

# 计算机组成原理

# THINPAD 大实验

## 实验报告

涂珂 2011011273 计 14      傅左右 2011011264 计 14

December 12, 2013

## Contents

<b>1 实验目标</b>	<b>4</b>
<b>2 指令集任务</b>	<b>4</b>
2.1 THCO MIPS 基本指令集 . . . . .	4
2.2 扩展指令集 (402) . . . . .	4
<b>3 实验成果参数</b>	<b>5</b>
<b>4 实验成果简列</b>	<b>5</b>
<b>5 整体设计图</b>	<b>5</b>
<b>6 重新设计的指令集</b>	<b>8</b>
6.1 指令设计 . . . . .	8
6.2 R 型指令 . . . . .	8
6.3 I 型指令 . . . . .	8
6.4 B 型指令 . . . . .	9
6.5 J 型指令 . . . . .	9
6.6 NOP 指令 . . . . .	9
6.7 指令流水细节 . . . . .	10

<b>7 统一的信号及编码</b>	<b>10</b>
7.1 控制信号 . . . . .	10
7.2 寄存器编址 . . . . .	11
7.3 字符编码 . . . . .	11
7.4 指令集与控制信号关系表 . . . . .	11
<b>8 主要模块设计</b>	<b>13</b>
8.1 硬件 . . . . .	13
8.1.1 ALU . . . . .	13
8.1.2 BranchSelector . . . . .	13
8.1.3 Controller . . . . .	13
8.1.4 Decoder . . . . .	13
8.1.5 MemoryTop . . . . .	14
8.1.6 Passer . . . . .	14
8.1.7 RegFile . . . . .	14
8.1.8 RiskChecker . . . . .	14
8.1.9 TReg . . . . .	14
8.1.10 KeyTop . . . . .	14
8.1.11 VGA_top . . . . .	14
<b>9 主要模块实现</b>	<b>15</b>
9.1 硬件 . . . . .	15
9.1.1 ALU . . . . .	15
9.1.2 BranchSelector . . . . .	15
9.1.3 Decoder . . . . .	16
9.1.4 MemoryTop . . . . .	16
9.1.5 Passer . . . . .	19
9.1.6 RegFile . . . . .	19
9.1.7 RiskChecker . . . . .	19
9.1.8 VGA_top . . . . .	20
9.1.9 KeyTop . . . . .	21
9.2 硬件 · 补 . . . . .	21

9.2.1	FLASH 自启动 . . . . .	21
9.2.2	地址映射 . . . . .	21
9.3	软件 . . . . .	22
9.3.1	Assembler (thcoas.py) . . . . .	22
9.3.2	简单记事本 (test_keyboard.s) . . . . .	22
<b>10</b>	<b>实验成果展示</b>	<b>24</b>
10.1	Xilinx 编译报告 . . . . .	24
10.2	RTL 图 . . . . .	24
10.3	操作流程 . . . . .	24
10.3.1	串口读写 . . . . .	26
10.3.2	记事本 (外设) . . . . .	26
<b>11</b>	<b>实验心得和体会</b>	<b>29</b>
11.1	自定义指令集 . . . . .	29
11.2	数据通路的设计 . . . . .	29
11.3	代码同步与管理 . . . . .	29
11.4	代码调试 . . . . .	29
11.5	小记 . . . . .	31

# 1 实验目标

基于 THINPAD 教学计算机，设计：

- 基于 MIPS16 指令集的流水线 CPU
- 使用基本存储、扩展存储、Flash、IO 设备
- 能够运行 kernel、监控程序、project1 程序

## 2 指令集任务

### 2.1 THCO MIPS 基本指令集

序号	指令	序号	指令
1	ADDIU	14	LW_SP
2	ADDIU3	15	MFIH
3	ADDSP	16	MFPC
4	ADDU	17	MTIH
5	AND	18	MTSP
6	B	19	NOP
7	BEQZ	20	OR
8	BNEZ	21	SLL
9	BTEQZ	22	SRA
10	CMP	23	SUBU
11	JR	24	SW
12	LI	25	SW_SP
13	LW		

### 2.2 扩展指令集 (402)

- JRRA
- SLTI
- ADDSP3

- NOT
- SLT

### 3 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
- TODO

### 4 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

### 5 整体设计图

数据通路可详见 datapath.png 文件。

RTL 综合图可见下，或者 cpu-schematic.pdf。可以无限放大该页面。

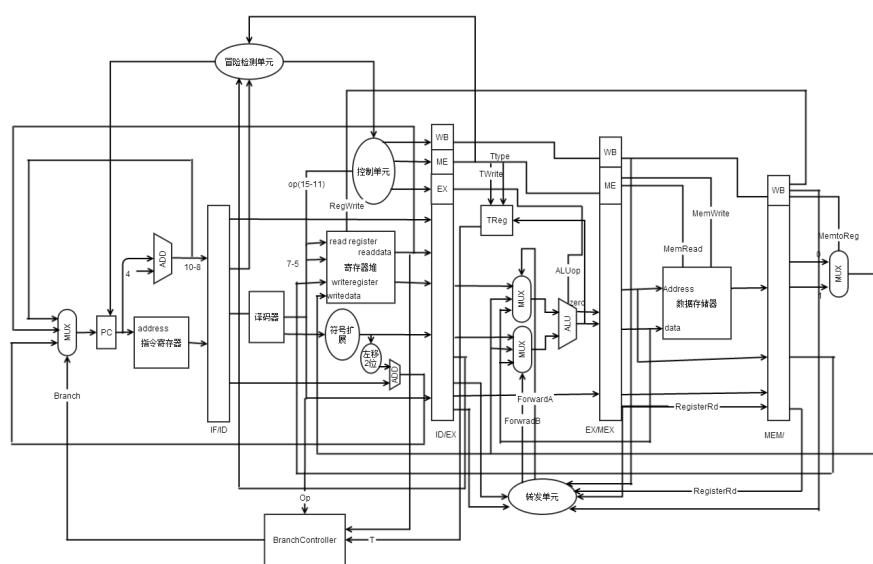


Figure 1: 数据通路图 datapath.png

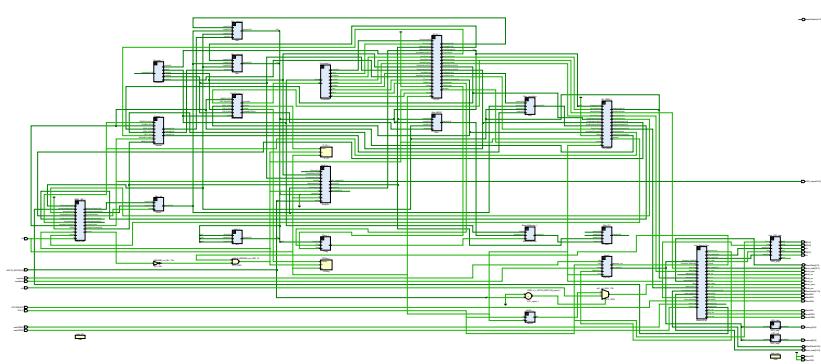


Figure 2: RTL 综合图 (可放大)

## 6 重新设计的指令集

### 6.1 指令设计

- 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。

### 6.2 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

### 6.3 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	10101XXXYYYIIIII
SW	10110XXXYYYIIIII
SW_SP	10111XXXYYYIIIII

---

#### 6.4 B 型指令

B	指令结构
B	11000IIIIIIIIII
BTEQZ	11001IIIIIIII000
BEQZ	11010XXXIIIIIIII
BNEZ	11011XXXIIIIIIII

---

#### 6.5 J 型指令

J	指令结构
*JRRA	1110000000000000
JR	11101XXX00000000

---

#### 6.6 NOP 指令

NOP	0000000000000000
-----	------------------

---

## 6.7 指令流水细节

关于每一条指令在流水的五个步骤中具体做了什么。表格无法正常显示。请见相关设计文档instruction.xlsx。

## 7 统一的信号及编码

### 7.1 控制信号

每一级流水阶段的寄存器均会储存相应信号。(显然的，当前指令与当前流水信号相对应)

Table 7: 控制信号表

控制信号	发生阶段	置 0 时	置 1 时	置 10 时
PCWrite	IF	null	写 PC	
Branch(2 位)	ID	PC+4	PC+4+immediate	Reg1
ForwardA(2 位)	EX	来自寄存器堆的输出 Reg1	转发写回的值	转发上一次 ALU 运算结果
ForwardB(2 位)	EX	来自寄存器堆的输出 Reg2	转发写回的值	转发上一次 ALU 运算结果
ALUsrc	EX	来自寄存器堆的输出	来自符号扩展的立即数	
ALUop(3 位)	EX	加	001: 减, 010: 与, 011: 或, 100: 非, 101: 左移, 110: 右移	
Ttype	EX	小于 T 为 1	不等于 T 为 1	
Twrite	EX	null	写入 T	
MemRead	MEM	null	读内存	
MemWrite	MEM	null	写内存	
MemtoReg	WB	写入 ALU 输出值	写入内存输入值	
RegWrite	WB	null	写入寄存器堆	

## 7.2 寄存器编址

我们将特殊寄存器也看作普通的寄存器。将寄存器编码长度（3位）扩展一位，与特殊寄存器统一编址（4位）。这样在 CPU 内部处理的时候就能简化流程。

我们设置了 SP、PC、RA、IH、Zero 四个特殊寄存器。8 个普通寄存器。特殊寄存器高位均为 1。

Table 8: 寄存器编址表

编码 4 位	寄存器
0+XXX	8 个普通寄存器
1000	Zero
1010	PC
1011	IH
1100	RA
1101	SP

## 7.3 字符编码

我们将 95 个 ASCII 可打印字符的等宽字符写进了 VGA 控制模块的存储里。我们以空格（0x20）为偏移量，将每一个字符对应的 ASCII 码减去 0x20 作为内部字符编码。这样就有两个好处：一是减少地址量，二是可以用连接两个 `std_logic_vector` 的方法 (`std_logic_vector & std_logic_vector`) 计算字符地址。

## 7.4 指令集与控制信号关系表

Table 9: 指令与控制信号关系

指令	Branch	ALU-op	ALU-src	T-type	T-write	Mem-Read	Mem-Write	Mem-to-Reg	Reg-Write
NOP	00	000	0	0	0	0	0	0	0
MFIH	00	000	0	0	0	0	0	0	1
MFPC	00	000	0	0	0	0	0	0	1
MTIH	00	000	0	0	0	0	0	0	1
MTSP	00	000	0	0	0	0	0	0	1
AND	00	010	0	0	0	0	0	0	1
OR	00	011	0	0	0	0	0	0	1
NOT*	00	100	0	0	0	0	0	0	1
SLT*	00	001	0	0	1	0	0	0	0
CMP	00	001	0	1	1	0	0	0	0
SLL	00	101	1	0	0	0	0	0	1
SRA	00	110	1	0	0	0	0	0	1
ADDU	00	000	0	0	0	0	0	0	1
SUBU	00	001	0	0	0	0	0	0	1
ADDSP	00	000	1	0	0	0	0	0	1
LW_SP	00	000	1	0	0	1	0	1	1
SW_SP	00	000	1	0	0	0	1	0	0
ADDIU	00	000	1	0	0	0	0	0	1
SLTI*	00	001	1	0	1	0	0	0	0
ADDSP3*	00	000	1	0	0	0	0	0	1
LI	00	000	1	0	0	0	0	0	1
ADDIU3	00	000	1	0	0	0	0	0	1
LW	00	000	1	0	0	1	0	1	1
SW	00	000	1	0	0	0	1	0	0
B	01	000	0	0	0	0	0	0	0
BTEQZ	01	000	0	0	0	0	0	0	0
BEQZ	01	000	0	0	0	0	0	0	0
BNEZ	01	000	0	0	0	0	0	0	0
JRRA*	10	000	0	0	0	0	0	0	0
JR	10	000	0	0	0	0	0	0	0

## 8 主要模块设计

这一部分将简明扼要的介绍主要或关键模块的设计考量。更详细的设计细节可见下一个章节“[主要模块实现](#)”。

### 8.1 硬件

#### 8.1.1 ALU

运算逻辑单元，实现了以下 7 种基本运算（加法、减法、逻辑与、逻辑或、逻辑非、逻辑左移、逻辑右移）：

ALUop	运算
000	$a+b$
001	$a-b$
010	$a \& b$
011	$a$
100	$\sim a$
101	$a \ll b$
110	$a \gg b$

#### 8.1.2 BranchSelector

跳转选择器，根据不同的跳转指令及跳转条件决定 PC 是否跳转。

#### 8.1.3 Controller

控制器，输入  $Op_p$ ，输出  $Op_p$  对应的控制信号（对应表见 [signal.xlsx](#)）。由于自定义了指令集使得不同的指令对应的  $Op_p$  不同且类型相近的指令相邻，让这一部分得以简化。

#### 8.1.4 Decoder

译码器，将 16 位指令进行译码，向外输出 5 位指令编号  $Op_p$ ，3 个寄存器地址  $reg1$ 、 $reg2$ 、 $reg3$  和 16 位立即数  $imm$ 。

在译码器中根据  $Op_p$  进行了立即数的扩展。3 个寄存器地址中  $reg1$ 、 $reg2$  为下一步将要读取的寄存器、 $reg3$  为将要写入的寄存器。

### **8.1.5 MemoryTop**

MemoryTop 是总线上负责所有外围设备的通信的一个模块，相当于实际计算机中连接南北桥的芯片，目的是向 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。

其职能可描述为：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。

### **8.1.6 Passer**

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。

### **8.1.7 RegFile**

寄存器堆，负责所有寄存器，寄存器编址 4 位。具体可参见[寄存器编址分配表](#)。所有寄存器均是下降沿写入。同时具有相应的调试接口。

### **8.1.8 RiskChecker**

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。

### **8.1.9 TReg**

T 寄存器，T 寄存器的值需要额外的判断写入，故将其从寄存器堆中分离出来。置于 EXE 阶段，读取 ALU 的输出。

所有指令中涉及到的 T 的写入分为 2 种，不等于和大于，所以我们通过 ALU 做减法，不等于即为 ALU 的输出不为 0，大于即为 ALU 输出符号位为 0。于是我们就可以直接根据控制信号 Ttype、Twrite 及 ALU 的输出直接计算出 T 寄存器的值输出。简化了内部逻辑。

### **8.1.10 KeyTop**

键盘模块。附属两个子模块 **Keyboard.vhd**（完成接收键盘发送的信号和分析，进行滤波和去除毛刺，输出得到的扫描码），**KeySignaltoChar.vhd**（对键盘扫描码进行译码，翻译成内部的 ASCII 字符方言。）

### **8.1.11 VGA\_top**

VGA 管理模块。具有一个显存和一个字模存储。外部可以通过传入指定地址和内部字符约定编码，就能更改显示内容。同时在开发阶段中被用来进行调试。

## 9 主要模块实现

这一部分将详细的介绍各个主要模块的实现细节。简明的设计思路可往回参看上一个章节“[主要模块设计](#)”。

### 9.1 硬件

#### 9.1.1 ALU

完全组合逻辑。接受外部传入的 ALUop 信号，通过 case...when...语句进行相应运算并输出计算结果。

#### 9.1.2 BranchSelector

对不同 Op 表示的不同的跳转指令所需的条件进行判断，输出数据选择器 Mux\_PC 的控制信号 Branch，从而完成对跳转分支的选择功能。

Listing 1: 分支选择器 BranchSelector

```
...
case Op is
    when "11000" => -- B
        Branch <= "01";
    when "11001" => -- BTEQZ
        if T = '0' then
            Branch <= "01";
        end if;
    when "11010" => -- BEQZ
        if RegInput = Int16_zero then
            Branch <= "01";
        end if;
    when "11011" => --BNEZ
        if RegInput /= Int16_zero then
            Branch <= "01";
        end if;
    when "11101" => --JR
        Branch <= "10";
    when "11100" => --JRRA*
        Branch <= "10";
    when others => null;
end case;
...
```

### 9.1.3 Decoder

1. 通过 Case 语句判断高五位的值得到指令类型 (5 位指令编号 Op)
2. 根据指令类型 Op 的不同决定在后续流程中将要读写的寄存器和立即数扩展的方式 (每一条指令的执行细节可参见指令文档instruction.xlsx)
3. 输出下一步即将要读写的寄存器编址 reg1 (读)、reg2 (读)、reg3 (写) (没有使用寄存器则赋为零寄存器地址), 以及扩展后的 16 位立即数 imm。

### 9.1.4 MemoryTop

MemoryTop 是总线上负责所有外围设备的通信的一个关键模块, 相当于实际计算机中连接南北桥的芯片, 目的是向 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。

其职能可描述为: 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。

最后我们的实现是 MemoryTop 运行在 CPU 主频四倍的时钟频率上, 为了保证 CPU 与外围设备时钟的一致, 我们将总的时钟管理也交给 MemoryTop, 由 MemoryTop 四分频后的时钟输出给 CPU 使用。

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)

MemoryTop 以实现的功能分为以下几个时序流程:

a. BOOT

初始状态机, rst。转入 BOOT\_FLASH。

b. BOOT\_FLASH

调整 FLASH 信号, 开始从 FLASH 中读取数据, 由于 FLASH 延迟比较大, 所以设置一个八位计时器。完成从 FLASH 读取指令之后, 将取指令的地址 +1, 存入 RAM2 的地址亦 +1。转入 BOOT\_RAM2

c. BOOT\_RAM2

将从 FLASH 中读取出的指令写入 RAM2。转入 BOOT\_READY。

d. BOOT\_READY

检查读出指令的数量是否达到 0xffff 条。如果是，则跳出启动循环，转入 READ1，进入正常的 CPU 工作循环。否则的话，转入 BOOT\_FLASH 状态，继续循环。

e. READ1

在 READ1 与 IDEL1 两个状态中，向 CPU 输出的时钟信号为高位。在 RW1 与 IDEL2 中，向 CPU 输出的时钟信号为低位。这样就能四分频后输出给 CPU 一个稳定的时钟。

READ1 是 CPU 正常工作时的第一个工作流程。在这个状态机中，RAM 的地址 address1 是下一条待取指令的地址，保持 RAM\_OE 为低位，开始读出指令。转入 IDEL1。

f. IDEL1

IDEL1 是第二个工作流程。在这个状态中，已经读出下一条指令 (output1)。如果 CPU 要求串口通信的话（即访问地址为 0xBF00），根据 Decoder 传来的指令信号 (MemoryRead or MemoryWrite)，就拉低串口读写使能。转入 RW2。

g. RW1

RW1 是第三个工作流程，在这个状态中，程序将内部已经准备好的 CPU 取存的数据准备好。并通过 output2 转发出去。转入 IDEL2。

h. IDEL2

IDEL2 是最后一个工作流程，是为了保证外围设备正常工作的一个缓冲周期。转入 READ1。

address1, address2 均为 18 位拓展后的数据地址（为了适应总线宽度）。

固定的控制信号：

- VGA 显存地址  $\leq$  address2
- VGA 数据  $\leq$  dataInput(7 downto 0)
- RAM1 使能  $\leq$  1, RAM2 使能  $\leq$  0
- FLASH 写使能、擦除使能  $\leq$  1
- FLASH 数据  $\leq$  ``Z''

固定的状态信号：

a. 串口状态信号 (访问地址 0xBF01)

根据约定, dataready (可读) 放在倒数第二位, 串口是否阻塞 (可写) 放在最后一位

$BF01 \leq 0 \& serial\_dataready \& (serial\_tsre \text{ and } serial\_trbe)$

b. 键盘状态信号 (访问地址 0xBF03)

根据约定, keyboard\_dataready 在最后一位, 表示是否可读入数据。

$BF03(0) \leq Keyboard\_Dataready$

Table 11: 外围设备的控制信号一览

信号 / 状态	BOOT-FLASH	BOOT-RAM1	BOOT-RAM2	BOOT-READY	READ1	IDEL1	RW1	IDEL2
基本总线	"Z"	"Z"	"Z"	"Z"	"Z"	Not MR (0xBF00) or "Z"	Not MR (0xBF00) or "Z"	Not MR (0xBF00) or "Z"
扩展总线	"Z"	flash-addr-input	flash-addr-input	"Z"	"Z"	data-input	data-input	"Z"
RAM2 地址	FLASH-PC	FLASH-PC	FLASH-PC	ZERO	address1	address1	address2	address2
RAM2-OE	1	1	1	0	0	0	Not MR	0
RAM2-RW	1	1	0	1	1	1	1 (0xBF00) or NOT MW	1
serial-rdn	1	1	1	1	1	Not MR (0xBF00) or 1	Not MR (0xBF00) or 1	Not MR (0xBF00) or 1
serial-wrn	1	1	1	1	1	1	Not MW (0xBF00) or 1	1
FLASH 读使能	0	1	1	1	1	1	1	1
键盘读使能	0	0	0	0	0	0	MR (0xBF02)	0
VGA 写使能	0	0	0	0	0	0	MR (0xF*)	0
CPU-CLOCK	0	0	0	0	1	1	0	0

### 9.1.5 Passer

根据流水寄存器的值和译码阶段的寄存器的值判断是否有数据冲突，输出控制信号。

Listing 2: 数据冲突检测单元

```
if (EXMEM_RegWrite = '1' and EXMEM_W /= Zero_Reg and EXMEM_W =
    INDEX_R1) then
    ForwardA <= "10";
end if;
if (EXMEM_RegWrite = '1' and EXMEM_W /= Zero_Reg and EXMEM_W =
    INDEX_R2) then
    if INDEX_alusrc = '0' then
        ForwardB <= "10";
    else
        ForwardC <= "10";
    end if;
end if;
if (MEMWB_RegWrite = '1' and MEMWB_W /= Zero_Reg and EXMEM_W =
    INDEX_R1 and MEMWB_W = INDEX_R1) then
    ForwardA <= "01";
end if;
if (MEMWB_RegWrite = '1' and MEMWB_W /= Zero_Reg and EXMEM_W =
    INDEX_R2 and MEMWB_W = INDEX_R2) then
    if INDEX_alusrc = '0' then
        ForwardB <= "01";
    else
        ForwardC <= "01";
    end if;
end if;
```

### 9.1.6 RegFile

寄存器堆，负责所有寄存器，寄存器编址 4 位。具体可参见[寄存器编址分配表](#)。所有寄存器均是下降沿写入。

同时具有相应的调试接口。在调试时能够通过拨码即时的向 LED 或者 VGA 输出。极大的方便了调试。

### 9.1.7 RiskChecker

同 Passer 写法相同。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。

### 9.1.8 VGA\_top

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。

可见下面的代码，*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。减轻硬件负担。

Listing 3: 零扩充简化了寻址

```
signal vector_x : std_logic_vector(9 downto 0);      --X 10b 640
signal vector_y : std_logic_vector(8 downto 0);      --Y 9b 480

...
5   -- store char
ram: char_mem port map(clka => clk, addra => char_addr, douta
    => pr);
-- display cache
cache: fifo_mem port map(
    -- a for write -- wea: 1 is write signal
10   clka => clk, wea => wctrl, addra => waddr, dina => wdata,
    -- b for read
    clkb => clk, addrb => caddr, doutb => char
);

15   -- cache addr 5 + 6 = 11
caddr <= vector_y(8 downto 4) & vector_x(9 downto 4);

-- char access addr 7 + 4 + 4 = 15
-- last 2 control the display-point(x, y)
20   -- first char control which char
char_addr <= char(6 downto 0) & vector_y(3 downto 0) &
    vector_x(3 downto 0);
```

该模块在调试时被编写成可以用来即时的十六进制显示各种输入数据，以VGA\_play.vhd形式存在于工程中。在之后的阶段被移除了接口。

### 9.1.9 KeyTop

对 50MHz 进行十分频，使用 5MHz 的时钟信号用于滤波。通过检测F0信号来判断一次按键。再将键盘的扫描码翻译成内部通信的 ASCII 方言（减去 0x20）。其中键盘的 dataready 会在上升沿赋为 0，表示数据已经失效。

## 9.2 硬件・补

### 9.2.1 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（0xFFFF）。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。

### 9.2.2 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

## 9.3 软件

### 9.3.1 Assembler (thcoas.py)

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。

具体使用方法：`python binToHex.py a.s`。汇编器会将 MIPS 汇编代码 `a.s` 编译输出 `a.s.bin` 的二进制文件，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。

### 9.3.2 简单记事本 (test\_keyboard.s)

程序读取 PS2 键盘的输入，然后将结果发送至串口，同时显示在 VGA 上。在 VGA 上的显示支持换行。

```

D:\study\2013fall\ComputerOrganization\lastproject\makecomputer\assembler>python
thcoas.py test_helloworld.s
L:1 1010001011011111 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 0001000000000000 0x1400 C:3 ['MFPC', 'R4']
L:6 1000110000000011 0x8c03 C:4 ['ADDIU', 'R4', '0x0003']
L:7 0000000000000000 0x0 C:5 ['NOP']
L:8 1100000000011101 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']
L:17 1100000000010101 0xc015 C:14 ['B', 'TESTW']
L:18 0000000000000000 0x0 C:15 ['NOP']
L:19 1011110101100000 0xbd60 C:16 ['SW', 'R5', 'R3', '0x0']
L:20 0000000000000000 0x0 C:17 ['NOP']
L:22 1010001100001010 0xa30a C:18 ['LI', 'R3', '0x0a']
L:23 0001010000000000 0x1400 C:19 ['MFPC', 'R4']
L:24 1000110000000011 0x8c03 C:20 ['ADDIU', 'R4', '0x0003']
L:25 0000000000000000 0x0 C:21 ['NOP']
L:26 110000000001101 0xc00d C:22 ['B', 'TESTW']
L:27 0000000000000000 0x0 C:23 ['NOP']
L:28 1011110101100000 0xbd60 C:24 ['SW', 'R5', 'R3', '0x0']
L:29 0000000000000000 0x0 C:25 ['NOP']
L:31 1010001100001101 0xa30d C:26 ['LI', 'R3', '0x0d']
L:32 0001010000000000 0x1400 C:27 ['MFPC', 'R4']
L:33 1000110000000011 0x8c03 C:28 ['ADDIU', 'R4', '0x0003']
L:34 0000000000000000 0x0 C:29 ['NOP']
L:35 110000000000101 0xc005 C:30 ['B', 'TESTW']
L:36 0000000000000000 0x0 C:31 ['NOP']
L:37 1011110101100000 0xbd60 C:32 ['SW', 'R5', 'R3', '0x0']
L:38 0000000000000000 0x0 C:33 ['NOP']
L:41 1101111000000000 0xef00 C:34 ['JR', 'R7']
L:42 0000000000000000 0x0 C:35 ['NOP']
L:46 0000000000000000 0x0 C:36 ['NOP']
L:47 1010011010111111 0xa6bf C:37 ['LI', 'R6', '0x00BF']
L:48 0101011011000000 0x56c0 C:38 ['SLL', 'R6', 'R6', '0x0000']
L:49 1000111000000001 0x8e01 C:39 ['ADDIU', 'R6', '0x0001']
L:50 1010110000000000 0xb600 C:40 ['LW', 'R6', 'R0', '0x0000']
L:51 1010011000000001 0xa601 C:41 ['LI', 'R6', '0x0001']

```

Figure 3: 汇编器截图（代码原文行号、二进制、十六进制代码、翻译后的代码地址号、相应指令）

```

J J J J J J L R M K I B H U V H B U N J M K I H K I > , I J I O J
0
H I U H U I H U I H I U H I U H U I
N K O P K O P N K O P N N N M M M
J O I J I O
J O I J O I J
J O I I I I I I I I I s ( A L U o p , a , b )
begin
    casee A L U o p i s
    wheen " 0 0 0 " =>
        res <= a + b ;
    wheen " 0 0 1 " =>
        res <= a - b ;
    wheen " 0 1 0 " =>
        res <= a and b ;
    wheen " 0 1 1 " =>
        res <= a or b ;
    wheen " 1 0 0 " =>
        res <= not a ;
    wheen others =>
        res <= (others => ' 0 ' ) ;
    end case;
end process

```

Figure 4: 简单记事本

## 10 实验成果展示

CPU 可运行所有指令，能正常运行 Term。

### 10.1 Xilinx 编译报告

顺利完成所有综合、布线、生成二进制 bin 文件。

### 10.2 RTL 图

(可参见 cpu-schematic.pdf)

### 10.3 操作流程

0. kernel 的二进制代码已经预先写入 Flash。
1. 正确连线，将开关第 5 位拨到 1 (将手动单步 clk 时钟转换到 50M 时钟)
2. 打开 FlashAndRam.exe，将用户程序代码写入到 RAM 的 0x4000 处
3. 打开 Term
4. 在 Xilinx 中将 CPU 代码烧写进 FPGA 中，按 rst 复位系统

CPU Project Status (12/06/2013 - 02:18:42)			
Project File:	CPU.xise	Parser Errors:	No Errors
Module Name:	CPU	Implementation State:	Programming File Generated
Target Device:	xc3s1200e-4fg320	* Errors:	
Product Version:	ISE 14.6	* Warnings:	
Design Goal:	Balanced	* Routing Results:	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	* Timing Constraints:	All Constraints Met
Environment:	System Settings	* Final Timing Score:	0 ( <a href="#">Timing Report</a> )

Device Utilization Summary				[+]
Logic Utilization	Used	Available	Utilization	Note(s)
Total Number Slice Registers	636	17,344	3%	
Number used as Flip Flops	627			
Number used as Latches	9			
Number of 4 input LUTs	1,370	17,344	7%	
Number of occupied Slices	938	8,672	10%	
Number of Slices containing only related logic	938	938	100%	
Number of Slices containing unrelated logic	0	938	0%	
Total Number of 4 input LUTs	1,440	17,344	8%	
Number used as logic	1,370			
Number used as a route-thru	70			
Number of bonded IOBs	140	250	59%	
Number of RAMB16s	3	28	10%	
Number of BUF64MUXs	4	24	16%	
Average Fanout of Non-Clock Nets	3.75			

Figure 5: Xilinx 编译报告

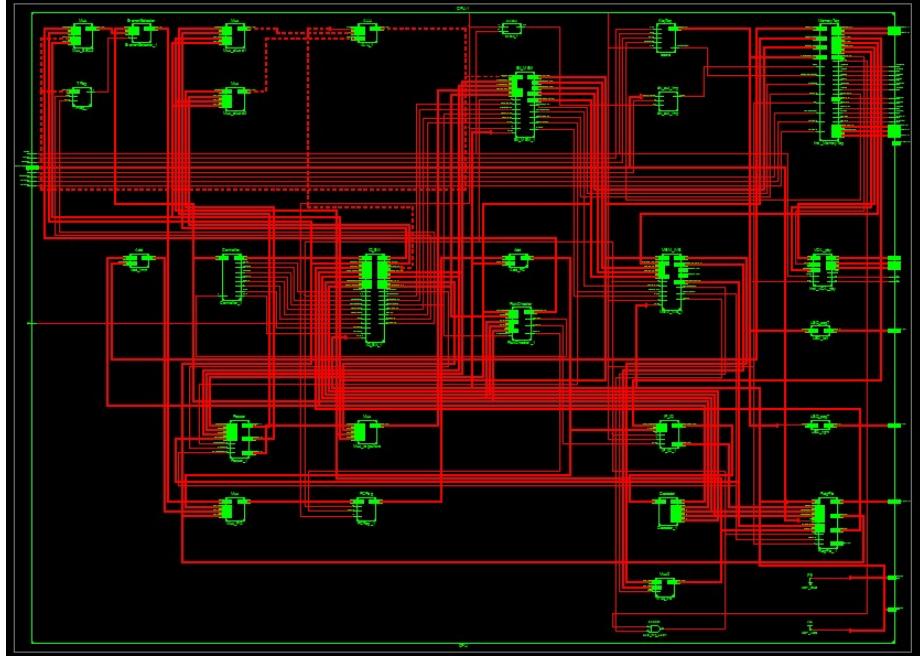


Figure 6: RTL 图

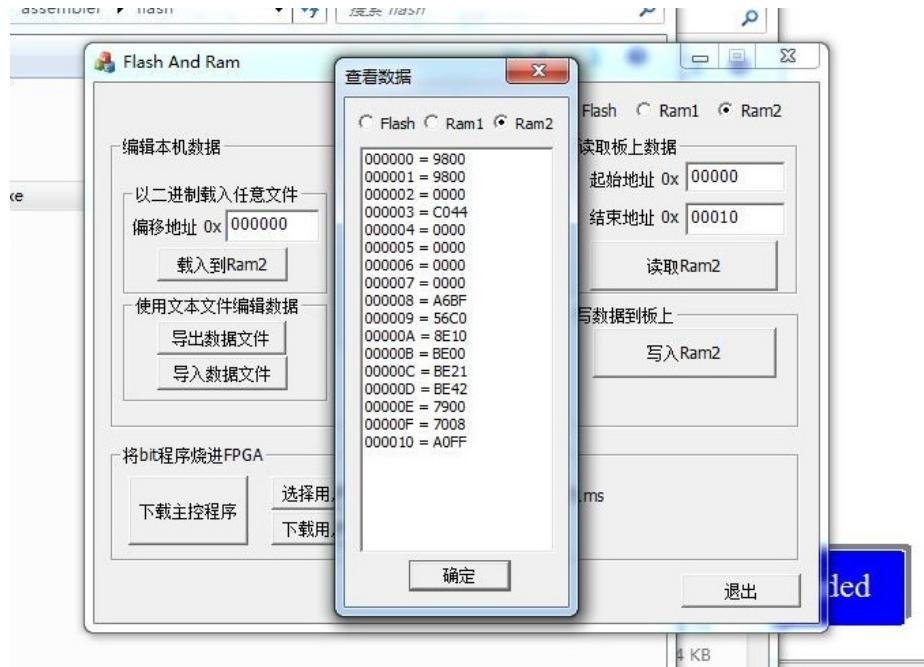


Figure 7: 使用工具程序，向 RAM2 写入用户程序

5. 系统将自动完成启动步骤，从 Flash 载入 kernel 代码并运行。
6. 在 Term 中连接串口，按 G 运行 0x4000 的用户程序，查看结果，R 查看寄存器。

### 10.3.1 串口读写

程序读写串口。输出 ``23\r\n''。

### 10.3.2 记事本（外设）

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，就像在真正的电脑上一样。

```

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 8: 斐波那契数列运行结果

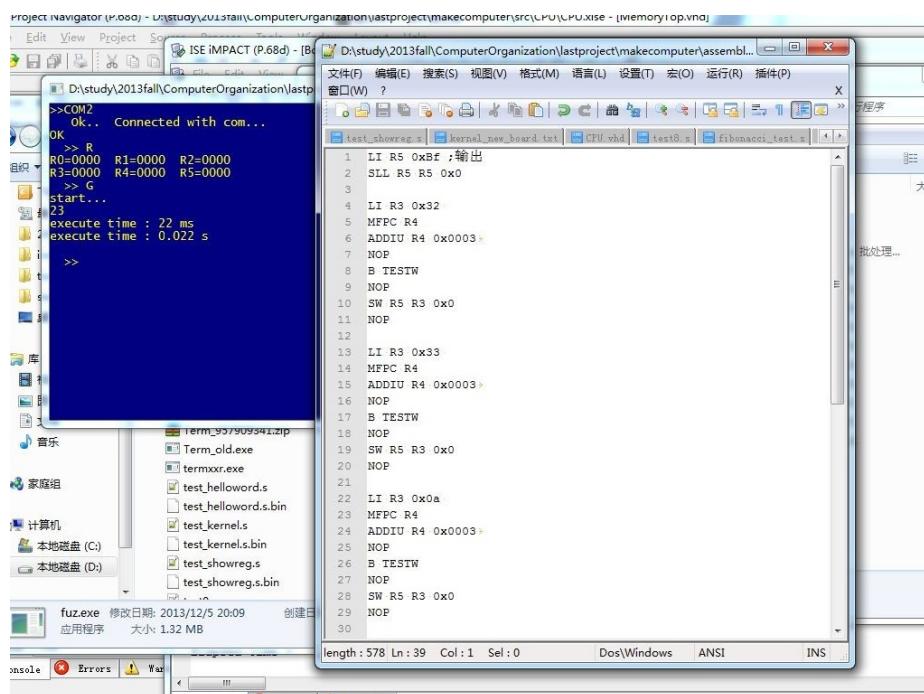
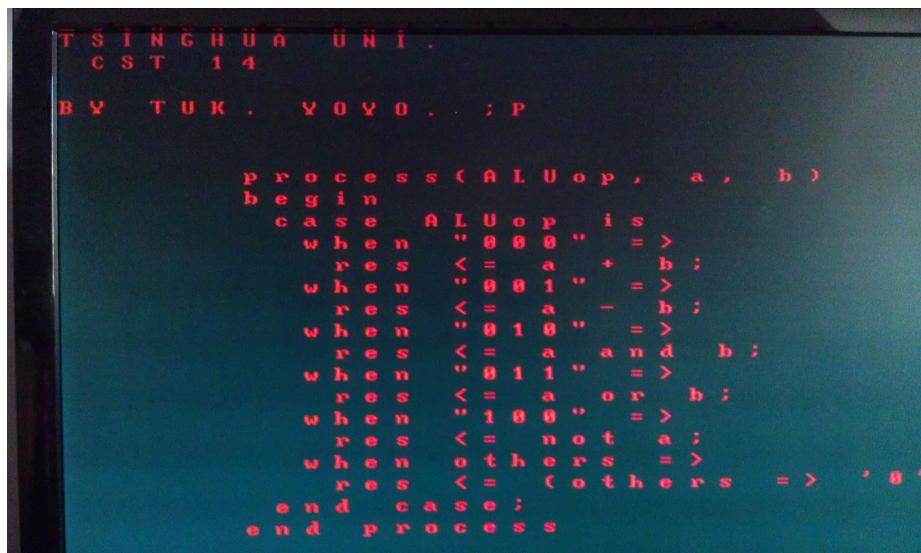


Figure 9: 串口读写，写串口 ``23\r\n''



```
TSINGHUA UNI.  
CST 14  
  
B Y T U R . V 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 10: 键盘、串口及 VGA 显示的记事本

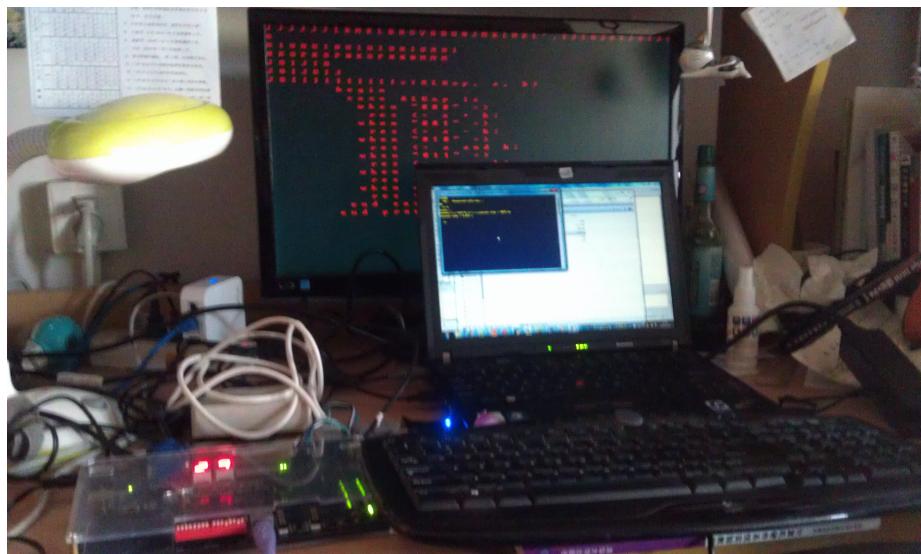


Figure 11: 整体测试截图

## 11 实验心得和体会

### 11.1 自定义指令集

课本上的指令集为了实现 44 条指令，不仅通过前 5 位作为 Op 判断指令类型，还需要通过其他的位才能判断具体是什么指令。这在 decoder 时，在一层 case 中要需要嵌套一层 case 语句来判断具体指令，而且在后面需要根据指令类型的运算中需要传入整个 16 位指令，不能只通过前 5 位 op 来判断。这很不方便。而我们的 CPU 只要求实现 30 条指令（25 条基本 +5 条扩展），因次前 5 位 op 完全足够判断指令类型，因此我们重新定义了指令集的二进制编码。定义指令集的过程中遵守相似的指令相邻。这样就可以少写很多 case。如 controller 中对控制信号 ALUSrc 选择可以如下写，否则要通过大量 case 语句来实现，这里得到了大大的简化。

```
if Op = "01010" or Op = "01011" or (Op >= "01110" and Op <= "10111") then  
    ALUSrc <= '1';  
else  
    ALUSrc <= '0';  
end if;
```

### 11.2 数据通路的设计

数据通路完全由一个人设计，保证整个 cpu 设计的一致性。其中数据通路共修改了 10 多次。在写代码之前就改了 7、8 次。开始写 cpu 的模块后又发现了一些问题又改了数次。后来的修改主要在旁路的设计部分，刚开始写的时候考虑不周。写代码时完全按照数据通路来写，保证了程序的一致。写的时候深深感受到数据通路一定要先设计好，这样写的时候思路才流畅，犯的错误会相对少一些，检查起来也十分方便。

### 11.3 代码同步与管理

由于是两个人同时写代码，所以需要有效的代码管理和同步，我们使用了 Git9 对代码进行了管理。不仅能独立的在各自分支中开发相应功能，也能很快的同步对方的代码、查看小组的整体进度，还可以使用 git 工具对之前的提交版本进行对比从而更加高效的从错误中恢复。

### 11.4 代码调试

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。

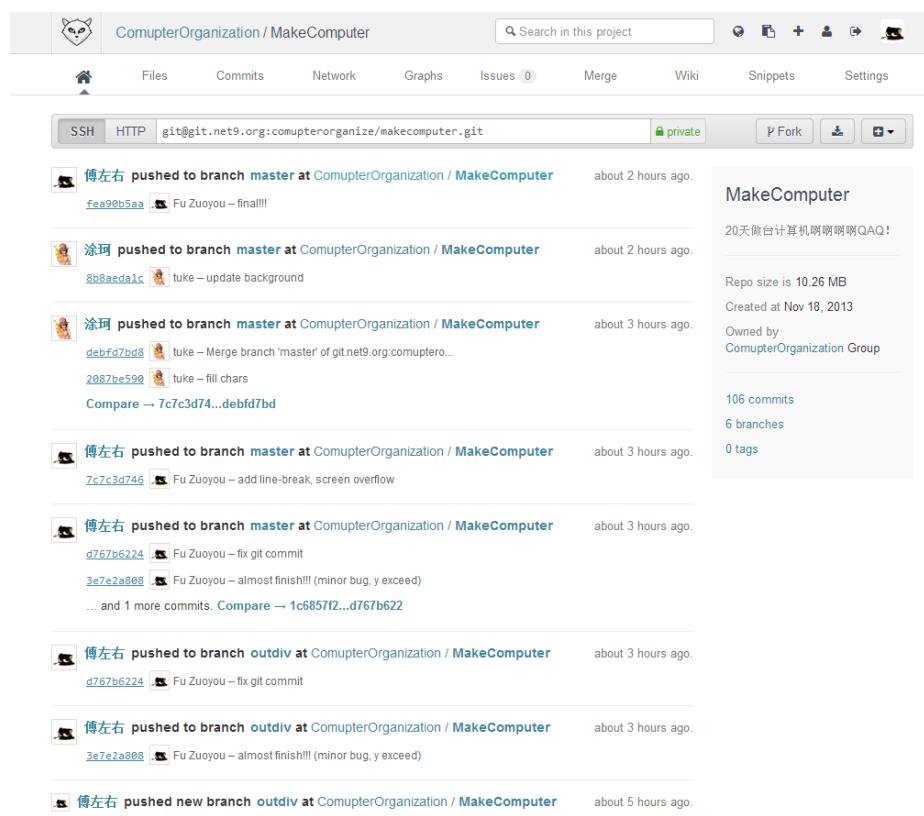
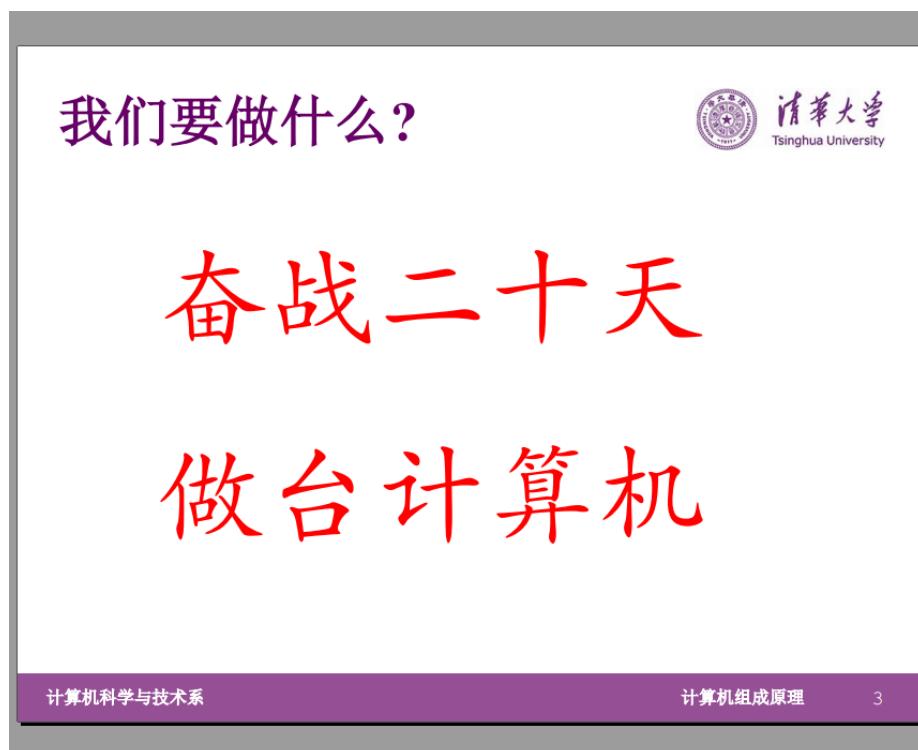


Figure 12: git 使用

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。

## 11.5 小记

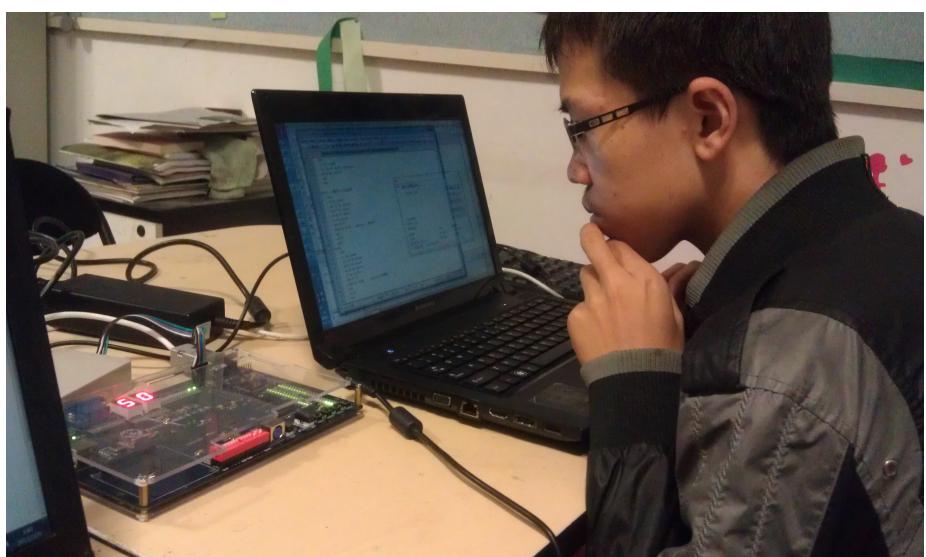
从“奋斗 20 天，做台计算机”。

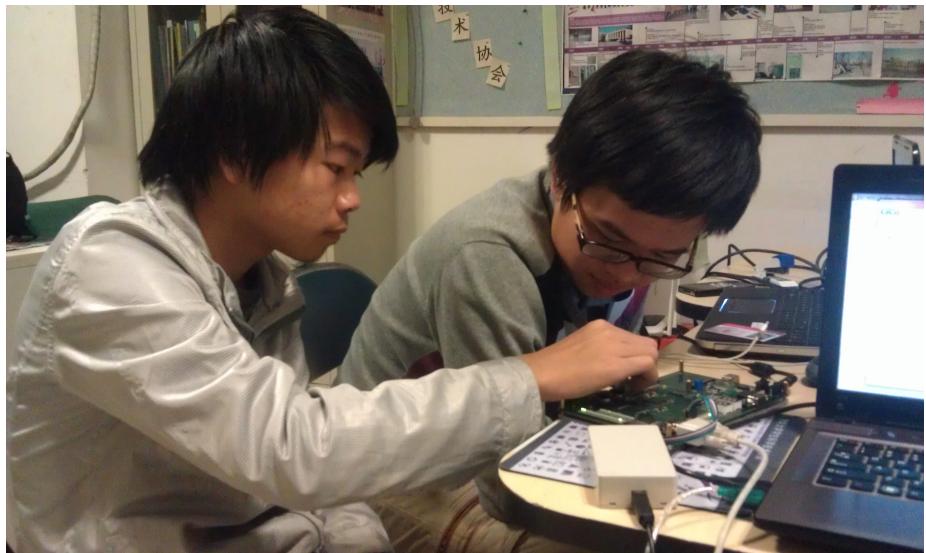


一路走过来，有用一堆莫名其妙的硬件第一次点亮了屏幕的喜悦。



也有 DDL 当天在 C 楼刷通苦苦调 bug 的内心抑郁。





不过令人开心的是，最后还是做出来了呢。画了那么久的图，研究了那么久的代码，调试了那么久。天不负我，总算有一个马马虎虎还行的结果了。

```
TSINGHUA UNI.  
CST 14  
  
BY TURK. VOVOK. ;P  
  
process(ALUop, a, b)  
begin  
case ALUop is  
when "000" =>  
    res <= a + b;  
when "001" =>  
    res <= a - b;  
when "010" =>  
    res <= a and b;  
when "011" =>  
    res <= a or b;  
when "100" =>  
    res <= not a;  
when others =>  
    res <= (others => '0');  
end case;  
end process
```

A screenshot of a computer terminal window. The title bar reads "TSINGHUA UNI." and "CST 14". Below the title bar, it says "BY TURK. VOVOK. ;P". The main area of the terminal shows assembly language code. The code defines a procedure named "process" that takes three parameters: "ALUop", "a", and "b". It uses a "case" statement to handle different values of "ALUop". For each case, it assigns a value to "res" based on the operation specified in the case label. For example, when "ALUop" is "000", "res" is assigned the value of "a + b". There are seven cases in total, including one for "others". The code ends with an "end process" statement.

真好。谢谢老师和助教。