|  |
| --- |
| 清华大学 计算机科学与技术系 |
| 计算机组成原理大作业 |
| 实验报告 |

|  |
| --- |
| 计34 何钦尧 2012010548  计34 王凯 2013XXXXXX  计34 黄科 2012012231  2015-12-10 |

目录

[1. 概述 3](#_Toc437628563)

[2. 系统整体设计 3](#_Toc437628564)

[2.1. CPU 4](#_Toc437628565)

[2.2. IOBridge 5](#_Toc437628566)

[3. CPU具体设计 5](#_Toc437628567)

[3.1. 概览 5](#_Toc437628568)

[3.1.1. 寄存器统一编址 5](#_Toc437628569)

[3.1.2. 部分组件独立译码 6](#_Toc437628570)

[3.2. CPU控制信号设计 6](#_Toc437628571)

[3.2.1. 控制信号一览 7](#_Toc437628572)

[3.2.2. 控制信号的实现 8](#_Toc437628573)

[3.3. 各主要组件实现详解 10](#_Toc437628574)

[3.3.1. ALU 10](#_Toc437628575)

[3.3.2. ALUController 10](#_Toc437628576)

[3.3.3. BranchSelector 10](#_Toc437628577)

[3.3.4. Controller 10](#_Toc437628578)

[3.3.5. Extender 10](#_Toc437628579)

[3.3.6. ForwardUnit 10](#_Toc437628580)

[3.3.7. HazardUnit 10](#_Toc437628581)

[3.3.8. Register 10](#_Toc437628582)

[3.3.9. StallUnit 10](#_Toc437628583)

[3.3.10. TSelector 10](#_Toc437628584)

[4. IOBridge具体设计 10](#_Toc437628585)

[5. VGAdapter具体设计 10](#_Toc437628586)

[6. KeyboardAdapter具体设计 10](#_Toc437628587)

[7. 写字板软件实现 10](#_Toc437628588)

[8. 最终成果展示 10](#_Toc437628589)

[9. 实验心得与体会 12](#_Toc437628590)

[9.1. 良好的设计的重要性 12](#_Toc437628591)

# 概述

在本实验中，我们在THINPAD教学计算机的硬件平台上，实现了能够执行THCO-MIPS指令集（MIPS16e的改编实现）的CPU。本组实现的指令为30条，包括25条基础指令和5条扩展指令。

CPU支持指令流水，能够以的主频运行。妥善的处理了流水线中可能有的各种结构冲突，数据冲突，控制冲突。

能够从Flash中加载程序到内存当中，完成自启动。支持串口读写，从PS/2键盘中输入数据，以及分辨率为的VGA外接显示。

此外实现了一个写字板的软件程序用于演示键盘和VGA的功能。实现了字符从键盘中的输入，回车换行，通过上下左右键移动光标等功能。

# 系统整体设计

系统的总体设计分为几个方面。一个是CPU；一个是CPU与各种IO外设（包括内存）交互的一个总控单元（因为所有对外设的访问都通过对某个特定的内存地址的访问来进行），这个有点类似于一般个人电脑的硬件结构中的北桥芯片。另外就是其他的外设，如VGA控制单元，键盘控制单元，Flash控制单元等。

各个组件在VHDL程序中的命名为：

* CPU
* IOBridge
* VGAdapter
* KeyboardAdapter
* FlashAdapter

在后文中，我们也将直接使用这些名字。

CPU本身只直接连接IOBridge（这里我们的“北桥芯片”），由IOBridge来完成对包括内存的其他外设的访问。同时，CPU所用的时钟信号是由IOBridge产生的。这样的实现方式是由于，IOBridge控制内存的访问，需要一个状态机来控制。而这个状态机的状态变换必须要和CPU的操作（也即时钟上升沿产生的时机）保持一个同步，这样才能保证CPU的执行中在需要的时候获取到了需要的信息。

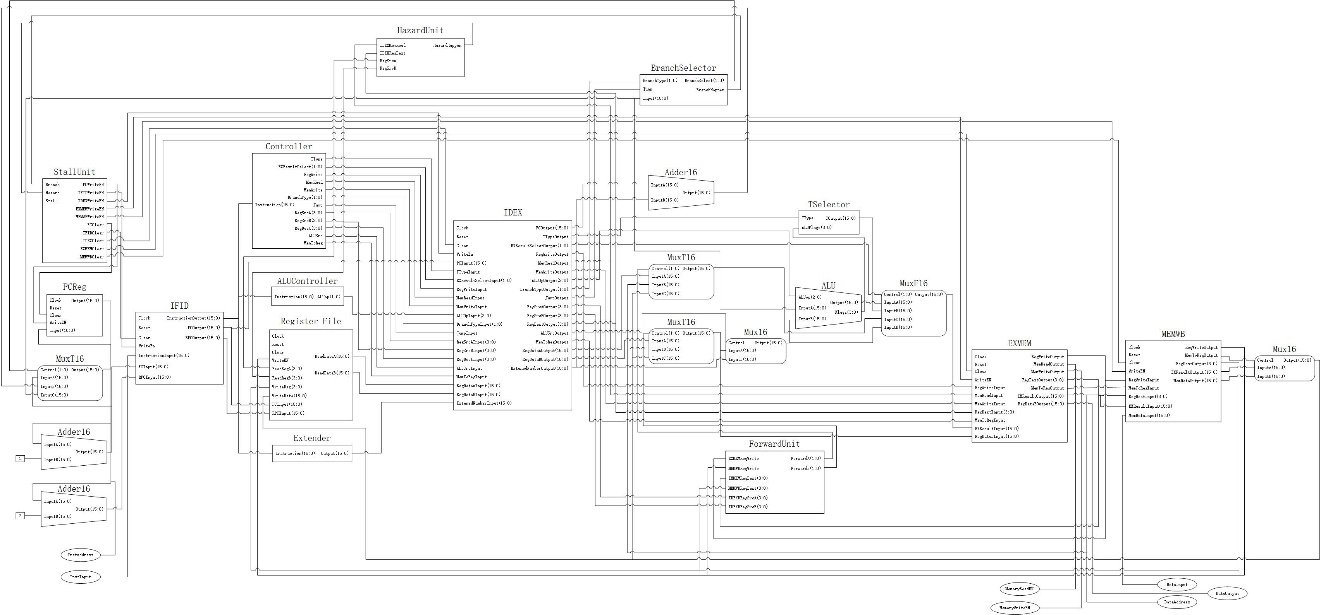
于是CPU在这里仅仅完成指令执行的流水线，而不含外设的访问逻辑。这样使得CPU的整个设计比较规整，易于实现和优化。

由于监控程序的实际需求（写指令内存），我们这里不将指令和数据内存分离，统一都放置在一块内存上。这里我们使用了Ram2，而没有使用Ram1。

## CPU

CPU采用MIPS通用的五段流水线的实现方式，五段流水线分别实现，IF(Instruction Fetch)，ID(Instruction Decode)，EXE(Execution)，MEM(Memory)，WB(Write Back)的功能。

以下为详细的CPU设计数据通路图：



也可以查看该图的源文件：datapath.vsdx, datapath.jpg。在图中，椭圆形标注的信号为外部的输入或者输出。另外所有组件的Clock（时钟信号）和Reset（复位信号）都接自同一来源，因此为了清晰没有在图中画出。

图中所有的矩形或者梯形框为组成CPU的各个组件（具体的也对应到实现中的一个VHDL entity）。约定所有的元件，左边标注的为输入，右边标注的为输出。图中所有信号的标注名称，都是在实现中VHDL内部使用的名称。不引起混淆的情况下，下文中也将使用这些名称。

CPU作为一个整体的元件，其输入输出信号有：

* Clock，输入，时钟信号
* Reset，输入，复位信号
* InstAddress，输出，指令地址
* InstInput，输入，取得的指令的输入
* DataAddress，输出，数据地址
* DataInput，输入，访存获得的内存的输入
* DataOutput，输出，需要往内存中写的数据
* MemoryReadEN，输出，读使能。仅用于控制数据的读。
* MemoryWriteEN，输出，写使能。仅用于控制数据的写。

以上的所有输入输出信号都和IOBridge相连接。也可以认为CPU仅直接和IOBridge通信。在只考虑CPU时，可以认为IF和MEM阶段的内存访问都是完全的组合逻辑操作。因而在CPU中，只有各寄存器用到了时序逻辑。

CPU使用五级流水线的实现，并且处理了流水线中的各种冲突。使用旁路单元（ForwardUnit）和冒险检测单元（HazardUnit）处理了数据冲突。也用StallUnit和BranchSelector的配合处理了跳转控制冲突。而IF和MEM段同时需要访问内存的结构冲突，留给IOBridge进行处理。

## IOBridge

IOBridge这里设计为几乎是整个计算机系统的最中央的控制单元，控制给CPU的时钟信号的发生，以及与各种IO设备的交互。

由于内存读写的时序的复杂性，以及指令和数据同时访存的结构冲突，IOBridge这里必须要运行在比CPU主频更高的频率上。

IOBridge运行一个状态机模型。在系统加电（或者Reset之后），首先从Flash中读取运行的程序并加载到内存当中，这一过程完成之后，开始给CPU产生时钟信号，开始运行。

通过对内存地址进行划分的方式，来处理不同外设的访问。0xBF00设定为串口的读写地址，0xBF01为串口状态信息地址；0xBF02为键盘数据地址，0xBF03为键盘输入状态信息地址；0xF800到0xFFFF划分出来作为显存的地址。对这些特殊地址进行的Load或Store指令就可以完成对IO外设的访问。

# CPU具体设计

## 概览

概览中将给出一些设计上的细微的处理之处。

### 寄存器统一编址

在MIPS架构当中，由于指令的格式比较规整，所以一般来说，指令中的固定的某几位就是表示的寄存器的编号，这样可以和控制器译码并行的，直接取出指令中的某几位，送给寄存器堆用来取出相应的操作数。但是我们在研究了这套指令集的格式之后发现，其指令并没有通常所说的MIPS32那样的规整，目的寄存器编号可能出现在3个寄存器位置中的任何一个，源寄存器的编号出现的位置也很不规整，并不能套用MIPS32通常使用的rs，rt，rd这样的做法（这种做法里面，源寄存器总是rs和rt，目的寄存器可能是rt或者rd）。

以及指令集中存在着很多操作特殊寄存器的指令，特殊寄存器包括：PC，RPC，IH，RA，T，SP。而操作特殊寄存器的指令的格式也不是那么的规整，我们研究认为并不能通过指令中的某几位来对这些特殊寄存器给出一个唯一的编号。

于是结合上面两个信息，我们决定对寄存器的编码采取另外的特殊手段。将通用寄存器以及特殊寄存器统一编码，将寄存器号扩展到4位。最高位为0代表通用寄存器，最高位为1代表特殊寄存器（此时后三位就是正常的通用寄存器的编码）。特殊寄存器的编码如下：

* 0000，特殊的零寄存器
* 0001，PC寄存器
* 0010，SP寄存器
* 0011，IH寄存器
* 0100，RA寄存器
* 0101，T寄存器
* 0110，RPC寄存器

由这些编码就可以直接在寄存器堆中取得相应的寄存器的值。

同时，为了处理源和目的寄存器号的位置的复杂性，我们选择先让Controller进行指令的译码，得到了该条指令对应的源和目的寄存器号了之后，再接到寄存器堆中获取数据。使用这样的方法而不是使用取数后再进行多路选择的办法更加的清晰一点，控制器的逻辑的复杂度增加还在可控的范围之内，译码后再取寄存器的串行执行的方法，比原来的并行其实也没有增加太多的等待时间。

### 部分组件独立译码

与寄存器号相似的，立即数扩展和ALU操作的不同方式的识别，在这里也有很大的问题，指令非常的不规整。

立即数的拓展有各种不同的位数（11位，8位，5位，4位，3位）以及有符号，无符号拓展。一种可行的方式是，让Controller产生一个扩展类型选择的信号，然后 对不同的扩展类型的结果进行多路选择。这样会导致设计变得更加的复杂，包括Controller以增加的若干不同的扩展单元。于是我们这里选择让立即数扩展的单元自己处理指令的译码，然后自行判断不同的情况的扩展，并输出结果。这样做减轻了Controller的实现复杂度。

同样的对ALU我们也采取了这样的处理。指令中不同的ALU操作总共有6种。使用独立的ALUController，接收16位的指令，然后产生三位的ALUOp控制信号。

这样的独立译码处理让Controller只处理最必要的流水线控制逻辑相关的信号，复杂度上更低。更细致的模块划分也让分模块测试变得更加的容易。

## CPU控制信号设计

CPU的各种状态的控制信号由Controller产生，通过各级阶段寄存器逐级向后传播。Controller输入指令，产生若干的控制信号用于控制流水线接下来的操作。下面列出由Controller产生的控制信号以及其作用。

### 控制信号一览

#### TType

用于处理CMP和SLT这两条需要写T寄存器的指令，选择两个操作数在相等和大于的情况下T等于1还是0。

TType=0表示CMP指令，即当两个操作数不相等时T为1。

TType=1表示SLT指令，即当x小于y时为1。

#### EXResultSelect(1:0)

选择EXE级的执行结果来自于ALU还是来自某一个寄存器，或是T的结果。可以用于完成move类指令。用于驱动一个四路选择器。

* 00 选择来自ALU的计算结果
* 01 选择来自T的计算结果
* 10 选择来自RegDataA
* 11 选择来自RegDataB（或者立即数的扩展，这个在之前的选择器进行了选择）

#### RegWrite

等于1时表示需要写一个寄存器，用于在WB阶段控制寄存器组。

#### MemRead

等于1表示需要读内存。这个可以用于做流水线数据冲突时候，需要暂停流水线的判断。具体见HazardUnit。此外MemRead信号也需要传给IOBridge用于数据总线控制。

#### MemWrite

等于1表示需要写内存。

#### BranchType(1:0)

分支类型。有四种取值：

* 00 表示不跳转
* 01 表示总是跳转
* 10 表示等于0跳转
* 11 表示不等于0跳转

#### Jump

等于1标识是跳转（J型）指令。

#### RegSrcA(3:0)

第一个源寄存器的编号。

#### RegSrcB(3:0)

第二个源寄存器的编号。

#### RegDest(3:0)

目的寄存器编号。

#### ALUSrc

用于选择ALU的第二个操作数来自寄存器还是立即数。0代表寄存器B，1代表立即数。

#### MemToReg

等于1表示写会寄存器的值是从内存中读取的（load类指令）。

### 控制信号的实现

下表给出各条指令的每个控制信号。具体详见指令信号.xlsx。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 指令码 | 简化 | RegSrc1 | RegSrc2 | RegDest | ExtendType | ALUSrc | MemToReg | BranchType | Jump | ALUop | Ttype | RegWrite | MemWrite | MemRead | EXEResultSelect |
| ADDIU | 01001 | first5 = "01001" | '1'&(10:8) |  | '1'&(10:8) | 100 | 1 | 0 |  |  | 000 |  | 1 | 0 | 0 | 00 |
| ADDIU3 | 01000 | first5 = "01000" | '1'&(10:8) |  | '1'&(7:5) | 101 | 1 | 0 |  |  | 000 |  | 1 | 0 | 0 | 00 |
| ADDSP | 01100011 | first8 = "01100011" | 0010 |  | 0010 | 100 | 1 | 0 |  |  | 000 |  | 1 | 0 | 0 | 00 |
| ADDU | 11100-01 | (first5 = "11100" and last2 = "01") | '1'&(10:8) | '1'&(7:5) | '1'&(4:2) |  | 0 | 0 |  |  | 000 |  | 1 | 0 | 0 | 00 |
| AND | 11101-01100 | (first5 = "11101" and last5 = "01100") | '1'&(10:8) | '1'&(7:5) | '1'&(10:8) |  | 0 | 0 |  |  | 010 |  | 1 | 0 | 0 | 00 |
| B | 00010 | first5 = "00010" |  |  |  | 110 |  |  | 01 | 0 | - |  | 0 | 0 | 0 | 00 |
| BEQZ | 00100 | first5 = "00100" | '1'&(10:8) |  |  | 110 |  |  | 10 | 0 | - |  | 0 | 0 | 0 | 00 |
| BNEZ | 00101 | first5 = "00101" | '1'&(10:8) |  |  | 110 |  |  | 11 | 0 | - |  | 0 | 0 | 0 | 00 |
| BTEQZ | 01100000 | last8 = "01100000" | 0101 |  |  | 100 |  |  | 10 | 0 | - |  | 0 | 0 | 0 | 00 |
| BTNEZ | 01100001 | last8 = "01100001" | 0101 |  |  | 100 |  |  | 11 | 0 | - |  | 0 | 0 | 0 | 00 |
| CMP | 11101-01010 | (first5 = "11101" and last5 = "01010") | '1'&(10:8) | '1'&(7:5) | 0101 |  | 0 | 0 |  |  | 001 | 0 | 1 | 0 | 0 | 01 |
| JR | 11101-00000000 | (first5 = "11101" and last8 = "00000000") | '1'&(10:8) |  |  |  |  |  |  | 1 | - |  | 0 | 0 | 0 | 00 |
| JALR | 11101-11000000 | (first5 = "11101" and last8 = "11000000") | '1'&(10:8) | 0110 | 0100 |  | 0 | 0 |  | 1 | - |  | 1 | 0 | 0 | 11 |
| JRRA | 1110100000100000 | Instruction = "1110100000100000" | 0100 |  |  |  |  |  |  | 1 | - |  | 0 | 0 | 0 | 00 |
| LI | 01101 | first5 = "01101" |  |  | '1'&(10:8) | 001 | 1 | 0 |  |  | - |  | 1 | 0 | 0 | 11 |
| LW | 10011 | first5 = "10011" | '1'&(10:8) |  | '1'&(7:5) | 111 | 1 | 1 |  |  | 000 |  | 1 | 0 | 1 | 00 |
| LW\_SP | 10010 | first5 = "10010" | 0010 |  | '1'&(10:8) | 100 | 1 | 1 |  |  | 000 |  | 1 | 0 | 1 | 00 |
| MFIH | 11110-00000000 | (first5 = "11110" and last8 = "00000000") |  | 0011 | '1'&(10:8) |  | 0 | 0 |  |  | - |  | 1 | 0 | 0 | 11 |
| MFPC | 11101-01000000 | (first5 = "11110" and last8 = "01000000") |  | 0001 | '1'&(10:8) |  | 0 | 0 |  |  | - |  | 1 | 0 | 0 | 11 |
| MOVE | 01111-00000 | (first5 = "11110" and last5 = "00000") |  | '1'&(7:5) | '1'&(10:8) |  | 0 | 0 |  |  | - |  | 1 | 0 | 0 | 11 |
| MTIH | 11110-00000001 | (first5 = "11110" and last8 = "00000001") |  | '1'&(10:8) | 0011 |  | 0 | 0 |  |  | - |  | 1 | 0 | 0 | 11 |
| MTSP | 01100100-00000 | (first8 = "01100100" and last5 = "00000") |  | '1'&(7:5) | 0010 |  | 0 | 0 |  |  | - |  | 1 | 0 | 0 | 11 |
| NOP | 0000100000000000 | Instruction = "0000100000000000" |  |  |  |  |  |  |  |  | - |  | 0 | 0 | 0 | 00 |
| OR | 11101-01101 | (first5 = "11101" and last5 = "01101") | '1'&(10:8) | '1'&(7:5) | '1'&(10:8) |  | 0 | 0 |  |  | 101 |  | 1 | 0 | 0 | 00 |
| SLL | 00110-00 | (first5 = "00110" and last2 = "00") | '1'&(7:5) |  | '1'&(10:8) | 010 | 1 | 0 |  |  | 011 |  | 1 | 0 | 0 | 00 |
| SLT | 11101-00010 | (first5 = "11101" and last5 = "00010") | '1'&(10:8) | '1'&(7:5) | 0101 |  | 0 | 0 |  |  | 001 | 1 | 1 | 0 | 0 | 01 |
| SRA | 00110-11 | (first5 = "00110" and last2 = "11") | '1'&(7:5) |  | '1'&(10:8) | 010 | 1 | 0 |  |  | 100 |  | 1 | 0 | 0 | 00 |
| SUBU | 11100-11 | (first5 = "11100" and last2 = "11") | '1'&(10:8) | '1'&(7:5) | '1'&(4:2) |  | 0 | 0 |  |  | 001 |  | 1 | 0 | 0 | 00 |
| SW | 11011 | first5 = "11011" | '1'&(10:8) | '1'&(7:5) |  | 111 | 1 |  |  |  | 000 |  | 0 | 1 | 0 | 00 |
| SW\_SP | 11010 | first5 = "11010" | 0010 | '1'&(10:8) |  | 100 | 1 |  |  |  | 000 |  | 0 | 1 | 0 | 00 |

表中仅对写出了有效的控制信号，其他的不写。在当前的设计里面，所有信号的0都表示无效。当所有控制信号全0的时候，该条指令不会产生任何作用（相当于一条NOP指令）。这样给指令的译码和处理冲突时插气泡的动作都带来了更方便，如译码时只需要考虑有意义的情况，其他不需要考虑的统一置为0即可。

## 各主要组件实现详解

### ALU

### ALUController

### BranchSelector

### Controller

### Extender

### ForwardUnit

### HazardUnit

### Register

### StallUnit

### TSelector

# IOBridge具体设计

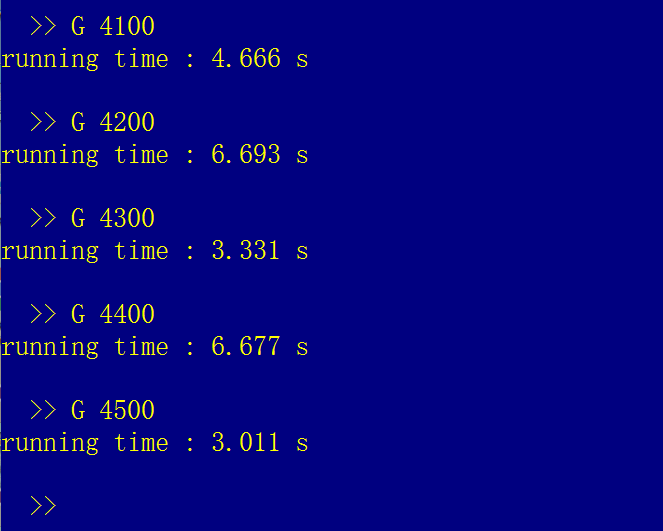
# VGAdapter具体设计

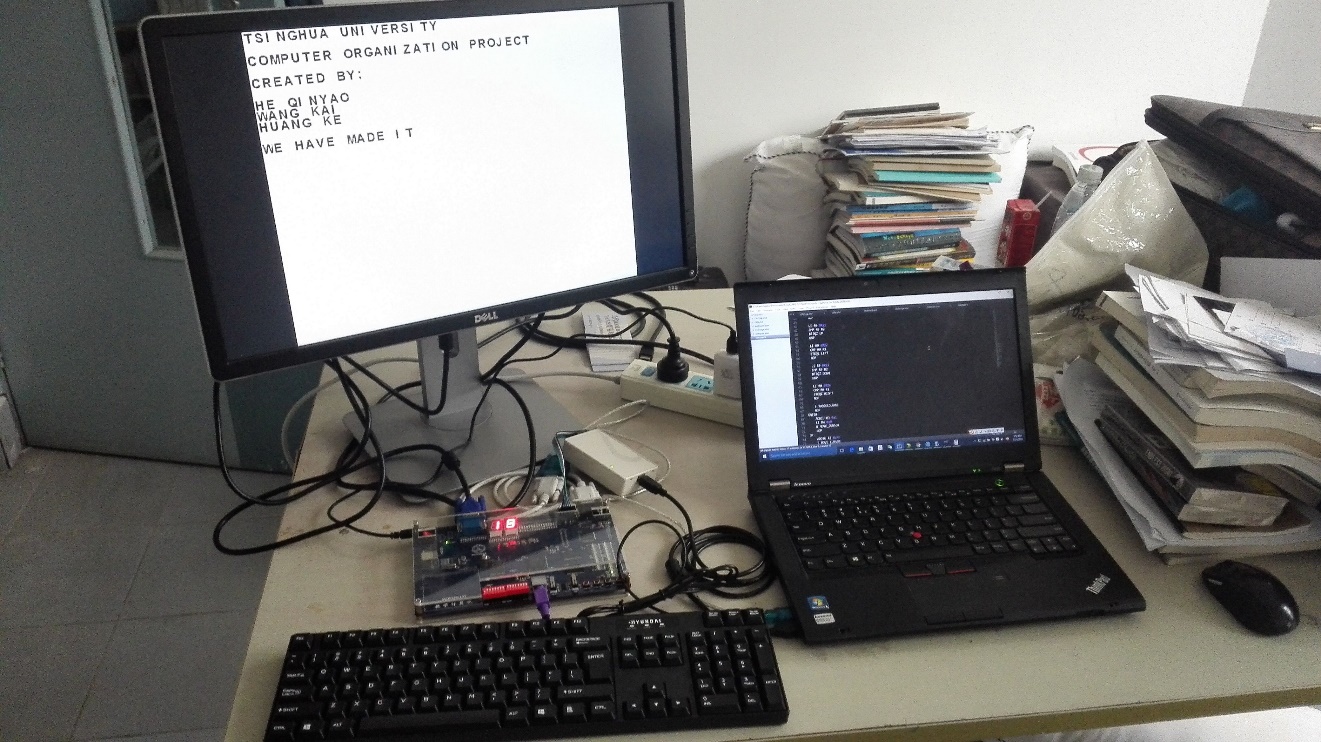
# KeyboardAdapter具体设计

# 写字板软件实现

# 最终成果展示

我们能够最终运行在，最高37.5MHz，最低（在有数据内存访问时）25MHz的可变频率上。下图为运行给出的5个检查使用的程序的结果，运行时间基本符合使用主频计算出来的理论值：





# 实验心得与体会

## 良好的设计的重要性

在“造台计算机”的三个星期的进程当中，我们花了前两个周用来形成一个较为完整，没有问题的设计。