

计原 - 大实验成果交流

涂珂 2011011273
傅左右 2011011264
计 14 - 402 组

December 29, 2013

实验成果一览

实验成果参数

- ▶ CPU 主频为 6.25MHz (12.5MHz 有时会出一些问题，所以只能二分之，6.25MHz 是稳定频率)
- ▶ RAM 频率为 25MHz
- ▶ 正常运行 kernel 内核程序，正常运行所有 project1 程序。
- ▶ VGA 分辨率为 640*480，VGA (显存) 运行频率 25MHz

实验成果简列

- ▶ 清晰的模块分工
- ▶ 指令集改进，指令集汇编工具
- ▶ 数据旁路元件
- ▶ 冒险检测单元
- ▶ 完整的 VGA、LED、可调时钟调试工具链
- ▶ FLASH 自启动
- ▶ 使用地址映射，统一管理外围 I/O 设备
- ▶ 串口通信
- ▶ VGA、键盘交互
 - ▶ VGA 等宽 ASCII 字符集显示
 - ▶ VGA 双端 FIFO 显存
 - ▶ 键盘输入、支持换行、发送串口与 VGA 的记事本程序

设计方案

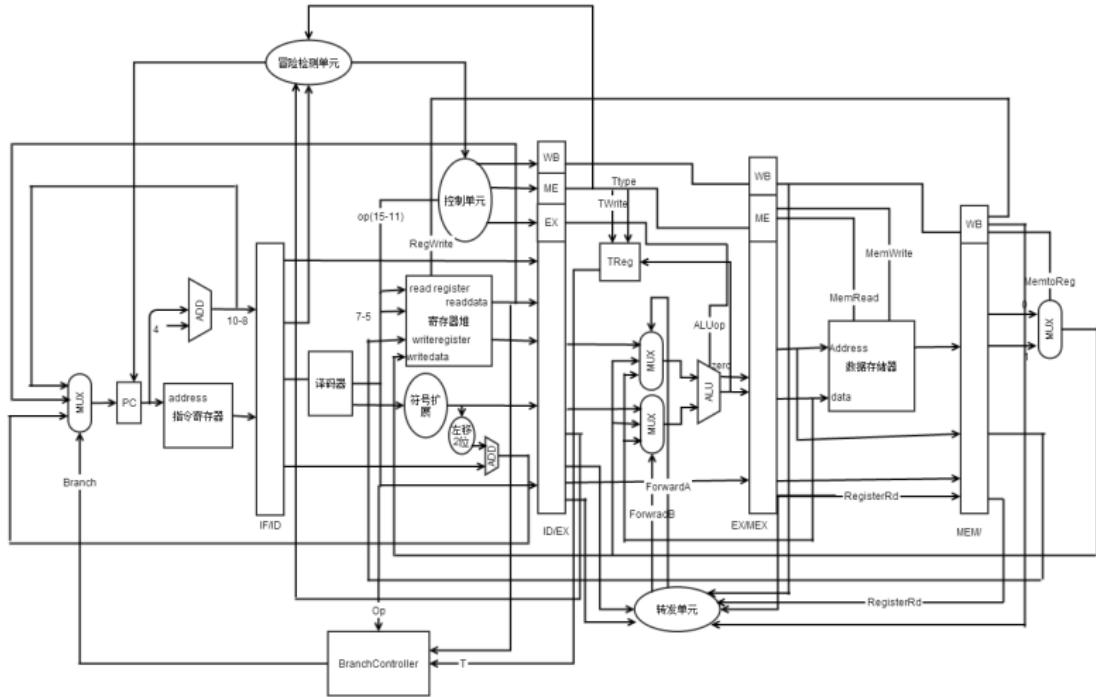


Figure: 数据通路图 datapath.png

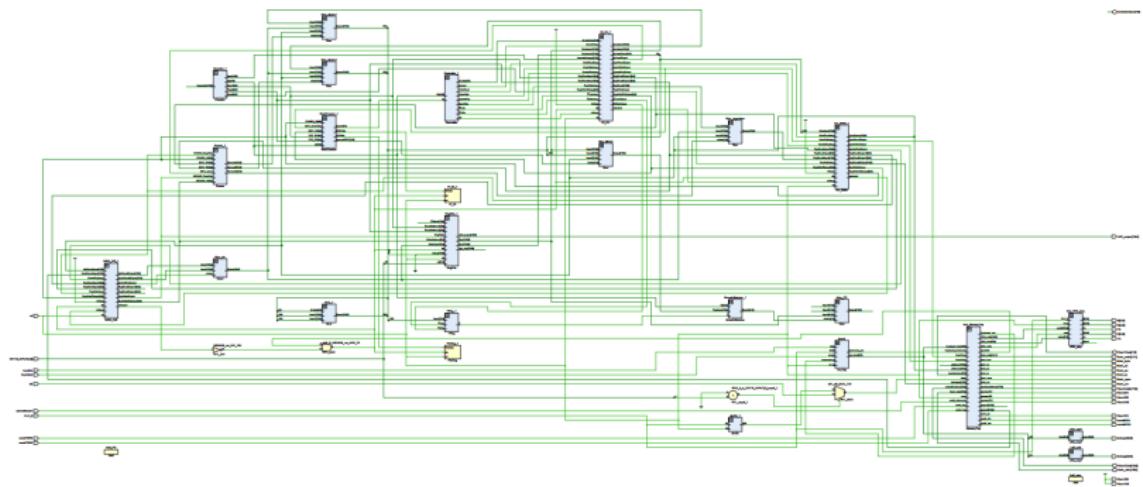


Figure: RTL 综合图 cpu-schematic.pdf

重新设计、并且成功实现指令集任务

指令集 (402)

1. THCO MIPS 基本指令集

2. 扩展指令集

- ▶ JRRA
- ▶ SLTI
- ▶ ADDSP3
- ▶ NOT
- ▶ SLT

重新设计指令

- ▶ 用前 5 位表示 op。共 30 条。
- ▶ 加 * 为扩展指令。
- ▶ XXX, YYY, ZZZ 为寄存器标号。
- ▶ III 为立即数。
- ▶ 把类型相近的 op 连续起来，这样写代码就可以用大于小于判断了。

R 型指令

R	指令结构
MFIH	00001XXX00000000
MFPC	00010XXX00000000
MTIH	00011XXX00000000
MTSP	00100XXX00000000
AND	00101XXXYYY00000
OR	00110XXXYYY00000
*NOT	00111XXXYYY00000
*SLT	01000XXXYYY00000
CMP	01001XXXYYY00000
SLL	01010XXXYYYIII00
SRA	01011XXXYYYIII00
ADDU	01100XXXYYYZZZ00
SUBU	01101XXXYYYZZZ00

I型指令

I	指令结构
ADDSP	01110IIIIIIII000
LW_SP	01111XXX000000000
ADDIU	10000XXXIIIIIIII
*SLTI	10001XXXIIIIIIII
*ADDSP3	10010XXXIIIIIIII
LI	10011XXXIIIIIIII
ADDIU3	10100XXXYYY0IIII
LW	10101XXXYYYIIIIII
SW	10110XXXYYYIIIIII
SW_SP	10111XXXYYYIIIIII

B 型指令

B 指令结构

B 11000IIIIIIIIII

BTEQZ 11001IIIIIIII000

BEQZ 11010XXXIIIIIIII

BNEZ 11011XXXIIIIIIII

J型、NOP 指令

J 指令结构

*JRRA 1110000000000000

JR 11101XXX00000000

NOP 0000000000000000

借助汇编器 thcoas.py (by taccoraw)

我们在武祥晋同学 (2011011278) 的基于 python 实现的汇编翻译器的基础上进行了改进，实现了对我们自己定义的指令集的编译工作。

改进后的汇编器可以将 MIPS 汇编代码编译输出二进制文件，并在命令行上输出丰富的调试信息（二进制、十六进制代码、代码原文行号、翻译后的代码地址号、相应指令），从而极大的方便了后续的调试。（尤其是需要蛋疼的单步跟踪的时候 ==）

```
C:\Windows\system32\cmd.exe
D:\study\2013fall\ComputerOrganization\lastproject\makecomputer\assembler>python thcoas.py test_helloworld.s
L:1  1010010110111111 0xa5bf C:0  ['LI', 'R5', '0xBf']
L:2  0101010110100000 0x55a0 C:1  ['SLL', 'R5', 'R5', '0x0']
L:4  1010001100110010 0xa332 C:2  ['LI', 'R3', '0x32']
L:5  0001010000000000 0x1400 C:3  ['MFPC', 'R4']
L:6  1000110000000011 0x8c03 C:4  ['ADDIU', 'R4', '0x0003']
L:7  0000000000000000 0x0   C:5  ['NOP']
L:8  11000000000011101 0xc01d C:6  ['B', 'TESTW']
L:9  0000000000000000 0x0   C:7  ['NOP']
L:10 1011110101100000 0xbd60 C:8  ['SW', 'R5', 'R3', '0x0']
L:11 0000000000000000 0x0   C:9  ['NOP']
L:13 1010001100110011 0xa333 C:10 ['LI', 'R3', '0x33']
L:14 0001010000000000 0x1400 C:11 ['MFPC', 'R4']
L:15 1000110000000011 0x8c03 C:12 ['ADDIU', 'R4', '0x0003']
L:16 0000000000000000 0x0   C:13 ['NOP']
```

扩展成果分享

1. 数据旁路, 冒险检测

- ▶ 数据冲突检测单元, 看寄存器堆取出的值是否是 EXE 或 MEM 阶段未写回的值, 若是, 则通过旁路引回。共引入了 4 条旁路。
- ▶ 控制冲突检查单元, 主要是检测跳转指令所导致的寄存器未写回的冲突, 并产生信号通过辅助的旁路元件处理解决相应的冲突。主要是提前判断跳转指令而寄存器未写回的冲突。引入了 2 条旁路.

旁路与冒险检测

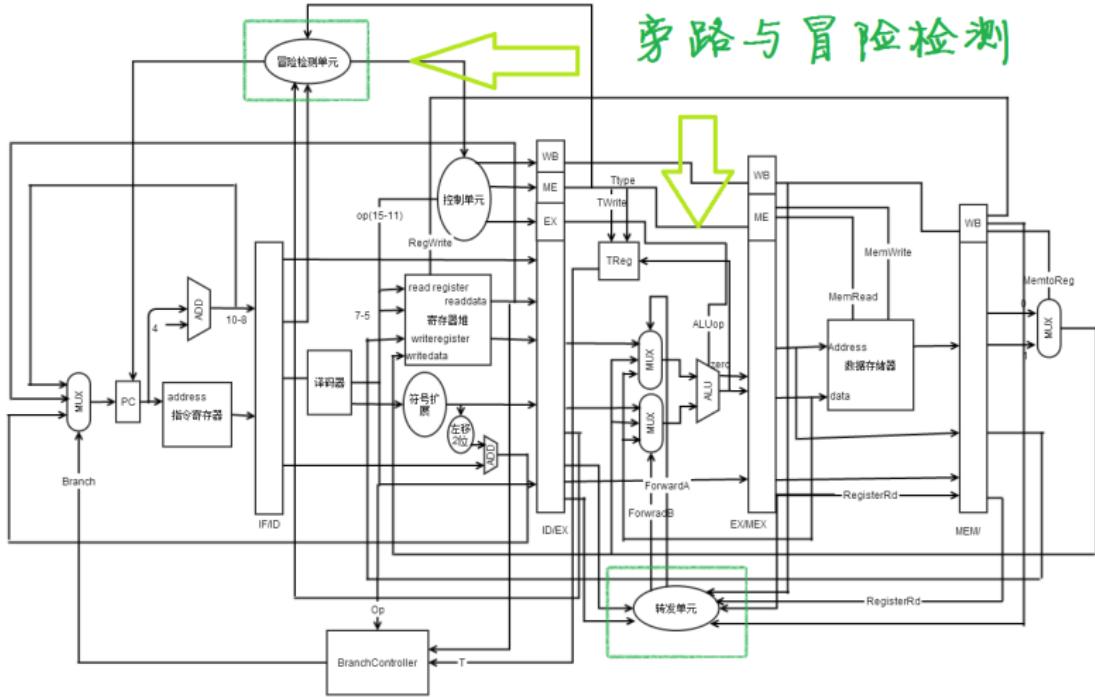


Figure: 旁路与冒险检测数据通路示意

2. 外围设备综合管理

我们在总线上设计了一个负责与所有外围设备的通信的模块 MemoryTop，向 CPU 封装所有的外围设备的逻辑，由此分担工作、并简化 CPU 内部设计。

其原理可描述为：CPU 从 MemoryTop 获取指令和数据，写入数据也需要通过 MemoryTop。我们的 MemoryTop 负责管理 RAM、FLASH、串口、PS2、VGA，以及相应的地址映射。通过“SW”、“LW”指令，并配合相应的地址，就能完成读写外围设备的功能。

MemoryTop 以管理的设备分为下列几个子模块：

- ▶ Flash 管理（实现 Flash 自启动）
- ▶ RAM 读写（负责取指、写存的关键部位）
- ▶ 键盘信息转发
- ▶ 串口管理
- ▶ 显存管理（VGA）

2.1 论如何科学的使用 VGA (其实很弱的 - -)

可接受一个 11 位地址、8 位数据、1 位写使能的输入。用来写 FIFO 显存。显存和字模骨架是通过 Xilinx 的 IP Core 实例化得到。

字模 (char_mem): 通过编写的辅助程序，从等宽字库中抽取 ASCII 可打印字符的字模。进行地址映射和补 0。最后生成一个 $10 * 15$ 点阵。零扩充之后得到一个整的 $16 * 16 * 95$ 内存空间。

显存 (fifo_mem): 整个 VGA 被设计为以一个字符显示 ($16 * 16$) 为单位，故相当于横向能放 40 个字符，竖向能放 30 个字符。也就是说显存只需要保存 $40 * 30$ 个字符 (7 位) 即可，将其零扩充得到 $64 * 32 * 8$ 位深度的显存。

零扩充：零扩充虽然浪费了一定的门空间，但是带来的好处是显然的，即 不需要乘法运算，只需要进行 *vector* 的连接运算就能计算显存与字模地址。

2.1 论如何科学的使用 VGA - 2

`vector_x` 是 0-639 的横坐标, `vector_y` 是 0-479 的纵坐标。
VGA 是水平逐行扫描的。于是只需取 `vector_x` 前 6 位和
`vector_y` 的前 5 位进行显存寻址 (获得 7 位表示字符编码的
`vector`), 就能表示某一个点正落在哪个字符的区域内。

于是接下来取他们的后 4 位, 再连接上 7 位表示字符的 `vector`,
进行字模寻址 (获得 1 位当前点是否为亮点的 `boolean`), 就能表
示当前字符下, 该点的显示内容。我们没有考虑颜色, 1 表示当
前点显示, 即为红色。

按照如上的原理就能完成整个屏幕的显示。只需要简单的连接指
令, 不需要额外的乘法除法运算。减少计算延迟和减轻硬件负担。

3. FLASH 自启动

由于是 MemoryTop 管理时钟。MemoryTop 一开始（或者 rst）会进入一个从 FLASH 拷贝代码到 RAM2 上的自动机状态。拷贝代码的长度是一定的（0x0FFF）。然后 MemoryTop 再在其四个状态机中循环的时候向 CPU 输出时钟。于是 CPU 就从内核代码开始执行，于是启动过程顺利完成。

4. 地址映射

主要存储：

- ▶ 系统程序区：0x0000-0x3FFF,16K
- ▶ 用户程序区：0x4000-0x7FFF,16K
- ▶ 系统数据区：0x8000-0xBEFF,16K
- ▶ 用户数据区：0xC000-0xF000,16K

外围设备：

- ▶ 串口
 - ▶ 串口 1 数据寄存器：0xBF00
 - ▶ 串口 1 状态寄存器：0xBF01
- ▶ 键盘
 - ▶ 串口 2 数据寄存器：0xBF02
 - ▶ 串口 2 状态寄存器：0xBF03
- ▶ VGA
 - ▶ FIFO 显存：0xF000-0xFFFF,3K

成果展示

D:\study\2013fall\ComputerOrganization\lastproject\makecomputer\assembler\fuz.exe

```
[8508] 0022  
[8509] 0037  
    >> D 8500 20  
[8500] 0001  
[8501] 0001  
[8502] 0002  
[8503] 0003  
[8504] 0005  
[8505] 0008  
[8506] 000d  
[8507] 0015  
[8508] 0022  
[8509] 0037  
[850a] 0059  
[850b] 0090  
[850c] 00e9  
[850d] 0179  
[850e] 0262  
[850f] 03db  
[8510] 063d  
[8511] 0a18  
[8512] 1055  
[8513] 1a6d  
[8514] 2ac2
```

Figure: 正确无误的运行斐波那契数列

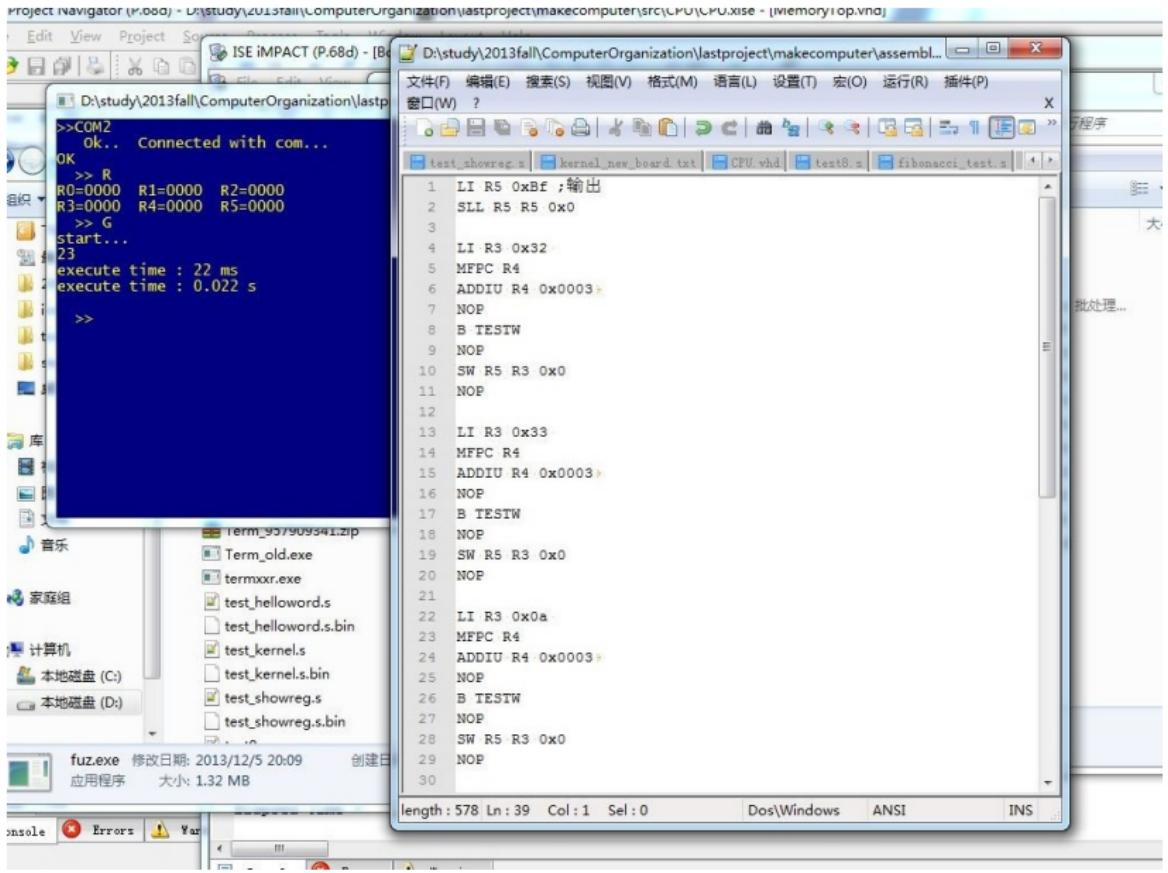


Figure: 与计算机串口通信，写串口“23\r\n”

T S I N G H U A U N I .
C S T 1 4

B Y T U K . Y O Y O . ; P

```
process (A L U o p , a , b )
begin
    case A L U o p is
        when "0 0 0" =>
            res <= a + b ;
        when "0 0 1" =>
            res <= a - b ;
        when "0 1 0" =>
            res <= a and b ;
        when "0 1 1" =>
            res <= a or b ;
        when "1 0 0" =>
            res <= not a ;
        when others =>
            res <= (others => '0')
    end case;
end process
```

Figure: 通过 PS2 键盘输入，然后将输入译码为 ASCII 码，将结果发送至串口，同时显示在 VGA 上。在 VGA 上的显示支持换行。读取键盘输入和 VGA 显示、串口发送，这些都是通过编写程序读取设备的映射地址获取设备数据而做到的。

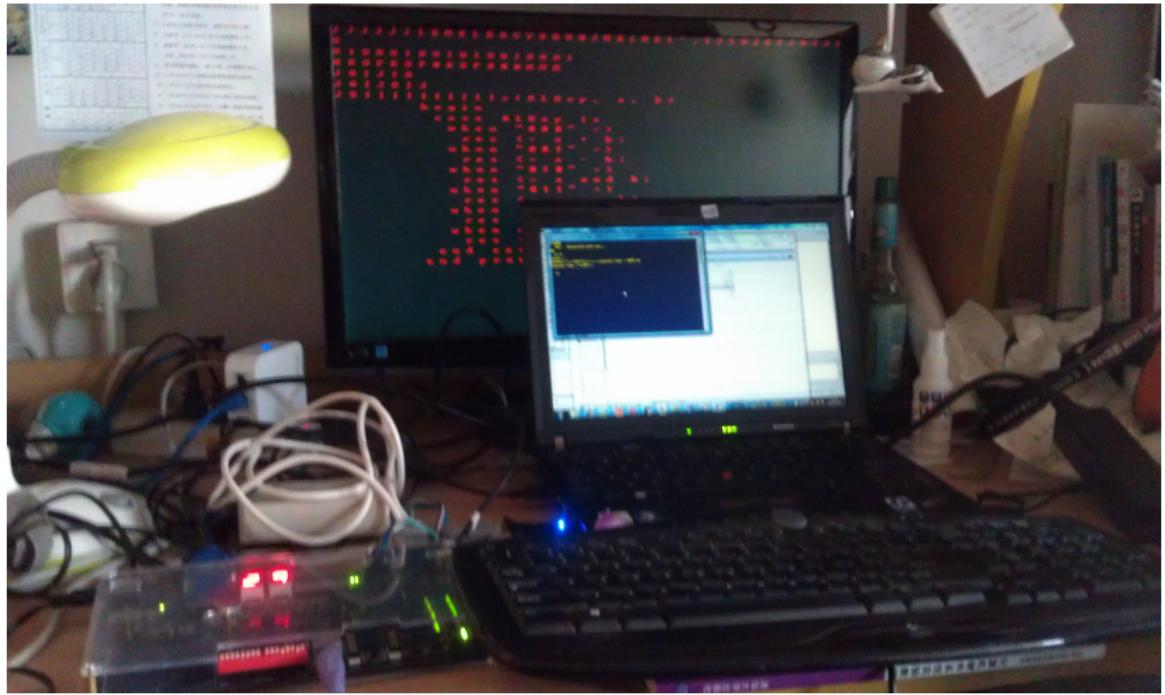


Figure: 整体测试图

分享一点点调试的经验

代码调试 - VGA

硬件的 `debug` 真是一件困难的事，它不能像软件一样输出一堆调试信息。我们只能利用有限的 LED 灯来 `debug`，于是我们在 `RegFile` 中实现了一个 `debug` 模块，他能实时的通过改变开关的输入来改变显示的寄存器。后来我们写好了 `vga`，就能输出信息在 `vga` 上来调试了，效率更高。

代码调试 - 传说中的单步跟踪

整个实现过程中遇到的最大的 bug 就是在我们刚开始运行 kernel 代码的时候，我们通过 Term+ 串口 +R 命令试图读寄存器的值，总是在一个寄存器里出现了随机的结果，但是我们用 led 看到寄存器里的值的确是正确的。一开始我们认为是我们串口写得不对，后来用串口精灵仔细检查了一下，发现我们串口应该没有错，而只是多输出了一个值，我们认为那应该是执行 kernel 代码时的出现的错误，于是我们把 kernel 中每一个功能部分（于是第一次作业阅读 kernel 源代码就用上了）单独取出来一个阶段一个阶段的进行调试。最后我们把读寄存器的部分单独拿出来，然后通过手按 clk 单步执行代码来 debug。这里涉及到手按与自动时钟的切换，我们写了一个模块来通过开关来快速切换两种时钟。最后单步执行观察寄存器的值发现时 BTEQZ 的一个冲突没解决好，导致多循环了一次。于是我们就恍然大悟原来是多循环了一次所以向串口多输出了一个无效值。于是我们马上在 riskchecker 中多加入一条旁路检查，然后程序顺利正常运行。

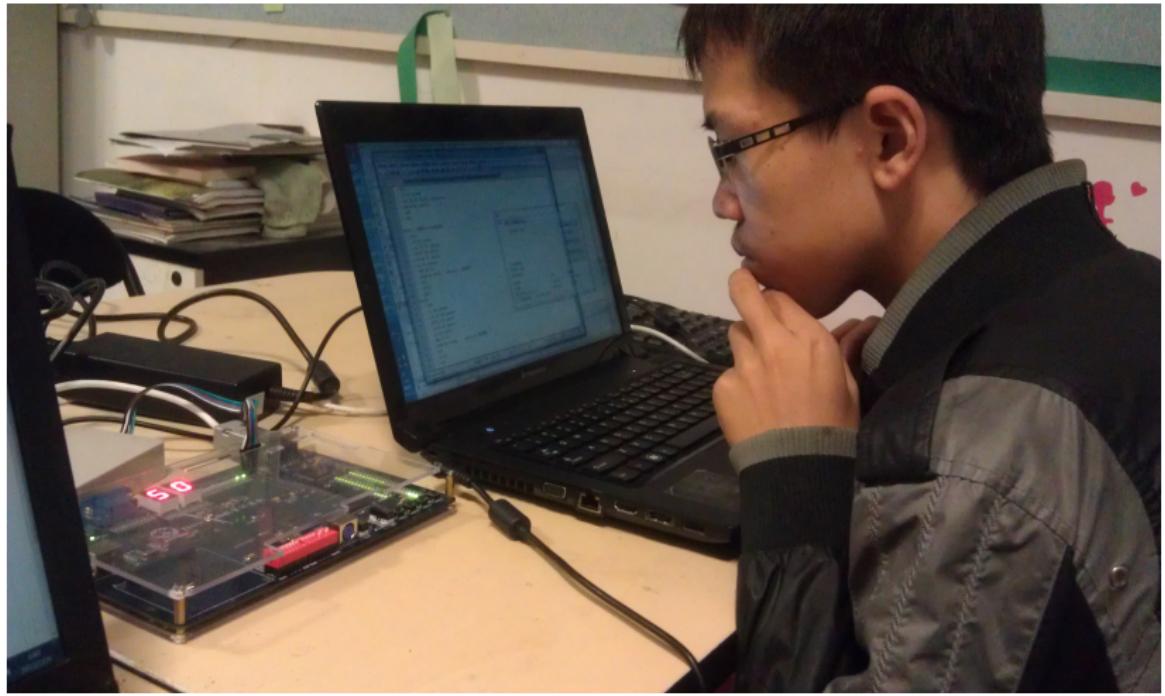


Figure: 做的过程实在是太苦逼了！但是做出来又是非常快乐的 =w=

The End. Thank you All :P!