武汉大学计算机学院

本科生实验报告

计算机组成原理实验

专 业 名 称　：计算机科学与技术

课 程 名 称　：计算机组成原理

指 导 教 师　：关　钢

学 生 学 号　：2022302111487

学 生 姓 名　：陈胤中

二〇二三年十二月

目录

1. 实验内容及目的

1. 设计实现包括 rom，rf，alu，dm 四大主要模块在内的各个模块，实现 cpu 内部的各个部件功能
2. 将写好的 rom，rf，alu，dm，控制信号模块，符号扩展模块，显示模块连接起来，实现一个简单 cpu 的功能并实现简单的指令调试
3. 了解 Fpga 芯片中内存的架构方式、实现原理；
4. 掌握 Vivado 内存 ip 核的设计过程；
5. 学习和掌握利用 verilog 调用、调试 ip 核的方法。
6. 达到对《计算机组成与设计》课程的深入理解，提升使用 verilog 进行系统设计的实战能力，为后续综合实验打下坚实的基础。

2. 实验步骤

1. 将四大模块以及显示模块全部导入至同一个工程文件目录下
2. 通过主模块 SCPU_TOP 的创建，在主模块中对各大模块进行例化并建立数据通路
3. 实现从取指令，指令译码，指令计算，指令读写寄存器堆，读写内存等操作

2.1 四模块的连接方法

在组装以四模块为基础的 cpu 之前我们已经完成了四大模块的构建与实现，在四模块的连接方法这个部分中我将介绍由 ROM,RF,DM,ALU 四模块以及几个其他模块连接成的框图以及分别叙述我的代码实现中各个模块的数据通路。

一、 四模块以及其他控制模块的连接框图

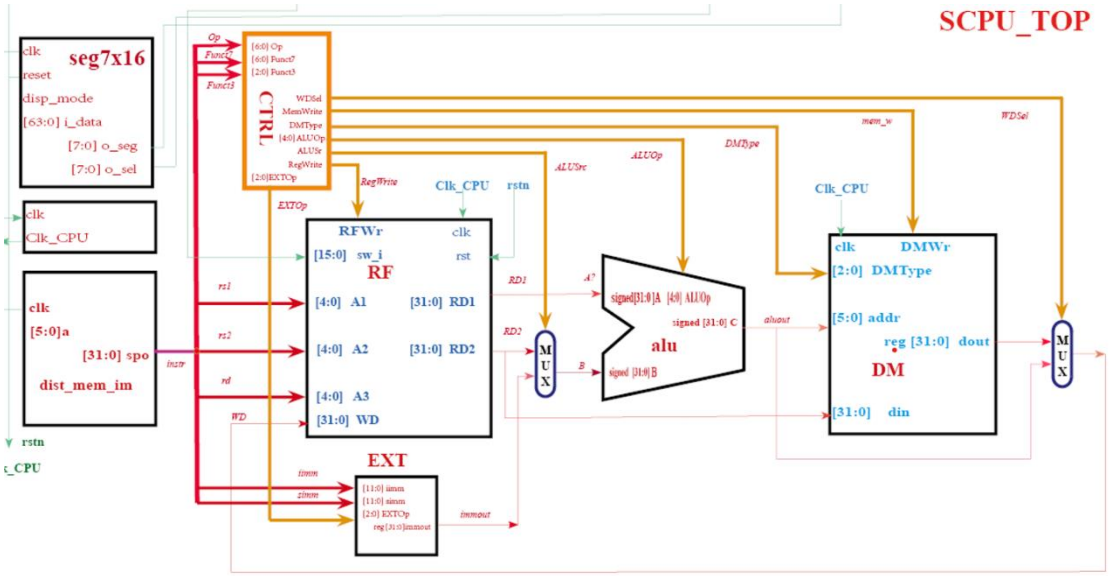


图 2.1.1 总体连接框图

如上图所示即为四大模块与显示模块，符号扩展模块，控制信号产生模块连接的总框图，接下来将具体展示 SCPU_TOP 模块中各个部件间的数据通路构成

二、 时钟周期分频

```
reg[31:0] clkdiv;
wire Clk_CPU;
//分频
always @(posedge clk or negedge rstn) begin
    if(!rstn) clkdiv <= 0;
    else clkdiv <= clkdiv + 1'b1;
end
//sw_i[15]选择慢速模式或快速模式,决定分屏系数是2的27次方（慢）还是2的25次方（快），总频/分频系数=刷新频率
assign Clk_CPU = (sw_i[15])? clkdiv[27] : clkdiv[25];
```

图 2.1.2 时钟周期分频代码片段

如上图所示，在主模块中对时钟周期进行分频，将其设置为可以由第十五号开关控制的，分频系数分别为 2^25 与 2^27 的 Clk_CPU，作为后续控制各个模块读写信号以及操作的时钟周期。

三、 取指令与指令译码

① 取指令

```
wire[31:0]inst_in;
reg[4:0]rom_addr;
//rom每周期地址顺序取数
always @(posedge Clk_CPU or negedge rstn) begin
    if(!rstn) begin rom_addr =5 'b0; end
    else begin
        if (rom_addr < 12) begin
            if(sw_i[1]==1'b0)begin
            rom_addr = rom_addr + 1'b1;
            end
            end
        else
            rom_addr = 5'b00000;
        end
end
//从coe文件中按地址取值
dist_mem_gen_0 u_im(
    .a(rom_addr),
    .spo(inst_in)
);
```

图 2.1.3.1 从 ROM 中取指令代码段

左图为从初始化的 ROM IP 核中的 coe 中取指令的代码，分为两个部分：第一部分为根据时钟周期的变化改变所要取的地址来实现每个周期取 coe 文件中的一条指令

第二部分为例化 dist_mem_gen_0，即根据地址取指令，并将取出的指令放入 instr 变量中。

② 根据指令的机器格式初始化符号扩展模块，控制信号产生模块以及 rf 模块。

a)根据机器格式获得指令各个字段的值

```
always@(*)begin
    Op = inst_in[6:0];  // op
    Funct7 = inst_in[31:25]; // funct7
    Funct3 = inst_in[14:12]; // funct3
    rs1 = inst_in[19:15];  // rs1
    rs2 = inst_in[24:20];  // rs2
    rd = inst_in[11:7];  // rd
    iimm=inst_in[31:20];//addi 指令立即数，lw指令立即数
    simm={inst_in[31:25],inst_in[11:7]}; //sw指令立即数
end
```

图 2.1.3.2 指令译码代码段

指令译码过程如图，根据指令的机器格式获得指令 Funct7,RS2,RS1,Funct3,RD,opcode 六个字段的值，以及针对 load 类型指令的立即数字段的赋值，针对 store 类型指令的立即数字段的赋值。

b)控制信号生成

```
ctrl U_CTRL(
.Op(Op),
.Funct7(Funct7),
.Funct3(Funct3),
.Zero(Zero),
.RegWrite(RegWrite),
.EXTOp(EXTOp),
.ALUOp(ALUOp),
.ALUSrc(ALUSrc),
.DMType(DMType),
.WDSel(WDSel),
.MemWrite(MemWrite)
);
```

图 2.1.3.3 控制信号生成模块 ctrl 的例化

控制信号模块即控制信号生成的部分，根据 opcode，Funct7，Funct3，Zero 四个输入值来产生：

1.控制 rf 模块的 RegWrite 寄存器堆写控制信号；

2.控制符号扩展模块的 EXTOp 符号扩展模式控制信号；

3.决定 ALU 第二个操作数的数据选择器的 ALUSrc 信号；

4.控制 dm 模块存储字节，半字还是字的存储模式的 DMTypte 控制信号；

5.写回寄存器堆的数据选择器的控制信号 WDSel；

c)符号扩展模块生成 ALU 可能需要操作的立即数

```
EXT U_EXT(.iimm(iimm),.simm(simm),.EXTOp(EXTOp),.immout(immout),.iimm_shamt(iimm_shamt),.bimm(bimm),.uimm(uimm),.jimm(jimm));
```

图 2.1.3.4 符号扩展模块 EXT 的例化

如上图所示，根据控制信号产生模块传入的控制信号 EXTOp 决定符号扩展结果的立即数输出 immout，并作为决定 ALU 第二个操作数的数据选择器的第二个输入。

四、 RF 模块的连接

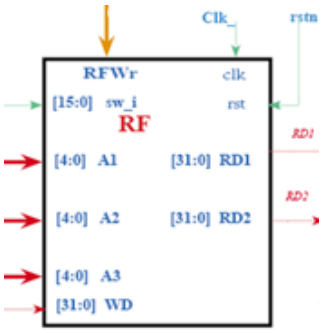


图 2.1.4.1 RF 模块图

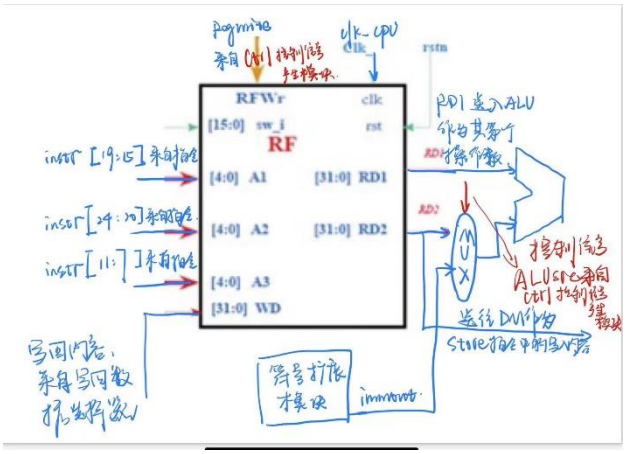


图 2.1.4.2 RF 模块数据通路图

RF 的输入：A1,A2,A3(RD)，均来自于指令机器码，写控制信号 RegWrite 来自于控制信号生成模块。以及来自于硬件的控制信号 rstn 以及经过分频的时钟周期信号 Clk_CPU。WD 信号来自于最后的写回数据选择器。

RF 模块的输出：RD1 即从地址为 A1 的寄存器中读出的数据送入 ALU 模块作为 ALU 的第一个操作数，RD2 即从地址为 A2 的寄存器中读出的数据送入数据选择器决定是寄存器的值还是指令立即数字段作为 ALU 的第二个操作数并且同时送往 DM 数据存储其中作为 store 型指令的存入内存中的内容。

```
//rf模块初始化
wire[31:0]RD1;
wire[31:0]RD2;
reg signed [31:0] WD;
rf U_RF(.clk(Clk_CPU),.rstn(rstn),.RFWr(RegWrite),.sw_i(sw_i[15:0]),.A1(rs1),.A2(rs2),.A3(rd),.WD(WD),.RD1(RD1),.RD2(RD2));
```

图 2.1.4.3 RF 模块的例化

代码部分（RF 模块在主模块中的例化调用部分）：

例化时较上述说明中多出了 sw_i 信号，因为在具体的 rf 模块实现时加入了写有效信号并由第一号开关控制，因此加入了 15 个开关的输入。其他的输入输出信号均与上述说明的各个信号的数据通路相同。

五、 ALU 模块的连接

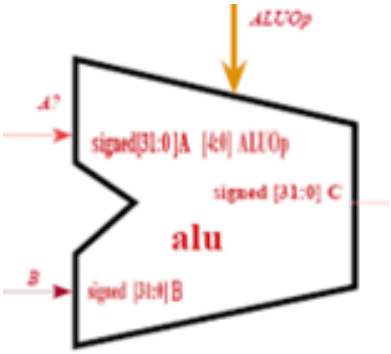


图 2.1.5.1ALU 模块图

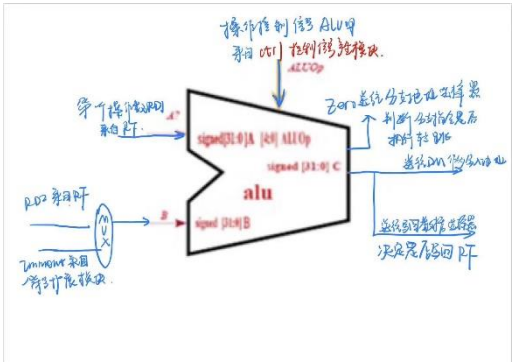


图 2.1.5.2ALU 模块各信号及数据通路图

ALU 模块共有三个输入两个输出，输入分别是两个操作数作为输入，一个操作码控制信号；其中第一个操作数直接来自从寄存器读出的内容 RS1，第二个操作数来源于寄存器读出的内容或是符号扩展后的立即数，例如 addi 与 add 指令就分别选择了立即数和寄存器的值作为其第二个 ALU 操作数。

输出分别是结果 C 以及两个变量相等判别信号 Zero，结果 C 送入 DM 作为写内存的地址信息，同时也直接送往写回数据选择器，可能作为写回寄存器堆的结果。

代码部分（ALU 模块在主模块中的例化调用部分）：

```
wire signed[31:0]B;
assign B=(ALUSrc==1)?immout:RD2;
//alu模块初始化
wire [31:0]aluout;

alu U_ALU(.A(RD1),.B(B),.ALUOp(ALUOp),.C(aluout),.Zero(Zero));
```

图 2.1.5.3 ALU 模块的例化

此段代码用中间变量 B 接收数据选择器的结果，并且使用 ALUSrc 控制信号作为判断条件来选择 ALU 的第二个操作数。用 aluout 作为 alu 操作的结果送往 DM 以及写回数据选择器。

六、 DM 模块的连接

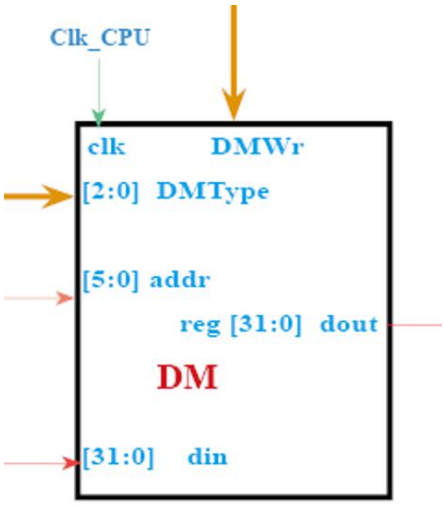


图 2.1.6.1DM 模块图

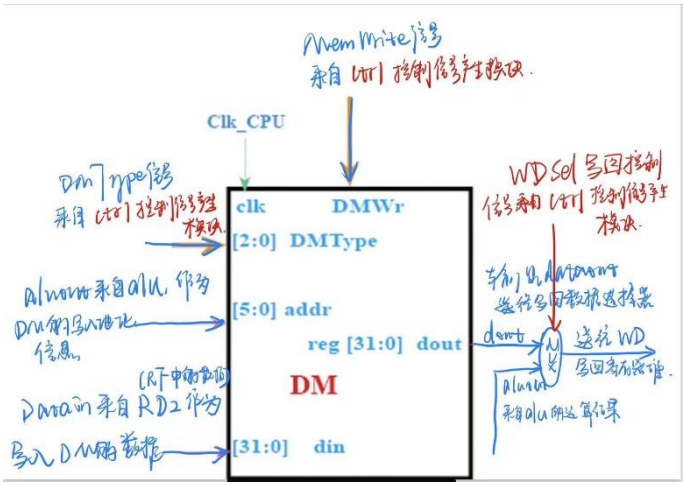


图 2.1.6.2 DM 模块各数据通路图

DM 模块为数据存储器即内存模块，有四个输入，一个输出。其中四个输入有两个是来自控制信号产生模块的控制 dm 写信号的 MemWrite 以及控制读写模式是半字，字节还是字的 DMtype 控制信号。另外两个输入分别是来自 alu 的运算结果 aluout 变量，作为读/写内存的地址信息以及来自寄存器堆中寄存器值的 RD2 变量，作为数据存储器的写入信息。输出信号 dout 送往写回数据选择器，由来自 ctrl 控制信号产生模块的 WDSel 信号来决定写回寄存器堆的是来自 ALU 的运算结果 aluout 还是来自数据存储器 DM 读出的数据 dout。

代码部分（DM 模块在主模块中的例化调用）；

```
wire signed[31:0]dout;
dm U_DM(.DMType(DMType),.clk(Clk_CPU),.addr(aluout),.din(RD2),.DMWr(MemWrite),.dout(dout));

always@(*)
begin
    case(WDSel)
        `WDSel_FromALU: WD<=aluout;
        `WDSel_FromMEM: WD<=dout;
        //`WDSel_FromPC: WD<=PC_out+4;
    endcase
end
```

图 2.1.6.3 DM 模块的例化

由于 DM 涉及到数据的写操作，因此要增加一个时钟信号 Clk_CPU 来控制写操作的执行。

定义的新变量 dout 来接收 dm 的输出，并送往写回数据选择器决定哪个信号将被写回寄存器堆。

2.2 四模块的调试方法

在 2.1 四模块的连接部分中主要介绍了连接的方式以及代码中的数据通路部分，在 2，2 四模块的调试方式中我将介绍作为一个整体的 cpu 的主模块如何运行各条指令。

一、 写信号触发方式

要想了解指令执行的过程和结果，写信号的触发和执行时机是必不可少的条件。本次实验中我选择的方式是将 DM 模块和 RF 模块的写信号设置为上升沿有效。

因此结合前序代码中在时钟周期来时取指令并译码，产生控制信号，我选择的上升沿触发写信号带来的结果是对应的写入数据的内容 din 以及 WD 将在下一个时钟周期的上升沿被写入，也就是说当八段数码管上显示正在执行的指令时，

```
//初始化和数据写入
always@(posedge clk or negedge rstn)
  if(!rstn)begin
      for(i = 0;i < 32;i =i + 1)
          rf[i] <= i;
  end
  else
   if(RFWr && (!sw_i[1])&&A3!=0) begin
      rf[A3] <= WD;
      $display("r[%2d]=0x%8X",A3,WD);
      end
```

图 2.2.1.1 RF 模块中写信号的触发模式

```
always@(posedge clk)begin
if(DMWr==1)
case(DMType)
   `dm_byte:dmem[addr]<=din[7:0];
   `dm_halfword:begin
   dmem[addr]<=din[7:0];
   dmem[addr+1]<=din[15:8];end
   `dm_word:begin
   dmem[addr]<=din[7:0];
   dmem[addr+1]<=din[15:8];
   dmem[addr+2]<=din[23:16];
   dmem[addr+3]<=din[31:24];end
   endcase
   end
```

图 2.2.1.2DM 模块中写信号的触发模式

其上一条指令对于 RF 或 DM 的写操作才执行完成。具体控制写信号执行的 RF 和 DM 模块代码如下图所示，都是在时钟周期来时触发写信号。

二、　单步执行指令的实现

想要实现单步执行指令，也就是让指令的执行受到我们人为的控制，有许多办法。例如使用开关的“0”，“1”状态作为 always@语句的触发条件，或者是像取指令时使用的 Clk_CPU 作为 always@语句取指令的触发条件。

我选择的是后者，在使用分屏后的时钟时钟 Clk_CPU 情况下加上控制取指令的信号 sw_i[1]作为能够停止指令取出（执行）的方式。实现的具体代码如下：

即在指令地址范围为 0-11 的条件下加入 sw_i[1]是否为“0”的条件，如为

```
//rom每周期地址顺序取数
always @(posedge Clk_CPU or negedge rstn) begin
    if(!rstn) begin rom_addr =5 'b0; end
    else begin
        if (rom_addr < 12) begin
            if(sw_i[1]==1'b0)begin
            rom_addr = rom_addr + 1'b1;
            end
            end
        else
            rom_addr = 5'b00000;
        end
end
```

图 2.2.2 单步执行指令实现

“0”则停止执行，如为“1”则继续执行。

三、　指令执行结果的展示方式

前四次实验中所使用的展示结果的方式都是利用循环显示各个部件的各个单元的形式来显示结果，在四模块综合实验中，我选择的也是这样的方式。通过 11，12，13，14 号开关的不同组合来实现不同的部件展示。其中 14-11 号开关的组合“0001”为展示 DM 单元；“0010”为展示 alu 的两个操作数以及两个输出；“0100”展示 RF 寄存器堆中 32 个寄存器的值；“1000”为展示当前正在被执行的指令的机器码。

代码部分以 DM 和 RF 模块为例：为使得各个内存单元的展示更加容易辨认，我将最左侧的一（二）个数码管用于展示被展示单元的编号，剩余的部分用于展示单元内的数据本身。

对于 RF 模块：32 个寄存器需要两个数码管来显示，因地址只有五位，故补

齐高位的“0”，低 24 位为真实的数据部分，构成 32 位八位数码管字符模式所需的编码。

对于 DM 模块：对于需要显示的前 16 个单元需要四位地址，而每个 dm 单元都是一个八位的寄存器，因此除去高位的单元编号和低位的数据部分之外，用“0”补齐八位数码管字符模式所需的 32 位编码。

```
    //寄存器每周期顺序取数
    always@(posedge Clk_CPU or negedge rstn)begin
        if (!rstn) begin reg_addr=5'b0;end
        else if(sw_i[13]==1'b1)
         begin
            reg_data={{3'b0},{reg_addr[4:0]},U_RF.rf[reg_addr][23:0]};
            reg_addr=reg_addr+ 1;
        end
    end
```

图 2.2.3.1 RF 模块循环显示时加入单元编号

```
always @(posedge Clk_CPU or negedge rstn) begin
        if(!rstn) begin
            dmem_addr<= 7'b000000;
            dmem_data<= 32'hFFFFFFFF; end
        else if(sw_i[11]==1'b1)begin
            dmem_data = {dmem_addr[3:0],{20'b0},U_DM.dmem[dmem_addr][7:0]};
            dmem_addr<= dmem_addr + 1'b1;
            if(dmem_addr >= DM_DATA_NUM)begin
                dmem_addr<= 7'b0000000;
                dmem_data<= 32'hFFFFFFFF;end
        end
    end
```

图 2.2.3.2 DM 模块在循环显示时加入单元编号的方式

3. 实验中的问题与解决方法

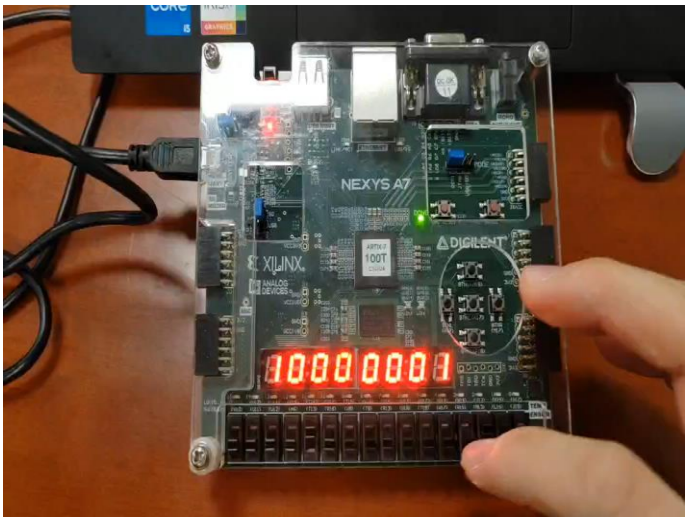
1. dm 循环显示十六个单元的内容时将八个单元的内容显示了两遍

   错误动画：

   后续出现的动画包括：dm 初值赋值失败、dm 前 16 个单元显示时将前八个单元显示两遍，dm 成功初始化及显示，指令执行结束后各个部件的值展示四个视频，若无法观看请点击下方链接观看：

   链接：https://pan.baidu.com/s/1P_k-urUT4u-f0QeZesnSkA?pwd=1203

   提取码：1203



视频 3.1.1 错误代码执行结果动画

（观看视频时先双击视频，在点击观看，后续视频皆如此）

错误原因：框中语句 reg[7:0]dmem[6:0]错误。

改正方法：将 dmem 单元的定义语句改为 reg[7:0]dmem[127:0]才能符合实验指导书中要求的 128 个内存单元

```
module dm(
input clk,
input DMWr,
input[5:0]addr,
input[31:0] din,
input[2:0] DMType,
output reg[31:0]dout
);
reg [7:0]dmem[6:0];
integer i;
    initial begin
    for(i=0;i<128;i=i+1)
        dmem[i]=i;
    end
```

图 3.1.1 错误代码中的定义

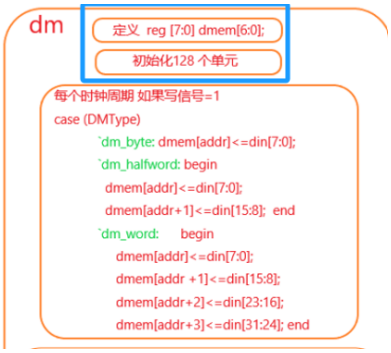


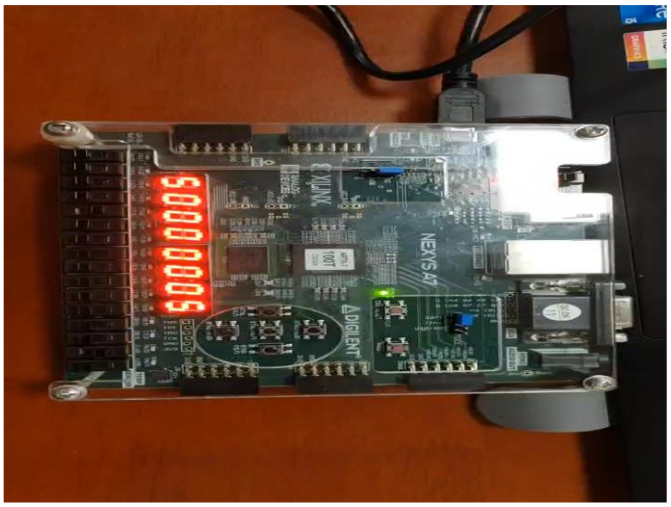
图 3.1.2 实验指导书中的错误

改正后的代码：

```
module dm(
input clk,
input DMWr,
input[5:0]addr,
input[31:0] din,
input[2:0] DMType,
output reg[31:0]dout
);
reg [7:0]dmem[127:0];
integer i;
    initial begin
    for(i=0;i<128;i=i+1)
        dmem[i]<=i;
    end
```

图 3.1.3 改正后的代码段

改正后执行结果：



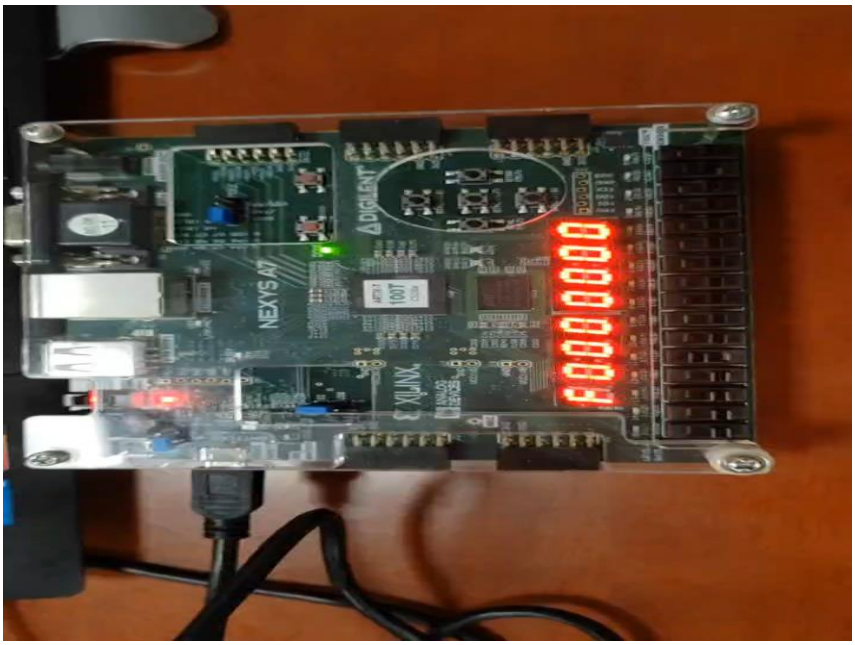
视频 3.1.2 改正后执行结果动画

2. dm 初始化失败，全部为 0

错误原因：dm 初始化的失败与初始的赋值语句有误有关，dmem[i]变量在 reg[7:0]dmem[6:0]的定义下，i<127 的循环条件下将会超出范围，产生错误，与上述循环显示的代码问题相同。

改正方法：将 dmem 单元的定义语句改为 reg[7:0]dmem[127:0]才能符合实验指导书中要求的 128 个内存单元，才能正确运行原来代码中的初始化语句。

错误动画：



视频 3.2.1 错误代码执行结果动画

改正后正确执行结果（与第一个问题的正确执行结果相同，故不再放）

3. 加减法 ALUOp 宏定义更改

本问题涉及到本次四模块的拼接的综合实验与我们之前所实现的单个模块的功能的实现方式的兼容性有关，因宏定义的不同而产生了指令在 ALU 阶段无法执行的问题。

错误问题：注意到之前实验时我们定义加法，减法的 ALUOp 宏定义为：ALUOp_add:5’ b00001,ALUOp_sub:5’ b00010 而大模块中的宏定义要求却不同如下：

```
注明: [4:0]ALUOp 可以如下类似定义
// ALU control code
`define   ALUOp_add   5'b00001
`define   ALUOp_sub   5'b00010
```

图 3.3.1 先前 ALU 模块实验中对于加减法宏定义的要求（错误代码）

```
`define ALUOp_nop 5'b00000
`define ALUOp_lui 5'b00001
`define ALUOp_auipc 5'b00010
`define ALUOp_add 5'b00011
`define ALUOp_sub 5'b00100
`define ALUOp_bne 5'b00101
`define ALUOp_blt 5'b00110
`define ALUOp_bge 5'b00111
`define ALUOp_bltu 5'b01000
`define ALUOp_bgeu 5'b01001
`define ALUOp_slt 5'b01010
`define ALUOp_sltu 5'b01011
`define ALUOp_xor 5'b01100
`define ALUOp_or 5'b01101
`define ALUOp_and 5'b01110
`define ALUOp_sll 5'b01111
`define ALUOp_srl 5'b10000
`define ALUOp_sra 5'b10001
```

图 3.3.2 本次四模块实验中对于加减法宏定义的要求

改正方法：根据指令解码产生控制信号的逻辑，以及宏定义头文件修改加减法的宏定义

改正后的 ALU 模块宏定义部分：

```
`define ALUOp_add 5'b00011
`define ALUOp_sub 5'b00100

module alu(
input signed[31:0]A,B,
input[4:0]ALUOp,
output reg signed[31:0]C,
output reg[7:0]Zero
);
always@(*) begin
case(ALUOp)
```

图 3.3.3 修改后的正确代码段

4. 符号扩展控制信号生成位置错误

错误原因：对于符号扩展模块的控制信号 EXTOp 各个位所表示的含义不清晰，与宏定义头文件中的定义不相符，以及实验指导书中对于控制信号 EXTOp 的赋值错误，从而导致 store 型指令和 load 型指令在计算常数时的不成功，导致指令执行问题。

操作指令生成常数扩展操作

```
assign EXTOp[0] =  stype;
assign EXTOp[1] =  itype_l | itype_r ;
```

图 3.4.1 实验指导书中的赋值语句错误

```
//EXT CTRL itype, stype, btype, utype, jtype
`define EXT_CTRL_ITYPE_SHAMT 6'b100000
`define EXT_CTRL_ITYPE  6'b010000
`define EXT_CTRL_STYPE  6'b001000
`define EXT_CTRL_BTYPE  6'b000100
`define EXT_CTRL_UTYPE  6'b000010
`define EXT_CTRL_JTYPE  6'b000001
```

图 3.4.2 宏定义文件中对于 store 型指令和 load 型指令 EXTOp 控制信号的定义

```
//符号扩展操作信号生成
assign EXTOp[0] =  stype;
assign EXTOp[1] =  itype_l | itype_r ;
```

图 3.4.3 错误代码

改正方案：根据宏定义文件中的内容修改 EXTOp 控制信号，使之与 ctl 控制信号产生的控制信号能够一一对应，正确解码 store 型和 load 型指令

改正后的代码：

```
//符号扩展操作信号生成
assign EXTOp[3] =  stype;
assign EXTOp[4] =  itype_l | itype_r ;
```

图 3.4.4 修正后的正确代码段

4. 实验结果与总结

一、 实验结果

本次实验我所执行的十二条指令以及其机器码，指令功能如下：

指令内容	指令机器码 （十六进制格式）	指令功能
add x1,x0,x0	000000b3	将寄存器 x1 的值改为 0
addi x1,x1,1	00108093	将寄存器 x1 的值改为 1
add x2 x0 x0	00000133	将寄存器 x2 的值改为 0
addi x2 x2 2	00210113	将寄存器 x2 的值改为 2
add x3 x0 x0	000001b3	将寄存器 x3 的值改为 0
addi x3 x3 3	00318193	将寄存器 x3 的值改为 3
sw x1 1(x0)	001020a3	将第一号内存单元的值改为 1
sw x2 2(x0)	00202123	将第二号内存单元的值改为 2
sw x3 3(x0)	003021a3	将第三号内存单元的值改为 3
lw x4 1(x0)	00102203	将第一号单元开始的一个字载入寄存器 x4 中
lw x5 2(x0)	00202283	将第二号单元开始的一个字载入寄存器 x5 中
lw x6 3(x0)	00302303	将第三号单元开始的一个字载入寄存器 x6 中

执行结果：

首先，该 12 条指令运行完成后各个部件的展示由于受限于文件大小无法展示，故不在此放时评展示完整运行结果，均以截图的形式替代。

a) 运算指令以 addi x3 x3 3 为例：

图 4.1.1 addi x3 x3 3 **执行结果**

如图所示，该指令成功将三号寄存器的值修改为 3（注意：32 个寄存器需要两个数码管来显示其编号，故编号为 03，值为 3。）

其余的 add 指令以及 addi 指令起到的效果均为修改寄存器，为其赋值，结果类似，故不再附图注释。

b)　存储型指令以 sw x3 3(x0)为例：

图 4.1.2 sw x3 3(x0)执行结果

如上图所示，本条存储指令成功将三号内存单元的值修改为 3（注意：dm 只显示前 16 个单元，故一个数码管表示其编号就够，以区别于 RF 中两个数码管表示寄存器编号。）

其余的两条 sw 指令与本条指令均为修改内存单元，为其赋新值，与本条指令的执行效果较为类似，故不在叙述。

c)　载入型指令以 lw x4 1(x0)为例：

图 4.1.3 lw x4 1(x0)执行结果

如上图所示，本条指令成功将第四号寄存器的值修改为 030201。

显示是此结果的原因：DM 中每个单元是一个八位寄存器，但是数码管的显示模式是四位表示一个数码管的内容，故单就一个单元而言，例如存在 DM 中的数据“1”而言，显示在数码管上的结果就是“01”。而前序的三条指令恰好是把 DM 的第一个单元修改为“1”，第二个单元修改为“2”，第三个单元修改为“3”。而 lw 指令为加载一个字，但是这个字的高 8 位全部为 0，存有内容的刚好为 24 位，

为显示的数据范围。再根据上述 DM 数据“1”显示为“01”可以得出，从高位到低位应该依次为“03”，“02”，“01”。因此显示结果为 030201 正确。

其余两条 lw 指令与本条指令的执行效果类似，因此不在阐述。

二、 实验总结：

本次实验从显示模块，分频的原理及其应用开始讲起，先后讲了 ROM,RF,ALU,DM 四个 cpu 内部的主要模块，一起帮助他们一起实现 cpu 功能的符号扩展模块，指令译码模块，宏定义等等内容，循序渐进，让我们从本次一系列的实验中不仅能够收获各个模块实现的思路，同时也能掌握指令执行的各个步骤与过程。通过 vivado 软件的配置调试和 verilog 语言的应用来实现这些这一系列的功能在提升我们动手编程能力的同时也让我们充分认识到了 cpu 内核各部件的构造思路及方法，为后续下学期的课程设计，支持更多 RISC-V 指令执行的 cpu 模块设计打下了坚实的基础。

当然，本次实验也存在许多不足之处等待后续的改进。例如指令的单步执行触发方式，使用时钟周期作为取指令触发信号导致需要根据时钟不断拨动开关来控制指令的取出执行，一旦手快或手慢都将造成不可挽回的错误而重新开始执行调试。在 12 条指令的指令集内从头执行可能比较容易，但在之后的多指令，更复杂的指令集的执行调试过程中将会变得不那么容易。又或是指令解码模块，符号扩展模块的控制信号能否支持更多不同格式的指令任然需要改进。如上的各种问题都是我在将来要改进，努力的方向。

最后，感谢各位实验室老师以及同学们的帮助，让我能够顺利完成本次《计算机组成与设计》课程的一系列实验，我也相信在未来有了本次实验打下的基础，我能在后续的实验开发中做的更好。

实验报告完成人：　　陈胤中

日期：　　2023 年 12 月 20 日