实验二：单周期CPU的IO模块设计

蒋傲凡 518030910275

1 核心代码及设计思路

1.1 输入端口

从提供的仿真结果可以看出，输入为两个五位二进制的数，即总共有十个位输入，从SW0到SW9。此外，还有reset控制信号以及总的时钟信号。因为CPU需要的是32位的输入，所以要将两个五位的输入转换为32位，通过模块in_port.v实现，即在两个输入前补零

输入的reset信号可以直接传递到子模块，而时钟信号则要进行二分频转换得到另一个信号共同作用控制CPU内部执行。则通过总时钟信号的上升沿判断，每次上升沿即翻转信号，由此得到二分频时钟，具体代码见clock_and_mem_clock.v文件。

1.2 输出端口

IO设备的输出通过三个output_port端口。按照仿真图的信息，不需要对其额外的处理。但根据任务要求，在这里对其进行十进制转换并通过七段译码器输出。首先通过取余和除十对输出数据进行算术运算得到其十位数与个位数。以个位数为例，它现在是四位表示的二进制数，对其进行case的switch，根据不同的数值与七段LED显示进行对应得到一个七位的输出代表七个输出指示灯的亮暗状态。具体代码见out_port_seg.v文件

1.3 数据存储器

这一部分主要是参看提供的参考代码。对存储器增加了两个32位输入端口以及三个32位输出端口。通过对地址第八位的判断来确定输出数据是来源于内部存储器还是IO。当数据由IO输入至CPU时，首先将两个输入口的数据传输至两个对应的寄存器，再通过指令值判断输入寄存器号是0还是1，最后通过某一寄存器传输至CPU。当数据由CPU输出至IO时，同样通过指令值判断输出端口。这一部分与参考代码一致，唯一不同的是在输出模块中添加了reset信号，当reset信号为0时，三个输出端口的值置为零，修改代码见io_output.v文件

1.4 代码截图

以下为代码截图

```
module in_port(SW4,SW3,SW2,SW1,SW0,in_port0);
    input SW4,SW3,SW2,SW1,SW0;
    output [31:0] in_port0;
    assign in_port0={27'b0,SW4,SW3,SW2,SW1,SW0};
endmodule
```

Figure 1: 输入端口部分代码

```
module clock_and_mem_clock(clk,clock_out);
    input clk;
    output clock_out;
    reg clock_out;
    initial clock_out = 0;

    always @(posedge clk) clock_out <= ~clock_out;

endmodule
```

Figure 2: 二分频部分代码

2 整体仿真结果

仿真结果和提供文件一致，四五六寄存器分别寄存的是两个输入端口的值以及它们的和。二分频产生的clock_out频率确实为主时钟频率的一半。而经过七段译码器的输出值与三个输出端口的值同样符合。例如0号输出端口第一次输出值为1，其高位为0，低位为1。1对应的七段码为1111001，将其转换为十进制后即为79，与仿真输出值相符。具体实验仿真结果截图见下

3 选做实验部分

选做的实验部分是自己设计一条新的指令并且仿真验证。在CPU内部设计阶段，主要修改的是alu.v文件以及cu.v文件。在alu中，对新的指令设计一个新的四位控制信号，通过对控制信号筛选，当信号对应该指令时，利用Verilog实现功能。例如在求hamming距离时，先对输入alu的两个数进行异或操作，再将各位数相加就求出了汉明距离。在cu中，增加新指令对应的六位opcode。同时修改aluc信号包含的指令，包括wreg信号对应的指令。自此即完成新指令的设计阶段

~~在仿真实现阶段存在问题，不知道如何添加新指令的32位码以及对应操作。虽然利用老师提供的编译应用可以根据汇编语言生成.mif文件，但现有的汇编语言并不能支持新的指令，或者说是我没有完全掌握汇编语言。因此选做部分卡在了仿真设计上，之前的设计构想也无法验证正确与否。~~

3.1 设计思路

对新的求hamming距离指令，我设计的指令格式为 hamm $7, $4, $5,其op码为000000，func码为000001，aluc的控制信号为1011.该指令设计属于R型指令，其三十二位指令码为000000 00100 00101 00111 000001。在IO模块的基础上需要修改的包括三个文件，alu.v，cu.v以及instmem.mif文件。alu.v和cu.v文件的修改与上文思路一致，具体代码附在文件夹中。mif文件的修改可通过Quartus修改，将三十二位的指令码每四个一组，转换为八位即00853801，插入到mif中作为一条新的指令。同时，由于新指令的加入，mif文件中的宽度DEPTH也要增加，这里从16增至64.

3.2 仿真结果

新加入的指令将四号与五号寄存器的值进行汉明距离的运算并将结果存放在七号寄存器。由于在IO实验的mif仿真中只添加了一条新的指令，总体结果与IO部分相似。在细节图上，当4,5号寄存器的值为1和2时，其二进制表示为01和10，hamming距离为2，与7号寄存器结果相同；当4,5号寄存器的值为3和4时，其二进制表示为011和100，hamming距离为3,模拟结果同样符合。

```
module out_port_seg(out_port0,HEX0,HEX1);
   input [31:0] out_port0;
   output reg [6:0] HEX0,HEX1;
   wire [3:0] high=out_port0/10;
   wire [3:0] low=out_port0-high*10;
   initial
   begin
      HEX0 = 0;
      HEX1 = 0;
   end
   always @(*)
      begin
      case(low) //1为熄灭，0为亮起。七位数对应七段位为 gfedcba
         0: HEX0[6:0] = 7'b1000000;
         1: HEX0[6:0] = 7'b1111001;
         2: HEX0[6:0] = 7'b0100100;
         3: HEX0[6:0] = 7'b0110000;
         4: HEX0[6:0] = 7'b0011001;
         5: HEX0[6:0] = 7'b0010010;
         6: HEX0[6:0] = 7'b0000010;
         7: HEX0[6:0] = 7'b1111000;
         8: HEX0[6:0] = 7'b0000000;
         9: HEX0[6:0] = 7'b0010000;
         default: HEX0[6:0] = 7'b1111111;
      endcase
      end
    always @(*)
       begin
       case(high)
          0: HEX1[6:0] = 7'b1000000;
          1: HEX1[6:0] = 7'b1111001;
          2: HEX1[6:0] = 7'b0100100;
          3: HEX1[6:0] = 7'b0110000;
          4: HEX1[6:0] = 7'b0011001;
          5: HEX1[6:0] = 7'b0010010;
          6: HEX1[6:0] = 7'b0000010;
          7: HEX1[6:0] = 7'b1111000;
          8: HEX1[6:0] = 7'b0000000;
          9: HEX1[6:0] = 7'b0010000;
          default: HEX1[6:0] = 7'b1111111;
       endcase
       end

endmodule
```

Figure 3: 输出端口部分代码

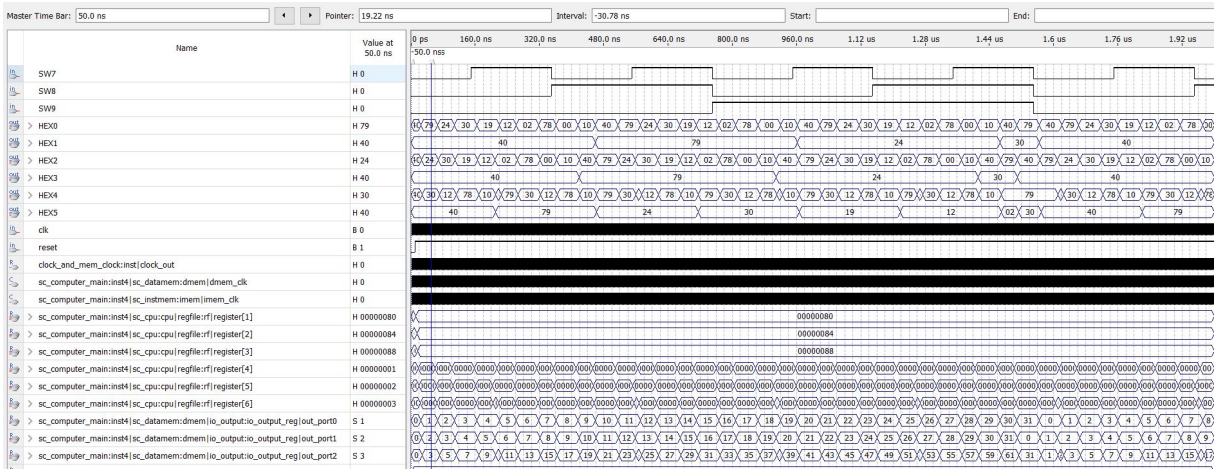

Figure 4: IO总体图

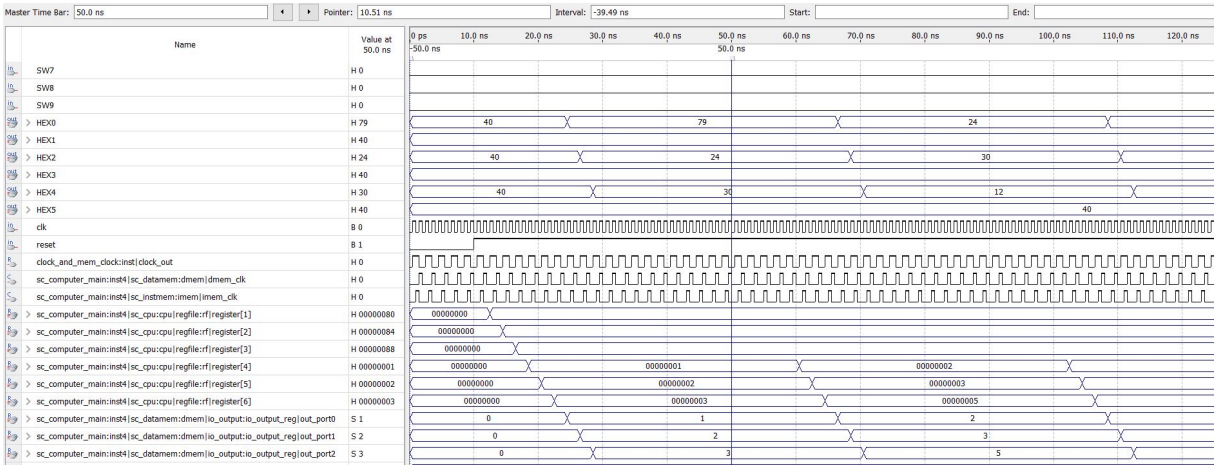

Figure 5: IO细节图

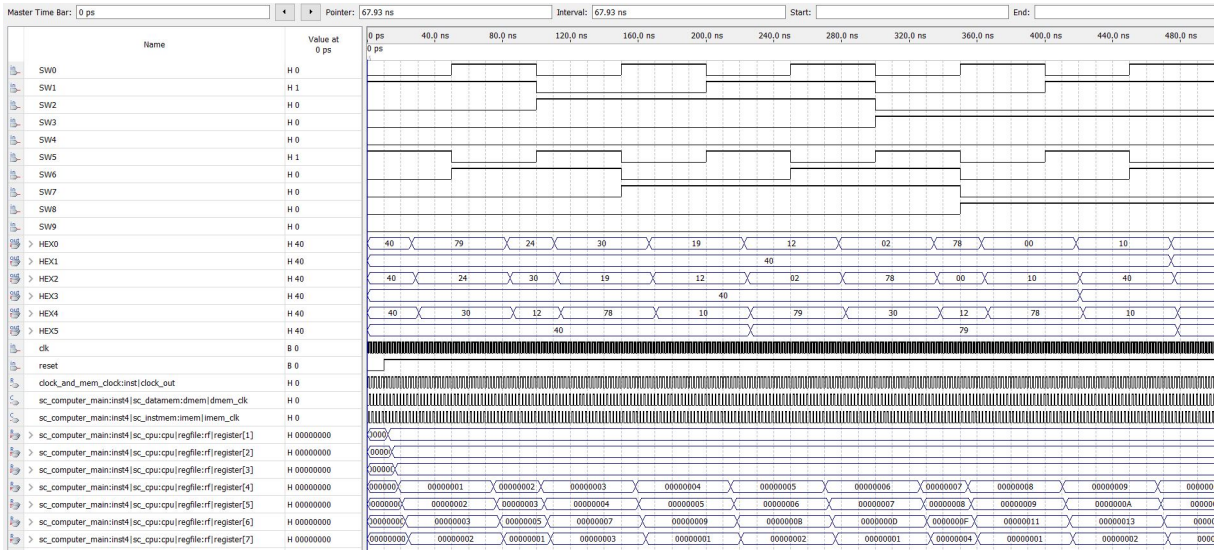

Figure 6: 新指令总体图

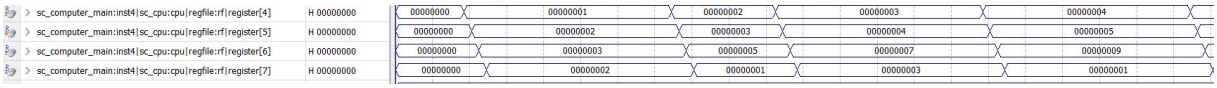

Figure 7: 新指令细节图

3.3 问题分析

之前卡在了仿真设计阶段，本来以为是要先用汇编实现再利用那个编译软件生成mif文件。多亏了陈老师的指导，原来在设计好32位的指令码之后，可以直接在mif文件中修改增加新的指令。之前看mif文件里面的代码都是八位，还以为和32位的指令码不同，但实际上是表示的是一样的，即将32位每四个一组计算出对应的十进制数实现32位指令码转mif文件中的指令。