

数字逻辑与处理器基础实验
小学期课程设计报告
BobAnkh

说明：本次课程设计根据相关要求，在孙老师提供的vivado 2014.3.1版本上调试运行，本报告所有调试和仿真以及相关数据均基于此。

一、设计方案（原理说明及框图）与关键代码

在参考理论课相关原理课件的基础上进行设计。本次设计主要是采用MIPS-32指令集的一个子集，每条指令均为32位，对应指令的格式如下：

分支指令：

Instruction	OpCode[31:26]	rs[25:21]	rt[20:16]	Imm[15:0]
beq	0x04	rs	rt	offset
bne	0x05	rs	rt	offset
blez	0x06	rs	0	offset
bgtz	0x07	rs	0	offset
bltz	0x01	rs	0	offset

跳转指令：

Instruction	OpCode[31:26]	target_address[25:0]
j	0x02	target
jal	0x03	target

Instruction	OpCode[31:26]	rs[25:21]	rt[20:16]	rd[15:11]	shamt[10:6]	funct[5:0]
jr	0	rs	0	0	0	0x08
jalr	0	rs	0	rd	0	0x09

存储访问指令：

Instruction	OpCode[31:26]	rs[25:21]	rt[20:16]	Imm[15:0]
lw	0x23	rs	rt	offset
sw	0x2b	rs	rt	offset
lui	0x0f	0	rt	imm

R 型计算指令：

Instruction	OpCode[31:26]	rs[25:21]	rt[20:16]	rd[15:11]	shamt[10:6]	funct[5:0]
add	0	rs	rt	rd	0	0x20
addu	0	rs	rt	rd	0	0x21
sub	0	rs	rt	rd	0	0x22
subu	0	rs	rt	rd	0	0x23
and	0	rs	rt	rd	0	0x24
or	0	rs	rt	rd	0	0x25
xor	0	rs	rt	rd	0	0x26
nor	0	rs	rt	rd	0	0x27
sll	0	0	rt	rd	shamt	0
srl	0	0	rt	rd	shamt	0x02
sra	0	0	rt	rd	shamt	0x03

sllv	0	rs	rt	rd	0	0x04
srlv	0	rs	rt	rd	0	0x06
slt	0	rs	rt	rd	0	0x2a
sltu	0	rs	rt	rd	0	0x2b

I 型计算指令：

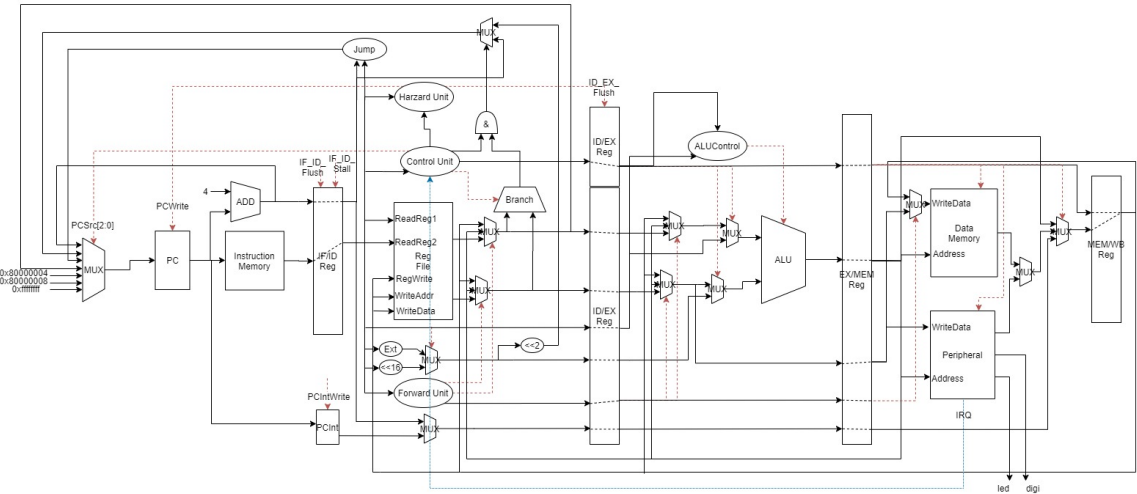
Instruction	OpCode[31:26]	rs[25:21]	rt[20:16]	Imm[15:0]
addi	0x08	rs	rt	imm
addiu	0x09	rs	rt	imm
andi	0x0c	rs	rt	imm
ori	0x0d	rs	rt	imm
slti	0x0a	rs	rt	imm
sltiu	0x0b	rs	rt	imm

该5级流水线的MIPS处理器，对于竞争冒险采用如下方式处理：

- 1) 对于数据关联问题采用转发解决
- 2) 对于load-use冒险在ID阶段判断，采用阻塞一个周期+转发的方式解决，发现load-use冒险时即会阻塞IF和ID两个阶段的命令
- 3) 对于分支指令在ID阶段判断，并在分支发生时刻冲刷掉IF阶段的命令
- 4) 对于跳转指令在ID阶段判断，并在跳转时冲刷掉IF阶段的指令

流水线CPU总体模块设计框图如下：

由于版面受限，所以画出了主要和关键的数据通路与控制信号，部分细节略去了。



各模块关键代码和设计说明如下：

总体顶层为CPU流水线，其中包含5个模块，分别是IF模块、PCInt模块、ID模块、EX模块、MEM模块。因为WB阶段主要就是选择是否将MEM/WB边界寄存器的数据写入到寄存器堆对应地址中，所以其已经在其他模块中实现了，故无需列

出单独模块。因为部分子模块是建立在单周期大作业的基础上进行了一些修改，所以简单介绍各个模块的设计，并将重点说明本次大作业比较独特的几个功能实现：外设、转发与冒险和中断与异常。

IF模块主要是通过PCSrc选择下一个时钟上升沿写入到PC中的信号，同时保存PC+4以及从指令存储器中读取出来的指令到边界寄存器。PC具有使能控制，边界寄存器也存在stall和flush，具体在后文中会说明。

PCInt模块主要是有条件地保存需要的PC信号以供中断或异常发生时，能够将正确的PC存入\$K0中从而能够在中断或异常结束后正确地跳转回来。该模块会在写入使能的情况下，在时钟上升沿来临时存入当前的PC。写入使能会在IF阶段既不发生stall也不发生flush时使能，从而确保发生在任何时候中断或异常都能够记录正确的需要的PC。在正常情况下，写入PCInt的PC就是对应写入IF/ID边界寄存器、进入ID阶段的指令的PC；而在发生flush或stall的情况下，因为不使能，则会保持记录其前一条指令（要么是分支指令，要么是跳转指令，要么是发生中断或异常而取消掉的ID阶段的指令），从而确保无论进入ID阶段的是何指令，在发生中断或异常时都能回到正确的点继续执行。这一模块部分关键代码如下：

```
always @(posedge clk or negedge rst_n)
begin
    if (~rst_n)
        PCInt <= 32'h00000000;
    else begin if (PCIntWrite)
        PCInt <= PC;
    end
end
```

```
PCInt pcint(
    .clk(clk),
    .rst_n(rst_n),
    .PCIntWrite(~(DataHazard || JumpHazard || BranchHazard || ((IRQ
|| Exception) && ~Supervise))),
    .PC(PC),
    .PCInt(PCInt)
);
```

之所以采用数据冒险、跳转冒险、分支冒险和中断或异常来控制PCIntWrite（即PCInt写使能），就是因为在ID阶段解码时发现这些情况则会flush或stall住IF/ID边界寄存器，因而此时不需要也不应该变更PCInt。

ID模块主要包含寄存器堆、控制信号模块、转发信号模块、冒险模块以及分支和跳转，产生的信号一部分送回IF模块使用，另一部分则送入边界寄存器供后续阶段使用。边界寄存器存在flush，具体见后文说明。同时在时钟上升沿来临

时，在这一模块中也实现WB阶段的功能——根据MEM/WB边界寄存器处的RegWrite信号决定是否向WriteAddr对应的寄存器写入WriteData。

EX模块根据前一级边界寄存器发来的相关数据和控制信号，结合转发选择合适的信号送入ALU，在产生的ALU控制信号的控制下输出结果，并存入边界寄存器中。

MEM模块主要是实现数据存储器和外设，根据地址和读写信号等进行相应操作，同时直接在本阶段根据控制信号选择WB阶段需要写入的信号，存入边界寄存器中。

外设模块。本次作业中可以说是首次接触到了比较多的外设，也按照要求实现了外设，主要是根据地址的首位（十六进制）判断是针对数据存储器还是外设的操作，来进行相应的读取数据选择和写入数据选择，具体的各外设实现主要参考课件和课堂讲授，主要也是配合软件（汇编代码）协同实现相关功能，如中断定时器、七段数码管等。中断定时器的使能等状态和定时的时长都由软件向对应地址存入对应数据决定。七段数码管则是借助查表法来实现，将表先存入到数据存储器中，根据软件中需要显示的数计算偏移量，从而查询到对应的数据，写入七段数码管对应地址中，从而显示出需要的结果。同时外设模块还会产生中断信号IRQ。外设模块关键核心代码块如下：

```
always @(*)
begin
    case (Address)
        32'h40000000: ReadData <= MemRead?TH:32'b0;
        32'h40000004: ReadData <= MemRead?TL:32'b0;
        32'h40000008: ReadData <= MemRead?{29'b0,TCON}:32'b0;
        32'h4000000c: ReadData <= MemRead?{24'b0,led}:32'b0;
        32'h40000010: ReadData <= MemRead?{20'b0,digi}:32'b0;
        32'h40000014: ReadData <= MemRead?systick:32'b0;
        default: ReadData <= 32'b0;
    endcase
end

always @(posedge clk or negedge rst_n) begin
    if(~rst_n) begin
        TH <= 32'b0;
        TL <= 32'b0;
        TCON <= 3'b0;
        led <= 8'b0;
        digi <= 12'b0;
```

```

        systick <= 32'b0;
    end
    else begin
        systick <= systick + 1;
        if (MemWrite)
            case(Address)
                32'h40000000: TH <= WriteData;
                32'h40000004: TL <= WriteData;
                32'h40000008: TCON <= WriteData[2:0];
                32'h4000000c: led <= WriteData[7:0];
                32'h40000010: digi <= WriteData[11:0];
            endcase
            if(TCON[0]) begin //timer is enabled
                if(TL==32'hffffffff) begin
                    TL <= TH;
                    if(TCON[1]) TCON[2] <= 1'b1; //irq is enable
                end
            else TL <= TL + 1;
            end
        end
    end
end
end

```

关于是否读取外设等判断和读取数据的选择在MEM模块中，部分代码如下，其中EX_MEM_ALU_OUT就是地址，因而据此判断是否对外设操作并选择输出外设还是数据存储器读取的数据：

```

wire IsPeripheral = (EX_MEM_ALU_OUT == 32'h40000000) ||
(EX_MEM_ALU_OUT == 32'h40000004) || (EX_MEM_ALU_OUT == 32'h40000008) ||
(EX_MEM_ALU_OUT == 32'h4000000c) || (EX_MEM_ALU_OUT == 32'h40000010) ||
(EX_MEM_ALU_OUT == 32'h40000014);
Peripheral peripheral(
    .clk(clk),
    .rst_n(rst_n),
    .MemRead(EX_MEM_MemRead && IsPeripheral),
    .MemWrite(EX_MEM_MemWrite && IsPeripheral),
    .Address(EX_MEM_ALU_OUT),
    .WriteData(EX_MEM_rt_data),
    .ReadData(ReadData2),
    .led(led),
    .digi(digi),
    .IRQ(IRQ)
);
assign MEM_DATA = IsPeripheral?ReadData2:ReadData1;

```

转发模块在ID模块中实现，根据后续3个边界寄存器中的相关信号，产生ID阶段、EX阶段和MEM阶段需要使用到的转发控制信号。关键代码如下，其中

ForwardA_ID和ForwardB_ID是在ID阶段就需要使用到的转发信号，前者对应的是rs，后者对应的是rt，根据该信号选择从寄存器堆读取的值还是EX/MEM或者MEM/WB边界寄存器转发回来的值。ForwardA_EX和ForwardB_EX则是在EX阶段需要使用的转发信号，同样针对rs和rt，而Forward_MEM则是应对lw-sw的情况，将上一条指令读取MEM的结果（在MEM/WB边界寄存器中）转发回MEM阶段输入，由该转发控制信号选择是EX/MEM边界寄存器的输入还是MEM/WB边界寄存器转发回来的值。这3个信号在ID模块中产生后，会先写入边界寄存器中再向后传递使用。

```
always @(*) begin
    if(~rst_n) begin
        ForwardA_ID = 2'b00;
        ForwardB_ID = 2'b00;
        ForwardA_EX = 2'b00;
        ForwardB_EX = 2'b00;
        Forward_MEM = 1'b0;
    end
    else begin
        if (EX_MEM_RegWrite && (EX_MEM_WriteAddr != 5'b0) && (EX_MEM_WriteAddr == rs)) ForwardA_ID = 2'b10;
        else if (MEM_WB_RegWrite && (MEM_WB_WriteAddr != 5'b0) && (MEM_WB_WriteAddr == rs) && ((EX_MEM_WriteAddr != rs) || ~EX_MEM_RegWrite)) ForwardA_ID = 2'b01;
        else ForwardA_ID = 2'b00;
        if (EX_MEM_RegWrite && (EX_MEM_WriteAddr != 5'b0) && (EX_MEM_WriteAddr == rt)) ForwardB_ID = 2'b10;
        else if (MEM_WB_RegWrite && (MEM_WB_WriteAddr != 5'b0) && (MEM_WB_WriteAddr == rt) && ((EX_MEM_WriteAddr != rt) || ~EX_MEM_RegWrite)) ForwardB_ID = 2'b01;
        else ForwardB_ID = 2'b00;

        if (ID_EX_RegWrite && (ID_EX_WriteAddr != 5'b0) && (ID_EX_WriteAddr == rs)) ForwardA_EX = 2'b10;
        else if (EX_MEM_RegWrite && (EX_MEM_WriteAddr != 5'b0) && (EX_MEM_WriteAddr == rs) && ((ID_EX_WriteAddr != rs) || ~ID_EX_RegWrite)) ForwardA_EX = 2'b01;
        else ForwardA_EX = 2'b00;
        if (ID_EX_RegWrite && (ID_EX_WriteAddr != 5'b0) && (ID_EX_WriteAddr == rt)) ForwardB_EX = 2'b10;
        else if (EX_MEM_RegWrite && (EX_MEM_WriteAddr != 5'b0) && (EX_MEM_WriteAddr == rt) && ((ID_EX_WriteAddr != rs) || ~ID_EX_RegWrite)) ForwardB_EX = 2'b01;
        else ForwardB_EX = 2'b00;
    end
end
```

```

        if (ID_EX_RegWrite && (ID_EX_WriteAddr != 5'b0) && (ID_EX_WriteAddr == rt)) Forward_MEM = 1'b1;
        else Forward_MEM = 1'b0;
    end
end

```

在转发的基础上，仍不能够解决全部的冒险问题。在已有转发模块的基础上，当ID阶段发现是跳转或者需要执行的分支指令时（我将分支指令提前到ID阶段判断和执行），则会将IF阶段边界寄存器flush掉，同时还有一个Hazard模块用来判断数据冒险——所谓数据冒险主要是指这3类情况：处于 EX 阶段的指令需要用到前一条指令的 MEM 阶段的结果、处于 ID 阶段的指令需要用到前一条指令的 EX 或 MEM 阶段的结果、处于 ID 阶段的指令需要用到前第二条指令的 MEM 阶段的结果。Hazard模块中相关部分的代码如下：

```

//处于 EX 阶段的指令需要用到前一条指令的 MEM 阶段的结果
wire EX_MEM_Hazard = (PCSrc == 3'b000) && ~ID_MemWrite && ID_EX_MemRead && ((ID_EX_WriteAddr != 5'b0) && (ID_EX_WriteAddr == rs || ID_EX_WriteAddr == rt));

//处于 ID 阶段的指令需要用到前一条指令的 EX 或 MEM 阶段的结果
wire ID_EX_MEM_Hazard = (PCSrc == 3'b001 || PCSrc == 3'b011) && ((ID_EX_MemRead || ID_EX_RegWrite) && (ID_EX_WriteAddr != 5'b0) && (ID_EX_WriteAddr == rs || ID_EX_WriteAddr == rt));

//处于 ID 阶段的指令需要用到前第二条指令的 MEM 阶段的结果
wire ID_MEM_Hazard = (PCSrc == 3'b001 || PCSrc == 3'b011) && EX_MEM_MemRead && ((EX_MEM_WriteAddr != 5'b0) && (EX_MEM_WriteAddr == rs || EX_MEM_WriteAddr == rt));

assign DataHazard = EX_MEM_Hazard || ID_EX_MEM_Hazard || ID_MEM_Hazard;

```

发生数据冒险时，需要flush掉ID/EX边界寄存器（即将RegWrite、MemRead、MemWrite信号强制置零），同时需要stall住IF/ID边界寄存器并使得PC不能写入新值（即不更新）。这就相当于是停住IF和ID阶段的指令，等待之前指令的数据。

中断或异常处理。异常处理在本次作业中主要是处理未定义指令，因而在ID模块中的Control模块中实现检查和反馈；中断处理在本次作业中主要是处理定时器中断，也就是外设发来的中断信号，因为中断需要变更一些控制信号，因而也在ID模块的Control模块中接受外设发来的IRQ信号进行处理。发生中断和异常后会进入内核态，此时就不再处理中断或异常了。

中断只需要接收外设发来的信号，而未定义指令异常则是根据OpCode和Funct判断该指令是否为已经定义的指令，否则将Exception置为1，表示发生异常，对应判断代码如下：

```

assign Exception = ((OpCode == 6'h00 && (Funct == 6'h08 ||

```



```

Funct == 6'h09 || Funct == 6'h20 || Funct == 6'h21 || Funct == 6'h2
2 || Funct == 6'h23 ||
Funct == 6'h24 || Funct == 6'h25 || Funct == 6'h26 || Funct == 6'h2
7 || Funct == 6'h0 ||
Funct == 6'h02 || Funct == 6'h03 || Funct == 6'h04 || Funct == 6'h0
6 || Funct == 6'h2a || Funct == 6'h2b)) ||
(Opcode == 6'h04 || Opcode == 6'h05 || Opcode == 6'h06 || Opcode ==
6'h07 || Opcode == 6'h01) ||
Opcode == 6'h02 || Opcode == 6'h03 || Opcode == 6'h23 || Opcode ==
6'h2b || Opcode == 6'h0f || Opcode == 6'h08 || Opcode == 6'h09 ||
Opcode == 6'h0a || Opcode == 6'h0b || Opcode == 6'h0c || Opcode ==
6'h0d)?1'b0:1'b1;

```

在中断或异常正常发生时（即在用户态发生中断或异常，内核态发生的中断或异常不予处理，这点通过监督位实现），控制信号模块中的部分控制信号也要相应变更，比如PCSrc需要变为3'b101，从而选择下次写入PC的值会是0x80000008，以此进入异常处理函数，或变为3'b100，从而选择下次写入PC的值会是0x80000004，以此进入中断处理函数。相关代码如下，其中Supervise是PC[31]的监督位，用于判断是否处于内核态：

```

assign PCSrc[2:0] =
    ( ~Supervise && Exception)?3'b101:
    ( ~Supervise && IRQ)?3'b100:
    (Opcode == 6'h00 && (Funct == 6'h08 || Funct == 6'h09))?3'b011:
    //jr jalr
    (Opcode == 6'h02 || Opcode == 6'h03)?3'b010:          //j jal
    (Opcode == 6'h04 || Opcode == 6'h05 || Opcode == 6'h06 || OpCod
e == 6'h07 || Opcode == 6'h01)?3'b001:    //branch
    3'b000;

```

同时将RegWrite置为1，表示要写入寄存器，并将目标寄存器改为\$k0（即\$26），同时选择写入对应PC值，并且是选择PCInt传递过来的值作为该需要写入的PC值，并且控制不读写数据存储器。异常信号还需要反馈到外部，如IF模块在接收到中断或异常信号时，会保证PC可以更新并flush掉IF/ID边界寄存器，这是无视其他冒险的，即只要发生中断或者异常，不论现在是执行指令到何种情况，IF/ID边界寄存器一定会flush并且PC一定会更新入新的值。而如何将正确的PC值保存如\$k0寄存器中，以便后续需要时可以正确跳转回来，这就是之前提到过的PCInt模块所做的工作了。对应代码分布在Control模块和ID模块中，部分关键代码如下：

```

assign RegWrite = (((IRQ || Exception) && ~Supervise) ||
~(Opcode == 6'h2b || Opcode == 6'h01 || Opcode == 6'h04 || Opcode =
= 6'h05 || Opcode == 6'h06 || Opcode == 6'h07 || Opcode == 6'h02 ||
(Opcode == 6'h00 && Funct == 6'h08)))?1'b1:1'b0;
// RegDst 2'b 11 是 IRQ 或 exception, 2'b10 是 jal, 2'b00 是 rt, 2'b0
1 是 rd

```

```
assign RegDst[1:0] = ((IRQ || Exception) && ~Supervise)?2'b11:(OpCode == 6'h03)?2'b10: (OpCode == 6'h0)?2'b01:2'b00;
assign MemRead = (OpCode == 6'h23 && ~((IRQ || Exception) && ~Supervise))?1'b1:1'b0;
assign MemWrite = (OpCode == 6'h2b && ~((IRQ || Exception) && ~Supervise))?1'b1:1'b0;
// MemToReg 2'b10 是 pc, 2'b01 是 mem.out, 2'b0 是 alu.out
assign MemToReg[1:0] = ((IRQ || Exception) && ~Supervise) || OpCode == 6'h03 || (OpCode == 6'h00 && Funct == 6'h09)?2'b10: (OpCode == 6'h23)?2'b01:2'b00;
```

```
assign WriteAddr = (RegDst == 2'b00)?inst_rt:
    (RegDst == 2'b01)?inst_rd:
    (RegDst == 2'b10)?5'd31: // $ra
    5'd26; // $k0 Exception
```

汇编语言设计

本次作业中，汇编程序是基于数字逻辑与处理器理论课程的汇编作业进行修改，采用的是冒泡排序算法。由于当时编写汇编代码时并未考虑到实验课程需要，所以使用了较多的分支和跳转指令以及并未刻意避免load-use冒险，这也导致采用本代码运行得到的CPI为1.797，相对偏高，具体计算和分析过程见后。

在此处仅说明部分关键代码，完整代码可以见附件。

头部通过三条跳转指令分别跳转到MAIN函数、INTERRUPT函数和EXCEPTION函数：

```
.text
j MAIN
j INTERRUPT
j EXCEPTION
```

在MAIN函数的开始再次通过一条jr指令跳出内核态：

```
MAIN:
li $t1, 0x00000014
jr $t1
```

在INTERRUPT函数的结束会有一条jr指令跳出内核态并回到此前运行的地方继续运行：

```
jr $k0
```

而EXCEPTION函数则因为不需要返回到原先运行的地方，所以只使用了一条分支指令进行无限循环：

```
EXCEPTION:
Loop2:
beq $zero, $zero, Loop2
```

本汇编代码的主体排序部分只需要接收待排序数组大小与数组首地址即可完成排序，本次作业的主要修改在于添加外设模块与添加了中断处理函数。

出于综合仿真的方便，还在汇编中添加了加载128个随机数的函数模块（这128个数另由python代码生成，具体说明见后文），并且在开始排序之前，读取并记录了系统时钟数，关闭了中断定时器。在完成了排序之后，先读取并记录当前系统时钟数，作差得到排序运行的时钟数，待后续显示使用。

因为综合会把数据存储器模块给综合掉，所以特别采用了汇编代码的软件方式来检查代码运行正确性：在汇编中加入了检查函数，在排序完毕后，将数据存储器中的数按顺序读出，在仿真时可观察寄存器值的变化判断排序正确性。相关代码如下：

```
# 将内存中的 128 个数全部读出来检查
addi $t1, $zero, 128          # 数据个数
addi $t2, $zero, 64           # 数据基地址
CHECK:
beq $t1, $zero, ENDCHECK
addi $t1, $t1, -1
lw $t3, 0($t2)
addi $t2, $t2, 4
j CHECK
ENDCHECK:
```

检查完毕后，使能中断定时器并进入结束函数（即无限循环）。定时器发出中断信号即进入中断处理函数中，将排序运行的时钟数的低4位（16进制）轮流显示在七段数码管上，每一次中断点亮一个数码管，轮流点亮，采用查表法向七段数码管写入数据，考虑到仿真时长，因而设计刷新频率为1MHz（即每1us点亮一根数码管），执行完中断处理函数后会跳转回中断发生的指令处继续执行并退出内核态。七段数码管要查的表也在开始阶段写入到了数据存储器中，对应的数据和数码管显示的数值对应如下：

查表数据	显示数值	查表数据	显示数值
0x003f	0	0x007f	8
0x0006	1	0x006f	9
0x005b	2	0x0077	A
0x004f	3	0x007c	b
0x0066	4	0x0039	C
0x006d	5	0x005e	d
0x007d	6	0x0079	E
0x0007	7	0x0071	F

中断函数相关代码如下：

```
INTERRUPT:
addi $t5,$zero,0x0001
sw $t5,0($s4)          #定时器中断禁止
li $t9,0x000f
```

```

move $s7, $v0
and $t5,$s7,$t9      #只取执行时钟周期数的低 16 位
srl $s7,$s7,4
and $t6,$s7,$t9
srl $s7,$s7,4
and $t7,$s7,$t9
srl $s7,$s7,4
and $t8,$s7,$t9

addi $s7, $zero, 0x0000 #七段数码管表的基地址
and $s6, $s6, $t9

beq $s6, $zero, INT1
addi $t9, $zero, 0x0004
beq $s6, $t9, INT2
addi $t9, $zero, 0x0008
beq $s6, $t9, INT3
addi $t9, $zero, 0x000c
beq $s6, $t9, INT4

INT1:
sll $t5, $t5, 2
add $t5, $t5, $s7
lw $t9, 0($t5)
addi $t9, $t9, 0x0800
j SCAN
INT2:
sll $t6, $t6, 2
add $t6, $t6, $s7
lw $t9, 0($t6)
addi $t9, $t9, 0x0400
j SCAN
INT3:
sll $t7, $t7, 2
add $t7, $t7, $s7
lw $t9, 0($t7)
addi $t9, $t9, 0x0200
j SCAN
INT4:
sll $t8, $t8, 2
add $t8, $t8, $s7
lw $t9, 0($t8)
addi $t9, $t9, 0x0100
j SCAN

SCAN:

```

```
sw $t9, 0($s5)
addi $s6, $s6, 0x0004
addi $t9, $zero, 0x0003
sw $t9, 0($s4)      #使能定时器中断
jr $k0
```

辅助工具说明:

因为在生成随机数和生成装载随机数的汇编代码, 以及根据机器码编写指令存储器内容这些方面有比较大量的重复工作, 所以我额外编写了一些python代码来辅助我完成这些工作, 这也大大加快了我debug的过程, 而不至于将时间浪费在这些繁杂的重复工作上。

我编写了一个cpu_inst.py, 可以根据Mars生成的机器码, 将其转换成指令存储器InstructionMemory中always块里的内容, 代码如下:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

i = 0
f1 = open('cpu_instruction.txt', 'w+')
with open('cputestinst', 'r') as f:
    for line in f:
        line = line.strip('\n')
        print("9'd", i, ":      Instruction <= 32'h", line, ";", sep=
'', file=f1)
        i = i + 1
```

我还编写了一个cpu_random_data.py, 用于在一定范围内产生一定数量的随机数, 保存其原始数据格式及顺序, 同时将其转换为十六进制后组合成用于载入数据的汇编代码输出, 并且给出了排序后的顺序结果, 代码如下:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import random
a = []
f1 = open('cpu_random_data.txt', 'w+')
f2 = open('cpu_random_data_sorted.txt', 'w+')
f3 = open('cpu_random_data_load.txt', 'w+')
for i in range(128):
    r_int = random.randint(1, 32767)
    print(r_int, file=f1)
    a.append(r_int)
    hex_r_int = hex(r_int)
    j = 4 * i + 64
    str = f"addi $t0, $zero, {hex_r_int}\nsw $t0, {j}($s4)"
```

```
print(str, file=f3)
a.sort()
for i in range(len(a)):
    print(hex(a[i]), file=f2)
```

二、文件清单

随本报告提交的文件如下，主要包含设计代码、testbench和约束文件、汇编代码、辅助工具代码以及其他文件（工程文件 and 设计图），分别放置在对应文件夹中。

设计代码：

ALU.v 加法器模块

ALUControl.v 加法器控制模块

Control.v 控制信号模块

CPU_pipeline.v 总体CPU流水线（顶层文件，主要包括IF模块、ID模块、EX模块、MEM模块和PCInt模块）

DataMemory.v 数据存储器

EX.v EX阶段全部模块（包括加法器模块、加法器控制模块与EX/MEM边界寄存器）

Forward.v 转发信号模块

Hazard.v 数据冒险信号模块

ID.v ID阶段全部模块（包括寄存器堆模块、控制信号模块、转发信号模块、数据冒险信号模块和ID/EX边界寄存器）

IF.v IF阶段全部模块（包括指令存储器模块、PC模块和IF/ID边界寄存器）

InstructionMemory.v 指令存储器模块

MEM.v MEM阶段全部模块（包括数据存储器模块、外设模块和MEM/WB边界寄存器）

PCInt.v 中断处理用PC记录模块

Peripheral.v 外设模块（包含全部外设）

RegisterFile.v 寄存器堆模块

testbench和约束文件：

x_cpu.xdc 约束文件

test_cpu.v testbench文件

汇编代码：

sort.asm 本次作业排序汇编代码

辅助工具代码：

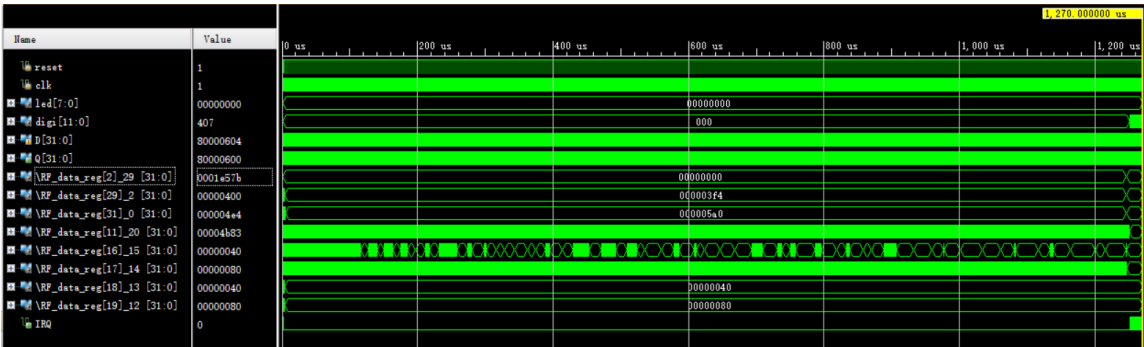
cpu_inst.py 将Mars得到的机器码转换成指令存储器中内容的辅助工具
cpu_random_data.py 生成随机数和相关装载数据的汇编代码的辅助工具

其他文件：

design.jpg 本次作业流水线CPU设计图
cpu_pipeline.rar 本次作业工程项目打包

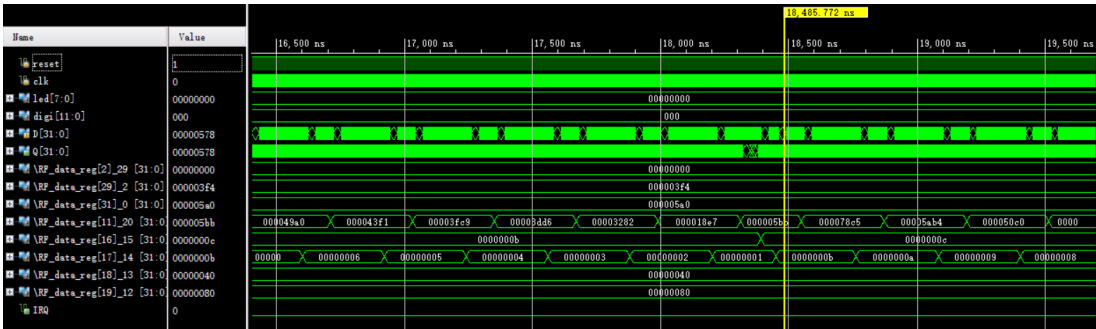
三、综合后仿情况

本次时序仿真主要针对排序功能进行验证，出于篇幅等的限制，所以主要说明排序及其正确性和中断按时发生轮流点亮数码管示数。整体波形如下，总仿真时长1270us：

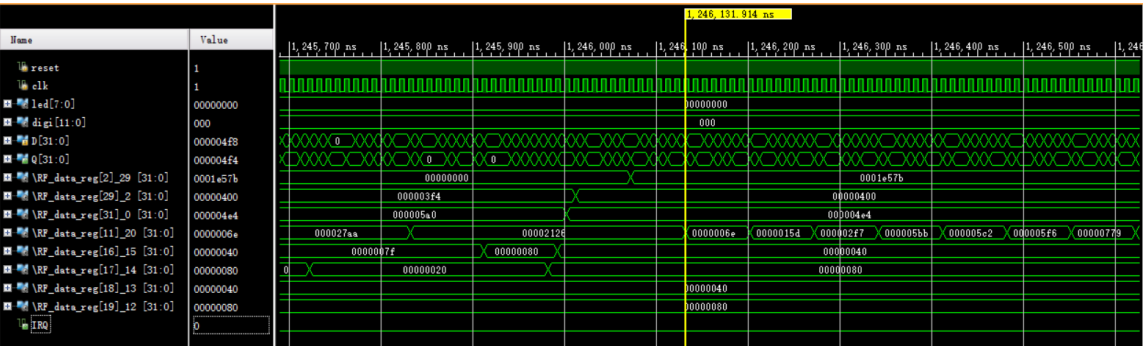


所呈现在仿真波形里的值从上到下分别为：复位信号reset，时钟clk，led灯led[7:0]，七段数码管digi[11:0]，程序计数器PC的值D[31:0]，PCInt模块存储的值Q[31:0]，2号寄存器\$vo的值RF_data_reg[2]_29，29号寄存器\$sp的值RF_data_reg[2]_29，31号寄存器\$ra的值RF_data_reg[2]_29，11号寄存器\$t3的值RF_data_reg[2]_29，16号寄存器\$s0的值RF_data_reg[2]_29，17号寄存器\$s1的值RF_data_reg[2]_29，18号寄存器\$s2的值RF_data_reg[2]_29，19号寄存器\$s3的值RF_data_reg[2]_29，中断信号IRQ

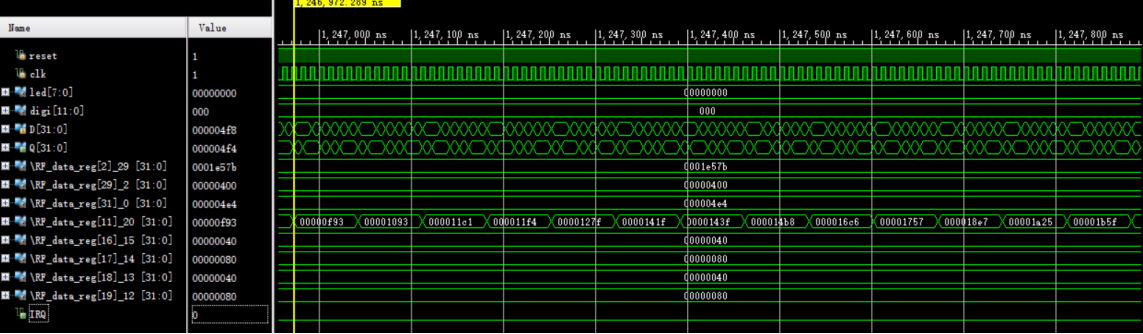
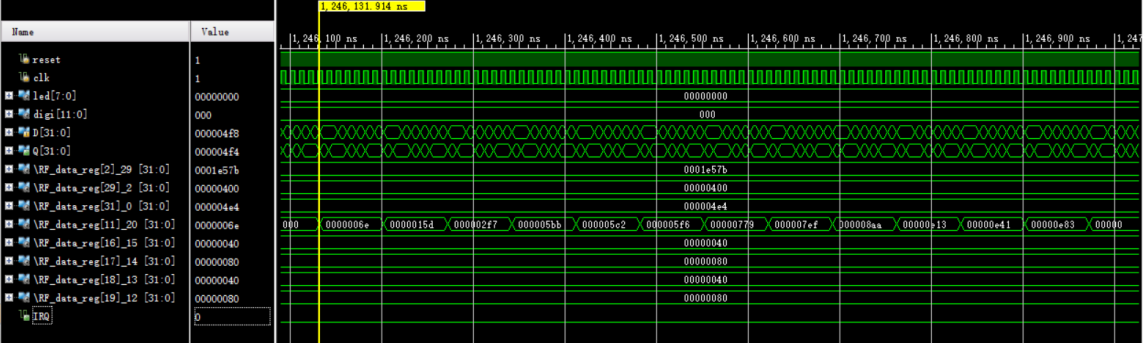
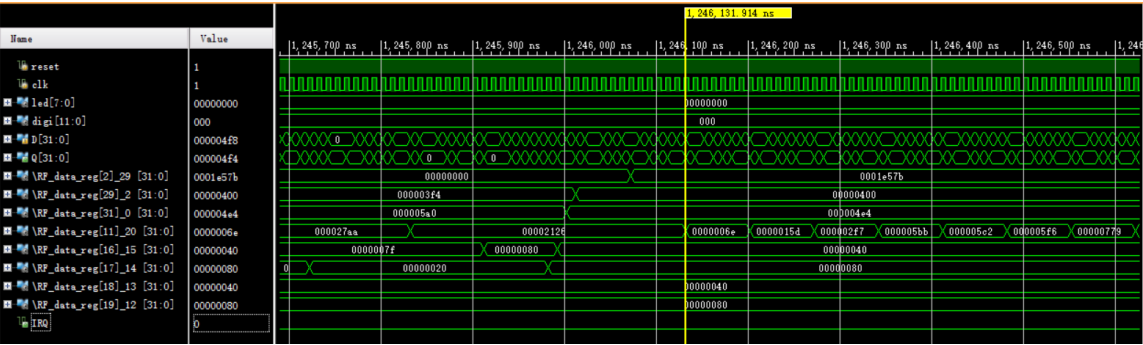
下图是仿真过程的波形截图，可以看到不断循环，数据也被不断读取出来比较或交换：

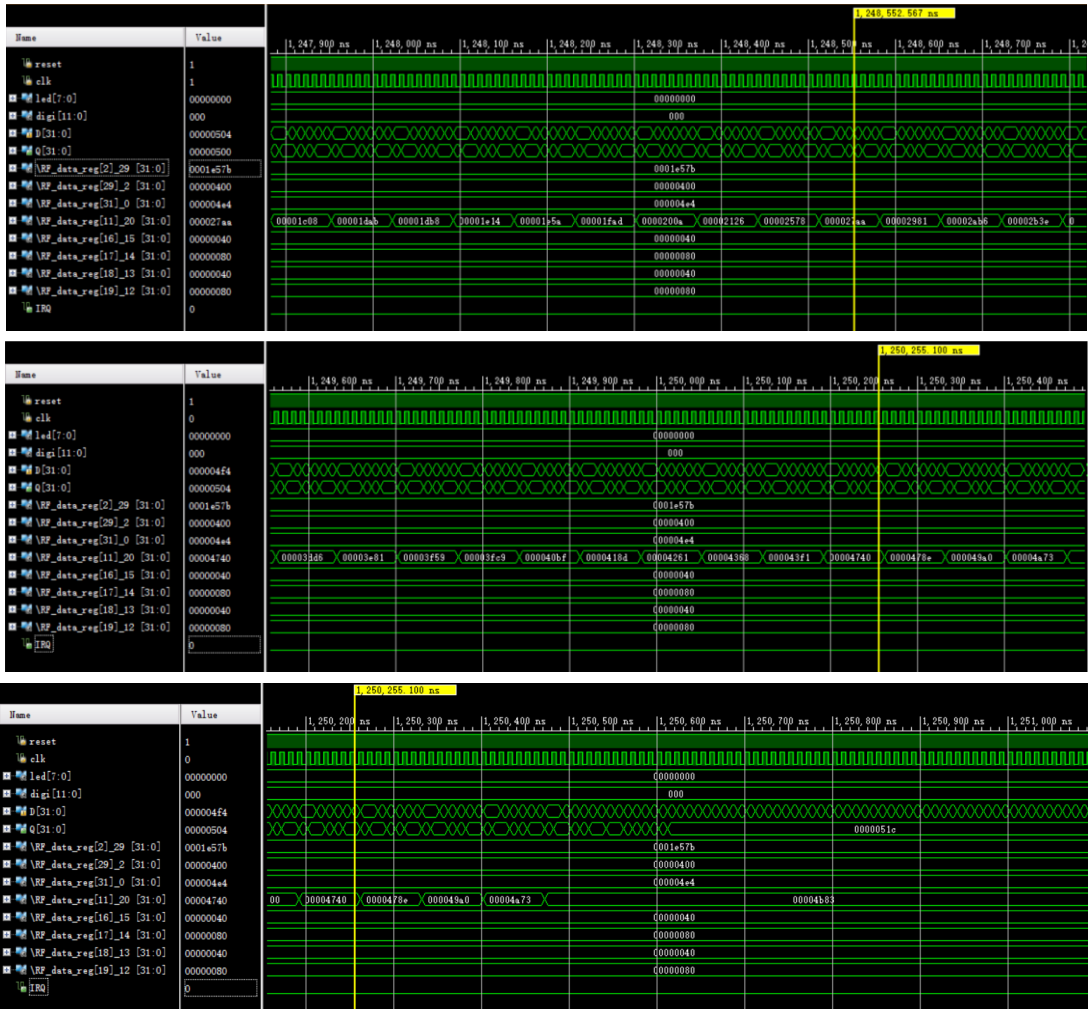


在完成排序后，会将排序的时钟周期数（即排序前后的系统时钟差值）存储到\$V0中，如下图所示，可以读出是0x0001e57b。此时\$sp的值也会回到0x00000400。

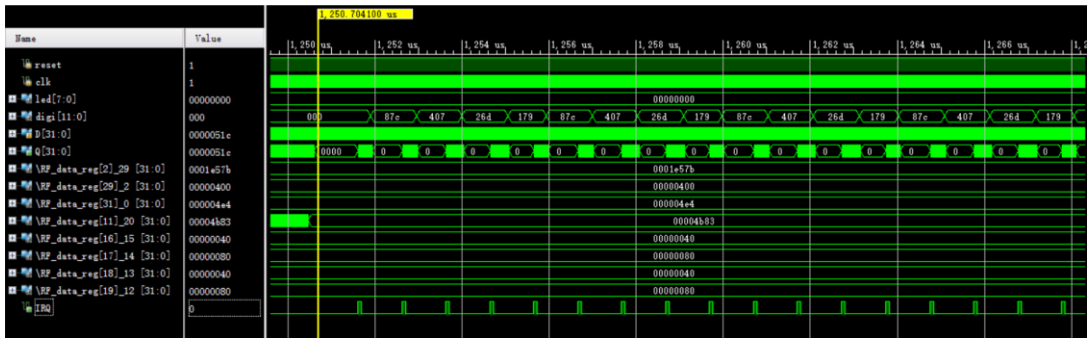


随后，根据上述汇编代码的设计，通过软件来检验排序的正确性，会将存储器中存储着的128个数按顺序读出来到\$t3中，所以观察\$t3在排序完成后的值变化即可检验排序的正确性。如下6图即选取了检验过程开始和中间和末尾的数值变化情况，可以看到其是完全按照升序排列的，并且与辅助工具生成的正确排序是一致的，由此可以验证排序的正确性。





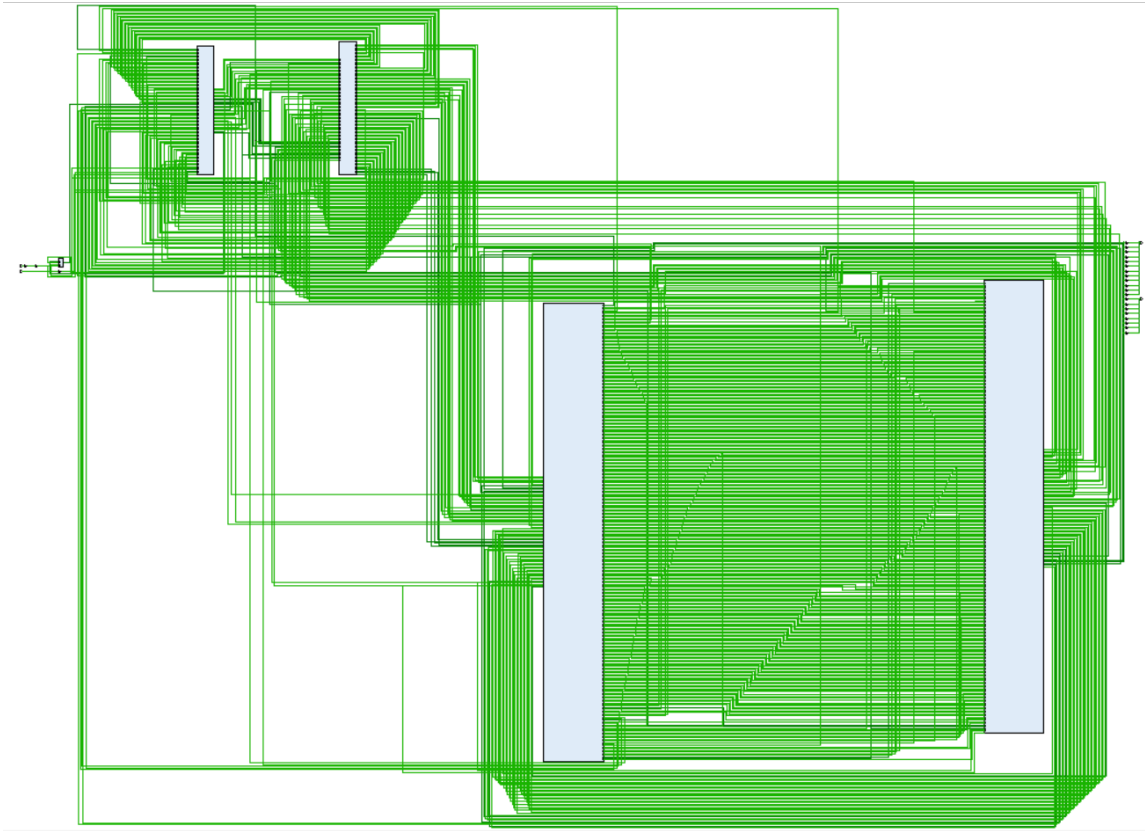
在检验排序正确性完成后，根据汇编代码可知，中断定时器就会使能，因而其也就会开始每隔1us发送一次中断信号IRQ（之前IRQ信号一直为0），这一点也可以在下图的IRQ信号变化中可以看出。当中断信号发出后，就会进入到中断处理函数中；而中断处理结束后又会回到此前的PC地址继续执行。中断处理函数就是将排序的时钟周期数（即当前存储在\$V0中的值）的低4位（十六进制）每次取1位轮流显示在数码管上。如下图所示，可以看到digi的值（十六进制）变化是从87c到407到26d到179，然后再度循环，这些值第一位就是表示点亮的数码管，8表示点亮最低位的数码管，4表示第二位数码管，2表示第三位数码管，1表示最高位数码管，而后面的数字根据此前设计中的表，可以知道就是循环刷新显示的e75b，确实是显示了需要显示的数，中断处理正确。



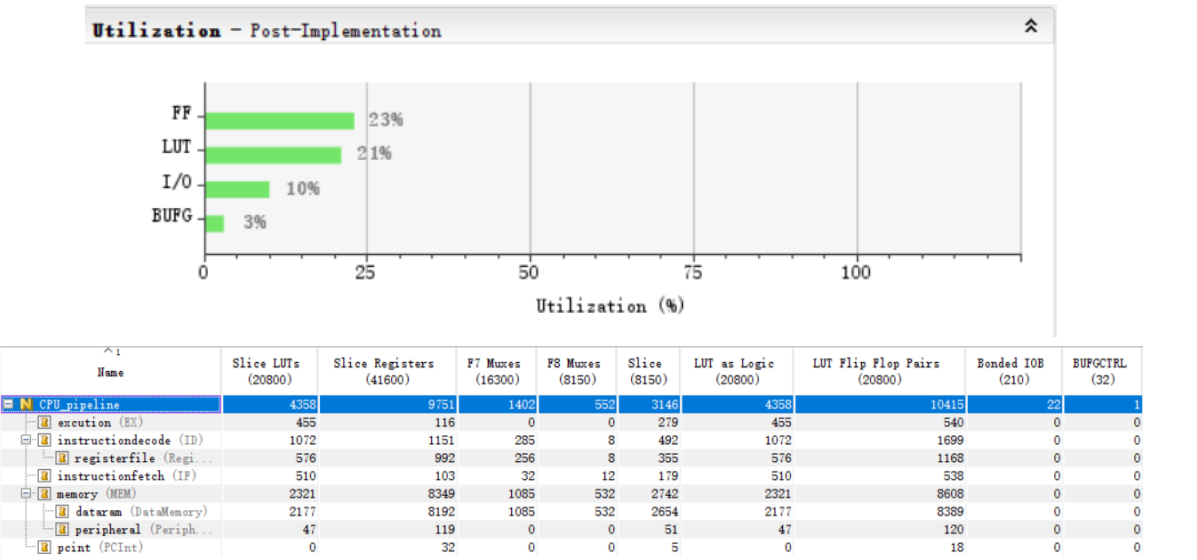
四、综合情况（面积和时序性能）

在vivado 2014.3.1中进行综合和实现，综合采用了Defaults策略，实现采用了Performance_Retiming的策略。根据实现后的相关报告，得到如下内容：

原理图：



占用资源情况：



可以看到，使用的寄存器的数量相对较多，达到了9751个，主要是数据存储器所使用，其次就主要是寄存器堆在使用；LUT的使用数量也不少，达到了4358个，相对而言仍是数据存储器占主要使用。与本人单周期作业进行对比，单周期

CPU使用了3750个LUT和8704个寄存器，可以看到使用资源稍多一些，但差距基本上并不大。

时序报告如下：

由下图可知，WNS为0.126ns，因而可以知道该CPU的理论主频为101.3MHz

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):		Worst Hold Slack (WHS):		Worst Pulse Width Slack (WPWS):	
Total Negative Slack (TNS):		Total Hold Slack (THS):		Total Pulse Width Negative Slack (TPWS):	
Number of Failing Endpoints:		Number of Failing Endpoints:		Number of Failing Endpoints:	
Total Number of Endpoints:		Total Number of Endpoints:		Total Number of Endpoints:	

All user specified timing constraints are met.

以下是时间裕度最小的10条关键路径：

Name	Slack	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Source...	Destin
Path 1	0.126	12	86	instructionfetch/IF_ID_Instruction_reg[21]_rep/C	instructionfetch/PC_reg[6]/D	9.767	3.539	6.228	CLK	CLK
Path 2	0.185	14	87	instructionfetch/IF_ID_Instruction_reg[17]_rep_0/C	instructionfetch/PC_reg[27]/D	9.794	3.140	6.654	CLK	CLK
Path 3	0.197	13	87	instructionfetch/IF_ID_Instruction_reg[17]_rep_0/C	instructionfetch/PC_reg[24]/D	9.766	3.105	6.661	CLK	CLK
Path 4	0.210	15	87	instructionfetch/IF_ID_Instruction_reg[17]_rep_0/C	instructionfetch/PC_reg[29]/D	9.805	3.231	6.574	CLK	CLK
Path 5	0.228	14	87	instructionfetch/IF_ID_Instruction_reg[17]_rep_0/C	instructionfetch/PC_reg[28]/D	9.784	3.222	6.562	CLK	CLK
Path 6	0.241	15	87	instructionfetch/IF_ID_Instruction_reg[17]_rep_0/C	instructionfetch/PC_reg[30]/D	9.771	3.346	6.425	CLK	CLK
Path 7	0.242	15	87	instructionfetch/IF_ID_Instruction_reg[17]_rep_0/C	instructionfetch/PC_reg[31]/D	9.766	3.250	6.516	CLK	CLK
Path 8	0.290	12	86	instructionfetch/IF_ID_Instruction_reg[21]_rep/C	instructionfetch/PC_reg[8]/D	9.603	3.532	6.071	CLK	CLK
Path 9	0.298	12	86	instructionfetch/IF_ID_Instruction_reg[21]_rep/C	instructionfetch/PC_reg[7]/D	9.597	3.450	6.147	CLK	CLK
Path 10	0.324	14	87	instructionfetch/IF_ID_Instruction_reg[17]_rep_0/C	instructionfetch/PC_reg[25]/D	9.690	3.114	6.576	CLK	CLK


以下是时间裕度最小的关键路径的具体信息：

Summary

Name	Path 1
Slack	0.126ns
Source	instructionfetch/IF_ID_Instruction_reg[21]_rep/C (rising edge-triggered cell FDCE clocked by CLK (rise@0.000ns fall@0.000ns period=10.000ns))
Destination	instructionfetch/PC_reg[6]/D (rising edge-triggered cell FDCE clocked by CLK (rise@0.000ns fall@0.000ns period=10.000ns))
Path Group	CLK
Path Type	Setup (Max at Slow Process Corner)
Requirement	10.000ns (CLK rise@10.000ns - CLK rise@0.000ns)
Data Path Delay	9.767ns (logic 3.539ns (36.235%) route 6.228ns (63.765%))
Logic Levels	12 (CARRY4=4 LUT3=1 LUT5=1 LUT6=5 MUXF7=1)
Clock Path Skew	-0.104ns
Clock Uncertainty	0.035ns

Source Clock Path

Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
(clock CLK rise edge)	(r) 0.000	0.000		
	(r) 0.000	0.000	Site: P17	clk
net (fo=0)		0.000		clk
			Site: P17	clk_IBUF_inst/I
IBUF (Prop ibuf I 0)	(r) 1.478	1.478	Site: P17	clk_IBUF_inst/O
net (fo=1, routed)		1.972		clk_IBUF
			Site: BUFCTRL_X0Y0	clk_IBUF_BUFG_inst/I
BUFG (Prop bufg I 0)	(r) 0.096	3.546	Site: BUFCTRL_X0Y0	clk_IBUF_BUFG_inst/O
net (fo=9751, routed)		1.633		instructionfetch/clk_IBUF_BUFG
			Site: SLICE_X59Y37	instructionfetch/IF_ID_Instruction_reg[21]_rep/C

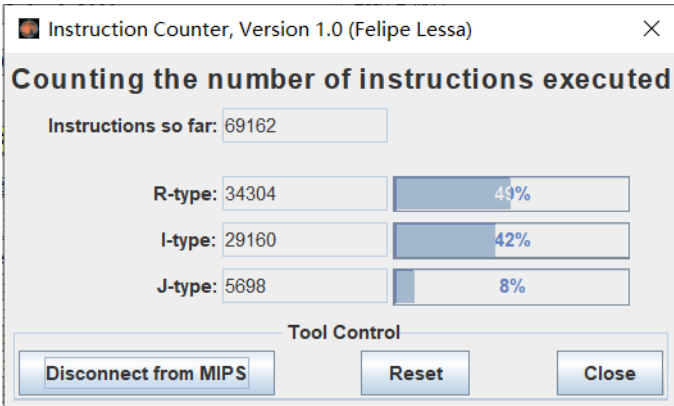
Data Path				
Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
FDCE (Prop fdce C Q)	(r) 0.456	5.635	Site: SLICE_X59Y37	 instructionfetch/IF_ID_Instruction_reg[21]_rep/Q
net (fo=86, routed)	1.795	7.431		 instructiondecode/registerfile/I11
			Site: SLICE_X42Y31	 instructiondecode/registerfile/ID_EX_rs_data[19]_i_7/I4
LUT6 (Prop lut6 I4 0)	(r) 0.124	7.555	Site: SLICE_X42Y31	 instructiondecode/registerfile/ID_EX_rs_data[19]_i_7/0
net (fo=1, routed)	0.000	7.555		 instructiondecode/registerfile/n_0_ID_EX_rs_data[19]_i_7
			Site: SLICE_X42Y31	 instructiondecode/registerfile/ID_EX_rs_data_reg[19]_i_3/I0
MUXF7 (Prop muxf7 I0 0)	(r) 0.209	7.764	Site: SLICE_X42Y31	 instructiondecode/registerfile/ID_EX_rs_data_reg[19]_i_3/0
net (fo=1, routed)	1.159	8.923		 instructiondecode/registerfile/n_0_ID_EX_rs_data_reg[19]_i_3
			Site: SLICE_X46Y33	 instructiondecode/registerfile/ID_EX_rs_data[19]_i_2/I0
LUT6 (Prop lut6 I0 0)	(r) 0.297	9.220	Site: SLICE_X46Y33	 instructiondecode/registerfile/ID_EX_rs_data[19]_i_2/0
net (fo=1, routed)	0.552	9.772		 instructionfetch/I60
			Site: SLICE_X48Y35	 instructionfetch/ID_EX_rs_data[19]_i_1/I5
LUT6 (Prop lut6 I5 0)	(r) 0.124	9.896	Site: SLICE_X48Y35	 instructionfetch/ID_EX_rs_data[19]_i_1/0
net (fo=3, routed)	0.868	10.763		 instructionfetch/jr_target[19]
			Site: SLICE_X54Y34	 instructionfetch/PCInt[31]_i_34/I3
LUT6 (Prop lut6 I3 0)	(r) 0.124	10.887	Site: SLICE_X54Y34	 instructionfetch/PCInt[31]_i_34/0
net (fo=1, routed)	0.000	10.887		 instructionfetch/n_0_PCInt[31]_i_34
			Site: SLICE_X54Y34	 instructionfetch/PCInt_reg[31]_i_17/S[2]
CARRY4 (Prop carry4 S[2] C0[3])	(r) 0.380	11.267	Site: SLICE_X54Y34	 instructionfetch/PCInt_reg[31]_i_17/C0[3]
net (fo=1, routed)	0.000	11.267		 instructionfetch/n_0_PCInt_reg[31]_i_17
			Site: SLICE_X54Y35	 instructionfetch/PCInt_reg[31]_i_5/CI
CARRY4 (Prop carry4 CI C0[2])	(r) 0.229	11.496	Site: SLICE_X54Y35	 instructionfetch/PCInt_reg[31]_i_5/C0[2]
net (fo=70, routed)	0.784	12.280		 instructionfetch/instructiondecode/data4
			Site: SLICE_X54Y36	 instructionfetch/PC[1]_i_5/I1
LUT5 (Prop lut5 I1 0)	(r) 0.310	12.590	Site: SLICE_X54Y36	 instructionfetch/PC[1]_i_5/0
net (fo=1, routed)	0.000	12.590		 instructionfetch/n_0_PC[1]_i_5
			Site: SLICE_X54Y36	 instructionfetch/PC_reg[1]_i_2/S[1]
CARRY4 (Prop carry4 S[1] C0[3])	(r) 0.533	13.123	Site: SLICE_X54Y36	 instructionfetch/PC_reg[1]_i_2/C0[3]
net (fo=1, routed)	0.000	13.123		 instructionfetch/n_0_PC_reg[1]_i_2
			Site: SLICE_X54Y37	 instructionfetch/PC_reg[12]_i_4/CI
CARRY4 (Prop carry4 CI 0[1])	(r) 0.323	13.446	Site: SLICE_X54Y37	 instructionfetch/PC_reg[12]_i_4/0[1]
net (fo=1, routed)	0.667	14.113		 instructionfetch/branch_target[6]
			Site: SLICE_X53Y37	 instructionfetch/PC[6]_i_2/I0
LUT3 (Prop lut3 I0 0)	(r) 0.306	14.419	Site: SLICE_X53Y37	 instructionfetch/PC[6]_i_2/0
net (fo=1, routed)	0.403	14.822		 instructionfetch/n_0_PC[6]_i_2
			Site: SLICE_X53Y38	 instructionfetch/PC[6]_i_1/I5
LUT6 (Prop lut6 I5 0)	(r) 0.124	14.946	Site: SLICE_X53Y38	 instructionfetch/PC[6]_i_1/0
net (fo=1, routed)	0.000	14.946		 instructionfetch/n_0_PC[6]_i_1
FDCE			Site: SLICE_X53Y38	 instructionfetch/PC_reg[6]/D
Arrival Time		14.946		
Destination Clock Path				
Delay Type	Incr (ns)	Path (ns)	Location	Netlist Resource(s)
(clock CLK rise edge)	(r) 10.000	10.000		 clk
	(r) 0.000	10.000	Site: P17	 clk
net (fo=0)	0.000	10.000		 clk_IBUF_inst/I
			Site: P17	 clk_IBUF_inst/0
IBUF (Prop ibuf I 0)	(r) 1.408	11.408	Site: P17	 clk_IBUF
net (fo=1, routed)	1.868	13.276		 clk_IBUF_BUFG_inst/I
			Site: BUFCTRL_X0Y0	 clk_IBUF_BUFG_inst/0
BUF6 (Prop buf6 I 0)	(r) 0.091	13.367	Site: BUFCTRL_X0Y0	 instructionfetch/clk_IBUF_BUFG
net (fo=9751, routed)	1.449	14.816		 instructionfetch/PC_reg[6]/C
			Site: SLICE_X53Y38	
clock pessimism	0.259	15.075		
clock uncertainty	-0.035	15.040		
FDCE (Setup fdce C D)	0.032	15.072	Site: SLICE_X53Y38	 instructionfetch/PC_reg[6]
Required Time		15.072		

根据此数据通路可以推测本流水线CPU的关键路径应该是在执行jr指令。对于这一方面，如果将jr指令放到EX阶段执行，我认为可以优化时序性能，但是设计要求在ID阶段进行跳转，因而就不再额外优化这一方面了。

CPI计算

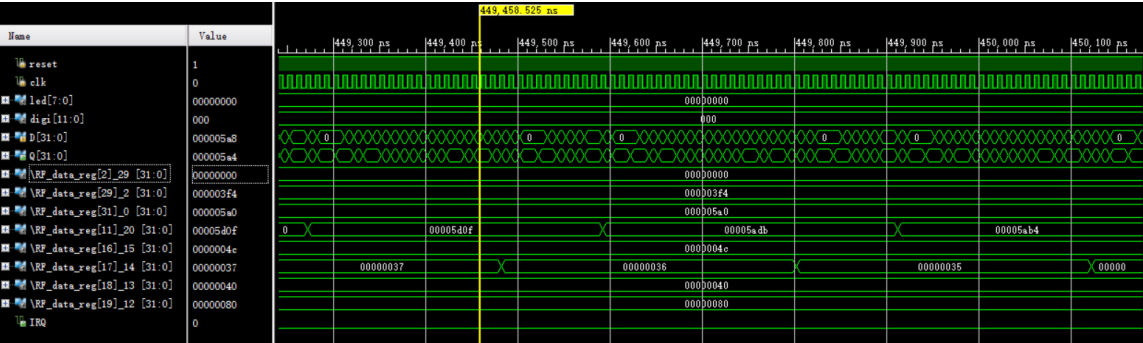
根据上述后仿过程，可以知道本流水线在本汇编情况下执行的排序时钟周期数为0x0001e57b，转换成十进制为124283。

修改部分汇编代码使得其能够在Mars中运行，统计运行的指令数为69162：



由此可以计算得到：CPI=124283/69162=1.797

可以看到CPI是相对偏高的，对于这一点，我个人分析是因为汇编代码排序部分在最初撰写的时候并未考虑到需要为CPU流水线服务，因而存在不少的分支和跳转指令，并且未优化load-use冒险，所以导致CPI相对偏高。这一点查看后仿记录当前PC和PCInt的波形也可以看到，较多出现stall或flush的情况：



其中的D[31:0]即当前的PC，而Q[31:0]则记录着当前进入ID阶段的指令的PC（发生stall或flush时该值并不发生变化）

由上述数据也可以计算得到平均每秒执行指令数目为：56.37M

这一方面我认为通过更换排序算法或者重新构建汇编指令即可得到较好的改善。

五、经验体会

这一次的作业让我对于流水线CPU有了更加深入和进一步的了解，也让我对于本作业涉及到的流水线数据通路的工作细节方面都有所掌握。同时，这一次流水线CPU也是建立在之前的单周期CPU的基础上设计的，所以两者之间异同的比较也让我有了不少收获和新的认识。这一次流水线CPU让我着重认知和理解了转发冒险、中断异常、外设等的设计处理，从无到有逐步调试逐步完成，还是很有收获和成就感的。

我在一开始就阅读了网络学堂上提供的往届的心得体会，同时认识到了设计需求的复杂性，我意识到了应该先从总体架构设计开始，先进行整体框架和各个模块的设计，而非一开始就钻入具体代码编写。所以我也是将整体功能按照阶段拆解开来，再逐步细化，然后重新统筹合并或者补充一些模块，这样使得我对于整体功能和实现（特别是一些跨模块的功能实现）有了较为清晰的认知和把握，从而顺利完成了设计。

本次实验中，根据要求通过软件和硬件结合的方式实现了诸多功能，或许仍有一些还存在待完善或者待优化的方面（比如可以重写汇编代码以获得更低的CPI），但是总体上让我对于相关设计和实现有了比较深刻的认识和理解。同时有部分软件实现功能与此前硬件实现进行了一定的对比印证，认知了其中的异同。

由于本次作业的软硬件调试比较复杂，所以我并没有一开始就采用完备的汇编指令调试，而是先行使用了单周期CPU作业中使用过的汇编代码进行调试，与之前的结果进行比较，用于检验正确性，从而较快也较方便的发现问题。而在调试通之后，要添加中断异常处理和外设调用等软件部分的代码，因为考虑到主体和这些部分是解耦的，所以我也是先在单周期汇编代码的基础上添加了这些部分并完成调试后，再将主体换成排序代码进行最终调试。这样的过程虽然看似多了一步，但是因为汇编代码本身相对简单，更容易发现问题，所以debug的过程也是比较顺利和快速的。

此外，我还自己编写了一些python辅助工具，用于帮助生成代码，将一些重复繁琐的工作交给程序实现，方便自己调试代码。

本次流水线CPU的设计总体来说是很意义的，不光让我对于流水线的设计实现细节有了比理论课更进一步的理解和认识，同时也让我体会到了数字电路的设计方法原理思路，对于数字电路的设计思想也有了更多的理解，这将来也会有益于我的学习科研。

最后要感谢老师和助教的帮助和指导，让我能够在编写大作业的过程中收获很多新知识技巧，并顺利完成课程设计。