

关于CPU测试

*注意：为了便于使用 Mars 进行测试，CPU 设计做如下规定：

Mars 中 CPU 采用冯诺依曼结构，我们使用它的 default 内存设置，故指令存储起始地址为：0x00400000，数据存储起始地址为 0x10010000，由于我们设计的 CPU 为哈佛结构，指令和数据的起始地址都为 0，所以最后请在 PC 模块、指令存储模块、数据存储模块进行地址映射（将逻辑地址转换为物理地址 0）。

a) 前仿真单条指令的测试

在 modelsim 中，可以使用如下代码初始化 iram:

```
initial begin
    $readmemh("lout.txt", ram);
end
```

但 initial 块不可综合，若下板时需要初始化 iram 请参看后面介绍的 ip 核使用示例。

测试指令时可在 cpu 的 testbench 中添加如图 8.4.11 代码打印结果，将结果打印到文件中（可以自行查找 verilog 的系统函数的使用方法）

```
integer file_output;
integer counter = 0;

initial begin
    //$dumpfile("mydump.txt");
    //$dumpvars(0,cpu_tb.uut.pcreg.data_out);
    file_output = $fopen("result.txt");
    // Initialize Inputs
    clk = 0;
    rst = 1;

    // Wait 100 ns for global reset to finish
    #50;
    rst = 0;
    // Add stimulus here
end
```

```

2      always @ (posedge clk)
3      begin
4          counter = counter + 1;
5
6          $fdisplay(file_output, "regfiles0 = %h", cputb.uut.rf.regfiles[0]);
7          $fdisplay(file_output, "regfiles1 = %h", cputb.uut.rf.regfiles[1]);
8          $fdisplay(file_output, "regfiles2 = %h", cputb.uut.rf.regfiles[2]);
9          $fdisplay(file_output, "regfiles3 = %h", cputb.uut.rf.regfiles[3]);
10         $fdisplay(file_output, "regfiles4 = %h", cputb.uut.rf.regfiles[4]);
11         $fdisplay(file_output, "regfiles5 = %h", cputb.uut.rf.regfiles[5]);
12         $fdisplay(file_output, "regfiles6 = %h", cputb.uut.rf.regfiles[6]);
13         $fdisplay(file_output, "regfiles7 = %h", cputb.uut.rf.regfiles[7]);
14         $fdisplay(file_output, "regfiles8 = %h", cputb.uut.rf.regfiles[8]);
15         $fdisplay(file_output, "regfiles9 = %h", cputb.uut.rf.regfiles[9]);
16         $fdisplay(file_output, "regfiles10 = %h", cputb.uut.rf.regfiles[10]);
17         $fdisplay(file_output, "regfiles11 = %h", cputb.uut.rf.regfiles[11]);
18         $fdisplay(file_output, "regfiles12 = %h", cputb.uut.rf.regfiles[12]);
19         $fdisplay(file_output, "regfiles13 = %h", cputb.uut.rf.regfiles[13]);
20         $fdisplay(file_output, "regfiles14 = %h", cputb.uut.rf.regfiles[14]);
21         $fdisplay(file_output, "regfiles15 = %h", cputb.uut.rf.regfiles[15]);
22         $fdisplay(file_output, "regfiles16 = %h", cputb.uut.rf.regfiles[16]);
23         $fdisplay(file_output, "regfiles17 = %h", cputb.uut.rf.regfiles[17]);
24         $fdisplay(file_output, "regfiles18 = %h", cputb.uut.rf.regfiles[18]);
25         $fdisplay(file_output, "regfiles19 = %h", cputb.uut.rf.regfiles[19]);
26         $fdisplay(file_output, "regfiles20 = %h", cputb.uut.rf.regfiles[20]);
27         $fdisplay(file_output, "regfiles21 = %h", cputb.uut.rf.regfiles[21]);
28         $fdisplay(file_output, "regfiles22 = %h", cputb.uut.rf.regfiles[22]);
29         $fdisplay(file_output, "regfiles23 = %h", cputb.uut.rf.regfiles[23]);
30         $fdisplay(file_output, "regfiles24 = %h", cputb.uut.rf.regfiles[24]);
31         $fdisplay(file_output, "regfiles25 = %h", cputb.uut.rf.regfiles[25]);
32         $fdisplay(file_output, "regfiles26 = %h", cputb.uut.rf.regfiles[26]);
33         $fdisplay(file_output, "regfiles27 = %h", cputb.uut.rf.regfiles[27]);
34         $fdisplay(file_output, "regfiles28 = %h", cputb.uut.rf.regfiles[28]);
35         $fdisplay(file_output, "regfiles29 = %h", cputb.uut.rf.regfiles[29]);
36         $fdisplay(file_output, "regfiles30 = %h", cputb.uut.rf.regfiles[30]);
37         $fdisplay(file_output, "regfiles31 = %h", cputb.uut.rf.regfiles[31]);
38
39         $fdisplay(file_output, "instr = %h", cputb.uut.iram.douta);
40         $fdisplay(file_output, "pc = %h", cputb.uut.pcreg.data_out);
41     end

```

图 8.4.11 testbench 示例代码

以下举一条指令的测试进行说明：

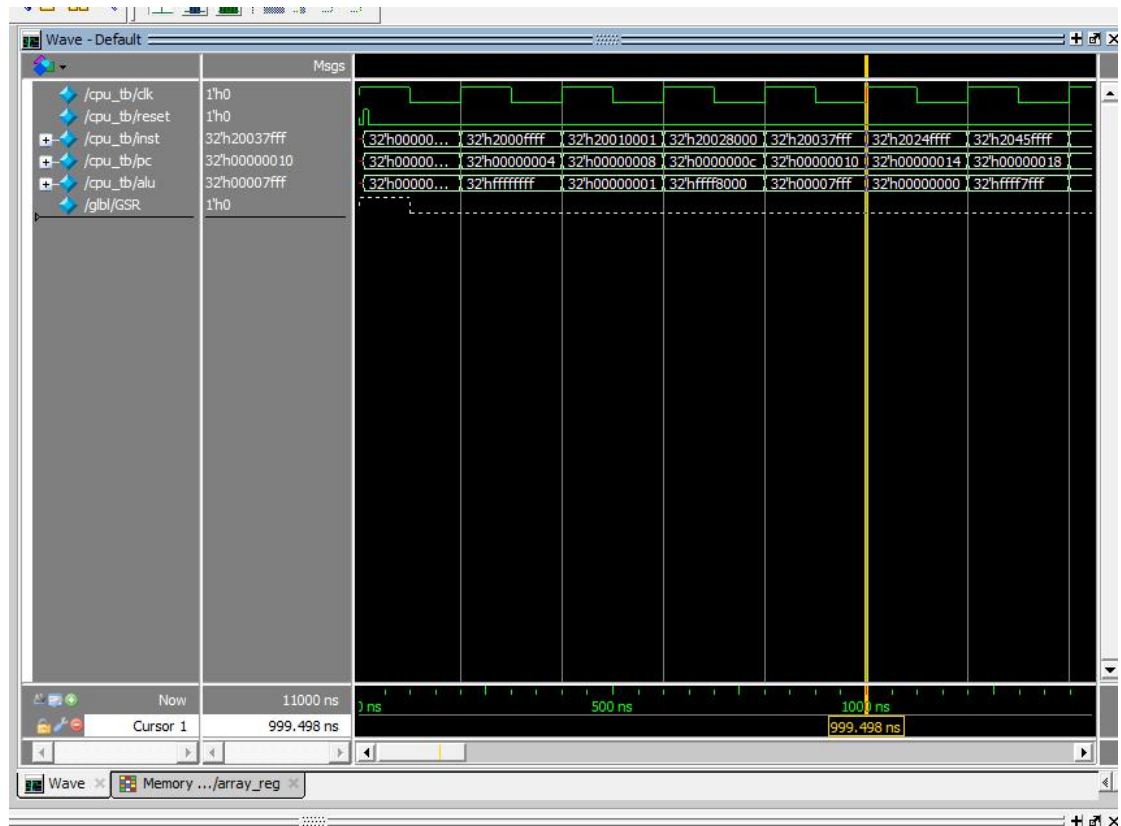
比如 addi 指令，测试指令如下：

```

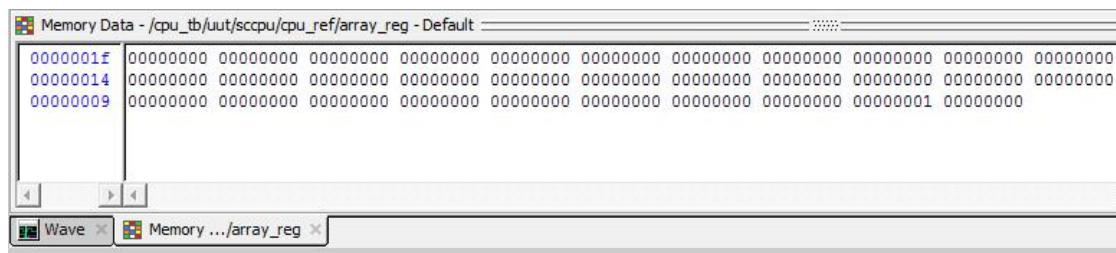
sll $0,$0,0
addi $0,$0,0xffff
addi $1,$0,0x0001
addi $2,$0,0x8000
.....
.....
.....
addi $29,$29,0xdbc1
addi $30,$30,0x7871
addi $31,$31,0x7771

```

使用 modelsim 进行仿真结果如图：



测试指令执行结果时对每一条指令的执行结果进行查看，可查看 Modelsim 里的内存模块，如执行完 `addi $1,$0,0x0001` 指令后寄存器堆的结果如图：



b) 指令的边界数据测试

我们 CPU 测试时需要对各条指令进行边界数据进行测试以保证设计的 CPU 的正确性。

还是以 addi 指令为例，可以用以下汇编指令进行测试：

```
addi $1,$0,0x0001
addi $2,$0,0x8000
addi $3,$0,0x7fff
addi $4,$1,0xffff
addi $5,$2,0xffff
addi $6,$3,0x0006
```

```
addi $7,$0,0x000f
addi $8,$0,0x00f0
addi $9,$0,0x0f00
addi $10,$0,0xf000
addi $7,$0,0x000f
addi $8,$0,0x00f0
addi $9,$0,0x0f00
addi $10,$0,0xf000
```

```
addi $11,$0,0x5555
addi $12,$0,0xAAAA
addi $11,$11,0x5555
addi $12,$12,0xAAAA
```

```
addi $13,$0,0x000f
addi $14,$0,0x00ff
addi $15,$0,0x0fff
addi $13,$13,0x000f
addi $14,$14,0x00ff
addi $15,$15,0x0fff
```

```
addi $16,$0,0x0010
addi $17,$0,0x0011
addi $18,$0,0x0012
addi $19,$0,0x0013
addi $20,$0,0x0014
addi $21,$0,0x0015
addi $22,$0,0x0016
```

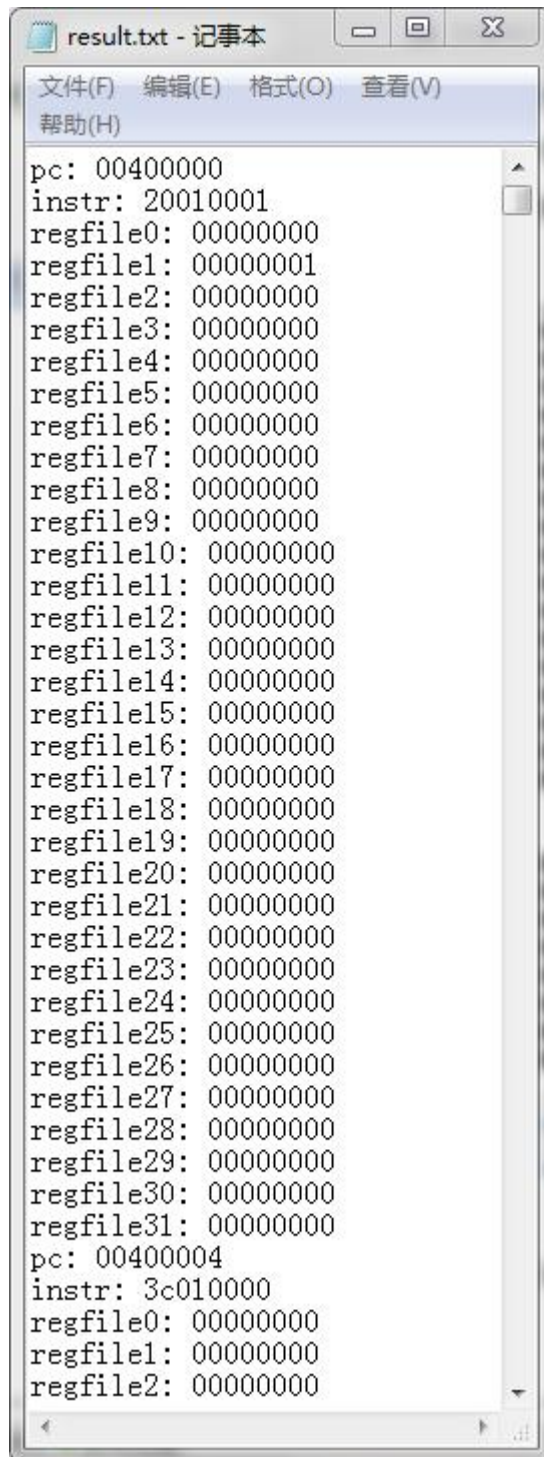
```
addi $23,$0,0x0017
addi $24,$0,0x0018
addi $25,$0,0x0019
addi $26,$0,0x001A
addi $27,$0,0x001B
addi $28,$0,0x001C
addi $29,$0,0x001D
addi $30,$0,0x001E
addi $31,$0,0x001F
```

```
addi $16,$16,0x0001
addi $17,$17,0xf001
addi $18,$18,0x8001
addi $19,$19,0x7001
addi $20,$20,0x0001
addi $21,$21,0x0211
addi $22,$22,0xab1
addi $23,$23,0xdc01
addi $24,$24,0xa001
addi $25,$25,0xb001
addi $26,$26,0xe001
addi $27,$27,0x7d01
addi $28,$28,0xcba1
addi $29,$29,0xdbc1
addi $30,$30,0x7871
addi $31,$31,0x7771
```

d) 随机指令测试

可以自行编写一些符合 MIPS 规范的指令序列。将这序列分别放到 CPU 仿真状态下执行和 MARS 上去执行，分别产生两个执行结果文件，比较执行结果文件来判断 CPU 执行指令是否对。

用 Mars 运行 MIPS 汇编程序后会在 Mars 目录下生成一个 result.txt 文件，内容如图 8.4.12，为运行每条指令后的 指令地址：pc、指令十六进制码：instr、寄存器堆十六进制码：regfile；



```
result.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V)
帮助(H)

pc: 00400000
instr: 20010001
regfile0: 00000000
regfile1: 00000001
regfile2: 00000000
regfile3: 00000000
regfile4: 00000000
regfile5: 00000000
regfile6: 00000000
regfile7: 00000000
regfile8: 00000000
regfile9: 00000000
regfile10: 00000000
regfile11: 00000000
regfile12: 00000000
regfile13: 00000000
regfile14: 00000000
regfile15: 00000000
regfile16: 00000000
regfile17: 00000000
regfile18: 00000000
regfile19: 00000000
regfile20: 00000000
regfile21: 00000000
regfile22: 00000000
regfile23: 00000000
regfile24: 00000000
regfile25: 00000000
regfile26: 00000000
regfile27: 00000000
regfile28: 00000000
regfile29: 00000000
regfile30: 00000000
regfile31: 00000000
pc: 00400004
instr: 3c010000
regfile0: 00000000
regfile1: 00000000
regfile2: 00000000
```

图 8.4.12 寄存器状态

用 Mars 导出编译后的 MIPS 指令十六进制格式文件，如图 8.4.13，点击红框处，导出格式选择十六进制。

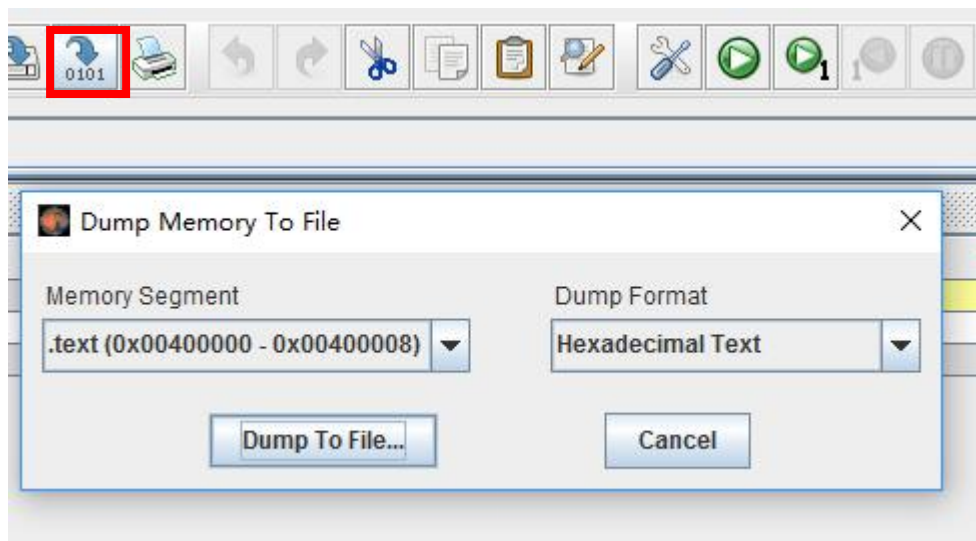


图 8.4.13 导出文件

用自己的 CPU 运行导出的 MIPS 汇编程序，按 `cpu_tb.v` 中，如图 8.4.14，将 `pc`、`instr`、`regfile` 结果输出到文件，在 CPU 工程目录下生成另一个 `result.txt` 文件；

```
integer file_output;
integer counter = 0;

initial begin
    //$dumpfile("mydump.txt");
    //$dumpvars(0,cpu_tb.uut.pcreg.data_out);
    file_output = $fopen("result.txt");
    // Initialize Inputs
    clk = 0;
    rst = 1;

    // Wait 100 ns for global reset to finish
    #50;
    rst = 0;
    // Add stimulus here

    // #100;
    //$fclose(file_output);
end

always begin
    #50;
    clk = ~clk;
    if(clk == 1'b0) begin
        if(counter == 400) begin
            $fclose(file_output);
        end
        else begin
            counter = counter + 1;
            $fdisplay(file_output, "pc: %h", test.uut.pcreg.data_out);
            $fdisplay(file_output, "instr: %h", test.uut.pipe_if.ram_outdata);
            $fdisplay(file_output, "regfile0: %h", test.uut.pipe_id.rf.reg0.data_out);
            $fdisplay(file_output, "regfile1: %h", test.uut.pipe_id.rf.reg1.data_out);
            $fdisplay(file_output, "regfile2: %h", test.uut.pipe_id.rf.reg2.data_out);
        end
    end
end
```

图 8.4.14 结果文件

比对 1 和 2 中生成的两个结果文件，比对可使用 UltraCompare 或 Notepad++ 等工具。

如图 8.4.15，对比发现在 pc 为 8，执行指令 3c038000 后 \$1 寄存器的结果出现了错误，需要进一步调试修改。

例子中运行的指令为：

```
sll $0,$0,0
addi $1,$0,0xf
addi $0,$1,0x1
```

结果发现第三条指令出现错误，\$0 寄存器结果异常，因为零号寄存器恒置零，所以应修改寄存器堆中对零号寄存器的写操作。

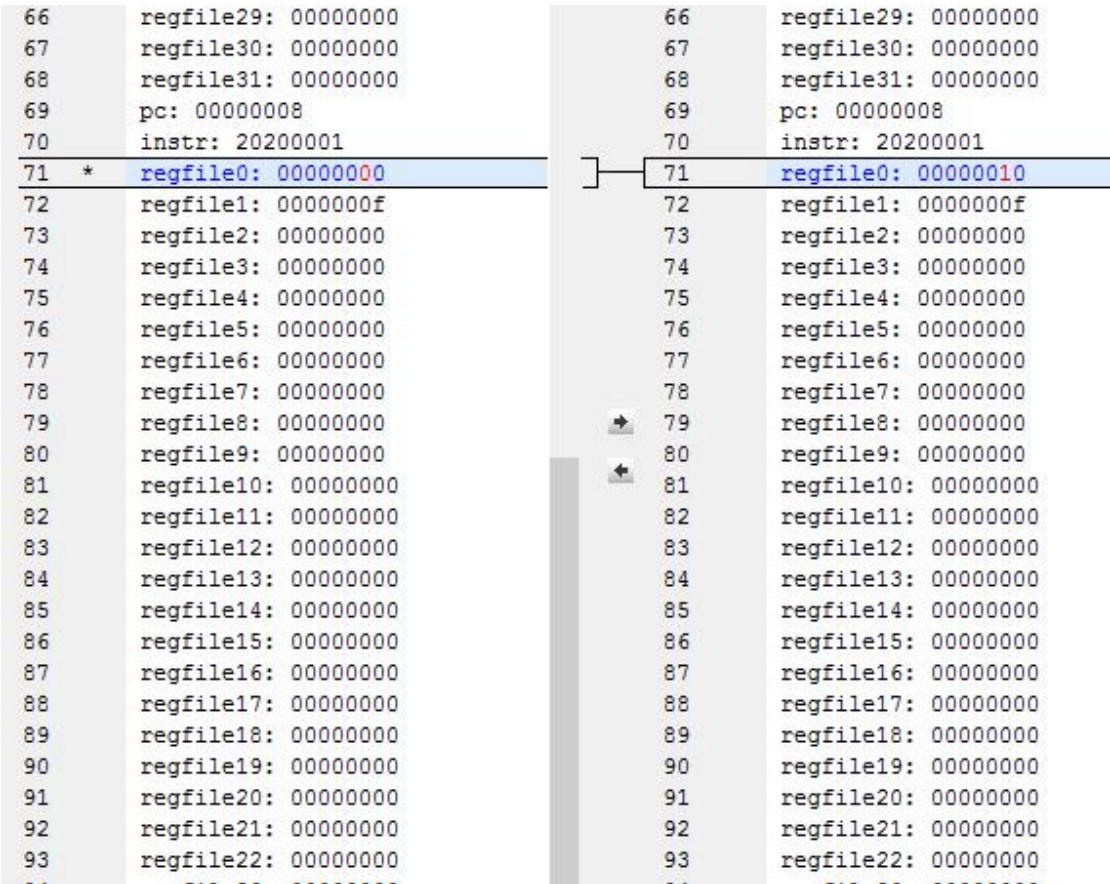


图 8.4.15 对比情况

e) 后仿真测试

Modelsim 的仿真分为前仿真和后仿真，下面具体介绍一下两者的区别。

- 前仿真

前仿真也称为功能仿真，主旨在于验证电路的功能是否符合设计要求，其特点是不考虑电路门延迟与线延迟，主要是验证电路与理想情况是否一致。可综合 FPGA 代码是用 RTL 级代码语言描述的，其输入为 RTL 级代码与 Testbench。

- 后仿真

后仿真也称为时序仿真或者布局布线后仿真，是指电路已经映射到特定的工艺环境以后，综合考虑电路的路径延迟与门延迟的影响，验证电路能否在一定时序条件下满足设计构想的过程，是否存在时序违规。其输入文件为从布局布线结果中抽象出来的门级网表、Testbench 和扩展名为 SDO 或 SDF 的标准时延文件。SDO 或 SDF 的标准时延文件不仅包含门延迟，还包括实际布线延迟，能较好地反映芯片的实际工作情况。一般来说后仿真是必选的，检查设计时序与实际的 FPGA 运行情况是否一致，确保设计的可靠性和稳定性。选定了器件分配引脚后在做后仿真。

f) 下板测试

由于我们自行编写的指令 RAM 用来初始化内存的 initial 指令是不可综合的，无法在开发板上运行，所以，我们可以使用 vivado 提供的 ip 核来替换我们的 ram，其可以使用一个 coe 文件来初始化内存。

Coe 为初始化 ROM 的配置文件，以下为 coe 文件格式实例

memory_initialization_radix=16; #16 表示指令以 16 进制表示，可以按需求更改

memory_initialization_vector= #由此开始为指令序列，以“,”隔开“;”结束

00000000,

241d03fc,

0800001a;

跟随以下步骤来添加 IP 核：

如图选择 IP 核列表

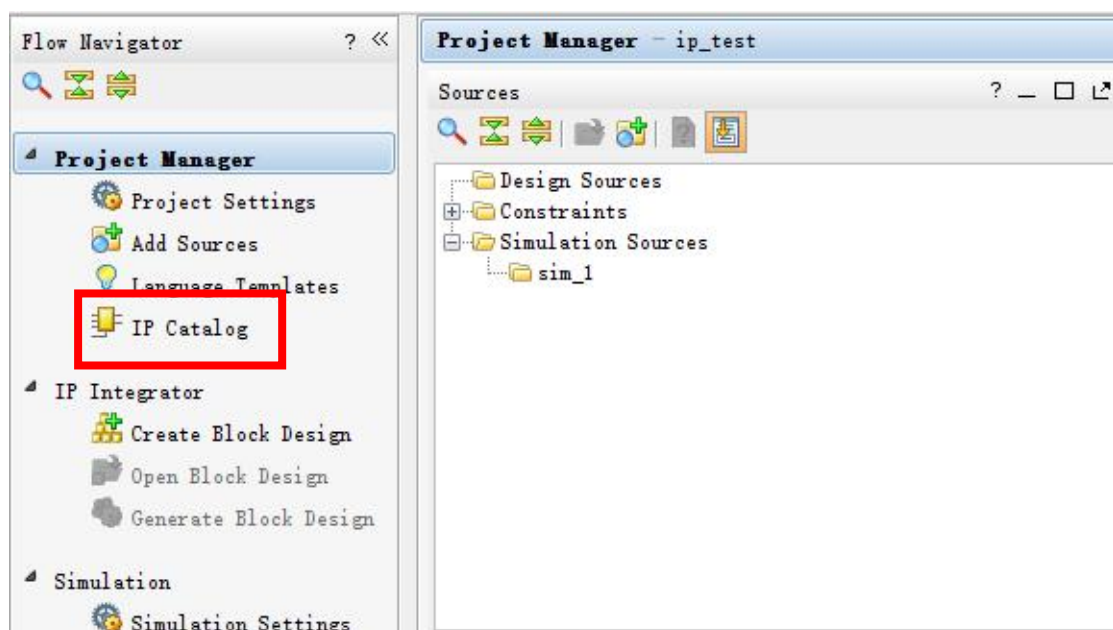


图 8.4.12 IP 核列表选择

在列表中选择 Memories & Storage Elements 中的 RAMs & ROMs，双击 Distributed Memory Generator

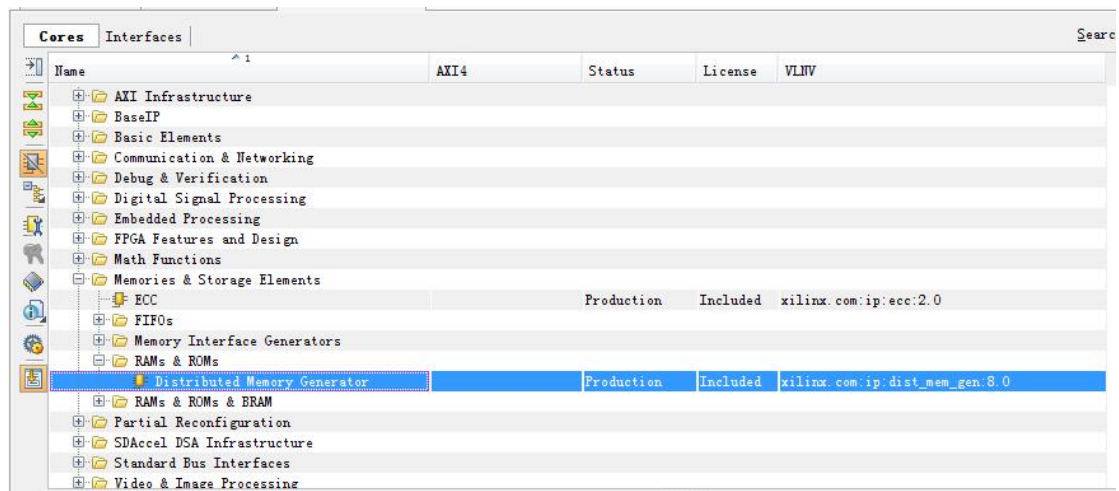
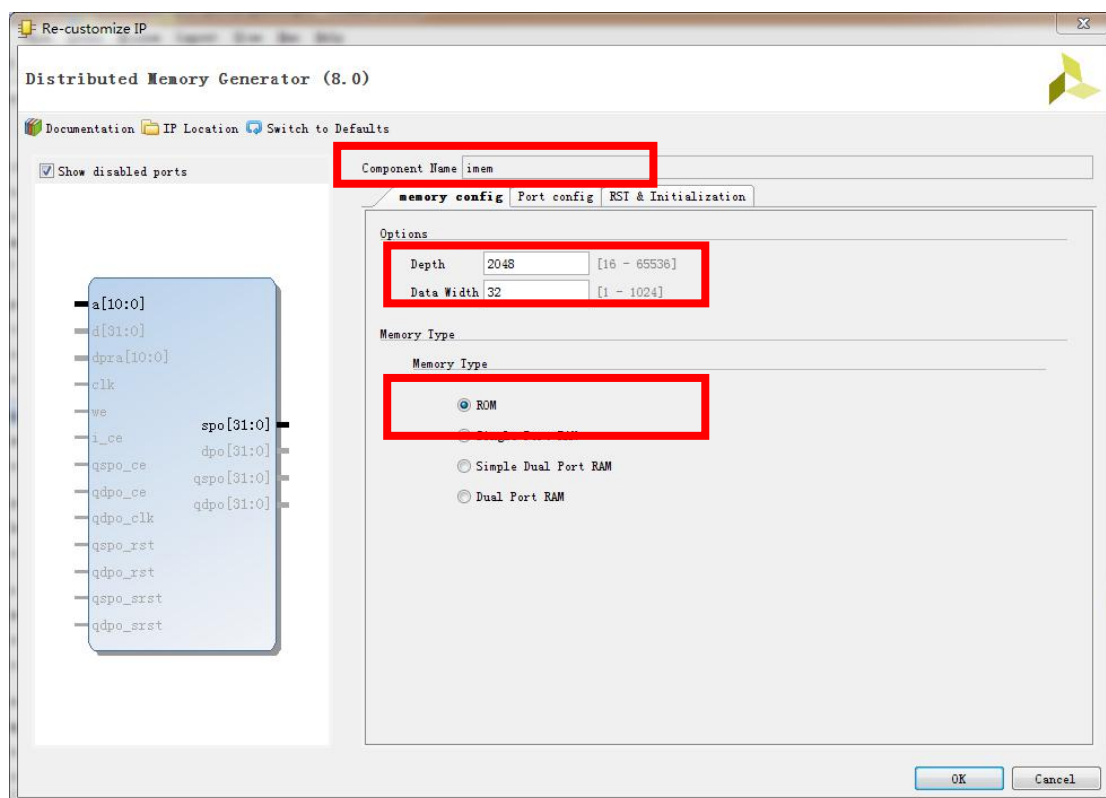


图 8.4.13 列表选择

根据图 8.4.14 配置 ip 核，下图均为推荐配置，可根据自己的需求来更改



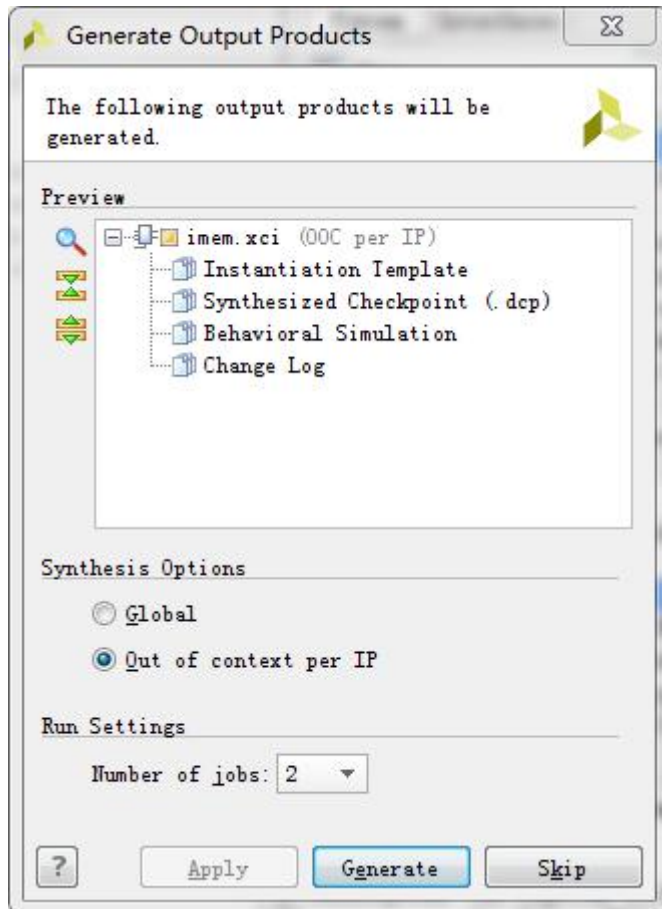
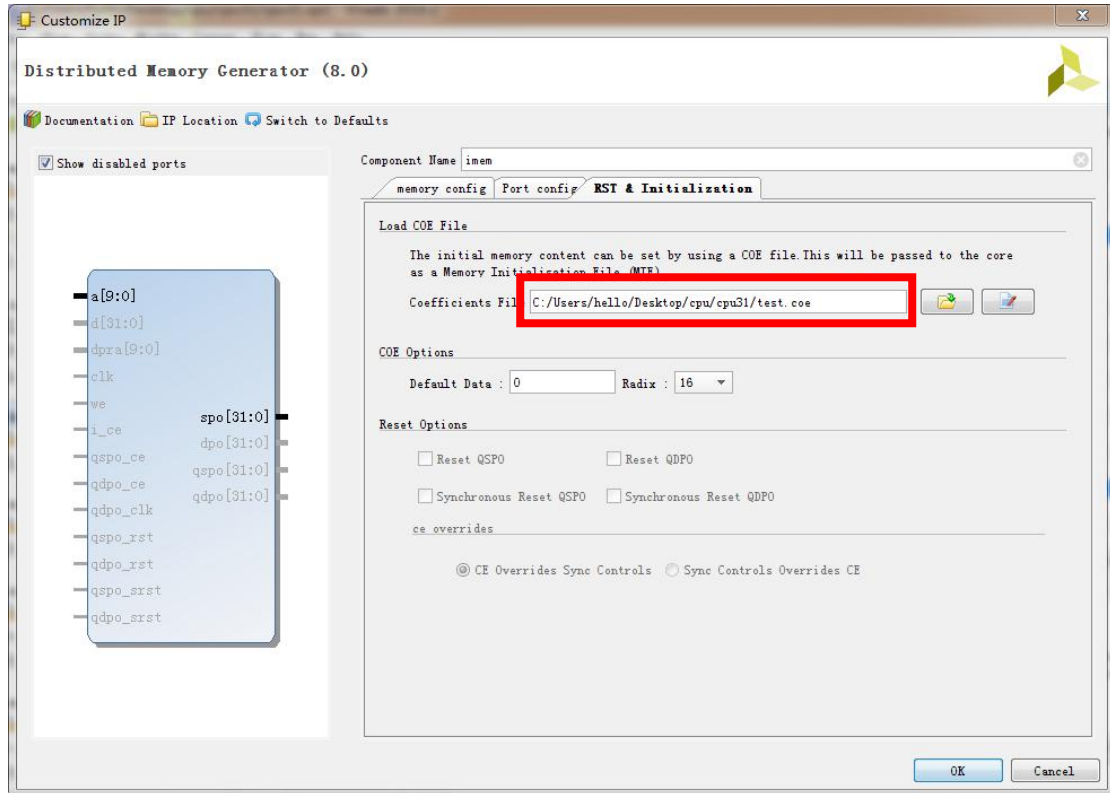


图 8.4.15 IP 核推荐配置

双击 ip 核模块，可以对 ip 核进行修改

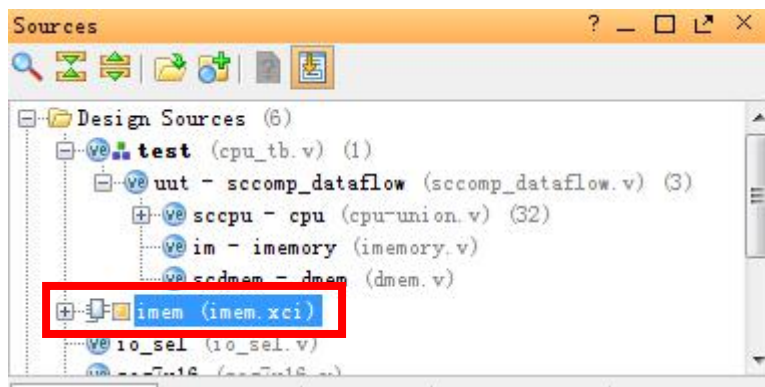


图 8.4.16 IP 核模块

在 CPU 调用 IP 核时可参照如图 8.4.17 方式

```
/*指令存储器*/  
imem imem(ip_in[12:2], inst);  
//imemory im(pc, inst);
```

图 8.4.17 IP 核调用方式