南開大學
Nankai University

计算机学院

计算机组成原理实验报告

实验五

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 5 月 13 日

目录

1 实验内容说明

请根据实验指导手册中的单周期 CPU 和多周期 CPU 实验内容，完成如下任务并撰写实验报告：

1、针对单周期 CPU 实验，复现并验证功能，同时对三种类型的 MIPS 指令，挑 1 至 2 条具体分析总结其执行过程。

2、针对多周期 CPU 实验，请认真分析指令 ROM 中的指令执行情况，找到存在的 bug 并修复，实验报告中总结寻找 bug 和修复 bug 的过程。

3、将 ALU 实验中扩展的三种运算，以自定义 MIPS 指令的形式，添加到多周期 CPU 实验代码中，并自行编写指令存储到指令 ROM，然后验证正确性，波形验证或实验箱上箱验证均可。

注意：单周期 CPU 使用异步存储器（.v）文件格式，多周期 CPU 使用的是同步存储器（IP 核形式），二者不要弄混。多周期实验中，每一个 IP 核请自行创建到项目中，不要使用源码中的 dcp 文件。

2 实验一

2.1 实验目的

1. 理解 MIPS 指令结构，理解 MIPS 指令集中常用指令的功能和编码，学会对这些指令进行归纳分类。

2. 了解熟悉 MIPS 体系的处理器结构，如延迟槽，哈佛结构的概念。

3. 熟悉并掌握单周期 CPU 的原理和设计。

4. 进一步加强运用 verilog 语言进行电路设计的能力。

5. 为后续设计多周期 cpu 的实验打下基础。

2.2 实验原理图

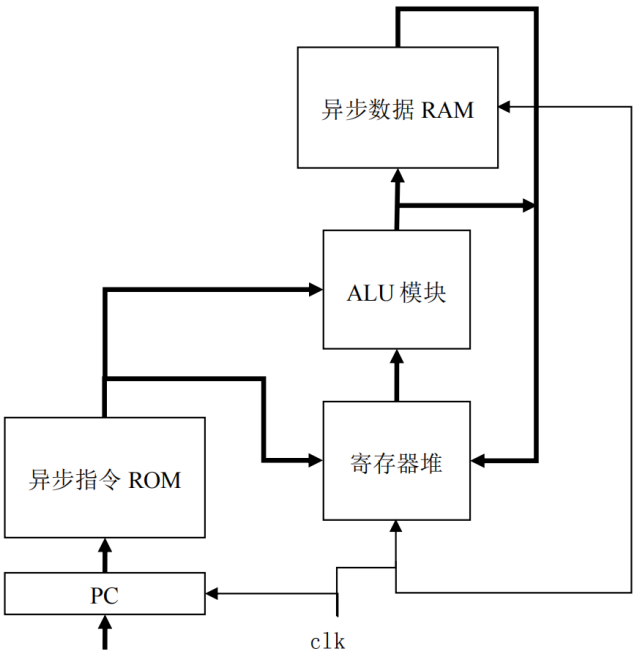

图 2.1: 单周期 CPU 的大致框图

2.3 实验步骤

2.3.1 复现单周期 CPU 及功能验证

原单周期 CPU 代码中有一行 bug，原来的逻辑只能向内存写入最低位，我们将其改为如下的逻辑。

```
    assign dm_wen = ({4{inst_SW}}) & {4{resetn}};
```

接着我们编写仿真程序，实现对不同寄存器和内存的监控。

```
timescale 1ns / 1ps
module tb;

    // Inputs
    reg clk;
    reg resetn;
    reg [4:0] rf_addr;
    reg [31:0] mem_addr;

    // Outputs
    wire [31:0] rf_data;
    wire [31:0] mem_data;
    wire [31:0] cpu_pc;
    wire [31:0] cpu_inst;

    // Instantiate the Unit Under Test (UUT)
    single_cycle_cpu uut (
        .clk(clk),
        .resetn(resetn),
        .rf_addr(rf_addr),
        .mem_addr(mem_addr),
        .rf_data(rf_data),
        .mem_data(mem_data),
        .cpu_pc(cpu_pc),
        .cpu_inst(cpu_inst)
    );

    initial begin
        // Initialize Inputs
        clk = 0;
        resetn = 0;
        rf_addr = 0;
        mem_addr = 0;

        // Wait 100 ns for global reset to finish
        #100;
        resetn = 1;
        rf_addr = 1;
        #10;
```

```
40          rf_addr = 2;
41          #10;
42          rf_addr = 3;
43          #10;
44          rf_addr = 4;
45          #10;
46          rf_addr = 5;
47          #10;
48          mem_addr = 20;
49          #10;
50          rf_addr = 6;
51          #10;
52          rf_addr = 7;
53          #10;
54          rf_addr = 8;
55          #10;
56          mem_addr = 28;
57          #10;
58          rf_addr = 9;
59          #10;//跳转
60
61          #10;
62          rf_addr = 10;
63          #10;//不跳转
64          #10;
65          rf_addr = 11;
66          #10;
67          #10;
68          mem_addr = 16;
69          #10;
70          rf_addr = 12;
71          #1000 $finish;
72
73      end
74      always #5 clk=~clk;
75  endmodule
```

并运行仿真程序，结果如下所示：

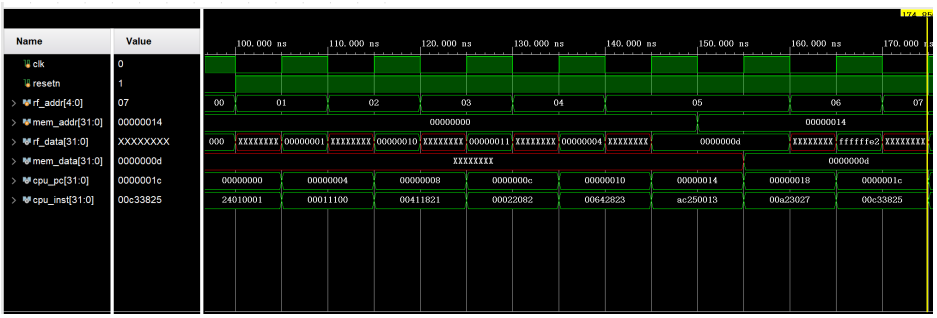

图 2.2

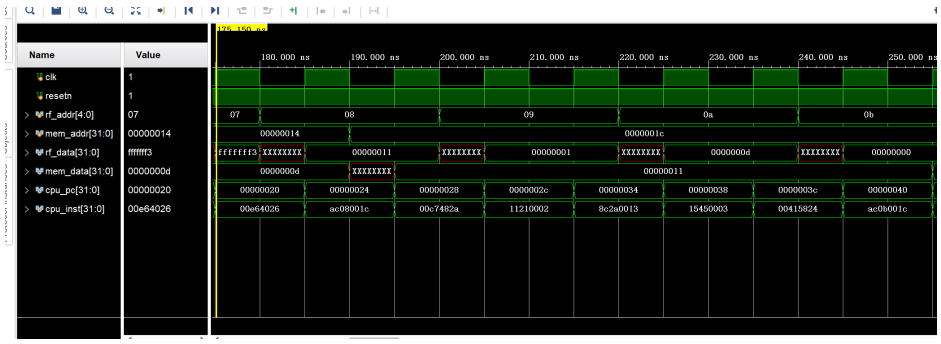

图 2.3

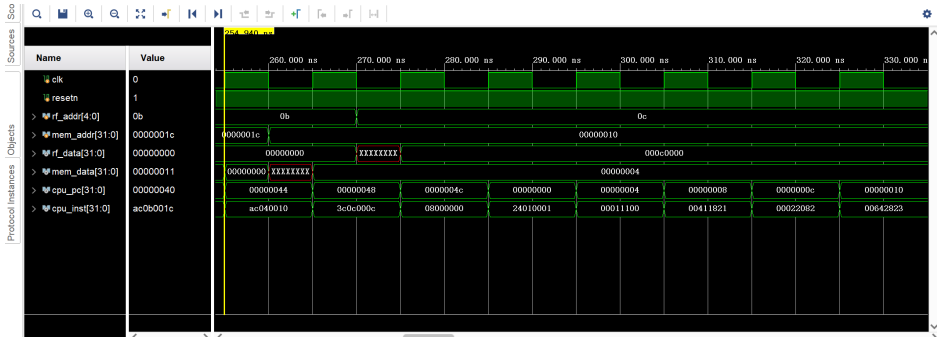

图 2.4

2.3.2 三种类型 MIPS 指令分析

首先我们先来分析一下 CPU 在各个阶段做了什么。

取指：从内存中读取下一条指令（在这里是读取 inst_rom 中的指令）

通过程序计数器 PC 存储当前指令的地址，通过地址总线找到内存中的指令，将指令内容复制到指令寄存器（IR）中，之后 PC 自动递增，指向下一条指令的地址（除非遇到跳转指令）。

译码：解析指令的含义和操作

控制单元（CU）分析指令的操作码（Opcode），确定需要执行的操作（如加法、跳转等）。

识别操作数的来源（寄存器、内存地址或立即数）。

执行：执行指令要求的计算或操作

算术逻辑单元（ALU）进行数学或逻辑运算（如加减乘除、位运算）。

若涉及跳转指令（如 JMP），计算目标地址并更新 PC。

若涉及内存操作，计算有效地址（如 LOAD 或 STORE 指令的地址）。

访存：与内存交互（读取或写入数据）

若指令需要读取内存（如 LOAD），从计算出的地址中获取数据。

若指令需要写入内存（如 STORE），将数据存入指定地址。

若无内存操作，此阶段可能跳过。

写回：将结果保存到寄存器或内存

将 ALU 的计算结果或从内存读取的数据写入目标寄存器。

更新寄存器的值，为下一条指令做好准备。

接着我们来分析一下三种类型的 MIPS 指令。

R 型指令（算数指令）

000000 (op)	R_s	R_t	R_d	shamt	funct
6bits	5bits	5bits	5bits	5bits	6bits

图 2.5: R 型指令格式

其中，op 指的是操作码，所有 R 型指令都全为 0；rs、rt、rd 为寄存器编号，分别对应第一个源操作数、第二个源操作数、保存结果；shamt 为常数，在移位指令中使用；funct 为功能码，指定指令的具体功能。

```
assign inst_rom[ 2] = 32'h00411821; // 08H: addu  $3 ,$2,$1    | $3 = 0000_0011H
```

我们以这个 addu 加法指令为例,其操作码为 00411821,将其展开成二进制,为 0000_0000_0100_0001_0001_1000_0010_0001, 前 6 位为 000000，代表是 R 型指令，接下来 5 位为 00010，对应 $2 寄存器，接着为 00001，对应 $1 寄存器，接着为 00011，对应 $3 寄存器，00000 表示不做移位，funct 为 100001 对应的就是 addu 指令。也就是说，这里的操作是，将 $2 寄存器与 $1 寄存器的值进行相加，并将结果保存在 $3 寄存器中，由仿真图像，$2 寄存器的值为 00000010H，$1 寄存器的值为 00000001H，执行完这次操作之后，$3 寄存器的值为 00000011H。

I 型指令（立即数指令）

OP	R_s	R_t	imm（立即数）
6bits	5bits	5bits	16bits

图 2.6: I 型指令格式

op 标识指令的操作功能,rs 是第一个源操作数，是寄存器操作数，rt 是目的寄存器编号，用来保存运算结果，imm 是第二个源操作数，立即数。

```
assign inst_rom[ 0] = 32'h24010001; // 00H: addiu $1 ,$0,#1    | $1 = 0000_0001H
```

这条指令的操作码为 24010001，展开成二进制为 0010_0100_0000_0001_0000_0000_0000_0001, 前 6 为 001001，表明进行的操作为 addui 立即数加法运算，接下来 5 位 00000 对应 $0 寄存器，接着 00001 对应 $1 寄存器，最后 16 位 0000000000000001 表明立即数是 1。所以这条指令就是把 $0 寄存器的值与立即数 1 进行相加，结果保存在 $1 寄存器中。由仿真图像，原来 $0 寄存器的值默认为 0，加 1 结果为 00000001h 保存在 $1 寄存器中。

J 型指令

OP	立即数
6bits	26bits

图 2.7: J 型指令格式

```
assign inst_rom[19] = 32'h08000000; // 4CH: j      00H          | 跳转指令00H
```

这条指令操作码为 08000000, 展开成二进制为 0000_1000_0000_0000_0000_0000_0000_0000，前 6 位 000010 对应指令 j，后 26 位全为 0 即表示跳转到 0 号指令处，由图像可以看到，在执行完这条指令之后，下一次取的指令是 0 号指令。

3 实验二

3.1 实验目的

1. 在单周期 CPU 实验完成的提前下，理解多周期的概念。
2. 熟悉并掌握多周期 CPU 的原理和设计。
3. 进一步提升运用 verilog 语言进行电路设计的能力。
4. 为后续实现流水线 cpu 的课程设计打下基础。

3.2 实验原理图

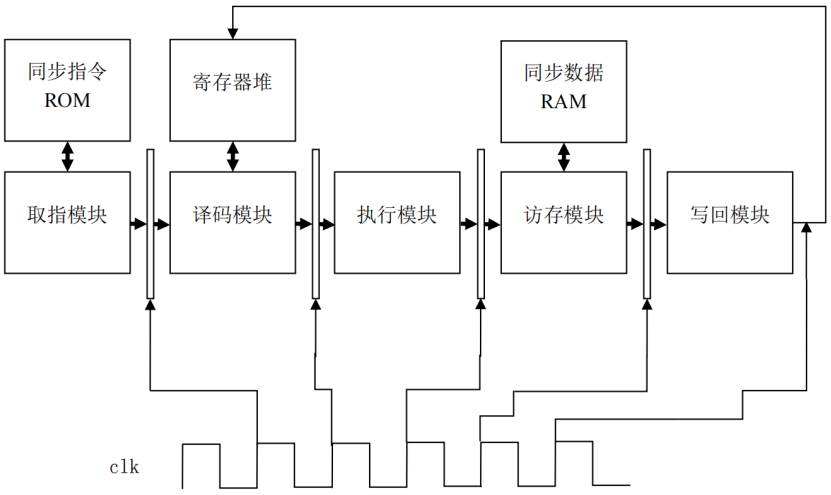

图 3.8: 多周期 CPU 的大致框图

不同于单周期，多周期在每个时钟到来时进行 5 个执行步骤中的 1 个步骤。

3.3 实验步骤

在原来的代码基础上，如果利用原来的 ip 核，会导致周期不匹配问题。应该就是当输出寄存器（勾选 Primitives Output Register）的时钟（clk_out）与数据来源的时钟（clk_in）不同步时，直接传输数据可能导致亚稳态，导致仿真中数据变为 X。

3.3.1 IP 核搭建

按之前的操作配置 ip 核，但注意不能勾选 primitives output register 这个选项，否则会引入一个时钟周期的延迟。

inst_rom

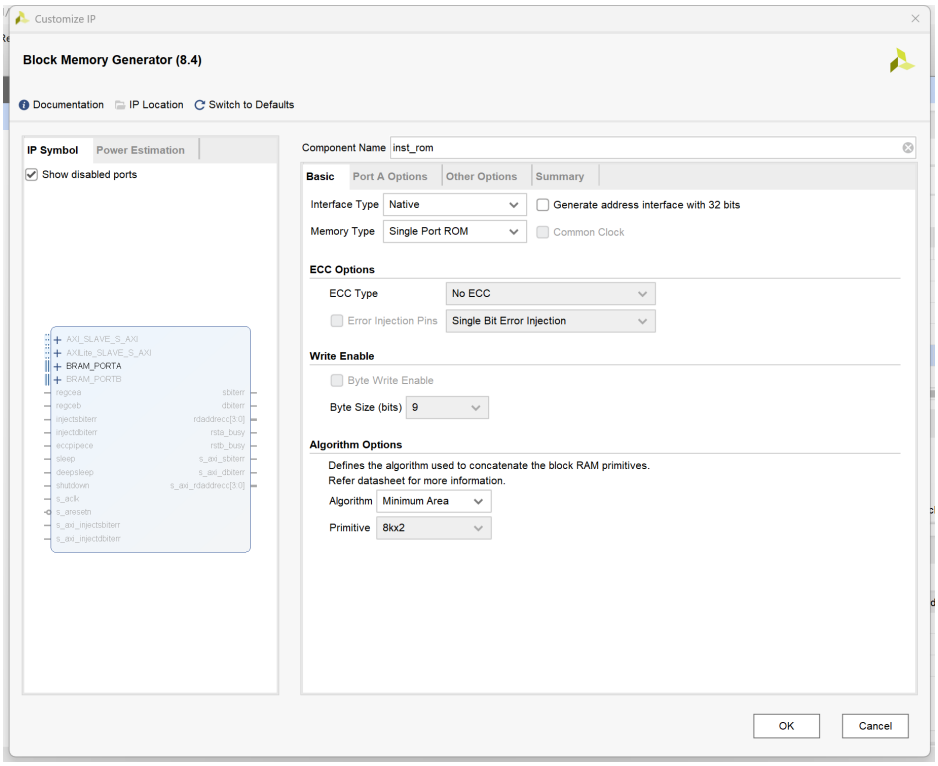

图 3.9: inst_rom 配置

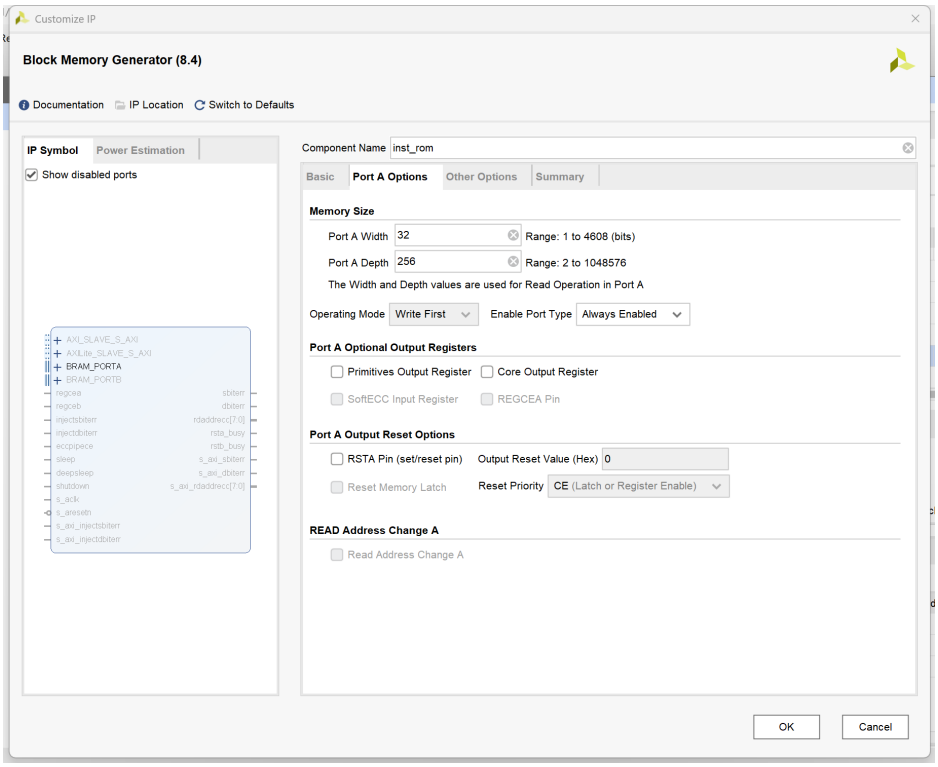

图 3.10: inst_rom 配置

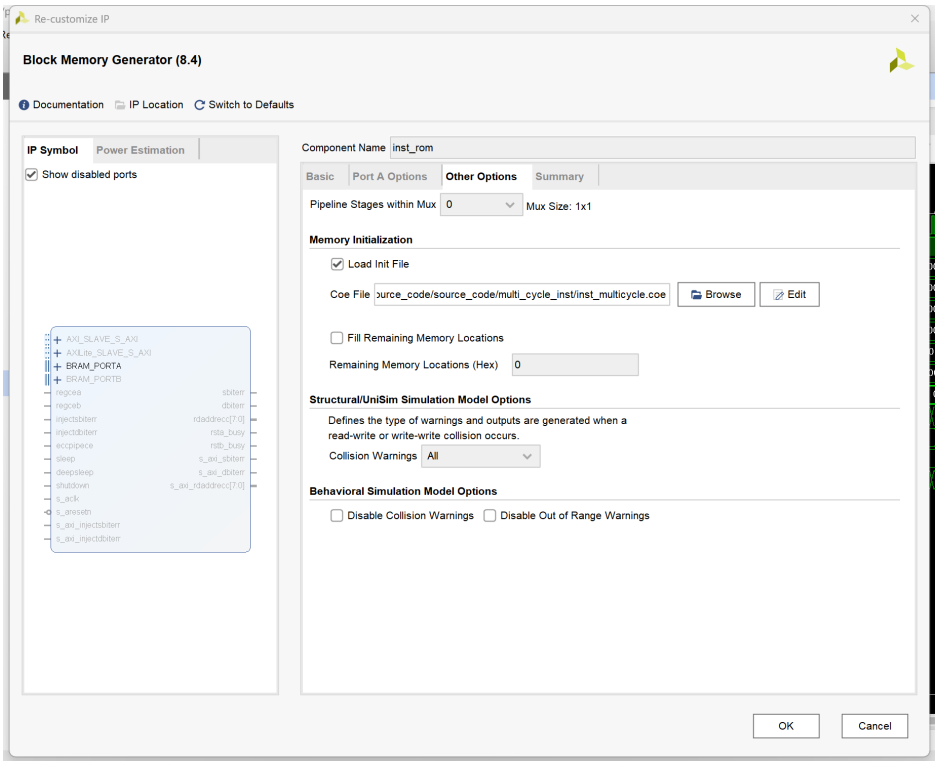

图 3.11: inst_rom 配置

data_ram

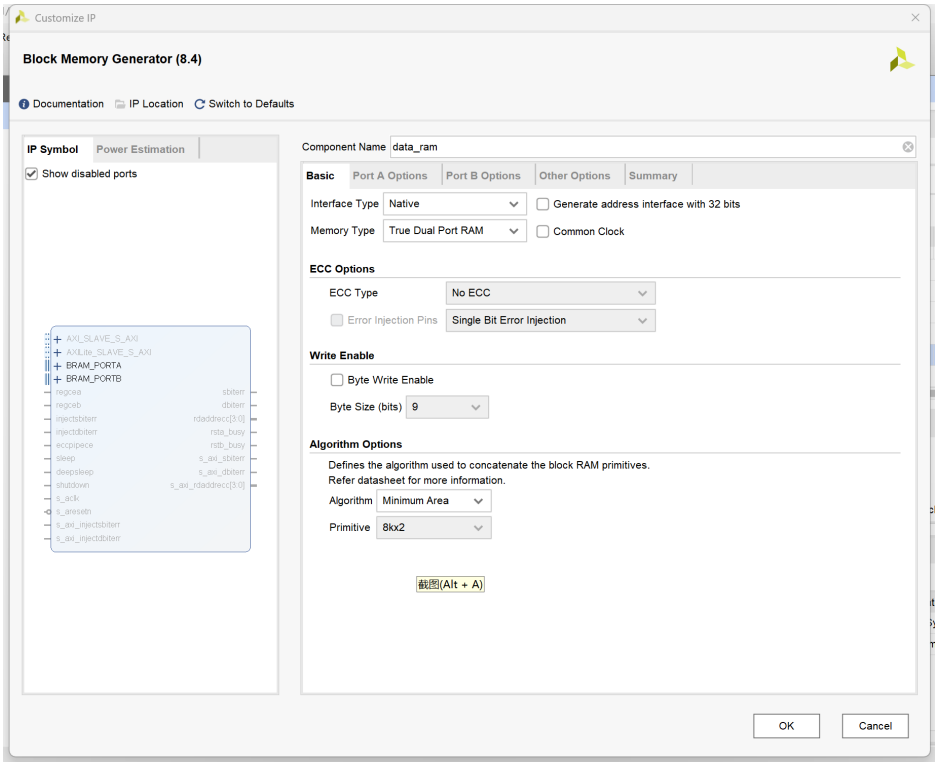

图 3.12: data_ram 配置

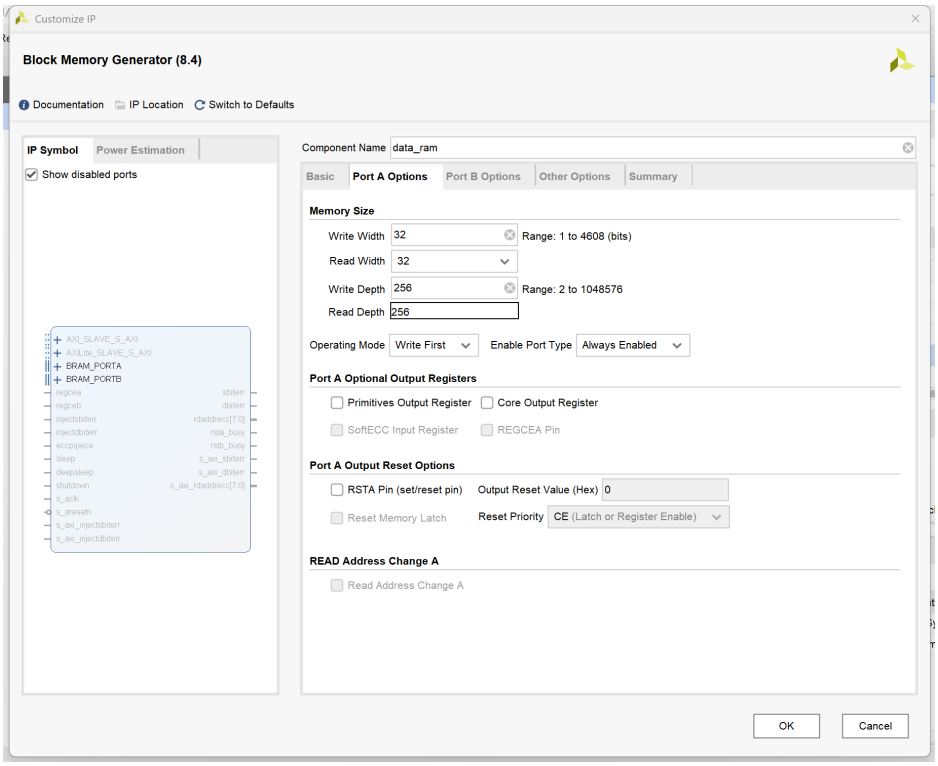

图 3.13: data_ram 配置

另外，需要把我们的指令导入到程序中，通过加载存着指令的.coe 文件。

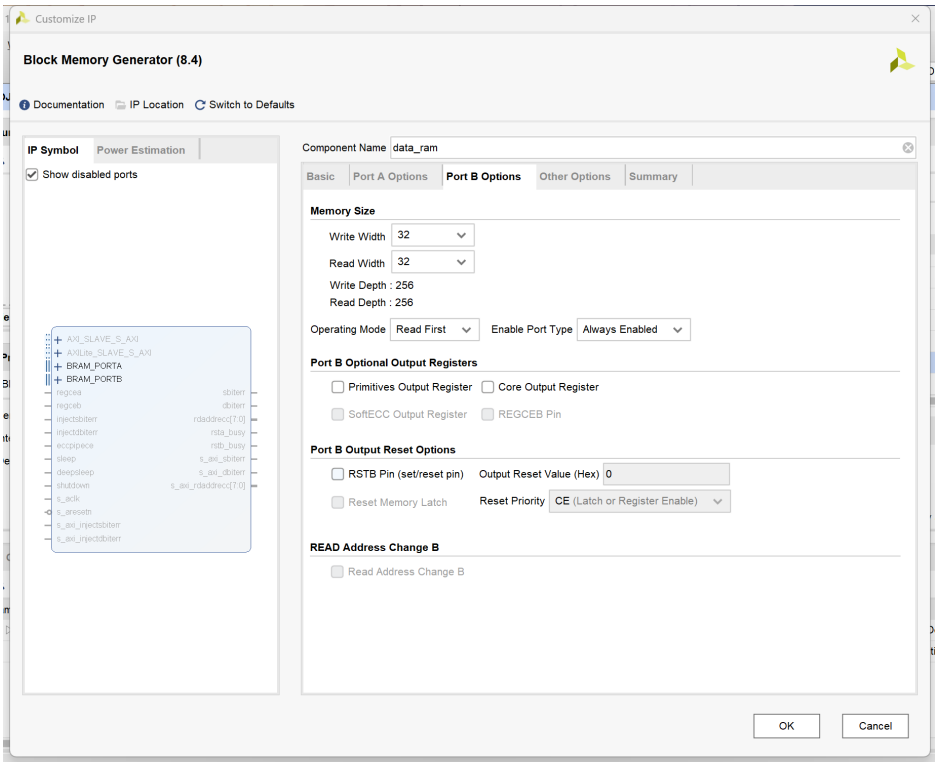

图 3.14: data_ram 配置

3.4 拓展三种运算

在完成了对 bug 的修改之后，此阶段我们需要加入扩展的三种运算，并修改源代码以能够正确运行。大致分为添加运算、添加指令、修改总线宽度（可以不修改，位宽已足够）等。

3.4.1 对 alu 模块的修改

```
    input  [3:0] alu_control, //修改成4位控制运算
    ...
    wire alu_sgt;   //有符号比较，大于置位
    wire alu_not;   //按位取反
    wire alu_hui;   //低位加载
    ...
    assign alu_add  = (alu_control == 4'b0001);
    assign alu_sub  = (alu_control == 4'b0010);
    assign alu_slt  = (alu_control == 4'b0011);
    assign alu_sltu = (alu_control == 4'b0100);
    assign alu_and  = (alu_control == 4'b0101);
    assign alu_nor  = (alu_control == 4'b0110);
    assign alu_or   = (alu_control == 4'b0111);
    assign alu_xor  = (alu_control == 4'b1000);
    assign alu_sll  = (alu_control == 4'b1001);
    assign alu_srl  = (alu_control == 4'b1010);
    assign alu_sra  = (alu_control == 4'b1011);
    assign alu_lui  = (alu_control == 4'b1100);
    assign alu_sgt  = (alu_control == 4'b1101);
    assign alu_not  = (alu_control == 4'b1110);
    assign alu_hui  = (alu_control == 4'b1111);
    ...
    wire [31:0] sgt_result;
    wire [31:0] not_result;
    wire [31:0] hui_result;
    ...
    assign hui_result = {16'd0, alu_src2[15:0]};
        //立即数装载结果为立即数移位至低半字节
    assign not_result = ~alu_src2;                  //按位取反
    ...
    wire not_equal;
    assign not_equal = (adder_result != 0);
    assign sgt_result[31:1] = 31'd0;
    assign sgt_result[0] = ~((alu_src1[31] & ~alu_src2[31]) |
        (~(alu_src1[31]^alu_src2[31]) & adder_result[31])) & not_equal;
    ...
        assign alu_result = (alu_add|alu_sub) ? add_sub_result[31:0] :
                            alu_slt           ? slt_result :
                            alu_sltu          ? sltu_result :
                            alu_and           ? and_result :
                            alu_nor           ? nor_result :
                            alu_or            ? or_result  :
```

```
                        alu_xor          ? xor_result :
                        alu_sll          ? sll_result :
                        alu_srl          ? srl_result :
                        alu_sra          ? sra_result :
                        alu_lui          ? lui_result :
                        alu_sgt          ? sgt_result :
                        alu_not          ? not_result :
                        alu_hui          ? hui_result :
                        32'd0;
endmodule
```

以上是对 alu 的修改，在上一个实验中已经进行分析，这里不再赘述。

既然对 alu 进行了修改，那么还需要对其顶层模块 EXE 进行相关修改。但我发现需要修改的只有总线宽度，原来 150 位已经足够，无需修改，其他相关模块也类似。

3.4.2 对 decode 模块的修改

```
    wire inst_SGT, inst_NOT,inst_HUI;
```

首先声明我们添加的 3 条指令。

```
    assign inst_SGT = op_zero & sa_zero & (funct == 6'b001001); // 大于置位
    assign inst_NOT = op_zero & sa_zero & (funct == 6'b001010); // 按位取反
    assign inst_HUI = (op == 6'b100111)& (rs==5'd0); // 低位加载
```

在这里对我们添加的 3 条指令进行定义，sgt 大于置位指令和 not 按位取反指令设置为 R 型指令，hui 立即数低位加载指令为 I 型指令，我们按照不同指令的格式进行定义，需要构造 6 位功能码，注意要与之前定义的操作的功能码不同。

```
    wire inst_sgt, inst_not, inst_hui;
    assign inst_sgt = inst_SGT;
    assign inst_not = inst_NOT;
    assign inst_hui = inst_HUI;
```

这里又新增了四行，定义三个变量用于后续选择 alu 控制信号。

```
    //依据立即数扩展方式分类
    wire inst_imm_zero; //立即数0扩展
    wire inst_imm_sign; //立即数符号扩展
    assign inst_imm_zero = inst_ANDI  | inst_LUI  | inst_ORI | inst_XORI | inst_HUI;
    assign inst_imm_sign = inst_ADDIU | inst_SLTI | inst_SLTIU
                         | inst_load  | inst_store;

    //依据目的寄存器号分类
    wire inst_wdest_rt;  // 寄存器堆写入地址为rt的指令
    wire inst_wdest_31;  // 寄存器堆写入地址为31的指令
    wire inst_wdest_rd;  // 寄存器堆写入地址为rd的指令
    assign inst_wdest_rt = inst_imm_zero | inst_ADDIU | inst_SLTI
                         | inst_SLTIU    | inst_load;
```

```
    assign inst_wdest_31 = inst_JAL;
    assign inst_wdest_rd = inst_ADDU | inst_SUBU | inst_SLT  | inst_SLTU
                         | inst_JALR | inst_AND  | inst_NOR  | inst_OR
                         | inst_XOR  | inst_SLL  | inst_SLLV | inst_SRA
                         | inst_SRAV | inst_SRL  | inst_SRLV | inst_SGT
                         | inst_NOT;
```

在这里需要把我们新增的三条指令添加进去，并按照不同功能进行分类，lui 操作需在高位补 0，分类到立即数 0 扩展这类中，另外两个 R 型指令都是写入 rd 中的指令，所以分类到 inst_wdest_rd 中。

```
    assign alu_control = (inst_add ? 4'b0001:
                          inst_sub ? 4'b0010:
                          inst_slt ? 4'b0011:
                          inst_sltu ? 4'b0100:
                          inst_and ? 4'b0101:
                          inst_nor ? 4'b0110:
                          inst_or ? 4'b0111:
                          inst_xor ? 4'b1000:
                          inst_sll ? 4'b1001:
                          inst_srl ? 4'b1010:
                          inst_sra ? 4'b1011:
                          inst_lui ? 4'b1100:
                          inst_sgt ? 4'b1101:
                          inst_not ? 4'b1110:
                          inst_hui ? 4'b1111:
                          4'b0000) ;
```

由于我们的 alu 是把原来的独热编码压缩成了 4bit 控制信号，所以这里也需对控制信号进行修改，4 位信号正好对应 16 个操作。

至此，我们完成了对程序的修改。现在需要加入 3 条指令到.coe 文件中。

3.4.3 添加三条指令

我们把新增的 3 条指令放在最后的跳转指令 08000000h 之前。

SGT 指令

我们添加第一条指令为 0000’0000’0100’0001’0001’1000’0000’1001，对应 16 进制为 00411809h，R 型指令前 6 位全为 0，接下来 5 位 00010 为第一个源操作数，即为 $2 寄存器，接下来 5 位 00001 为第二个源操作数，即为 $1 寄存器，接下来 5 位 00011 为第二个源操作数，即为 $3 寄存器，后 5 位 00000，代表特殊操作，置 0 即可，最后 6 位为功能码，对应 SGT 指令，与我们编写的 SGT 指令对应。这条指令的意思也就是比较 $2 寄存器和 $1 寄存器的大小，若 $2 寄存器大于 $1 寄存器，则把 $3 寄存器置为 1。

NOT 指令

第二条指令为 0000’0000’0100’0001’0001’1000’0000’1010，对应 16 进制为 0041180ah，最后 6 位功能码为我们定义的 NOT 指令操作码，这条指令的意思是把 $2 寄存器的值按位取反赋值到 $3 寄存器中。

HUI 指令

第三条指令为 I 型指令，我们基于 I 型指令的格式进行设计，操作码为 1001’1100’0000’0001’0000’1111’1111’1111，对应 16 进制 9c010fffh，前 6 位 100111 为功能码，对应 HUI 指令，接下来 5 位为寄存器操作数，是第一个源操作数，接下来 5 位 00001 是目的寄存器，对应 $1 寄存器，最后 16 位为立即数，是第二个源操作数。这条指令的意思就是把 16 位立即数加载到 $1 寄存器中，并在高位用 0 进行拓展。

3.5 实验结果分析

3.5.1 仿真验证

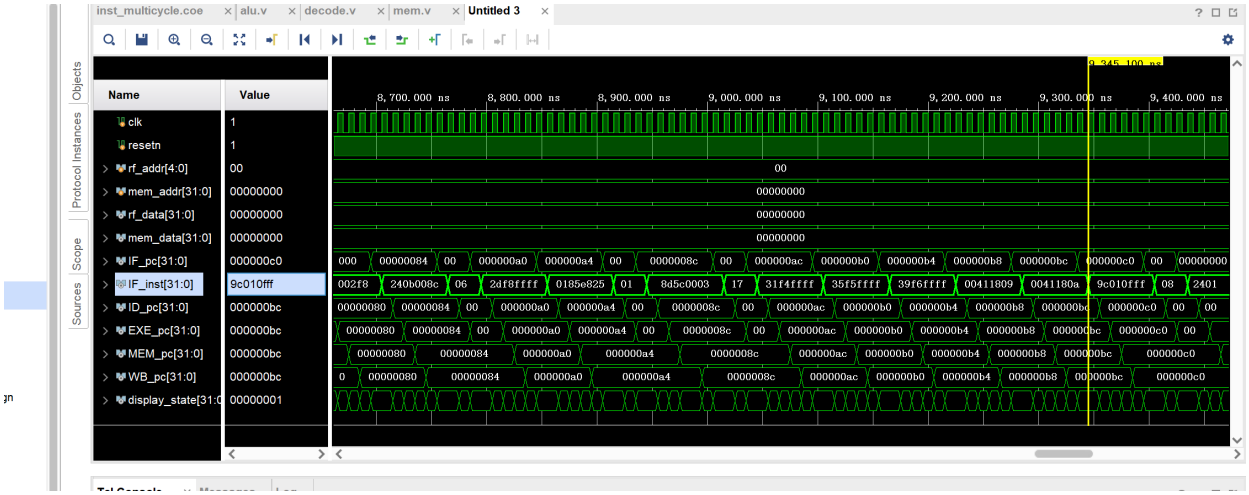

图 3.15: 仿真结果

可以看到我们成功加入了 00411809h、0041180ah、9c010fffh 三条指令。

3.5.2 上箱验证

SGT 指令

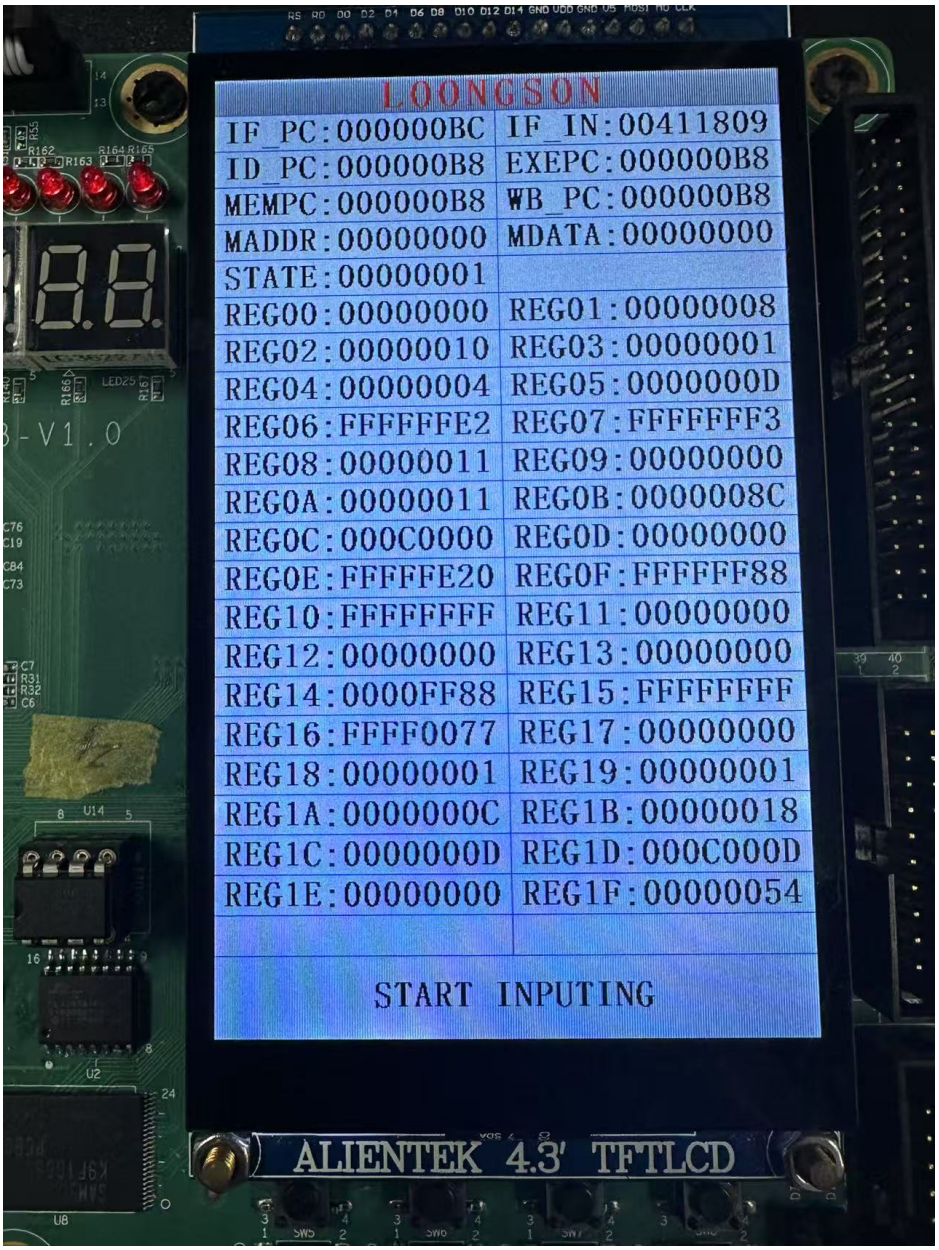

图 3.16: SGT 指令上箱结果

在执行完 00411809h 这条指令后，由于 $2 寄存器为 00000010h，$1 寄存器为 00000008h，所以 $3 寄存器置位 1，可以看到 $3 寄存器确实为 1。**NOT 指令**

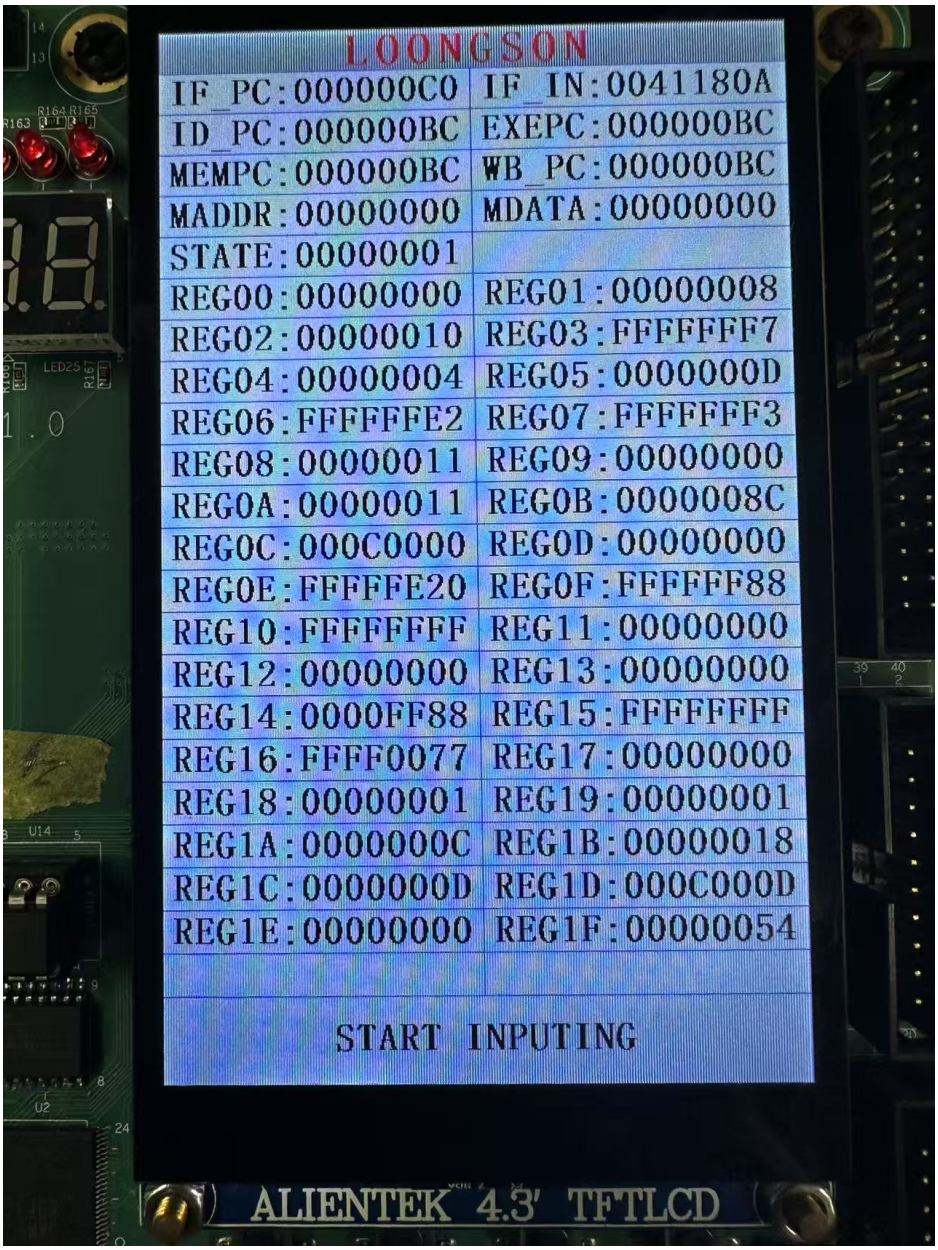

图 3.17: NOT 指令上箱结果

执行完 0041180ah 这条指令之后，由于 $1 寄存器为 00000008h，按位取反结果为 fffffff7h，保存在 $3 寄存器中，可以看到确实如此。**HUI 指令**

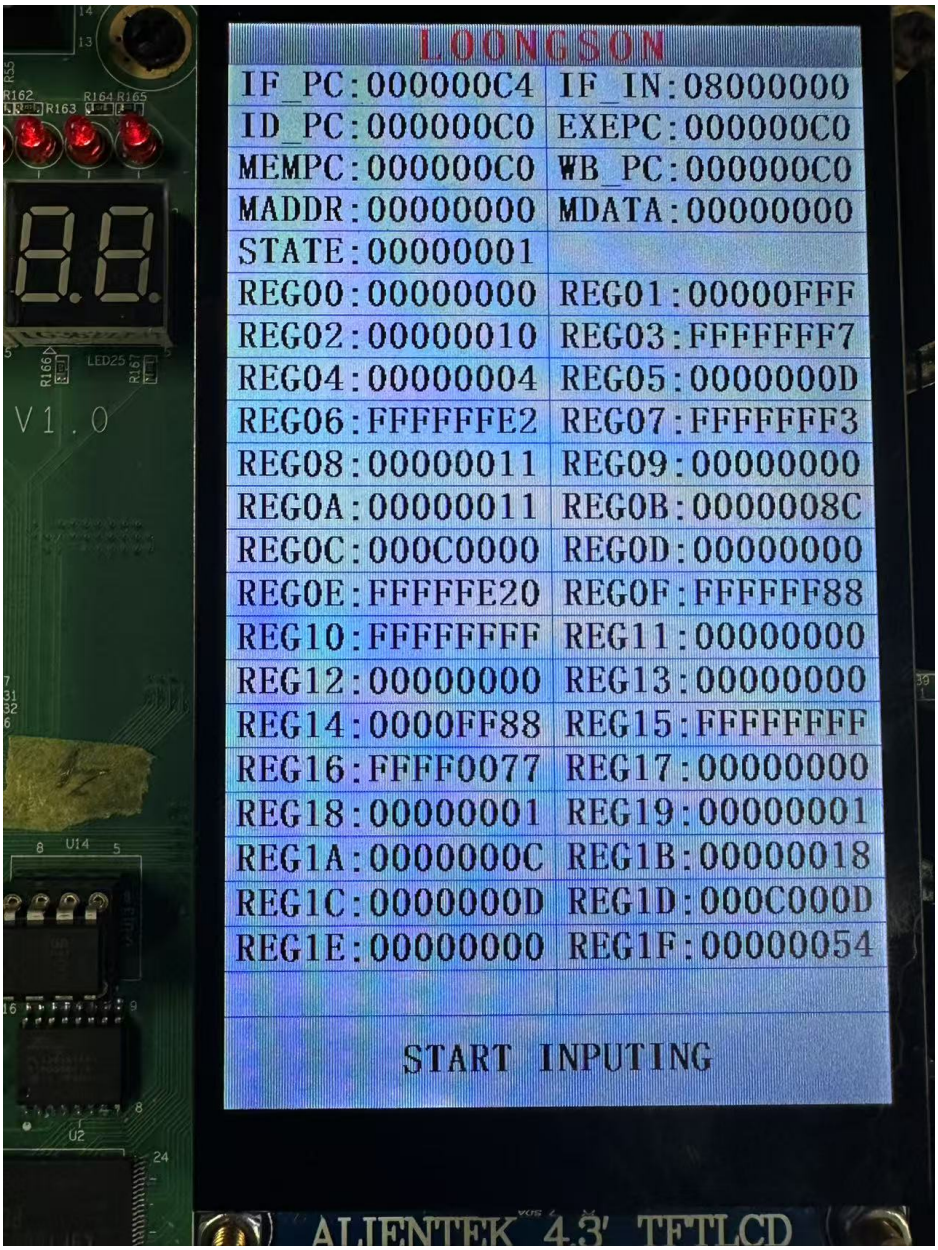

图 3.18: HUI 指令上箱结果

最后在执行完 HUI 这条指令后，把立即数 0fffh 加载到 $1 寄存器，并在高位用 0 扩展，可以看到结果为 00000fffh，完全正确。

通过以上结果，验证了我们添加的三条指令的正确性。

4 总结感想

1. 通过本次实验，我对于 cpu 的结构有了更深刻的认识，其内部主要是由运算器、控制单元、时钟、寄存器构成。

2.cpu 运作的过程，其实就是由取指、译码、执行、访存、写回这 5 个步骤组成，并关联了内存、寄存器、运算器、控制单元等逻辑器件。

3. 熟练了 IP 核的搭建过程，对于 primitives output register 这一选项的勾选有了更深刻的理解，

勾选这个可以提高系统的时钟频率，但要注意时序约束的问题。

4. 能够基于不同指令的格式进行设计，并得到了验证。