Tsinghua University

计算机组成原理大作业实验报告

基于 THINPAD 的微型计算机系统

计 05 2010011358 许欣然 2010011361 邹林希

2012-12-7

1 目录

1 目录 1
2 实验目标概述 3
3 实验设计细节 3
4 特色 3
5 整体结构 4
5.1 整体设计图 4
5.2 模块分区图 5
6 片内信号及约定 6
6.1 片内控制信号 6
6.2 统一寄存器编址 6
6.3 自控制部件的特殊处理 7
6.4 字符编码 7
6.5 指令集与控制信号对照表 8
7 主要模块设计 9
7.1 硬件 9
7.1.1 ALU 9
7.1.2 Controller 9
7.1.3 BranchSelector 9
7.1.4 RegisterFile 9
7.1.5 SignExtend 9
7.1.6 旁路控制器 9
7.1.7 MemoryController （IO） 10
7.1.8 TControl 11
7.1.9 HazardDetect 11

7.1.10 KeyboardAdapter 11
7.1.11 VGAAdapter 11
7.2 软件 12
7.2.1 记事本 12
7.2.2 bin -> hex 转换器 13
7.3 调试模块 13
7.3.1 VGA-Debugger 13
7.3.2 SmartClock 14
8 主要模块实现 14
8.1.1 ALU 14
8.1.2 Controller 14
8.1.3 BranchSelector 15
8.1.4 RegisterFile 15
8.1.5 SignExtend 15
8.1.6 旁路控制器 15
8.1.7 MemoryController （IO） 15
8.1.8 TControl 17
8.1.9 HazardDetect 18
8.1.10 KeyboardAdapter 18
8.1.11 VGAAdapter 18
9 扩展部分 18
10 实验成果展示 18
11 实验心得和体会 22
12 注 24

2 实验目标概述

- 设计基于 THINPAD 的计算机，其中 CPU 采用基于 MIPS16e 的修改的指令集
- 支持从 Flash 自启动
- IO 部分支持 PS/2 键盘、VGA 接口、串口

3 实验设计细节

- CPU 指令集设置有 Branch Delay Slot，可通过放置无关指令来提高运行效率
- CPU 主频为 12.5MHz
- VGA 分辨率为 640*480
- 内存部分通过倍频工作模拟实现双端口内存，不区分指令和数据，符合冯·诺依曼结构

4 特色

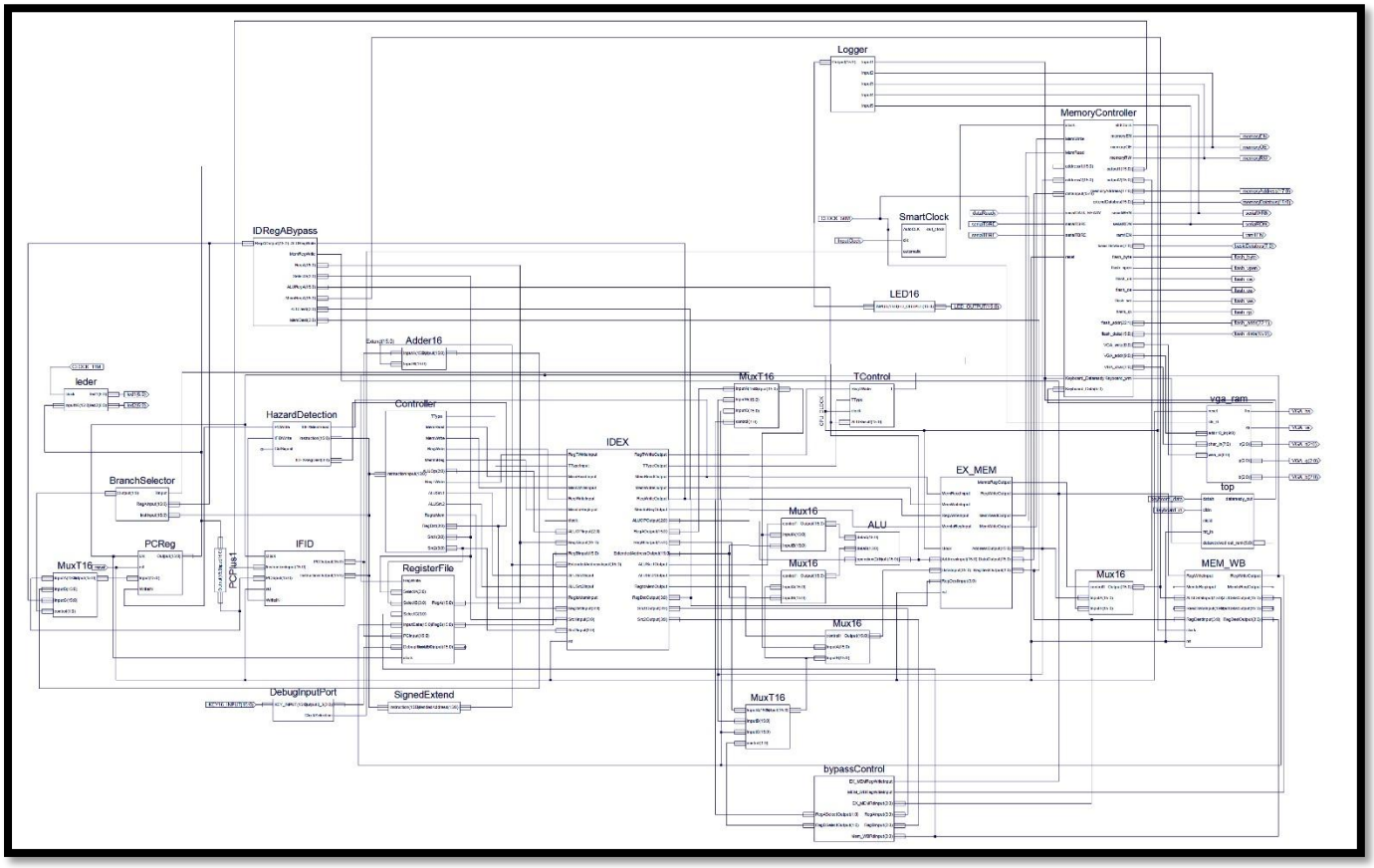
- 明确的模块性
- 统一寄存器编址
- VGA、键盘两种人机交互设备
- 含有 LED_Logger、VGA Debug 模块，使用方法类似于电压表，方便通过原理图接入
- 提供智能时钟控制模块，方便切换手动、自动时钟，提高了调试速度
- 自控制模块
- 扩展较多
  - 数据旁路
  - 控制冒险控制单元
  - VGA 显示器
  - 键盘
  - Flash 自启动
  - 记事本小程序（支持换行）

- ✧ Bin -> hex 转换器

5 整体结构

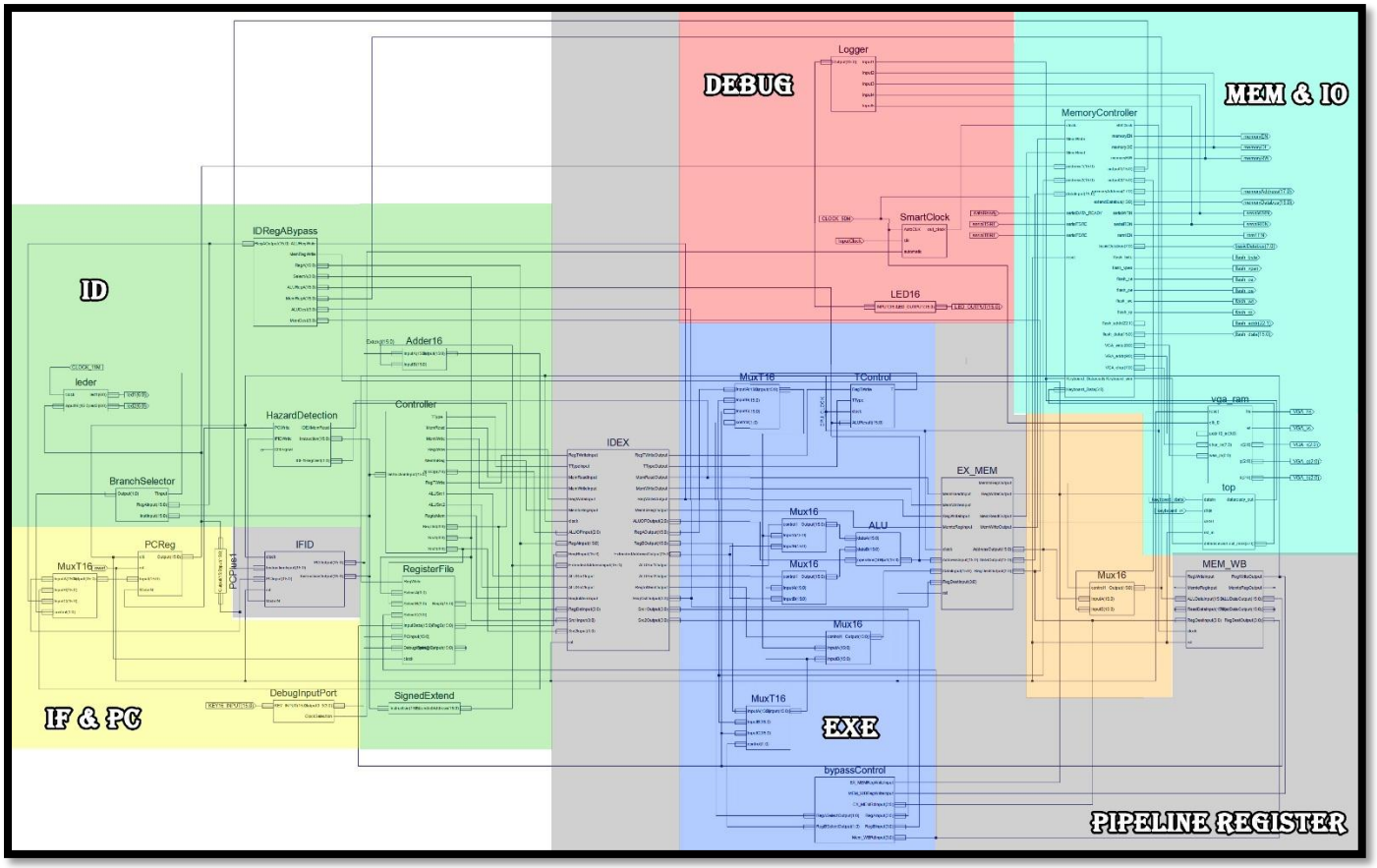
5.1 整体设计图

详见 datapath.xps 文件。



5.2 模块分区图

图中仅大致的区分了各个流水阶段，WB 阶段与 ID 阶段均共享寄存器堆，且 WB 阶段无额外电路，故无 WB 专用分区。



6 片内信号及约定

6.1 片内控制信号

每一级流水均会储存相应的信号，此处只列出信号名称。

MemRead	0：X	1：读内存				
MemWrite	0：X	1：写内存				
RegWrite	0：X	1：写入寄存器				
MemtoReg	0：ALU 输出	1：内存输出				
RegtoMem	0：RegA	1:RegB				
ALUOP(3)	000:加	001:减	010:与	011:或	100:左移	101:右移
TType	0：大于等于-> 0	1：等于 -> 0				
RegTWrite	0:X	1:写入				
Branch(2)	00:PC+4	01:PC+4+IMM	10：RegA			
ALUSrc1	RegA	立即数				
ALUSrc2	RegB	立即数				
RegASelect(2)	00：寄存器堆	01：EX/MEM	10：MEM/WB			
RegBSelect(2)	00：寄存器堆	01：EX/MEM	10：MEM/WB			

6.2 统一寄存器编址

由于 PC、IH、SP 等特殊寄存器对于 CPU 内部结构设计、旁路复杂度及设计简洁性均有较大影响，故在 CPU 内部将寄存器编码长度扩展至 4 位，并将特殊寄存器和普通寄存器统一至 4 位编码。此处借鉴了 MIPS32 的零寄存器的思想，设置了 ZERO 寄存器，始终对外输出 0 值。

四位编码	寄存器
0 + 三位编码	8 个普通寄存器
1000	ZERO 寄存器
1001	SP 寄存器
1010	PC 寄存器
1111	IH 寄存器

6.3 自控制部件的特殊处理

由于部分元件的控制信号仅与自己相关，为了减少 Controller 的复杂度以及减少模块间布线难度，一些部件采用自主控制方式。

采取自控制的部件有：地址扩展单元，跳转数据旁路单元，跳转控制单元。

以上部件均直接接受 16 位指令，部件自身对指令进行分析并输出正确的解码结果，大大的减少了片内模块间的传输信号量及控制器设计复杂度，同时也让两个人分工制作的工作量均衡一些。

6.4 字符编码

字符	编码	字符	编码	字符	编码
0	0	**e**	14	**s**	28
1	1	**f**	15	**t**	29
2	2	**g**	16	**u**	30
3	3	**h**	17	**v**	31
4	4	**i**	18	**w**	32
5	5	**j**	19	**x**	33
6	6	**k**	20	**y**	34
7	7	**l**	21	**z**	35
8	8	**m**	22		
9	9	**n**	23		
a	10	**o**	24		
b	11	**p**	25		
c	12	**q**	26		
d	13	**r**	27		

6.5 指令集与控制信号对照表

详细表格详见 Signal.xlsx，控制信号以 Controller 内部实现为准，由于后期 Debug 时未对该表进行修改，可能有少数几个控制信号不准确。

为了安全性着想，表中的空余项均使用 ZERO 寄存器替代。

指令	15-11	10-8	4-0	RegA	RegB	RegDst	ALUSrcA	ALUSrcB	MemRead	MemWrite	RegWrite	MemtoReg	RegtoMem	ALUOP(3)	TType	RegTWrite	Branch
ADDSP3 rx imm	00000			SP	IMM	8-10	0	1	0	0	1	0	0	000	X	0	00
NOP	00001					X	X	X	0	0	0	0	0	X	X	0	00
B imm	00010					X	X	X	0	0	0	0	0	XXX	X	0	01
BEQZ rx imm	00100					X	X	X	0	0	0	0	0	XXX	X	0	01
BNEZ rx imm	00101					X	X	X	0	0	0	0	0	XXX	X	0	01
SLL rx ry imm	00110		XXX00	5-7	IMM	8-10	0	1	0	0	1	0	0	100	X	0	00
SRA rx ry imm	00110		XXX11	5-7	IMM	8-10	0	1	0	0	1	0	0	101	X	0	00
ADDIU3 rx ry imm	01000			8-10	IMM	5-7	0	1	0	0	1	0	0	000	X	0	00
ADDIU rx imm	01001			8-10	IMM	8-10	0	1	0	0	1	0	0	000	X	0	00
SLTI rx imm	01010			8-10	IMM	X	0	1	0	0	0	0	0	001	0	1	00
ADDSP imm	01100	011		SP	IMM	1001	0	1	0	0	1	0	0	000	X	0	00
BTEQZ imm	01100	000				X	X	X	0	0	0	0	0	XXX	X	0	01
BTNEZ imm	01100	001				X	X	X	0	0	0	0	0	X	X	0	01
MTSP rx	01100	100		5-7	ZERO	1001	0	0	0	0	1	0	0	000	X	0	00
LI rx imm	01101			IMM	ZERO	8-10	1	0	0	0	1	0	0	000	X	0	00
LW_SP rx imm	10010			SP	IMM	8-10	0	1	1	0	1	1	0	000	X	0	00
LW rx ry imm	10011			8-10	IMM	5-7	0	1	1	0	1	1	0	000	X	0	00
SW_SP rx imm	11010			SP	8-10	X	0	1	0	1	0	0	1	000	X	0	00
SW rx ry imm	11011			8-10	IMM	X	0	1	0	1	0	0	1	000	X	0	00
ADDU rx ry rz	11100		XXX01	8-10	5-7	2-4	0	0	0	0	1	0	0	000	X	0	00
SUBU rx ry rz	11100		XXX11	8-10	5-7	2-4	0	0	0	0	1	0	0	001	X	0	00
JR rx	11101		00000			X	0	X	0	0	0	0	0	XXX	X	0	10
MFPC rx	11101		00000	PC	ZERO	8-10	0	0	0	0	1	0	0	000	X	0	00
SLT rx ry	11101		00010	8-10	5-7	X	0	0	0	0	0	0	0	001	0	1	00
SRAV rx ry	11101		00111	5-7	8-10	5-7	0	0	0	0	1	0	0	101	X	0	00
CMP rx ry	11101		01010	8-10	5-7	X	0	0	0	0	0	0	0	001	1	1	00
AND rx ry	11101		01100	8-10	5-7	8-10	0	0	0	0	1	0	0	010	X	0	00
OR rx ry	11101		01101	8-10	5-7	8-10	0	0	0	0	1	0	0	011	X	0	00
MFIH rx	11110		00000	IH	ZERO	8-10	0	0	0	0	1	0	0	000	X	0	00
MTIH rx	11110		00001	8-10	ZERO	1111	0	0	0	0	1	0	0	000	X	0	00

7 主要模块设计

7.1 硬件

设计中遵循一个原则：尽可能的减小时序电路所占的比例。最终的代码中含有时序电路的部分仅有各个寄存器及内存控制单元，保证了计算效率，提高了时序稳定性，故以下部分均忽略了时钟输入。寄存器堆及 T 寄存器内的寄存器均为负边沿写入，阶段寄存器均为正边沿写入。

7.1.1 ALU

ALU 单元根据长度为 3 的运算指令进行对应的运算。指令与运算的对应关系为：（设输入分别为 A、B）

Op	运算
000	A + B
001	A – B
010	A & B
011	A | B
100	A << B
101	A >> B

7.1.2 Controller

控制单元是 CPU 中代码最长的组件，但是它的逻辑是最简单的一个，依据信号表完全通过组合逻辑对输入的指令进行分析并输出对应的控制信号。

7.1.3 BranchSelector

跳转控制器，判断指令的跳转信号 Branch，详见指令集与控制信号对照表。从 PC+1，PC+1+IMM，RegA 中通过一个三选一选择器输出到 PC 写入端口。

7.1.4 RegisterFile

寄存器堆接受 PC 寄存器的值，两个四位的输入选择，输入数据及写入控制。寄存器堆组合了 14 个寄存器（4 个备用），根据统一寄存器编码输出普通寄存器、特殊寄存器、PC 寄存器的值。写入时，若写入地址为不合法地址，则写入无效。

寄存器堆额外提供了一套读取端口，方便单步 Debug 时，实时的查询每个寄存器的结果。

7.1.5 SignExtend

符号扩展器接受全 16 位指令，根据指令内容对立即数字段进行相应的扩展并输出 16 位结果，扩展的方式由扩展器内部通过组合逻辑对指令分析得到，无需控制器的额外控制。

7.1.6 旁路控制器

数据旁路被分为了两部分，正常的数据通路和跳转专用的数据旁路，两者的功能是有一定重叠的。

7.1.6.1 bypassControl（正常数据旁路）

此数据通路的主要功能为从 EX_MEM 端和 MEM_WB 两端流水寄存器处向 ALU 阶段返回寄存器的更新值，由于统一化寄存器编址的原因，使得该部分的控制单元非常简单。其中控制 RegASelect 和 RegBSelect 两个控制信号的逻辑表达式为：（两者类似，此处只列出一个，信号含义详见片内控制信号。

信号	赋值	条件	解释
RegASelectOutput <=	“01”	when ((EX_MEM_RegWrite = '1') and (EX_MEM_Dest /= "1000") and (EX_MEM_Dest = RegA)) else	EX_MEM 端写入寄存器 写入目标不为 ZERO 寄存器 写入目标等于 RegA 选择信号
	“10”	when ((MEM_WB_RegWrite = '1') and (Mem_WB_Dest /= "1000") and (Mem_WB_Dest = RegA) and (EX_MEM_Dest /=RegA)) else	MEM_WB 端写入寄存器 写入目标不为 ZERO 寄存器 写入目标等于 RegA 选择信号 EX_MEM 端无更新的数据
	“00”	"00" ;	默认写入来自寄存器堆的值

该元件仅为旁路的控制器，在 ALU 的输入前端还有额外的三选一选择器，用于选择输入源。

7.1.6.2 IDRegABypass（跳转专用数据旁路）

为了加快跳转，跳转控制被设计在了 ID 段，这使得根据寄存器的值跳转的指令无法使用正常的数据旁路。由于 Controller 部分的处理使得所有跳转相关的寄存器均从 RegA 端口读出，故此阶段的专用数据旁路仅针对 RegA，并且将数据选择器合并到控制器中，减少布线难度。

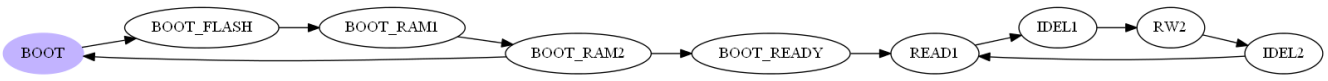
内部实现与普通的数据旁路基本一致，区别仅为返回源及返回目标不同。

7.1.7 MemoryController （IO）

MemoryController（下简称为 MC）被设计为负责 CPU 与一切 IO 和存储设备的交互组件，类似于 x86 计算机系统中的北桥、南桥芯片，内部逻辑最为复杂。为了提高可靠性，MC 运行在高于 CPU 主频四倍的时钟上，MC 的内部的状态机需要和外部时钟保持同步，所以 MC 接受 50M 时钟后对外分频，CPU 内其他部件均使用 MC 提供的时钟。

MC 内部分为了四个部分：Flash 启动，RAM 读写，键盘、串口端口映射，显存映射。

MC 的状态机拥有两类状态：启动状态（BOOT, BOOT_FLASH, BOOT_RAM1, BOOT_RAM2, BOOT_READY）和运行时状态（READ1, IDEL1, RW2, IDEL2），状态之间的转换关系如下图，转换均在时钟上升沿时发生。



当启动时处于 BOOT 状态，进入读取 FLASH，写入 RAM2 的循环，当载入完毕时进入 BOOT_READY 状态，此阶段对外不输出时钟，BOOT_FLASH 状态内部还包含了 512 个状态用于等待 FLASH 读出内容。

当进入 READ1 时开始对外输出时钟，READ1, IDEL1 为高电平，其他为低电平。READ1 为指令取址，RW2 对应 MEM 段内存操作，IDEL1 和 IDEL2 阶段为下一周期做准备工作。

串口端口按照实验要求被映射到了 BF00 和 BF01 上，键盘的 KeyboardAdapter 端口采用了相似的映射方式，映射到了 BF02 和 BF03。BF01 最低位为串口可写，次低位为串口可读，BF03 最低位为键盘可读。字符编码详见字符编码部分，编码之外的按键均返回空格对应的编码。

显存映射到了内存的 FFXX 段，从 FF00 起共 240 个单元对应屏幕上 14 行*16 列。字符编码详见字符编码部分。

7.1.8 TControl

由于 T 寄存器的写入需要额外的逻辑判断、写入方式与常规寄存器不同，且在 ID 阶段需要 T 寄存器的值帮助确定跳转位置，故将 T 寄存器及相关的逻辑单元组合成了一个 TControl 元件。元件放置于 ALU 段，接受 ALU 的运算结果和相关的控制信号，并将 T 寄存器的结果返回给 ID 段中的跳转控制器用于跳转控制。

该组件在时钟的负边沿根据 RegTWrite 信号来确定是否对 T 寄存器进行写入，根据 TType 信号确定写入的逻辑表达式。

7.1.9 HazardDetect

由于数据旁路的引入及不区分数据、指令内存，使得控制冒险仅有一种情况会发生：跳转指令所查询的寄存器在上一个语句中刚从内存中读出，此时需要等待一个周期，才能从 MEM 段引回至 ID 段。

当检测到 ALU 段的指令为 LW 指令且与 ID 段指令访问的寄存器相同时，发出信号使得 PC、IFID 两个寄存器暂停写入一个周期，我们将 IDEXE 阶段的清零信号忽略掉，因为控制器对于跳转信号的默认输出就是全 0，所以此处不清零也不会影响 CPU 的正常工作。

7.1.10 KeyboardAdapter

PS/2 键盘接口程序氛围两个主要模块。一是键盘数据接收及分析，对键盘产生的数据滤波去除毛刺，接受键盘串行数据并提取出扫描码。二是键盘数据译码及输出，对扫描码进行译码，并按照串口的通信原理实现对外接口。

7.1.11 VGAAdapter

对外提供显存的地址和数据接口，可以将显存中数据与字母码表对应并输出到屏幕相应位置上，显示屏分辨率为 640*480 像素，为了更好地支持换行功能，一行的字符数字被设定在了 16 个，方便通过移位指令模拟乘法。

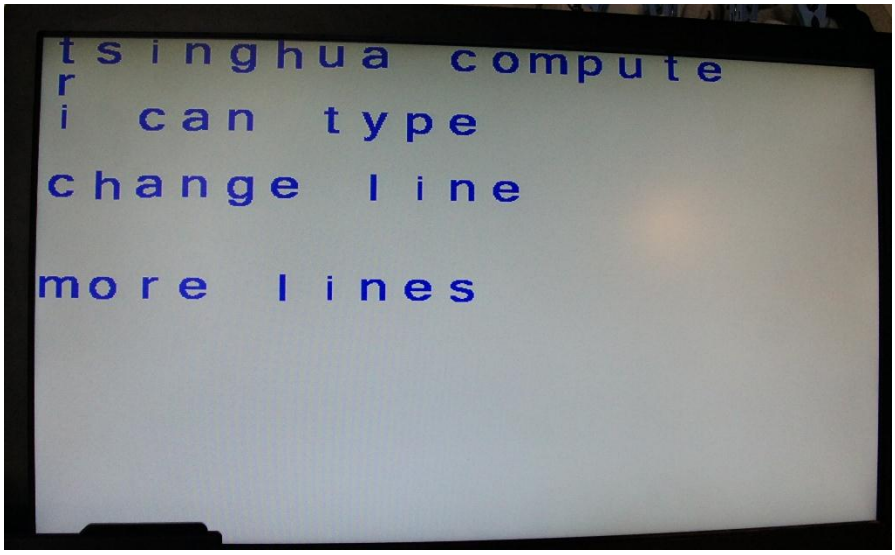
7.2 软件

7.2.1 记事本

代码详见 notepad.s。

```
    LI R3 0x0 ;行号
    LI R4 0x0 ;列号
START:  ;判断键盘是否可读
    NOP
    LI R0 0xBF
    SLL R0 R0 0x0
    NOP
    NOP
    NOP
    LW R0 R1 0x03
    NOP
    BEQZ R1 START
    NOP
LOAD:   ;读取键盘
    LW R0 R2 0x02
    LI R0 0x38
    CMP R0 R2
    BTNEZ JUDGECOLUMN
    NOP
    ADDIU R3 0x1
    LI R4 0x0
    B START
    NOP
JUDGECOLUMN:    ;判断是否达到行尾
    LI R0 0x10  ;每行 16 字
    CMP R0 R4
    BTNEZ JUDGEROW
    NOP
    ADDIU R3 0x1    ;行号+1
    LI R4 0x0       ;列号归 0
JUDGEROW:       ;判断是否到达屏幕尾端
    LI R0 0x0E  ;14 行
    CMP R0 R3
    BTNEZ PRINT
    NOP
    LI R3 0x0   ;行号清零
PRINT:
    LI R0 0xFF
    SLL R0 R0 0x0   ;显存起始地址
    SLL R1 R3 0x4   ;行号 * 16
    ADDU R0 R1 R0
    ADDU R0 R4 R0   ;偏移量 = 行号 * 16 + 列好
    ADDIU R4 0x1    ;列号+1
    SW R0 R2 0x0
B START     ;返回起始
NOP
JR R7
NOP
```

使用截图：



7.2.2 bin -> hex **转换器**

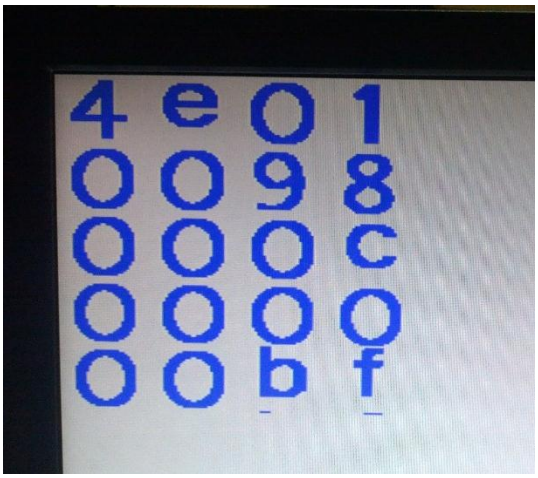
由于使用 Kernel 做为运行环境输入代码需要输入指令或者 16 进制表示，无法利用编译器提供的标号、注释等高级功能 ,导致书写程序极为不便。我们编写了 binToHex.py 程序帮助从编译器的.bin 文件翻译成 Term 接受的 16 进制文件。

使用方法为：python binToHex.py sourcefile.bin，会产生 sourcefile.hex，用记事本打开后可以直接复制到 Term 中。

7.3 调试模块

7.3.1 VGA-Debugger

在 CPU 项目的编写同时，我们同时进行着 VGA-Debugger 的编写。VGA-Debugger 提供了 5 个 16 位输入端口，可以将 CPU 内部的工作状态通过 4 位 16 进制字符显示在 VGA 屏幕上，使用图如下：



该图中：第一行为 PC 当前值，第二行为当前 MC 部件读取的内存值，第三、四行为 ALU 的两个输入值。最后一行为寄存器堆的 Debug 端口输出值，根据拨码管低四位的值，实时的读取出任意寄存器的值。

由于该模块完成之后对于 CPU 的调试提供了极大的便捷，对于发现 CPU 内部的信号问题极为有帮助，在有需求时更可以增加 Debug 端口的输出行数。该模块在项目后期独立发展为了 VGAAdapter 组件，为 CPU 提供了显示功能。

7.3.2 SmartClock

由于调试时反复的通过手动时钟运行非常的浪费时间，故设计了一个智能的时钟模块。该模块提供自动、手动时钟两种模式，且可以通过开关切换，在调试较长代码时，配合 VGA-Debbuger 显示的 PC 值，可以先使用较慢的自动时钟运行至出错位置附近，然后换为手动时钟单步调试。

该模块在最终的 CPU 中被保留了下来，自动时钟频率被调节到了 50M，切换开关被设定在了拨码管的第五位，设定为 1 时代表自动时钟。

8 主要模块实现

主要的设计思想已在主要模块设计中写明，此部分只粗略写明代码思路，具体请查看 Project 文件夹内对应 vhd 文件。

8.1.1 ALU

根据 Op 信号，通过 case 语句选择对应运算结果输出。

8.1.2 Controller

通过 Case 语句判断高五位的值来对控制信号选择输出，对于无法通过前五位区分的指令采取多层 case 嵌套的方式处理。

代码格式如下：

```
case InstructionInput(15 downto 11) is
    when "00000" => --ADDSP3
        RegDst <= "0" & InstructionInput(10 downto 8);
        Src1 <= "1001"; --SP
        Src2 <= "1000"; --ZERO
        ALUSrc1 <= '0';
        ALUSrc2 <= '1'; --立即数
        MemRead <= '0';
        MemWrite <= '0';
        RegWrite <= '1';
        MemtoReg <= '0';
        RegtoMem <= '0';
        ALUOp <= "000";
        TType <= '0';
        RegTWrite <= '0';
    …
    when "11100" => --ADDU & SUBU
        RegDst <= "0" & InstructionInput(4 downto 2); --2-4
        Src1 <= "0" & InstructionInput(10 downto 8); --8-10
```

```
        Src2 <= "0" & InstructionInput(7 downto 5); --5-7
        ALUSrc1 <= '0'; --立即数
        ALUSrc2 <= '0'; --0
        MemRead <= '0';
        MemWrite <= '0';
        RegWrite <= '1';
        MemtoReg <= '0';
        RegtoMem <= '0';
        case InstructionInput(1 downto 0) is
            when "01" =>
                ALUOp <= "000";
            when others =>
                ALUOp <= "001";
        end case;
        TType <= '0';
        RegTWrite <= '0';
    when others =>
        …
end case;
```

8.1.3 BranchSelector

跳转控制器，根据指令给出对应的 Branch 选择信号。

8.1.4 RegisterFile

时序部分交给实例化的寄存器内部处理，自身仅通过多层数据选择器实现纯组合逻辑的操作。

8.1.5 SignExtend

根据指令对指令中的立即数字段进行相应的符号扩展。

对 SLL 和 SRA 指令，如果立即数是 0，会输出 8，使得 ALU 不用再做判断。

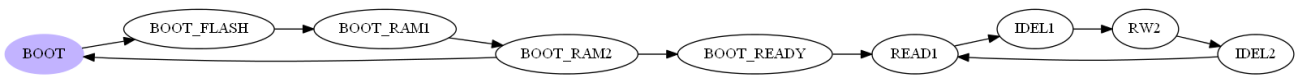
8.1.6 旁路控制器

无特殊特点。

8.1.7 MemoryController （IO）

MC 内部封装了一个 flash_io 元件，此元件通过 FLASH AND RAM 软件源代码中的 flash_io.vhd 实例化得到。

MC 内部尽量使用组合逻辑，时序部分完成的功能仅包含：状态机的切换，输出缓存寄存器的赋值，自启动中的 FLASH 读操作。



BOOT_RAM2 向 BOOT_READY 状态的转跳条件为载入地址已经达到 0x0FFF，故 KERNAL 代码最长可为 4k 条语句。

模块的输入为两个地址（address1, address2）分别对应取址和 MEM 段，MEM 段的读、写信号（Mw、Mr，1 为工作），MEM 段输入数据 dataInput。

内部储存信号有：buffer1、buffer2（两个读取输出），FLASHPC（启动载入地址指针），FLASH_HOLDER（启动数据缓存）。

8.1.7.1 *组合电路部分*

对外 IO 的控制信号输出参照下表进行数据选择：（BOOT 状态全部为 0，此处省略）

信号	BOOT_FLASH	BOOT_RAM1	BOOT_RAM2	BOOT_READY	READ1	IDEL1	RW2	IDEL2
RAM2 地址	FLASHPC	FLASHPC	FLASHPC	ZERO	Address1	Address2	Address2	
RAM2 OE	1	1	1	0	0	0	Not Mr	0
RAM2 RW	1	1	0	1	1	1	1(=BF00) Not Mw	1
扩展总线	高阻	FLASH_HOLDER	FLASH_HOLDER	高阻	高阻	dataInput	dataInput	高阻
基本总线	高阻	高阻	高阻	高阻	高阻	Not Mr (=BF00) Else 高阻	Not Mr (=BF00) Else 高阻	Not Mr (=BF00) Else 高阻
串口读使能	1	1	1	1	1	Not Mr (=BF00) Else 1	Not Mr (=BF00) Else 1	Not Mr (=BF00) Else 1
串口写使能	1	1	1	1	1	1	Not Mw (=BF00) Else 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 (=BF00)	0
VGA 写使能	0	0	0	0	0	0	Mr (=FFxx) 0	0
对外 clock	0	0	0	0	1	1	0	0

上表中假设 address1，address2 为已扩展为 18 位的数据。

硬连接：

VGA 地址 <= address2

VGA 数据 <= inputdata 后八位

RAM1 使能 <= 1，RAM2 使能 <= 0

FLASH 写使能、FLASH 擦除使能 <= 1

FLASH 数据线 <= 高阻

信号：BF01 <= 串口 dataready & (串口 TSER and 串口 TRBE)

信号：BF03 <= 键盘 dataready

8.1.7.2 时序电路部分

若无特殊说明，状态机按照状态表跳转至下一状态。

- BOOT 状态将 BOOT_FLASH 段计数器置 0，将 FLASHPC 置为 FFFF。
- BOOT_FLASH 状态内有一个 8 位长的计数器
  - 处于 0 时向 flash 地址线载入扩展后的 FLASHPC+1，计数器+1，FLASHPC 值+1
  - 处于 511 时从 flash 数据线读出数据至 FLASH_HOLDER，计数器清零，进入 BOOT_RAM1 状态。
  - 否则计数器+1
- BOOT_RAM2 状态判断 FLASH_PC 值是否已经超过 0x0FFF，若超过则跳至 BOOT_READY，否则返回 BOOT_FLASH 状态。
- READ1 状态将 RAM2 数据线上数据读出至 output1 缓存。
- RW2 状态判断地址线数据
  - 若为 BF01，将扩展后的基本总线数据载入 output2 缓存
  - 若为 BF01，将扩展后的 BF01 信号载入 output2 缓存
  - 若为 BF02，将扩展后的键盘数据线上数据载入 output2 缓存
  - 若为 BF03，将扩展后的 BF03 信号载入 output2 缓存
  - 否则载入扩展总线上数据

8.1.7.3 Flash 自启动的实现

在提供给 CPU 时钟信号之前，由 MC 内部将 Flash 芯片的 0x0000~0x0FFF 段复制到 RAM2 中，然后进入工作循环，提供给 CPU 时钟以完成启动过程。

8.1.8 TControl

大于等于信号为 ALU 输出最高位，相等信号为 ALU 输出全部位的或。

当处于时钟负边沿且 RegTWrite 信号为写入时，若 TType = 0，写入大于等于信号的值，否则写入相等信号的值。

8.1.9 HazardDetect

无特殊实现

8.1.10 KeyboardAdapter

对 50MHZ 时钟十分频 ,得到一个 5MHZ 的信号用作滤波时钟。由于每次按键按下再松开会产生三个信号，其中第二个信号是 F0，以此判断一次输入。

对外接口类似于串口，dataready 和 datareceived 对应串口中的 dataready 和 wrn，不同点是 datareceived 在上升沿时认为数据已经读取完毕，将 dataready 置为 0。返回值为字符的输出字符编码，其中空格和回车分别输出“100100”和“111000”。

8.1.11 VGAAdapter

对输入的 50MHZ 时钟二分频 ,产生的 25MHZ 信号用作 VGA 时钟。屏幕分辨率设为 640*480 ,考虑消隐区，每行像素 800 ,每场行数 525。将屏幕分成 16 列 20 行 ,使用 IP CORE 的双端 RAM 作为显存 ,可同时读写，显存中储存屏幕对应位置需要显示的字符的编码。对外提供显存地址和数据的接口。

9 扩展部分

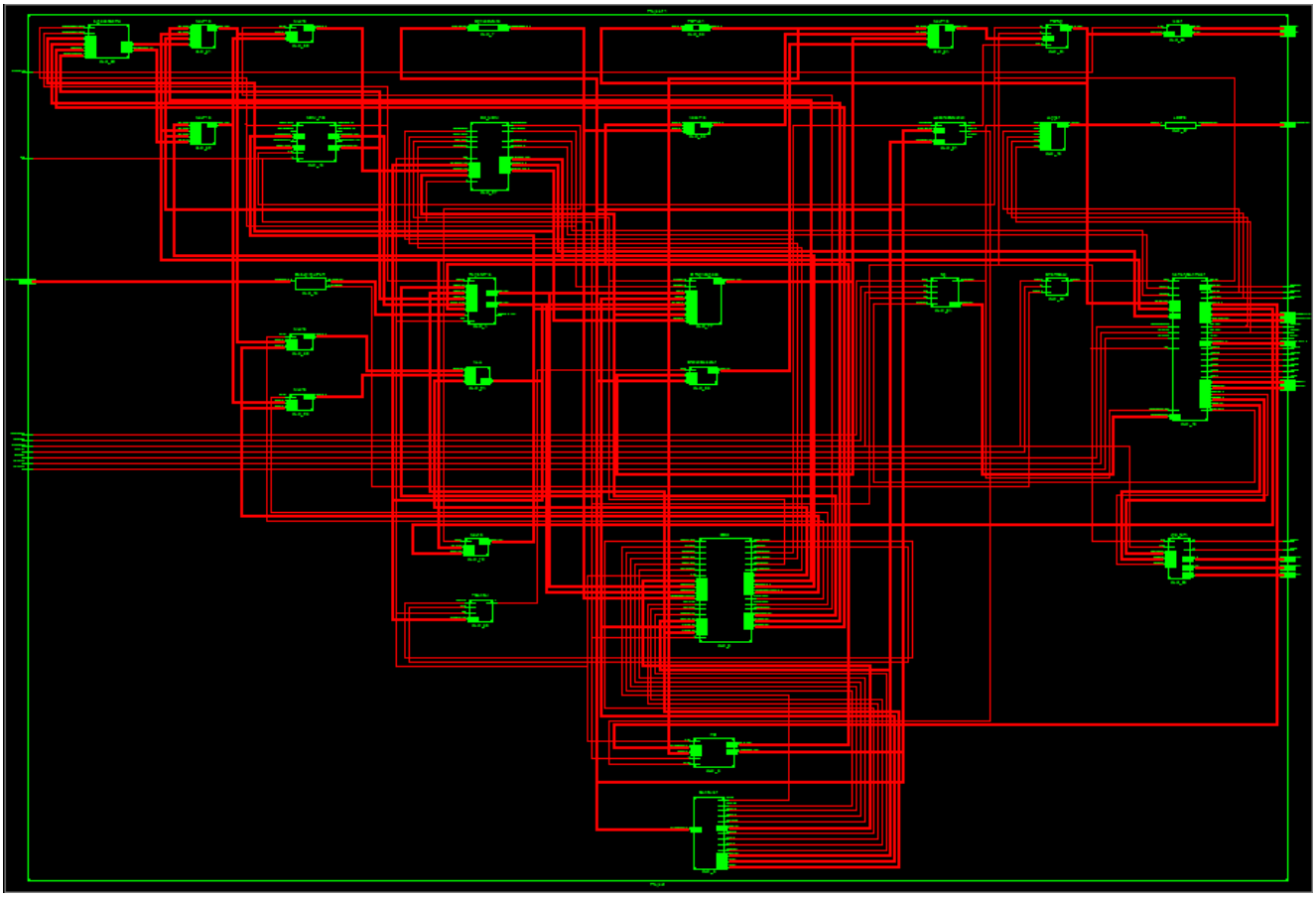
10 实验成果展示

CPU 可以正常运行全部指令，指令。

Xilinx 编译报告

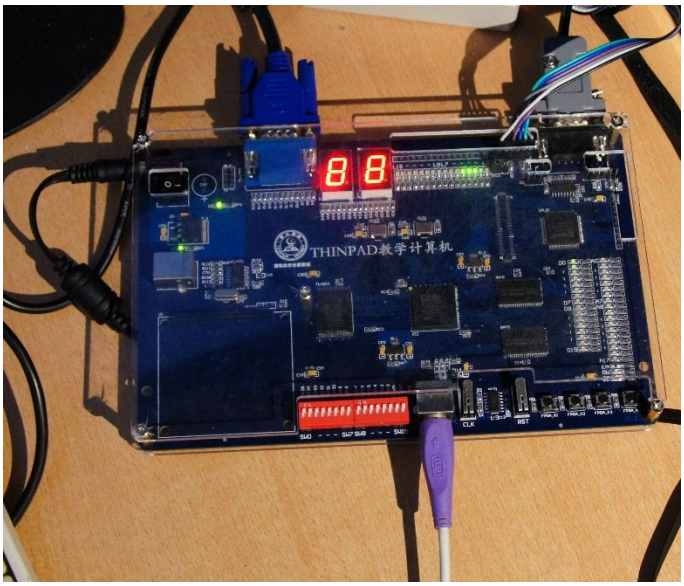
Device Utilization Summary				[-]
Logic Utilization	Used	Available	Utilization	Note(s)
Total Number Slice Registers	631	17,344	3%	
Number used as Flip Flops	623			
Number used as Latches	8			
Number of 4 input LUTs	1,338	17,344	7%	
Number of occupied Slices	908	8,672	10%	
Number of Slices containing only related logic	908	908	100%	
Number of Slices containing unrelated logic	0	908	0%	
Total Number of 4 input LUTs	1,424	17,344	8%	
Number used as logic	1,338			
Number used as a route-thru	86			
Number of bonded IOBs	158	250	63%	
Number of RAMB16s	4	28	14%	
Number of BUFGMUXs	4	24	16%	
Average Fanout of Non-Clock Nets	3.60			

顶层 RTL 图

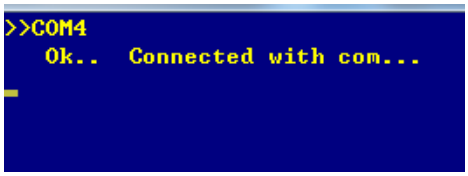


正常使用流程：

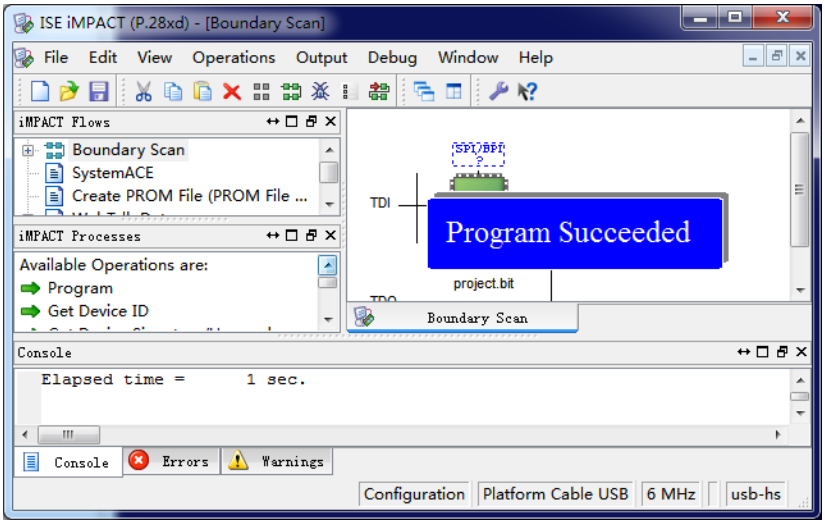
1. 正确连接数据线（电源线，JTAG 线，串口、PS\2，VGA 线）



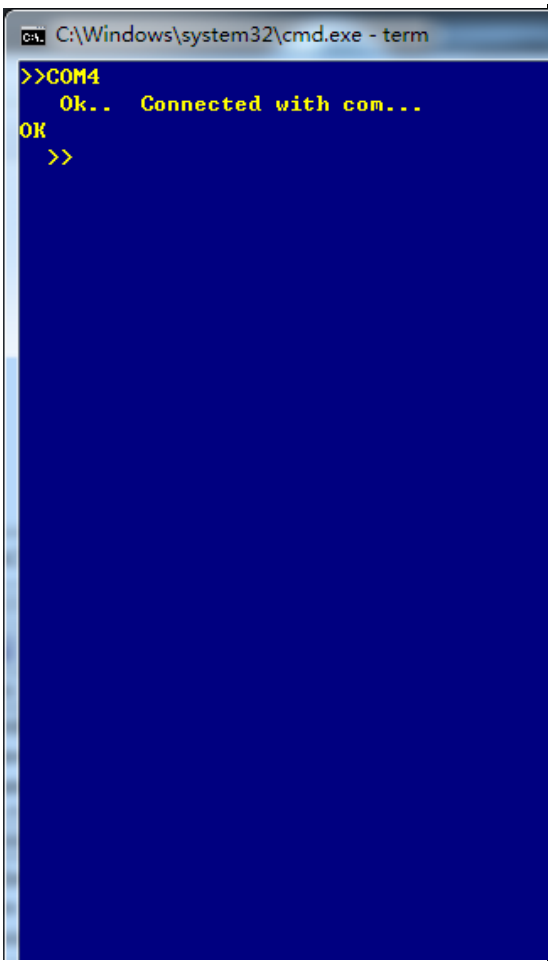
2. 打开 Term，准备好串口

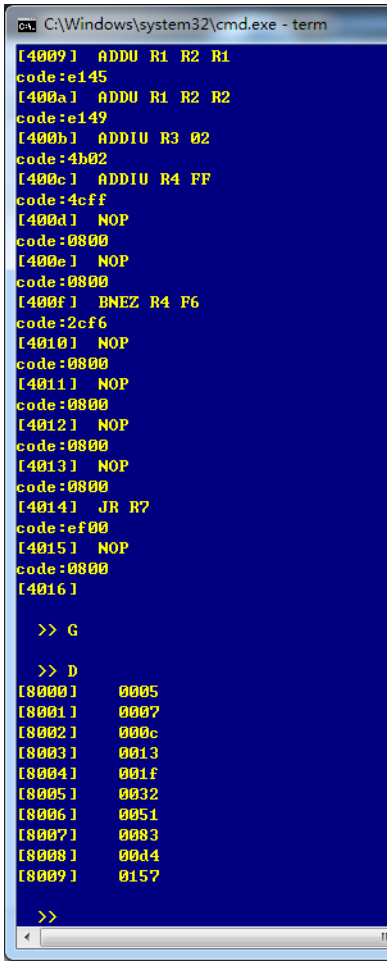


3. 烧录 CPU

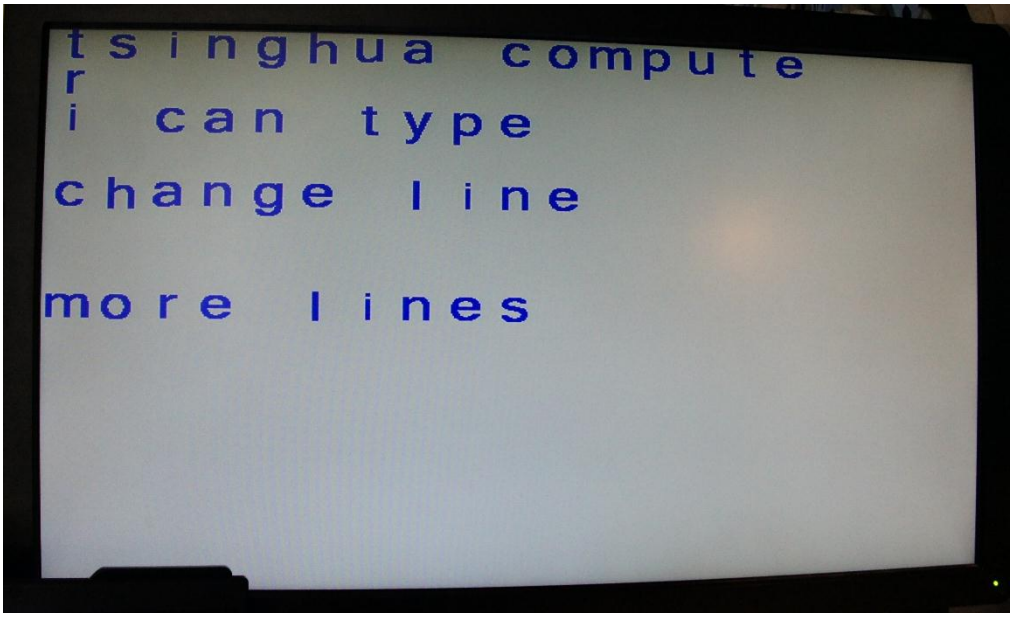


4. 由于有自启动功能，按一次 Reset 键即可正常使用，图为斐波那契数列的程序结果。

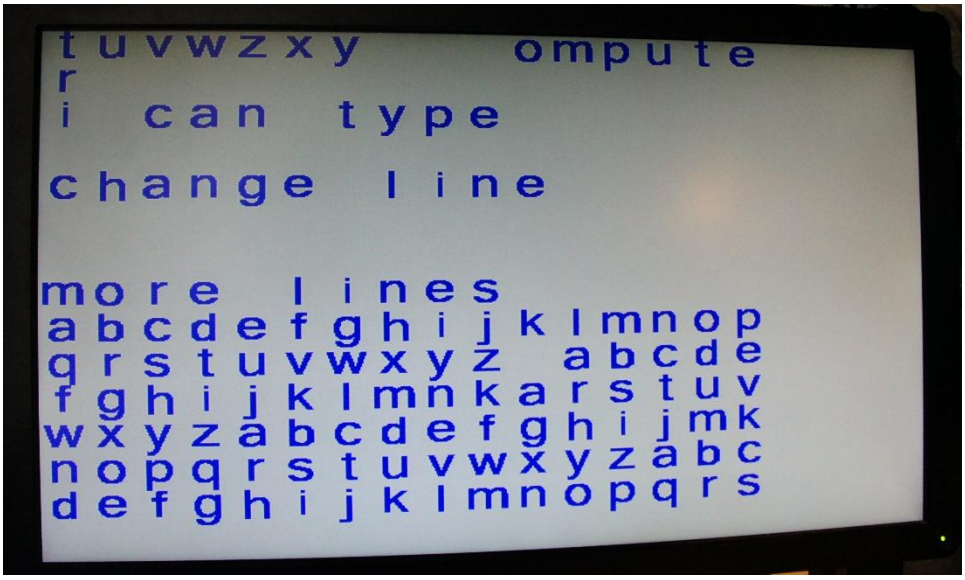




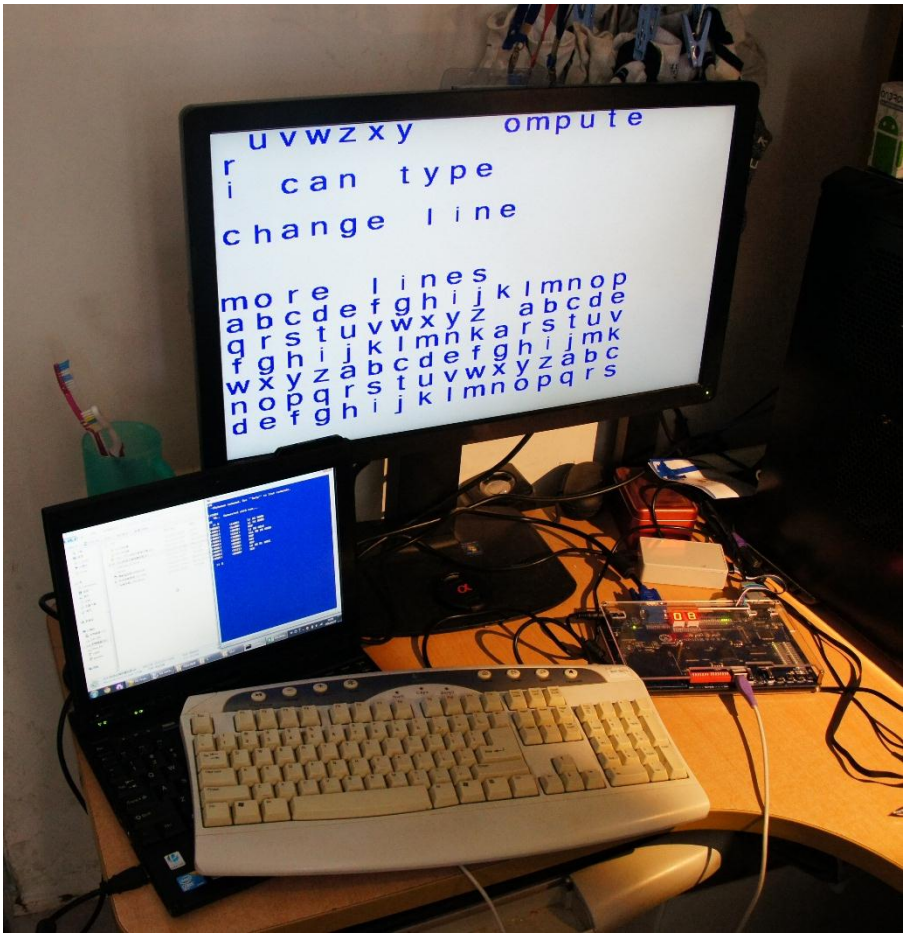
5. 写入编译好的记事本程序运行，测试空格、换行



6. 测试循环写入



7. 整体测试图片



11 实验心得和体会

- 良好的数据通路

我们组的数据通路结构基本由一个人设计，保证了数据通路的整体性。数据通路一共修改了 20 余次，在第 5 次修改之后才开始编写一些确定接口不会改变了的元件，直到第 7 次修改完成后才正式开始编写 CPU，之后的修改大多在更改元件所处的流水线位置（ALU/内存选择器移至了 MEM 段），添加新功能（VGA，键盘，Flash，控制冒险）等，对已经写过的代码基本没有影响。由于一开始考虑到了各种扩展，所以将各部分都设计为通用性较强、模块化较强，大大减少了后期写扩展部分的麻烦。

- 统一化编址的好处

统一化编址之后，数据旁路的判断条件和回路设计都大大简化，而且在 ALU 阶段的数据选择和布线复杂度都减少了很多，同时控制器对外的控制信号也可以完全通过统一化编址代替复杂的选择信号，从各种角度上减轻了编程负担。

◆ 负边沿

一开始全部的寄存器都采用的是正边沿写入，在内存控制单元并不复杂的时候还能勉强正常工作，但当内存时序变复杂时，我们发现 CPU 运转不正常。通过 VGA-Debuger 查看寄存器堆中数值时，我们发现即使 ALU 计算结果正确，寄存器堆中写入的数据也有错误，且错误值不是任何一种计算结果。经过反复检查后发现是由于写入数据是在上升沿时才写入到 MEM_WB 流水寄存器中，而此时寄存器堆就执行写入的话会导致数据没有准备好，把寄存器的实现改为负边沿写入之后问题解决。

后来调试跳转的时候又反复出现问题，后来发现时由于 TControl 内的寄存器并没有使用标准寄存器，而是自行实现的一位寄存器，写入仍为正边沿，所以导致问题。这让我意识到了同一功能的组件应该都实例化自同一个元件。

◆ Xilinx 原理图

Xilinx 从原理图生成 VHDL 文件的功能非常好用，但是原理图的绘制及修改都极为麻烦，在绘制原理图之后再修改接口往往会导致原理图中的元件结构和连线一团乱，最好能在画原理图之前就确定好所有元件的接口。

在使用 IO_Marker 时，由于会对总线的内部名称进行修改，若日后还需要将此 IO 连接到别的总线上，最好能加一层缓冲以简化重连线操作。原理图中的 Logger、DebugInputPort 及已经被删除掉的 VGA_Debuger 就是为了这个目的而存在。

◆ GIT 的使用

通过使用 git 控制版本，多次帮助我们从错误的修改中返回到正确的旧版本上。最好能养成一个习惯，实现确定好实现功能的若干步骤，每达到一个步骤的目标时 commit 一次，保证在下一个步骤修改错误时能及时的恢复。

◆ 良好的分工

由于 CPU 数据通路的复杂性，调试阶段两个人一起调试的工作效率并不会比一个人高太多，所以在整个 CPU 写完之后，我们组内两个人就分为了 CPU 和外设两个工作方向，许欣然负责 CPU 的正常工作和 Flash 自启动，邹林希负责完成 VGA 和键盘模块的尝试、包装工作，最后当 CPU 可以正常工作后，两个外设也已经包装完毕，只花了很短的时间就融合到 CPU 之中了。

虽然准确时间没法测试，不过总共熬夜了约一星期，大概在 100 小时内就完成了包含了这么多扩展的原因很大一部分是因为两个人都可以高效的工作，而不是一直在一起纠结调试问题。

◆ 管脚勘误

实验教程上的 VGA 行场同步管脚给反了，导致显示不正常，经过几个小时的调试后使用万用表测出，特此勘误：

名称	管脚	数据流向
VGA_HHYNC	D6	OUT
VGA_VHYNC	E6	OUT

12 注

代码仍然保留了调试部分，运行时需要将拨码管的第五位置为 1 才可以正常运行，否则为手动时钟。