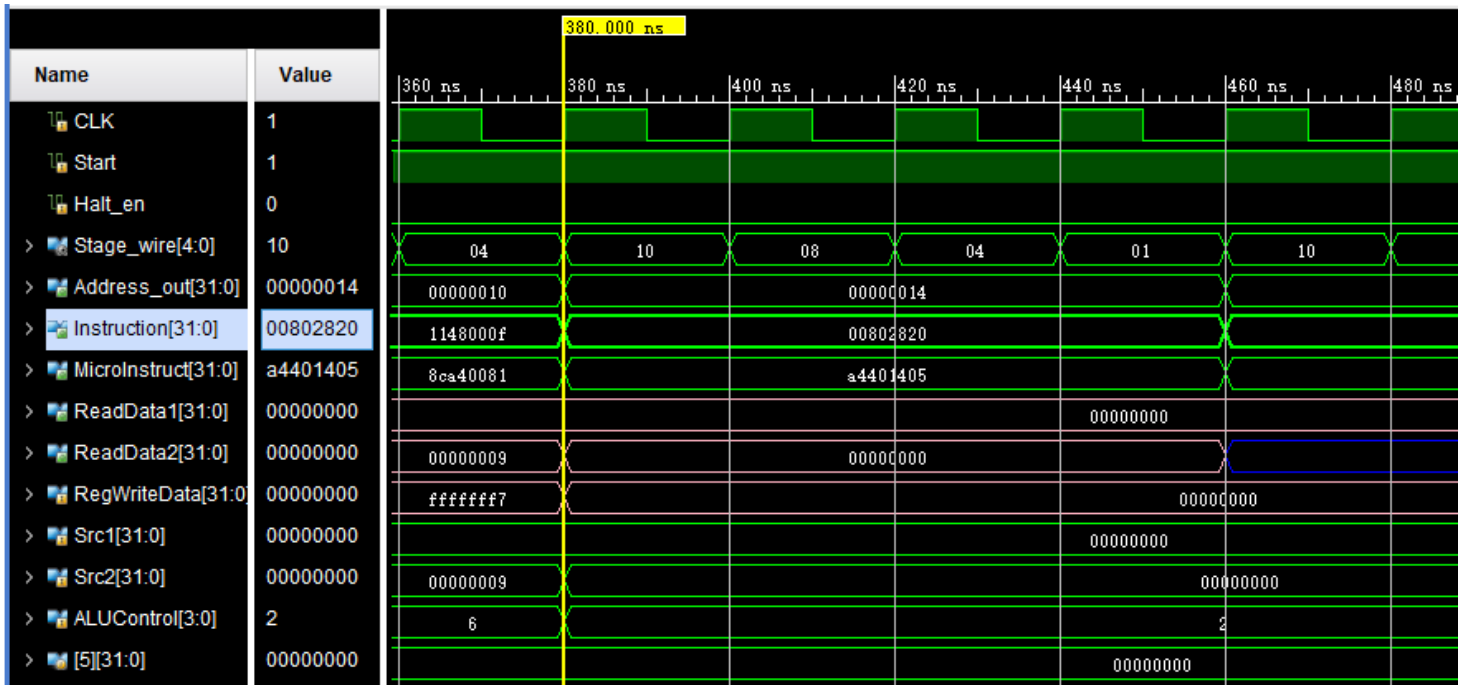
程序的第二条指令是 `addi $t0, $zero, 9` ， 十六进制码即 `20080009` ， 运行如图：



由控制单元生成的仿微指令是 a4002006 ， 解码可得 StageControl=101 ， 因此这条指令会经历4个时钟周期，经过 10000 ， 01000 ， 00100 ， 00001 这四个阶段，简便起见之后都用16进制来写。

同时解码可得 ALUSrc=1 ， RegDst=0 ， RegWrite_en=1 。这说明限制寄存器第二源操作数的读取， ALU 的第二源操作数来自指令的符号扩展。在多周期CPU中， RegWrite_en 实际与 StageControl 中的第三位数必然相等，因为写入寄存器必然发生在写回阶段。可以看到，在 01 阶段的时钟下降沿，运算结果写入了 $t0 即第8号寄存器。

指令 add $a1, $a0, $zero ， 十六进制码 00802820 ， 这实际相当于令 $a1=$a0 。在程序刚开始运行、未开始外层循环时， $a0=0 ，所以图中的5号寄存器的值没变化，但实际已经将 RegWriteData 的值给写入了。



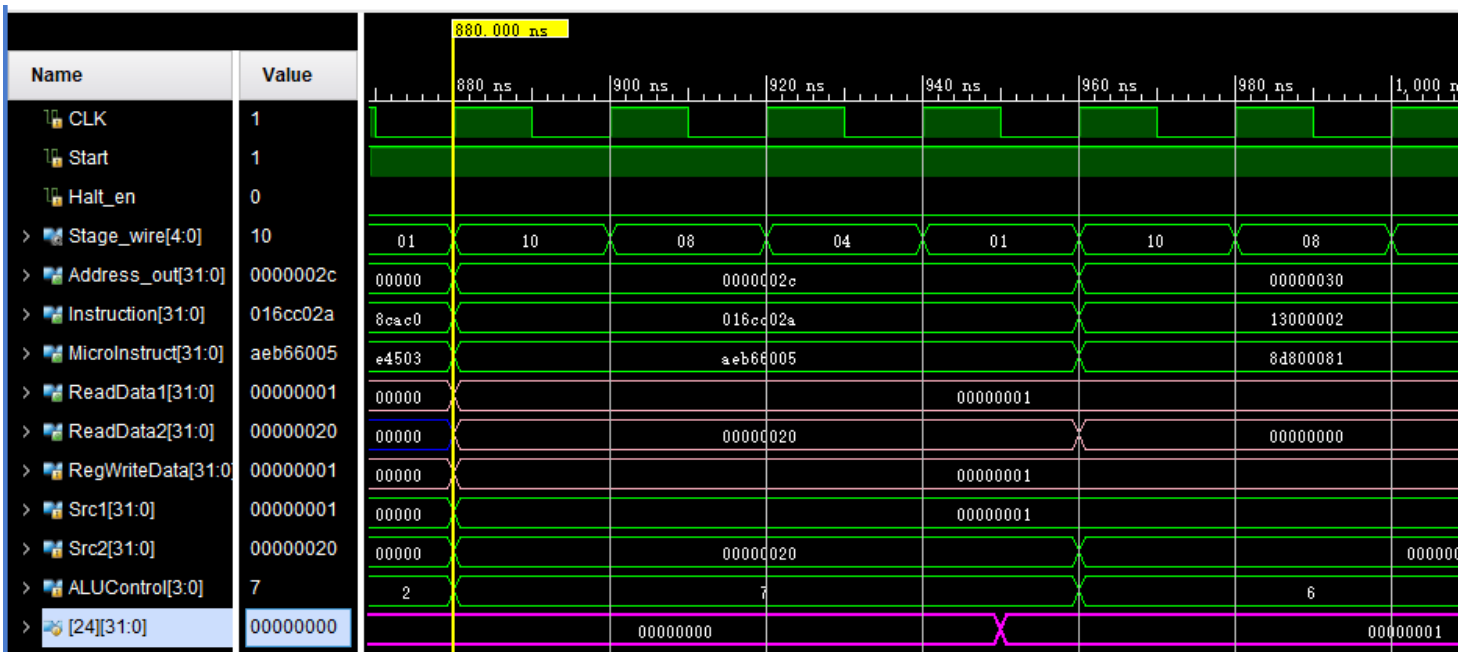
对仿微指令 a4401405 解码，可以看到 RegRead1 ， RegRead2 ， RegWrite 分别赋值为 00100 ， 00000 ， 00101 ，这与指令所指示的是一样的。

3. sub 与 slt

指令 `sub $t7, $t0, $t2` ， 十六进制码 `010a7822` ， 仿微指令 `ac853c05` 。运行如图：

与上述的相同，在 `01` 阶段的时钟下降沿将运算结果写回到寄存器。对于 `sub` 指令， `ALUControl=0110` ， 这也是正确的。

指令 `slt $t8, $t3, $t4` ， 十六进制码 `016cc02a` ， 仿微指令 `aeb66005` 。运行如图：

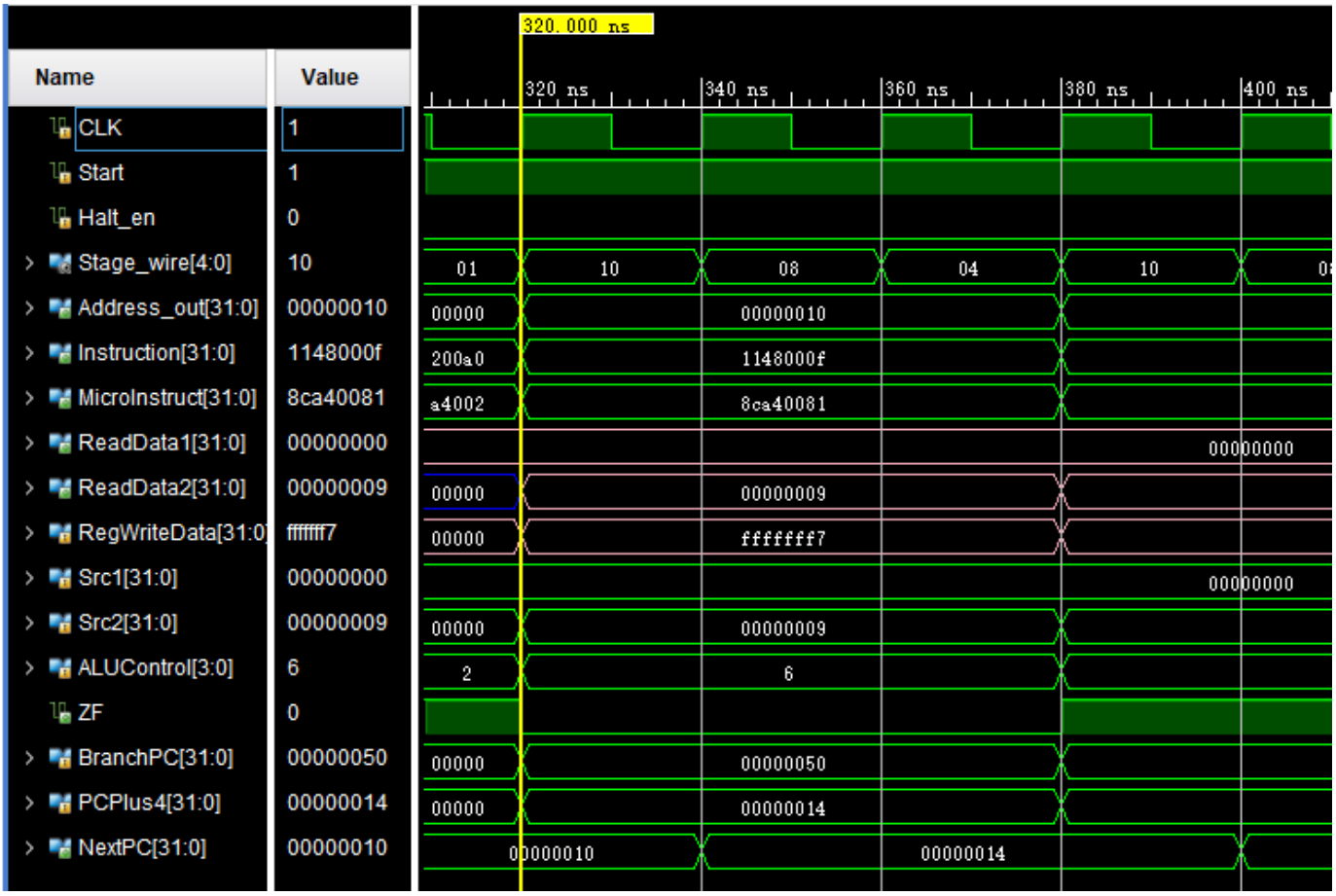


这是判断小于则置 `1` ， `ALUControl` 码是原来的或非，即 `0111` ， 这里第一源操作数为 `1` ， 小于第二源操作数的 `20` ， 故在 `01` 阶段的时钟下降沿将 `$t8` 置 `1` 了。

4. beq

`beq` 指令只需要经历前三个阶段，在 `EXE` 阶段结束后就回到 `IF` 阶段。首先来看一个不分支的情况：

指令 `beq $t2, $t0, done_outer_loop` ， 十六进制码 `1148000f` ， 仿微指令 `ac853c05` 。运行如图：



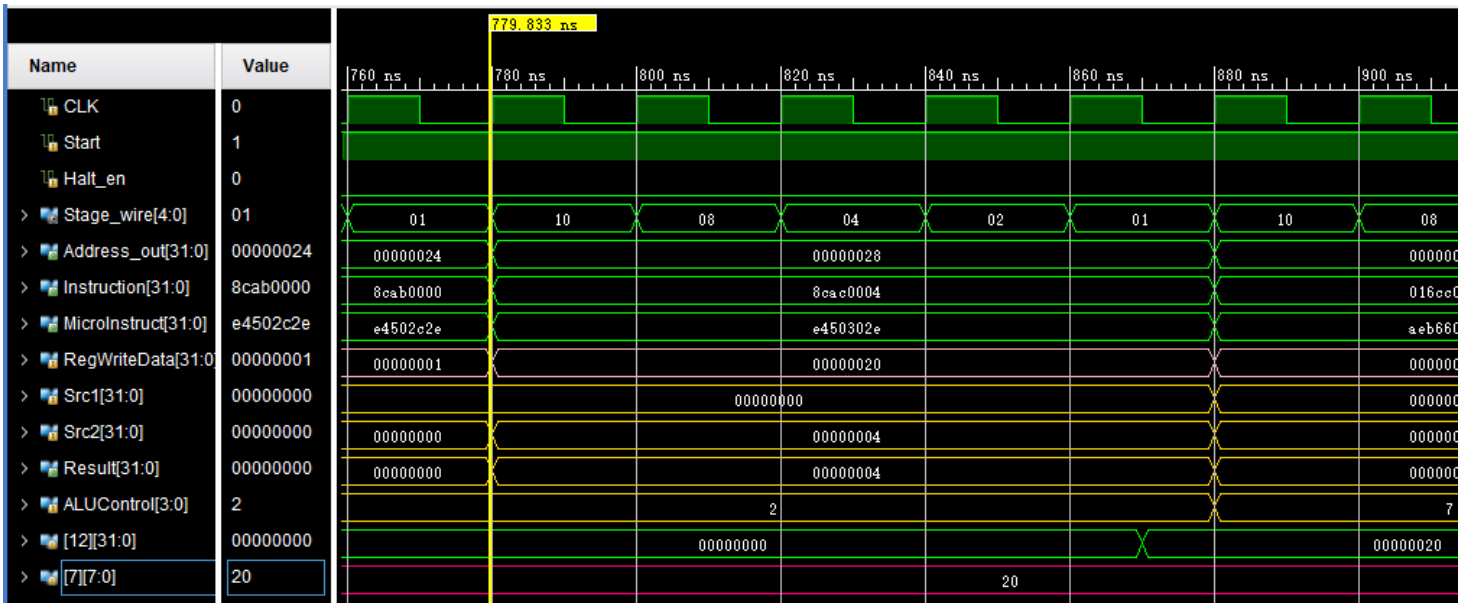
可以看到这个指令运行了三个时钟周期。 ALU 执行 - 操作，结果不为 0 ， 零标志位 ZF 为低电平。从仿微指令中可以解码得到 Branch_en=1 ， 只有这两者同时为 1 才会分支。可以看到 PCWriter 在 BranchPC 与 PC+4 之间选择了 PC+4 ， 也就是分支不发生。

接下来看一个分支发生的情况。指令 beq $t1, $t7, done_inner_loop ， 十六进制码 112f0009 ， 仿微指令 8c978081 。运行如图：

此时 ZF=1 ， 则分支发生。可以看到 NextPC 直到 EXE 阶段才选择了 BranchPC ， 这是因为 ID 阶段是判断是否选择跳转地址。

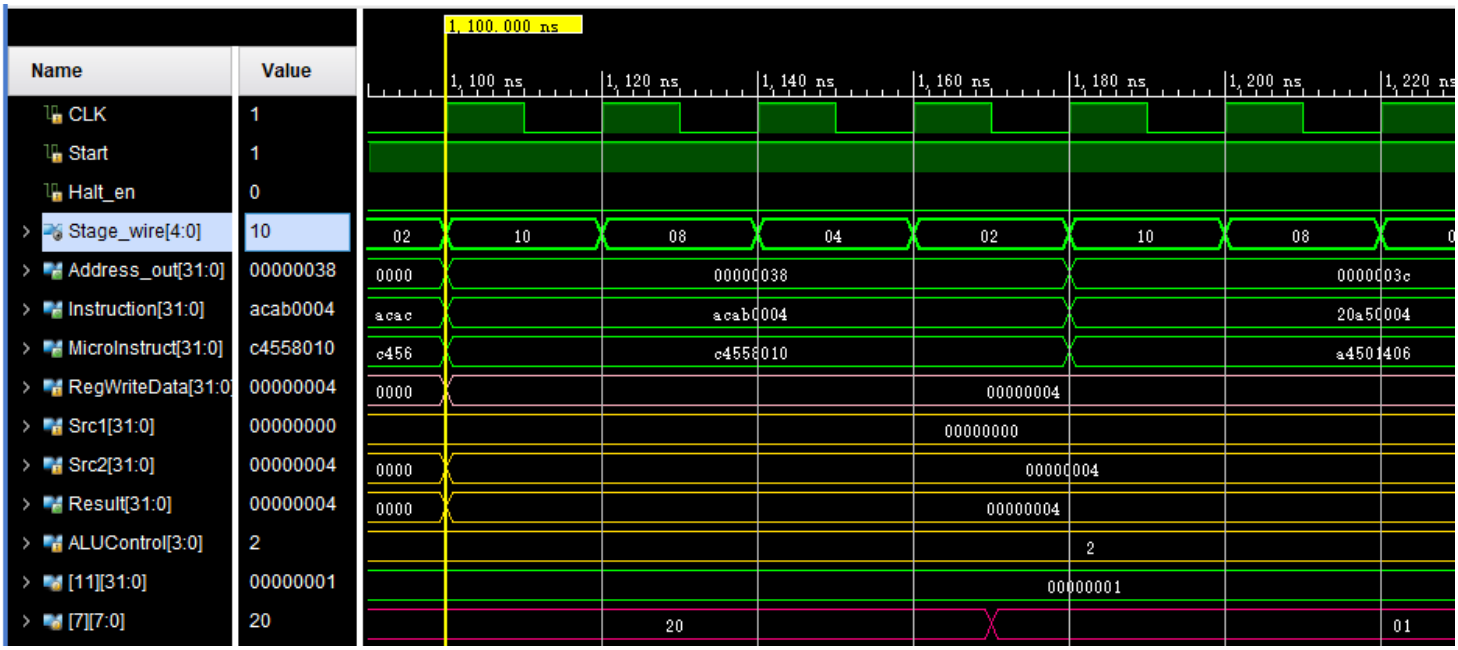
5. lw 与 sw

lw 指令是唯一需要经过五个阶段的指令。指令 lw $t4, 4($a1) ， 十六进制码 84ac0004 ， 仿微指令 e450352e 。运行如图：



此时 ALU 中送来的两个源操作数，分别是 $a1 中的值和立即数 4 。运算结果为 Result=4 ， 这指向了内存中的 4 及其后三个八位的字节。此时其中存储的数是第 7 字节中的 20 ， 之后在 01 即写回阶段的时钟下降沿将内存中的值写入到 $t4 寄存器中。

sw 指令是唯一需要访存但不写回的指令。指令 sw $t3, 4($a1) ， 十六进制码 84ac0004 ， 仿微指令 e450352e 。运行如图：

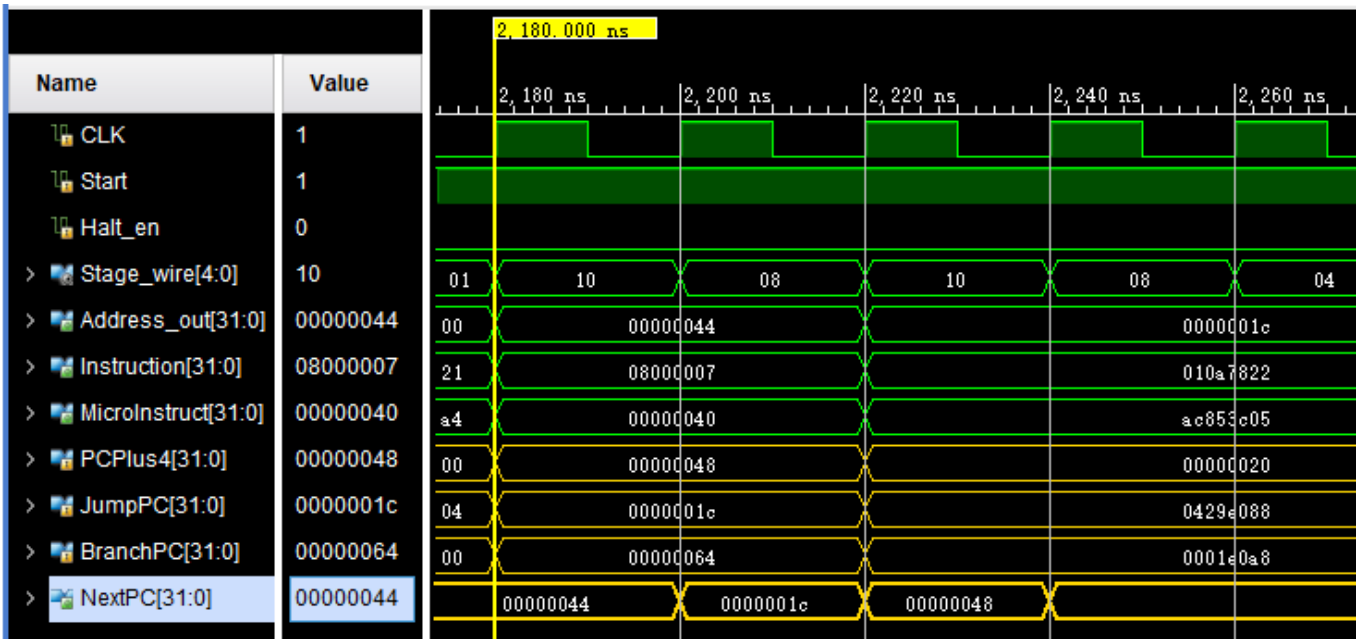


这条指令基本是上述 lw 指令的逆过程，要将 $t3 的值存入内存中的第 4 个字中。 4 由 ALU 计算出，指向内存中要存入位置的首地址。在 02 阶段，同样是在时钟下降沿将要写入的值写入指定内存位置中。

6. j

j 指令是需要经过阶段最少的指令，只有前两阶段。因此在 PCWriter 中，规定第一阶段 NextPC 就更新为 PC+4 ，第二段判断是否修改为跳转地址，第三阶段判断是否修改为分支地址，其它阶段必然也就默认为 PC+4 。

指令 j inner_loop ， 十六进制码 08000007 ， 仿微指令 00000040 。运行如图：



可以看到这条指令只运行了两个时钟周期，在第二阶段就无条件跳转到计算出的跳转地址。此时仿微指令几乎不携带任何有效信息，只有一位 Jump_en=1 。

七、实验总结

本次实验由于有单周期CPU从零开始造起的经验，本次多周期CPU实验完成的效率提高了很多很多。

在本次实验中我遵循三个标准：其一，就是单周期实验时的“多多复用”，复用是一种最有效的提高开发效率的方法；其二，是“长远考虑”，例如本次实验中我没有将通用算数单元也用在 IF 阶段计算 PC+4 、没有将指令存储器和数据存储器合并为一个存储器，主要是为了以后便于扩展为流水线CPU。

其三，“一切从简”。从实验开始，我就在思考“设计一种精妙的结构”，使得我能在单周期的基础上方便地扩展为多周期。我在设想阶段绘制了很多幅原理图，也稍微参照了一些参考资料，我发现在单周期架构下的多周期CPU的每个阶段之间实际并不需要多附设寄存器；同时单周期中的一些模块并不是必要的，比如 JumpAddressGenerator 。

同时，为了改进单独的一位线型变量太多的问题，我仿照微指令的思想，将大部分信息集中到这个 32 位的仿微指令中。最后从观感上确实要简洁一些，不过编写的时候也确实需要时时查询各位的对照表。

而究竟应该如何简便地将单周期扩展为多周期？我查阅的资料多显示应在 ControlUnit 中，应当根据当前状态结合指令内容决定产生什么控制信号，以及下一个状态的选择。但我认为这是将状态转移想复杂了。一条指令要经过什么阶段、要产生什么控制信号，这由指令本身本就能完全确定，与当前的运行状态并无关系，因此我认为 ControlUnit 这个模块并无必要将状态作为*因变量*(当然，这是要在单周期架构下直接改进为多周期才成立)。因此我直接设计了三位的 StageContol 来指示指令会经历什么阶段。

而CPU当前应处于什么阶段这个问题，是一个全局性的问题。在不大改 Control 模块的前提下，就必然要多设计一个有关阶段控制的模块来全局性地约束CPU，这就导致了 Stage 的诞生。 Stage 根据 StageContol 和时钟来改变状态。

在设想阶段，我也想了很多一些设计的细节。比如在 Stage 模块的基础上，我原本打算将各阶段所涉及的内容直接打包为一个“阶段模块”，当 Stage 指示当前运行为某阶段时，这个阶段就开始运行。但是这种设计必然导致两个阶段间必然要附设一组寄存器，这打破了“一切从简”的原则；同时也为寄存器堆这种在译码和写回阶段都需要运行的模块产生了麻烦。

所以最后设计出来的CPU，虽然表象上是按阶段运行，其本质却又是单周期CPU。模块运行与时序无关，例如本应在译码阶段才产生 StageControl 码、更新仿微指令，但实现的时候为了“一切从简”， Control 模块在指令更新的一瞬间就产生了这条指令对应的仿微指令，虽然方便、避免了很多时序上的bug，但我认为这倒是充满了单周期的意味。

最后，计组课是一门面向硬件底层的课。我虽然尚未学习过数电，但最终还是完成了单周期与多周期CPU的实验。关于CPU的理论知识光是听讲还是容易云里雾里，但经过一段时间的沉淀、经过真正上手从零开始设计的实验之后，确实可以算是比较深入地理解了MIPS CPU的设计与工作思想。