实验 2：单周期 MIPS 处理器

1 实验目的

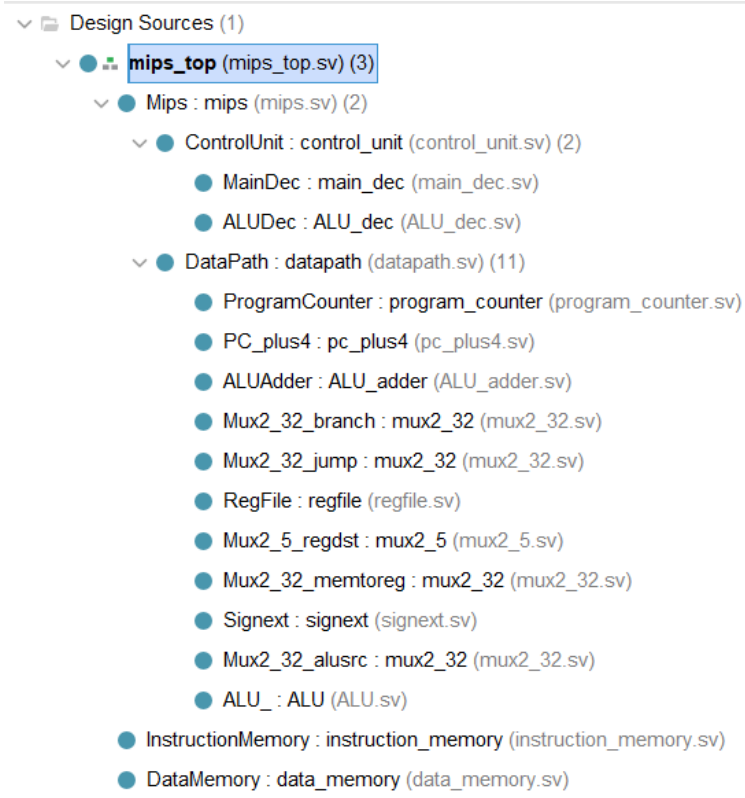
1. 熟悉 Vivado 软件；
2. 熟悉在 Vivado 软件下进行硬件设计的流程；
3. 设计单周期 MIPS 处理器，包括
   a) 完成单周期 MIPS 处理器的设计；
   b) 在 Vivado 软件上进行仿真；
   c) 编写 MIPS 代码验证单周期 MIPS 处理器；
   d) 在 NEXYS4 DDR 板上进行验证

2 实验内容

2.1 设计单周期 mips 处理器

实验结果

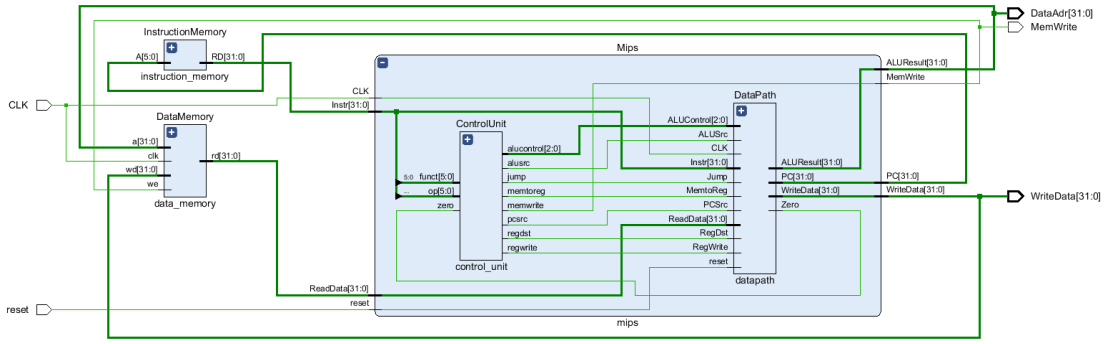
完成对与所有要求的指令的支持，封装为顶层文件，文件结构如下：



具体实现

本阶段主要参考 ppt 和《数字设计》一书。

最终的 RTL 图如下：



首先实现 PC 寄存器、指令存储器和数据存储器和寄存器文件。

pc 寄存器是一个带异步复位信号的 D 触发器，复位时置零，在时钟上升沿将输入的 pc_next 赋值给输出 pc：

```
always@(posedge clk, posedge res)
    begin
        if(res)
            pc<=0;
        else
            pc<=pc_next;
    end
```

指令寄存器需要一块 64*32bits 的的空间，根据输入的地址 A 输出相应位置的指令。在激励信号的作用下，会在开始时读入需要的指令文件。注意到由于 A 每次自增 1，就代表取得下一条指令，所以在输入时将 8 位 pc 的最后 2 位（始终为 0）丢弃，就可以在 pc 实际自增 4 时变为只会自增 1

```
logic [31:0] RAM[63:0];

initial
begin
    $readmemh("memfile_moreinstra.dat",RAM);
end

assign RD=RAM[A];
```

数据存储器（内存）也需要 64*32bits 的大小，根据读入的 a，输出相应的内容。并且在写使能为真时，时钟上升沿时对地址为 a 的数据进行写入。

```
logic [31:0] RAM[63:0];
assign rd=RAM[a[31:2]];

always_ff@(posedge clk)
    if(we)
        RAM[a[31:2]]<=wd;
```

寄存器文件开辟 32*32bit 的空间。在时钟上升沿时，如果寄存器写使能为真，就对相应的寄存器写入数据。两个输出与两个输入地址相匹配，并且输入地址为 0 时始终返回 0

```
    logic [31:0] rf[31:0];

    always_ff@(posedge clk)
        if(we3==1)
            rf[wa3]<=wd3;

    assign rd1=(ra1!=0)?rf[ra1]:0;
    assign rd2=(ra2!=0)?rf[ra2]:0;
```

接下来是根据 lw 指令，来对上述模块连接。由于 lw 指令需要一个加法器件，于是使用先增加一个 ALU 模块，根据 ALUControl 的取值做出相应的运算。首先实现加法，对应位 010。同时还需要实现 sub、and、or、slt。同时为了后面 beq 和 bne 指令，还需要一个 Zero 标志位，在运算结果为 0 时置 1

```
    always@(*)
begin
    case(ALUControl)
      3'b000: ALUResult <= A&B;
      3'b001: ALUResult <= A|B;
      3'b010: ALUResult <= A+B;
      3'b110: ALUResult <= A-B;
      3'b111: ALUResult <= A<B ? 1:0;//slt
      3'b101:ALUResult <= ~(A|B);
      default: ALUResult <= 0;
    endcase
end

assign Zero = (ALUResult==0);
```

然后需要对读入的立即数做出符号扩展，由 16 位扩展至 32 位。就是将最高位重复 16 次并与原数拼接。

接下来开始按照 lw 指令的数据流连接各个部件，由于需要写回寄存器，所以需要寄存器 RegWrite 写使能参与。并且再加上 pc 自增 4 的模块，用于更新 pc。

然后对于 sw 指令的数据流，再次连接部件。由于需要写入内存，就增加内存写使能控制器 MemWrite。

接下来是 r 型指令。由于需要写入寄存器，并且指令中写入寄存器的信息与之前 lw 写入寄存器的信息在 Instra 中位置不一样，于是需要一个二选一复用器，在 RegDst 的控制下选择合适的 Intra 部分作为写入寄存器。除此之外，ALU 的第二个操作数也需要一个复用器，在 ALUSrc 的判断下选择是从寄存器中读出的数据还是 Instra 中的符号扩展结果。由于 ALU 运算结果直接作为结果写回寄存器，不需要经过内存，就再增加一个复用器，在 MemtoReg

控制下选择是从内存中读出的数据还是 ALU 的运算结果作为写回寄存器的内容。

上述的复用器有 5 位的，也有 32 位的，实验中用 mux2_32 和 mux2_5 来实现。

再对 beq 指令做出扩展。经过符号扩展的立即数左移两位后加上 pc 自增 4 后的结果作为 PCBranch 的内容。增加一个复用器，再 PCSrc 的控制下判断是选择正常 PC 自增 4 还是 PCBranch 作为新的 PC。其中，这个加法没有使用 ALU，而是用一个专门的部件来实现加法。

接下来完成控制器，由 2 部分组成，分别是主译码器和 ALU 译码器。

主译码器根据读入 Instra 的 Opcode 部分对于先前提到的 RegWrite、RegDst、ALUSrc、Branch、MemWrite、MemtoReg 做出控制，并产生 ALUOp，在非 R 型指令时控制 ALU 的操作，当其为 010 时代表为 R 型指令。

ALU 译码器读入 Instra 的 funct 段，根据主译码器产生的 ALUOp，输出相应的 ALUControl 控制 ALU。

由于还未实现全部指令，主译码器和 ALU 译码器的相关代码将在所有指令实现后展示。

随后将主译码器和 ALU 译码器封装为控制单元，其中输出的 branch 由 ALU 的 Zero 标志和 Branch 共同决定。

此时基本的 CPU 已经完成。现在开始拓展更多指令。

首先拓展 addi 指令，只用在主译码器中加入此指令即可，现有数据路径可以完成此操作。

然后是 j 指令。首先数据路径需要增加一个复用器，根据 Jump 控制真正的 PC 是来源于立即数经过左移和扩展得到的 PCJump 还是先前已存在的 PC_next。同时还需要在主译码器、控制单元中增加新的输出项 Jump。

对于 andi、ori、slti 指令，除了 ALUOp 不一样，其它输出信号完全与 addi 一致，在主译码器和 ALU 译码器中增加相应的语句即可实现。

对于 bne 指令，在控制单元中增加一个 Branch_standard，当 Zero 标志位与 Branch_standard 相等时，并且 Branch 为 1 时，将 PCSrc 置为 1，代表需要 Branch。（这一点与 ppt 上的实现方法不一样）

最后是 nop 指令，不需进行任何操作，将所有控制单元全置为 0 即可。

至此，已完成对于所有指令的支持。主译码器和 ALU 译码器相关代码如下：

主译码器：

```
    logic [9:0] control;
    logic branch_standard;

    assign {regwrite,regdst,alusrc,branch,memwrite,memtoreg,aluop,jump}
= control;

    always@(*)
        case(op)
            6'b000000: control<=10'b1100000100;//R
```

```
            6'b100011: control<=10'b1010010000;//lw
            6'b101011: control<=10'b0010100000;//sw
            6'b000100: control<=10'b0001000010;//beq
            6'b000101: control<=10'b0001000010;//bne
            6'b000010: control<=10'b0000000001;//jump
            6'b001000: control<=10'b1010000000;//addi
            6'b001100: control<=10'b1010001000;//andi
            6'b001101: control<=10'b1010001010;//ori
            6'b001010: control<=10'b1010001100;//slti
            6'b100000: control<=10'b0000000000;//nop
            default:   control<=10'bxxxxxxxxxx;
        endcase
```

ALU 译码器：

```
    always@(*)
        case(aluop)
            3'b000: alucontrol<=3'b010;//+
            3'b001: alucontrol<=3'b110;//-
            3'b010:
                case(funct)
                    6'b100000: alucontrol<=3'b010;
                    6'b100010: alucontrol<=3'b110;
                    6'b100100: alucontrol<=3'b000;
                    6'b100101: alucontrol<=3'b001;
                    6'b101010: alucontrol<=3'b111;
                    default:   alucontrol<=3'bxxx;
                endcase
            3'b100:alucontrol<=3'b000;//andi
            3'b101:alucontrol<=3'b001;//ori
            3'b110:alucontrol<=3'b111;//slti
            default:
                alucontrol<=3'bxxx;
        endcase
```

接下来进行模块的封装。

首先将和数据有关的模块封装到 datapath 里，与先前的控制单元一起封装为 mips 模块：

```
    logic MemtoReg,Alusrc,RegDst,RegWrite,Jump,PCRsc,Zero;
    logic [2:0]ALUControl;
    control_unit
ControlUnit(Instr[31:26],Instr[5:0],Zero,MemtoReg,MemWrite,
        PCSrc,ALUSrc,RegDst,RegWrite,Jump,ALUControl);
    datapath DataPath(CLK,reset,MemtoReg,PCSrc,ALUSrc,RegDst,
        RegWrite,Jump,ALUControl,Zero,PC,Instr,ALUResult,WriteData,ReadD
ata);
```

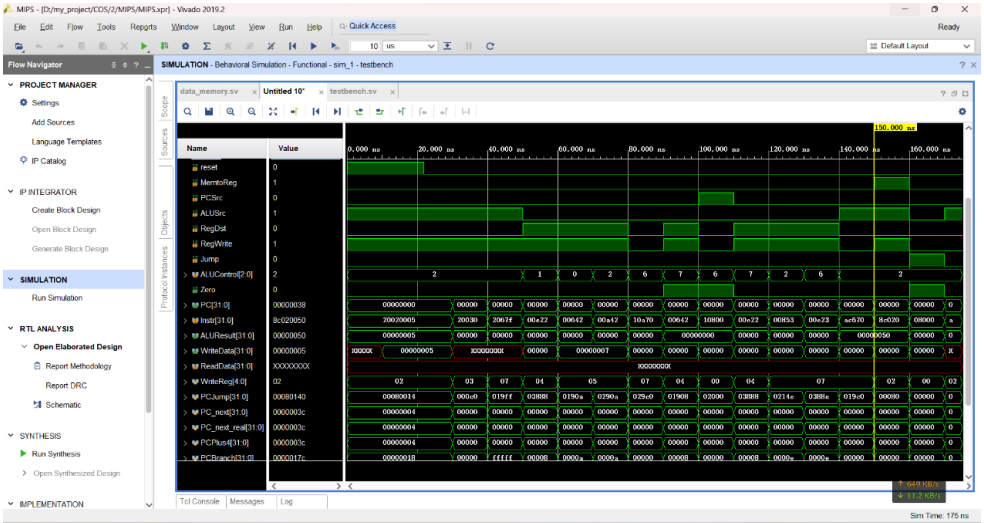
接着把 mips 模块和指令存储器、数据存储器封装至顶层文件 mips_top 中，完成 MIPS 指令 CPU 设计：

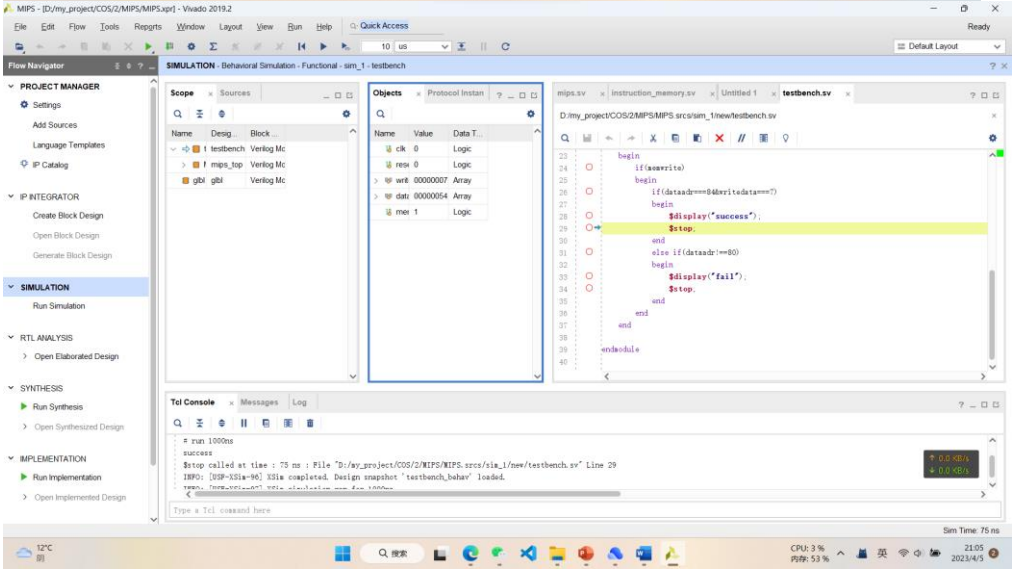
```
logic [31:0] PC,Instr,ReadData;
mips Mips(CLK,reset,PC,Instr,MemWrite,DataAdr,WriteData,ReadData);
instruction_memory InstructionMemory(PC[7:2],Instr);
data_memory DataMemory(CLK,MemWrite,DataAdr,WriteData,ReadData);
```

2.2 仿真

实验结果

仿真结果与预期结果一致。采用的第一个代码是《数字设计和计算机体系结构》书中的标准测试程序，第二个代码是 ppt 中的测试程序，并在其中添加 nop 指令（由于 MIPS 指令集中似乎没有 nop 指令，所以假定其操作码为 100000），采用书中的 testbench，得到的结果如下，与预期一致，得到 success 的反馈。



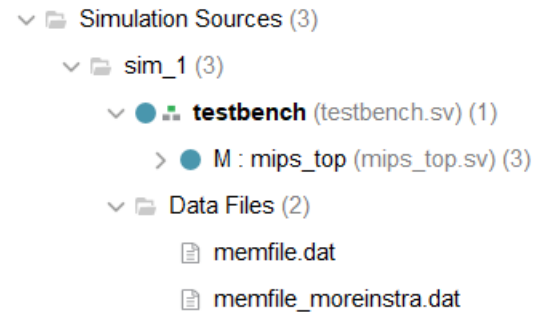


具体实现

首先根据《数字设计》的标准测试代码与 testbench 文件经行仿真测试，Tcl Console 得到 success 的反馈，波形图也与 ppt 上的结果一致。指令文件命名为 memfile.dat。

接下来根据 ppt 上的测试代码，并加上一句 nop 指令，进行仿真测试，Tcl Console 得到 success 的反馈，波形图也与 ppt 上的结果一致。指令文件命名为 memfile_moreinstra.dat。

仿真的文件结构如下：

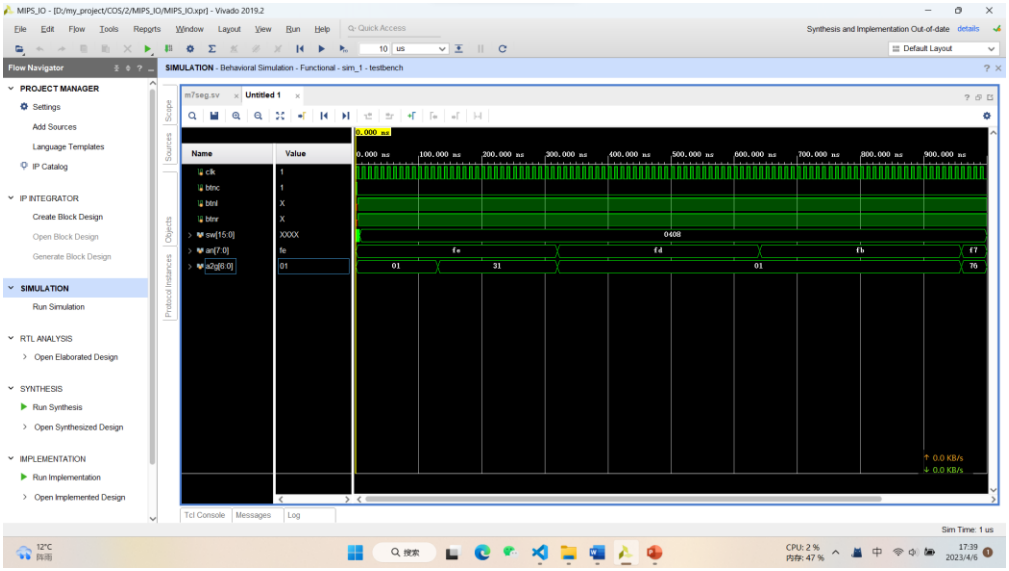


2.3 验证

实验结果

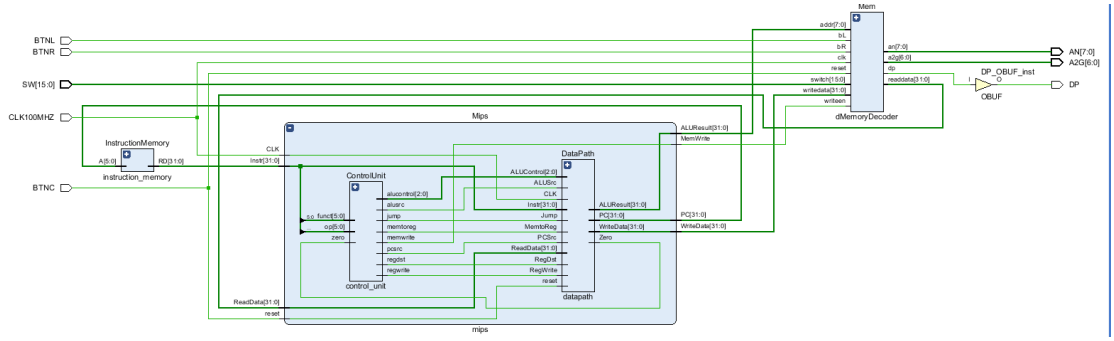
最终的代码文件结构如下：

基于 ppt 上的 testIO 测试指令，实现两个数的相加，写出相应的 testbench 仿真文件，运行仿真后比较波形图，与 ppt 上的内容一致。波形图如下，数码管显示的内容分别为 C00……，与预期结果一致：

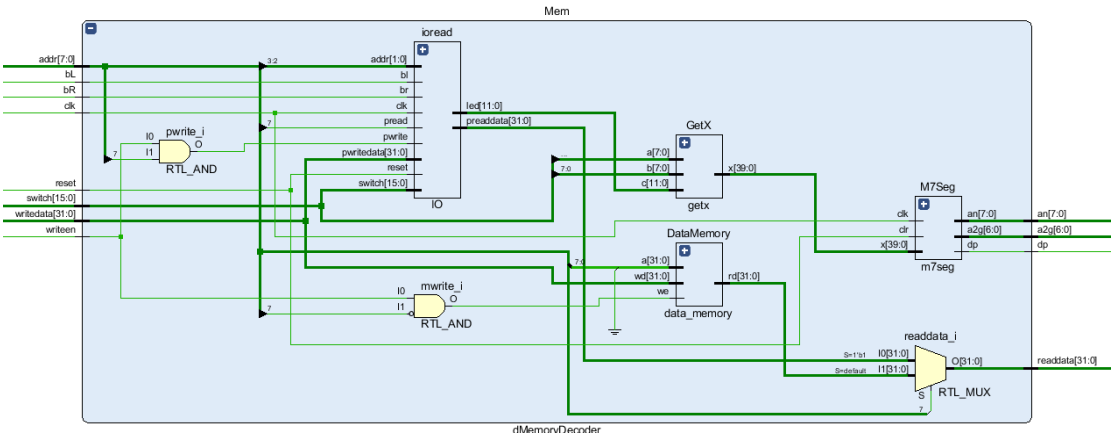


具体实现

最终实现的 RTL 图如下：



其中，Mem 的内部展开如下：



首先新建另一个工程，命名为 MIPS_IO，将原来 MIPS 模块的代码导入进该项目，再将测试的指令改为两个数相加的代码，命名为 testIO.dat，用于仿真和生成 bit 文件。

接下来分析要求，按下 BTNC 重置，BTNR 读入开关的高 8 位与低 8 位，作为两个加数，按下 BTNL 就输出相加的结果。

根据 ppt 上的原理图，对数据存储器进行修改，修改为数据存储器译码器，判断要读取的数据是来自于内存还是开关的输入。

设置一个两位变量 status。按下了 BTNR 后，读入开关的数据，并将 status 高位置为 1。按下了 BTNL 就输出两数相加的结果，将低位置为 1。当 IO 输入的写使能为真，并且是向 0x84（对应 addr 为 01）写入数据时，就写入两数相加的结果，然后将 status 低位置为 0。最后根据读使能判断是否需要读出数据，如果不需要读出就输出 0，需要读就输出相应的值。于是得到如下代码：

IO.sv：

```
always_ff@(posedge clk)
begin
    if(reset)
    begin
        status<=2'b00;
        led1<=12'h00;
        switch1<=16'h00;
    end
    else
    begin
        if(br)
        begin
            status[1]<=1'b1;
            switch1<=switch;
        end

        if(bl)
        begin
            status[0]<=1'b1;
            led<=led1;
        end

        if(pwrite&(addr==2'b01))
        begin
            led1<=pwritedata[11:0];
            status[0]<=0;
        end
    end
end

always_comb
    if(pread)
        case(addr)
```

```
                2'b11:preaddata={24'b0,switch1[15:8]};
                2'b10:preaddata={24'b0,switch1[7:0]};
                2'b00:preaddata={24'b0,6'b0,status};
                default:preaddata=32'b0;
            endcase
        else
            preaddata=32'b0;
```

接下来封装数据存储器译码器，在实例化原来的数据存储器和刚才完成的 IO 存储器后，还需要做一个简单的译码操作，就是根据 MIPS 的控制器输出的内存写使能 MemWrite（这里命名为 writeen，为避免引起混淆），和写入地址的最高位，判断是往内存里写入还是往 IO 存储器里写入，以及读出的数据是来自内存还是 IO 存储器。

除此之外，数据存储器译码器还需要实现对于七段数码管的显示支持，实例化 getx 模块后得到七段数码管需要显示的内容，实例化 m7seg 模块，进行相应的显示。

下面是封装后的数据存储器译码器的主要代码：

```
    assign pwrite=writeen&addr[7];
    assign mwrite=writeen&(~addr[7]);

    data_memory DataMemory(CLK,mwrite,addr,writedata,readdata1);
    IO
ioread(clk,reset,addr[7],pwrite,addr[3:2],writedata,readdata2,bL,bR,swi
tch,led);

    assign readdata=addr[7]?readdata2:readdata1;

    getx GetX(switch[15:8],switch[7:0],led,x);
    m7seg M7Seg(x,clk,reset,an,a2g,dp);
```

getX 模块就是根据开关的高位和低位，拼接输出的 led，形成要显示的内容。本次实验最开始没有注意到直接是用二进制表示的，还编写了 bin2bcd 代码，用于实现最高 12 位二进制数字转换为十进制 8421 码的功能，但是最后没有启用。

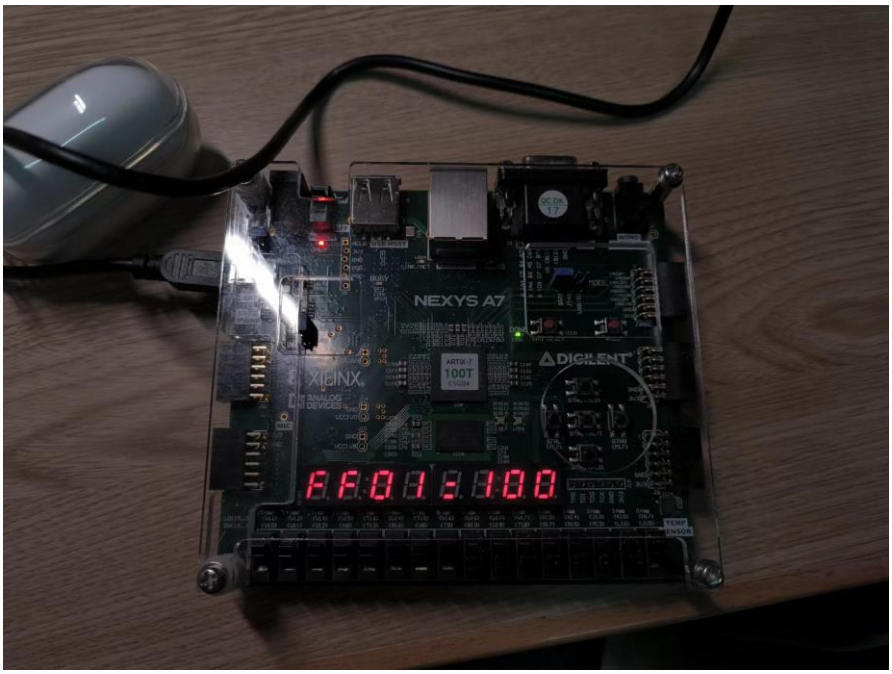
```
    assign x=  {1'b0,a[7:4],1'b0,a[3:0],
                1'b0,b[7:4],1'b0,b[3:0],
                5'b10000,1'b0,c[11:8],
                1'b0,c[7:4],1'b0,c[3:0]
                };
```

最后是采用时钟分频器的分时复用法控制七段数码管的模块。其中控制位在仿真阶段取时钟计数器的 5-7 位，以在波形图上展示更多的信息，在仿真阶段取通常的 17-19 位，以保证七段数码管正常工作。由于这些代码在上学期《数字逻辑》课程中多次使用，并且篇幅较长，就不展示了。

至此，带 IO 接口的单周期 MIPS 处理器已通过仿真测试。

2.4 板上验证

将七段数码管的时钟分频器改为取第 17-19 位来降频，导入约束文件，生成 bit 文件成功后，就上板验证。验证结果是正常工作，实现了要求的功能，即按下 BTNC 重置，BTNR 读入开关的高 8 位与低 8 位，作为两个加数，按下 BTNL 就输出相加的结果。验证的图片如下：



3 实验结论

本次实验通过使用 system Verilog 语言，在 vivado 软件上实现了单周期 CPU 的制作，支持 add，sub，addi，and，or，andi，ori，slt，slti，sw，lw，beq，bne，j，nop 指令，并封装为 CPU 模块，运行基础测试代码和扩展指令的测试代码均在仿真后通过测试。

之后通过结合所学知识，设计了带 IO 接口的单周期 MIPS 处理器，在运行指定的程序后，在仿真阶段通过测试，在导入约束文件后成功生成 bit 文件，再烧录入开发板后，按照既定的目标正常实现了相应的功能。

本实验第一阶段的单周期 CPU 命名为 MIPS，第二阶段的带 IO 接口的单周期 CPU 命名为 MIPS_IO，相关的项目文件在下面的链接中：

链接：https://pan.baidu.com/s/1x6r_RKk0Rg6r5DkoViws7Q
提取码：lab2

---

4 实验感想

本次实验的工作量比较大，涉及到的项目文件数比较多，如何清晰地区分、组织、封装这些文件模块还是很考验细心程度的。在编写代码的时候，往往会因为自己一个不小心，将 always 是时序逻辑还是组合逻辑搞错、对变量使用错误导致连线错误等等。尤其是在一开始从 imem、dmem、regfile 三大模块开始逐步向外扩展支持更多指令的时候，各种意想不到的连线错误，比如实例化时变量位置不对、连线时没区分变量大小写导致 n\c、数组的位数发生错误等等，都导致了意想不到的问题。

解决办法是除了仔细地输入代码、使用变量之外，还在每个新指令添加后及时查看 RTL 图，看看有没有模块的输入输出是 n\c 的，或者被不小心接地了，这个操作虽然比较费事，但真的规避掉了好多问题。

印象最深的一个 bug 就是第一阶段最后仿真测试的时候，波形图和 ppt 几乎完全一样，但是就是显示 fail，在 RTL 分析没有结果后，就只能把 datapath 添加到仿真的波形图里，一句一句指令读，看看是在哪一步出了问题。最后发现是 sw 指令中，数据存储器写入的时候，写入的并不是输入的 writedata，而是输出的 readdata。在改正后，终于正常运行，通过了测试。

不过毫无疑问，这次 lab 的作用是显著的。对于 CPU 的硬件实现各种指令的方式都有了比较深入的了解，特别是控制流、数据流相关的内容，以前都是抽象的名词，但是现在是真的理解到了其中的含义，这是不动手做所实现不了的。这次试验对于理解计算机软硬件交接的层面的有着极大的帮助。