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 2013010617
计 34 黄　科 2012012231

2015-12-10

目录

1. 概述 3
2. 系统整体设计 3
2.1. CPU 4
2.2. IOBridge 5
3. CPU 具体设计 5
3.1. 概览 5
3.1.1. 寄存器统一编址 5
3.1.2. 部分组件独立译码 6
3.1.3. 各信号默认值的处理 7
3.2. CPU 控制信号设计 7
3.2.1. 控制信号一览 7
3.2.2. 控制信号的实现 8
3.3. 各主要组件实现详解 9
3.3.1. ALU 9
3.3.2. ALUController 10
3.3.3. BranchSelector 10
3.3.4. Controller 10
3.3.5. Extender 10
3.3.6. ForwardUnit 10
3.3.7. HazardUnit 11
3.3.8. Register 11
3.3.9. StallUnit 12
3.3.10. TSelector 12
4. IOBridge 具体设计 12
4.1. 时序分析 12
4.2. 进一步优化 13
4.3. 变频 13
4.4. 探索速度的极限 13

4.5. 外设的处理 14
5. VGAdapter 具体设计 14
5.1. 整体设计思路 14
5.2. 设计图 14
5.3. 内部组件详细 15
5.3.1. VGAdapter 15
5.3.2. CharAdapter 15
5.3.3. VGACore 15
5.4. 设计特点 15
5.4.1. 封装性 15
5.4.2. 并行性 15
5.4.3. 扩展性 15
6. KeyboardAdapter 具体设计 15
6.1. KeyboardToAscii 16
6.2. Keyboard 17
7. 写字板软件实现 18
8. 最终成果展示 21
9. 实验心得与体会 24
9.1. 良好的设计的重要性 24
9.2. 分工的重要性 24
9.3. 重视可扩展性 24

1. 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37.5MHz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640×480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。

2. 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。

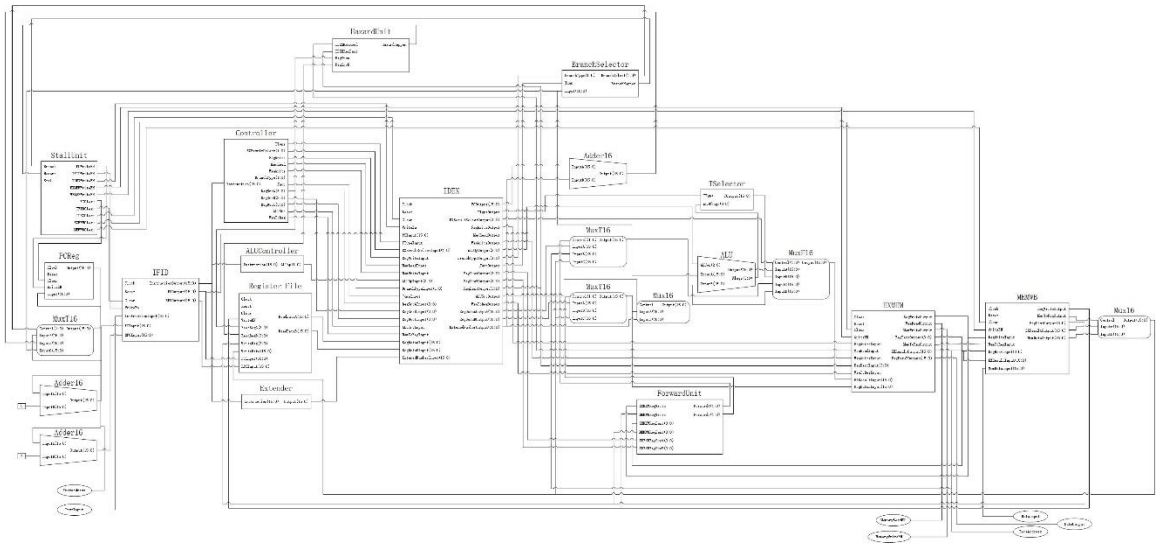
由于监控程序的实际需求（写指令内存），我们这里不将指令和数据内存分离，统一都放置在一块内存上。这里我们使用了 Ram2，而没有使用 Ram1。

开发板上提供的 50MHz 的时钟发生器限制了我们对于更快的运行这台计算机的需求。我们使用了 ISE 提供的时钟管理模块（DCM），来对输入的频率进行各种倍数的分频倍频，来得到一个我们合适的频率，从而使得计算机运行得更快。

2.1. 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。

2.2. 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。

3. CPU 具体设计

3.1. 概览

概览中将给出一些设计上的细微的处理之处。

3.1.1. 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。

3.1.2. 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。

3.1.3. 各信号默认值的处理

每个控制信号的有效都设置为 1, 0 表示无效。这样的设定可以使得在需要将一条指令清为 NOP 的时候更加的方便。所有控制信号都是 0 的话，相当于一条 NOP 指令。

3.2. 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。

3.2.1. 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）。

3.2.2. 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	1110100001000000	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 = "00001000000000000"									-		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。

3.3. 各主要组件实现详解

在各个组件的实现中，我们遵循一个原则，就是尽可能的减少时序电路所占的部分。所以在最后的实现里面，仅有各个寄存器需要用到时钟边沿进行触发，其他都是完全的组合逻辑电路。

3.3.1. ALU

ALU 需要执行钟操作，因此 ALUOp 需要 3 位的操作码编码就足够了。六个操作分别是：

- 加法
- 减法
- 按位与
- 按位或

- 左移
- 算术右移

对于移位操作，为了更好的提高运算的速度，我们没有使用 VHDL 自带的移位相关的函数，而是硬编码自行手动拼接。实际的这样做的效果非常好，ISE 硬件综合报告显示的最高主频由 57MHz 提高到了 73MHz。

3.3.2. ALUController

ALUController 完成对 ALUOp 的译码。

3.3.3. BranchSelector

BranchSelector 用于完成对于跳转的控制操作。根据跳转类型的不同（BranchType(1:0)），可以处理不跳转，等于零跳转，不等于零跳转，和总是跳转。以及和 Jump 指令配合一起，输出一个用于下一个 PC 的数值的三选器的 BranchSelect 控制信号。00 时不跳转，PC 为下一条指令（PC+1），01 时做 B 指令的跳转（即 PC+1+偏移），10 时做 J 指令的跳转（PC 变为 RegSrcA 读出的数值）。

在发生跳转时，都产生一个 BranchHappen 信号，传给 StallUnit，由他来处理相关的流水线的处理（这里为将 IDEX 寄存器清零）。

3.3.4. Controller

Controller 用于产生控制信号，之前已经详细阐述过了。

3.3.5. Extender

Extender 直接对指令译码，产生所需要的立即数的扩展。如果该条指令没有立即数的扩展的话，输出为 0。

3.3.6. ForwardUnit

ForwardUnit 用于实现解决数据冲突的数据旁路。其根据 EXMEM 和 MEMWB 两个阶段寄存器中的数据和 IDEX 中（当前指令）进行判断是否需要进行旁路，然后对两个源操作数产生两个三选器的控制信号。三选择器分别接的是：寄存器直接读出的结果，EXMEM 旁路回来的结果，和 MEMWB 旁路出来的结果。

ForwardUnit 的判断逻辑如下面的示意代码所描述：

```
if (EXMEMRegWrite = 1) and (EXMEMRegDest = IDEXRegSrcA)
    ForwardA = 01
else if (MEMWBRegWrite = 1) and (MEMWBRegDest = IDEXRegSrcA)
        and not((EXMEMRegWrite = 1) and (EXMEMRegDest = IDEXRegSrcA))
    ForwardA = 10
else
    ForwardA = 00
```

ForwardB 的计算类似。

大体上的思路是，如果上一条（或者上两条）指令，需要写寄存器，且要写的寄存器号和当前指令读的寄存器号相同，直接选择上一条的计算结果旁路回来，而不是选择寄存器读出的原始值。

比较微妙的一点是，当上一条和上两条指令同时都满足这一条件的时候，需要选择上一条指令的结果（即 EXMEM），因为这是最新的结果。

3.3.7. HazardUnit

HazardUnit 用于解决，上一条指令是从内存中读取，下一条指令马上用到了这个寄存器的情况。这样在该条指令到了 EX 级的时候，上一条指令才刚刚开始 MEM 级，数据还没有读出来，并不能进行旁路。所以这个时候必须要暂停流水线的运行，等待一个周期，下一个周期的时候数据已经从内存里面读出，就可以正常的进行旁路了。

Hazard 的检测可以在 ID 级进行。当 IDEXMemRead=1 且 IDEXRegDest 等于 RegSrcA 或者 RegSrcB 的任何一个的时候，在 IDEX 段（即当前正在执行 EXE 段）的指令是一条 load 指令，且 load 到的寄存器需要在 ID 阶段执行的指令被使用到。这时候给 StallUnit 发送一个 HazardHappen 信号，让 StallUnit 来处理，将 PCReg 和 IFIDReg 禁止写入，IDEXReg 清零（将该条指令设置为气泡）。大致的逻辑可以用以下的示意代码来描述

```
if (IDEXMemRead = 1)
    and ((IDEXRegDest = RegSrcA) or (IDEXRegDest = RegSrcB))
    HazardHappen = 1
else
    HazardHappen = 0
```

3.3.8. Register

寄存器是 CPU 设计中唯一需要时序逻辑的单元。寄存器包含有 PC，各阶段寄存器，寄存器堆（RegisterFile）。他们都用基本统一的方法来实现，所以统一在这里描述。

所有的寄存器，除了数据的输入输出之外，还有 4 个主要的控制输入：

- 时钟信号 Clock
- 异步清零信号 Reset
- 同步清零信号 Clear
- 写使能信号 WriteEN

异步清零信号全部用于接 THINPAD 上的 Reset 按钮。除了寄存器堆以外，其他的写使能信号和同步清零信号都由 StallUnit 控制。写使能信号用于暂停流水线，同步清零信号用于将某条指令清为 NOP（这里就是将所有的控制信号清零）。

插 NOP 必须要使用同步清零而不是异步的方式。这是由于，使用异步清零的话，在下一个时钟沿到来之前就改变了寄存器的值，进而影响到该段流水的组合逻辑的输出结果，这样就改变了不该改变的周期的结果。而使用同步清零，在时钟沿到来的时候，才进

行清零，相当于就是输入端接的全部是零，也就是可以认为下一个周期进来的是一条 NOP 指令。

所有的阶段寄存器都使用上升沿触发，而寄存器堆则使用下降沿触发。这是因为上升沿之后，需要一段时间，需要的数据才会从 MEMWB 寄存器回到寄存器堆，这时候才能执行写入的操作，并且让当前正处于 ID 阶段的指令能够正确的读出写回的数据。

3.3.9. StallUnit

StallUnit 设计用来处理流水线的启动和停止，以及插气泡相关的工作。有三个输入：Branch，Hazard，Stall。Branch 的时候需要将 IFID 清零（这也就是延迟槽后的一条指令，不能让他执行）。Hazard 的时候需要将 PCWrite 和 IFIDWrite 置为 0，将 IDEX 清零。Stall 的时候将所有的 WriteEN 都置为 0，这样也就是完全停住了流水线。

3.3.10. TSelector

TSelector 主要用于计算出 T 的值。输入参数包括 Controller 生成的 TSelect 信号（T 的类型），以及 ALU 计算得到的状态码（ZF，NF，OF，CF）。输出为计算得到的 T 的结果，将在后面接入一个四路选择器。

4. IOBridge 具体设计

由于采取了单块内存的设计，IOBridge 最重要的任务就是要处理指令和数据同时访问的冲突问题。我们采取通用的做法，让内存控制单元运行在更高的频率之上，这样 CPU 的周期对应着多个内存的周期，于是就可以完成两组数据（指令和数据）的读写。

由于内存的读写是有速度的上限的，尽可能的减少 CPU 时钟需要对应的内存时钟的数量，能够使得 CPU 运行得更快。

4.1. 时序分析

在我们以上的 CPU 的设计中我们可以看出，CPU 内部应该认为自己是在并行的同时对指令和数据进行读写操作。而在内存控制器这里实际的控制时，实际上是分时的对内存的数据和地址总线进行占用。

由于 CPU 上层需要同时的读出指令和数据，因此不能直接将内存的地址线接入到 CPU 的 InstInput 和 DataInput 上，而要在不同的时钟周期读出不同的数据，并用两个寄存器锁存住，然后两个寄存器再接输出到 CPU 的 InstInput 和 DataInput。

而内存要求在写入信号 WE 被拉低之前，需要事先准备好数据和地址。因此我们还需要一个单独的周期用来做数据准备。由于 load 和 store 类指令不可能同时存在，所以不可能需要同时做数据的写入和读取，于是写入和读取可以公用一个时钟周期。

于是我们可以划分一个四状态的模型，就可以良好的处理数据和指令的冲突。 示意图如下：

这样，在 PRE 状态的阶段，数据和地址从上一个阶段寄存器（EXMEM）中出来，下在一个时钟上升沿来临之前，到达内存总线上，这样就形成了数据建立时间。 DATA 状态的时候，根据读和写的不同，将 OE 或者 WE 控制信号拉低，读出或者写入数据。在 DATA 状态后的时钟上升沿，读出的数据被锁存在一个寄存器中。然后 INS 状态，内存的地址总线切换到指令的地址，拉低 OE，读出指令，INS 之后的上升沿将读出的指令写入寄存器。HOLD 状态是为了保证，从寄存器中出来的数据，在下一个时钟上升沿之前，到达下一个阶段寄存器（MEMWB）。

这样，CPU 运行在内存频率的四分之一上。

4.2. 进一步优化

在上面的设计当中，我们使用了两个寄存器来做锁存，因此需要额外的一个 HOLD 状态来等待数据的向下传播。

而我们可以用一种别的实现方式来省略掉这个 HOLD 状态。如果我们不使用两个寄存器，而是直接将 Inst 的输出接到内存的数据总线上，那么在 INS 阶段，我们直接读出的数据就可以直接被传输到 CPU 中，MEMWB 阶段寄存器上。这样我们就从四分频变成了三分频。

4.3. 变频

我们注意到，不是所有的指令都需要有内存访问的操作（除了读指令以外），那么我们现在的三个状态也还有优化的余地。是否需要访问内存可以通过 MemoryWriteEN 和 MemoryReadEN 来判断。如果不需要内存访问的话，我们可以在状态转移的时候，做一些变化，从 PRE 状态直接切换到 INS 状态。

由于 CPU 其实使用时钟信号的上升沿进行触发，所以即使是每个上升沿的时间间隔不同，对于 CPU 来说也是无所谓的。所以在这里我们使用这样的变频的方法也是可以运行的。我们只要在 IOBridge 中控制，当状态为 PRE 和 DATA 是，CPU 的时钟信号为高电平，INS 时为低电平，这样不仅可以实现变频，还可以确保内存和 CPU 运行的时钟的同步。

这样整个系统在没有访存指令时，为二分频，在有访存指令时为三分频。更进一步的提高了系统的运行速度。

4.4. 探索速度的极限

我们在上述的 2-3 变频的架构下，使用 ISE 的 DCM 模块产生不同的时钟频率，对整机进行测试。在我们的测试中，75MHz 的输入频率（对应 CPU 的最高主频为 37.5MHz）下可以比较稳定的运行，77MHz 下出现了一些内存读取数据的错误，但是仍然能够运行监控程序。

根据我们的分析，在更高的频率下，取出的指令有误，因此我们认为，该硬件上的内存能够支持的，给出地址之后，最快的能得到读出的数据并写入下一级寄存器，的频率约在 80MHz 左右。而在我们另外的架构尝试，使用单周期读取指令的情况下，只能运行到

38MHz 左右。也就是说，由于地址和数据从上一级寄存器中出来到稳定需要一段时间，所以从寄存器到寄存器的内存读取的最快速度在 40MHz 左右。上面的架构里面之所以可以在 70+MHz 的频率下，仍然能够读出数据，是因为之前的数据准备阶段，地址和数据已经从寄存器中出来并稳定，在地址已经给好了的情况下，内存的读取最快的速度可以达到 70+MHz。

上面的分频结构中的二分频时，实际上从寄存器到寄存器的全段的时间仍然在 38MHz 左右，也实际上符合我们上面的分析。

所以由于从时钟上升沿开始，到地址已经稳定到可以用来读取内存需要一定的时间，而我们上面的 2-3 分频的结构里面，由于实际上指令和数据的地址都用 PRE 的那一段时间就已经准备稳定，所以事实上节省了一些时间。可以认为这样的结构是在目前的硬件条件下最优的结果。如果使用 38MHz，然后在访存的时候二分频，这样最低的速度实际上会变慢。而我们这里可以达到最高 37.5MHz，最低 25MHz 的运行频率。这基本上是这个硬件系统上能够达到的极限速度了。

由于我们知道了系统速度的瓶颈是在寄存器的数据建立时间，所以如果用别的方法可以绕过寄存器而直接给出地址，就可以达到更高的速度。但是这种实现方式太不常规，对于各种时序的要求也更加的严格，更加的难以控制。我们在这里没有尝试这种实现。

4.5. 外设的处理

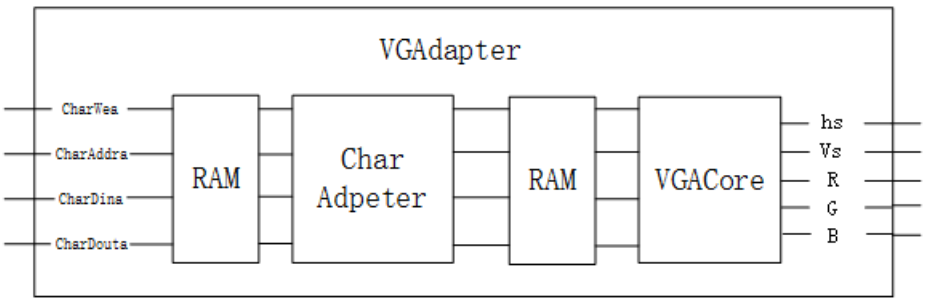
外设的处理都是通过与特定的内存地址的交互实现的。如 0xBF00 被映射为串口，0xF800 到 0xFFFF 被映射为显存。只需要在这里对这些特定的内存地址的操作，直接映射到相应外设的操作就可以了。

5. VGAdapter 具体设计

5.1. 整体设计思路

VGA 部分的设计的主要思路是，分离 VGA 显示信号的控制部分和读取 VGA 显示内容的部分，使得整个 VGA 的输出独立于 CPU 的运行流水，VGA 的控制器对外只表现为一个可以读写的内存段，CPU 通过对于该内存段的内容进行读写，将需要显示的内容传递给 VGA 的控制器，VGA 的控制器并行完成内容的显示工作。

5.2. 设计图



5.3. 内部组件详细

5.3.1. VGAdapter

控制整个 VGA 的运行，其中包含了对于 VGA 控制的全部组件，对 CPU 侧表现为一个内存，提供写使能 CharWea，写入地址 CharAddra，写入内容 CharDina 和读出内容 CharDouta，这四个接口构成一个完整的 Ram，CPU 只需要将需要显示的字符的 ASCII 码写入该内存段即可。

5.3.2. CharAdapter

负责将 CharRam 中的待显示字符的 ASCII 码转换为对应的点阵，并写入 Gram 即显存中供 VGACore 对显存进行读取并输出至屏幕上。对于每个字符的 8 位 ASCII 码，CharAdapter 先按照 ASCII 码读取对应的点阵图，将点阵图中的热内容复制到 Gram 中，等待 VGACore 进行显示。

5.3.3. VGACore

负责将显存中的内容输出至 VGA 中，其中的具体流程是，先按照时钟获取 X、Y 坐标，当 XY 满足需要输出同步信号的范围时，输出对应的同步信号，之后对于每个点的 RGB 值，根据显存中对应的点的值进行输出。

5.4. 设计特点

5.4.1. 封装性

对外只表现为内存，对于 CPU 的使用非常简单，可以在流水中对其显示内容进行更新。

5.4.2. 并行性

内部各组件间以及 VGA 控制器和 CPU 之间均并行，由内存进行数据通信，保证效率，同时简化控制。

5.4.3. 扩展性

易于扩展显示其他功能，对于其他的功能只需要在向 CPU 侧添加其他的数据接口，在 VGAdapter 中添加对应的 Adapter 将 CPU 传来的数据转化为显存中的点阵图即可，VGACore 和 VGAdapter 可以复用，同时可以满足通过软件进行不同显示内容间的切换。

6. KeyboardAdapter 具体设计

KeyboardAdapter 里含有两个元件，分别为 Keyboard 和 KeyboardToAscii 两个元件。KeyboardAdapter 做的作用就是将这两个元件组装在一起，其所有接口如下表所示：

接口名	IN/OUT	类型	作用
PS2Data	in	Std_logic	PS2 键盘读入数据
PS2Clock	in	Std_logic	PS2 键盘时钟信号

Clock	in	Std_logic	50M 时钟
Reset	in	Std_logic	Reset
DataReceive	in	Std_logic	从 MemoryController 处接收到的信号，判断是否可以进行下一次读写
DataReady	out	Std_logic	发出信号，判断键盘输入是否读入
Output	out	Std_logic_vector (7 downto 0)	输出键盘输入的字符相对应的编码

6.1. KeyboardToAscii

为了方便对键盘输入的内容进行显示，KeyboardToAscii 将键盘输入的字符相对应的编码转化为 ascii 码，并对一些 ascii 中不存在的特殊字符，将其对应的编码对应到 ascii 码中不可见的符号部分。输入输出结果均为 8 位编码。

整个对应如下表所示：

所对应的按键	输入的编码（十六进制）	输出的编码（十六进制）
-- A	41	1C
-- B	42	32
-- C	43	21
-- D	44	23
-- E	45	24
-- F	46	2B
-- G	47	34
-- H	48	33
-- I	49	43
-- J	4A	3B
-- K	4B	42
-- L	4C	4B
-- M	4D	3A
-- N	4E	31
-- O	4F	44
-- P	50	4D
-- Q	51	15
-- R	52	2D
-- S	53	1B
-- T	54	2C
-- U	55	3C

-- V	56	2A
-- W	57	1D
-- X	58	22
-- Y	59	35
-- Z	5A	1A
-- 0	30	45
-- 1	31	16
-- 2	32	1E
-- 3	33	26
-- 4	34	25
-- 5	35	2E
-- 6	36	36
-- 7	37	3D
-- 8	38	3E
-- 9	39	46
-- -	2D	4E
-- =	3D	55
-- \	5C	5D
-- BKSP	08	66
-- SPACE	20	29
-- TAB	09	0D
-- CAPS	25	58
-- L SHFT	00	12
-- L CTRL	01	14
-- L ALT	02	11
-- ENTER	12	5A
-- ESC	1B	76
-- [	5B	54
-- U ARROW	21	75
-- L ARROW	22	6B
-- D ARROW	23	72
-- R ARROW	24	74
--]	5D	5B
-- ;	3B	4C
-- '	27	52
-- ,	2C	41
-- .	2E	49
-- /	2F	4A

6.2. Keyboard

Keyboard 实现从 PS2 键盘读入一个按键，并输出按键对应的八位编码，以及输出 DataReady 为 1 时表示这次按键读入已经完成。当输入 DataRecive 为 0 时，将 DataReady 置为 0，则此时可以输入下一次按键。

输入的时钟信号有两个，分别为 50M 时钟和键盘的输入时钟（大约为 11M）。首先通过模拟 D 触发器对输入的键盘时钟信号和内容进行毛刺处理，得到整形后的时钟。根据变形后的时钟读入键盘输入，当按下一次键盘后，键盘的数据信号将会从高电平置为低电平，表示开始。之后的八个键盘时钟周期将会读入八个数据信号，在此之后数据信号将会恢复高电平。

根据这整个流程设计了相应的状态机即可时间 PS2 键盘的输入。

值得注意的是，PS2 键盘输入中，每次按下按键即执行上述的一个过程，得到按键所对应的编码，而松开按键的时候则会执行两次，得到一个终止码的编码（F0）和按键所对应的编码。所以后续实现的按键显示功能，就是根据当键盘数据为终止码时，下一个得到的键盘码数据即为所需要的按键编码。

7. 写字板软件实现

我们实现的打字板软件可以进入字母、数字以及一些标点符号的输入，并在最后显示光标。可以通过方向键调节光标的位置，并修改或者删除光标所在位置的符号。

打字板和显示器的交互方式，是通过读取 BF03 来判断键盘是否可读，然后读取 BF02 即使键盘输入的值，而显示器会显示 F800 后 1200 位内存上的字符（也即使显存），所以只需将键盘输入的字符写入相应的内存中，屏幕就会显示相应的字符。同样的，如果要擦除屏幕上的字符，只需把对应的内存写为 0 即可。同时当光标放置的地方有字符，那么就会显示字符的反色，并不断闪动，这个功能的实现过程是将字符对应的编码取反并写入对应的内存，就可以实现这种功能了。

程序如下：

```
START:
        LI R3 0x0 ;行号
        LI R4 0x0 ;列号

INIT:
        LI              R0 0XF8
        SLL             R0 R0 0X0
        LI              R2 0X04
        SLL             R2 R2 0X0
        ADDIU R2 0X50
        ADDIU R2 0X60
        LI              R3 0X0
INIT_F800:
        SW              R0 R3 0X0
        ADDIU R0 0X1
        ADDIU R2 0XFF
        CMP             R2 R3
        BTNEZ INIT_F800
        NOP

        MFPC R7
        ADDIU R7 0x2
        B PRINT_CURSOR
        NOP
WAIT:   ;判断键盘是否可读
        NOP
        LI R0 0xBF
```

```
        SLL R0 R0 0x0
        LW R0 R1 0x03
        NOP
        BEQZ R1 WAIT
        NOP
LOAD:   ;读取键盘
        LW R0 R2 0x02

        MFPC R7
        ADDIU R7 0X2
        B CLEAR_CURSOR
        NOP

        LI R0 0x12
        CMP R0 R2
        BTEQZ ENTER
        NOP

        LI R0 0X21
        CMP R0 R2
        BTEQZ UP
        NOP

        LI R0 0X22
        CMP R0 R2
        BTEQZ LEFT
        NOP

        LI R0 0X23
        CMP R0 R2
        BTEQZ DOWN
        NOP

        LI R0 0X24
        CMP R0 R2
        BTEQZ RIGHT
        NOP

        B JUDGECOLUMN
        NOP
ENTER:
        ADDIU R3 0x1
        LI R4 0x0
        B MOVE_CURSOR
        NOP
UP:
        ADDIU R3 0xFF
        B MOVE_CURSOR
        NOP
DOWN:
        ADDIU R3 0x1
        B MOVE_CURSOR
        NOP
LEFT:
        ADDIU R4 0xFF
        B MOVE_CURSOR
```

```
        NOP
RIGHT:
        ADDIU R4 0x1
        B MOVE_CURSOR
        NOP
MOVE_CURSOR:
        MFPC R7
        ADDIU R7 0X2
        B PRINT_CURSOR
        NOP
        B WAIT
        NOP
JUDGECOLUMN:        ;判断是否达到行尾
        LI R0 0x28      ;每行40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 0x1E      ;30行
        CMP R0 R3
        BTNEZ PRINT
        NOP
        LI R3 0x0       ;行号清零
PRINT:
        LI R0 0xF8
        SLL R0 R0 0x0   ;显存起始地址
        SLL R1 R3 0x5   ;行号 * 32
        SLL R5 R3 0X3   ;行号 * 8
        ADDU R1 R5 R1
        ADDU R0 R1 R0
        ADDU R0 R4 R0;偏移量 = 行号 * 40 + 列号
        ADDIU R4 0x1    ;列号+1
        SW R0 R2 0x0
        MFPC R7
        ADDIU R7 0X2
        B PRINT_CURSOR
        NOP
        B WAIT
        NOP
PRINT_CURSOR:
        LI R0 0xF8
        SLL R0 R0 0x0   ;显存起始地址
        SLL R1 R3 0x5   ;行号 * 32
        SLL R5 R3 0X3   ;行号 * 8
        ADDU R1 R5 R1
        ADDU R0 R1 R0
        ADDU R0 R4 R0;偏移量 = 行号 * 40 + 列号
        LW R0 R5 0X0
        LI R1 0X0
        SUBU R1 R5 R5
        ADDIU R5 0XFF
```

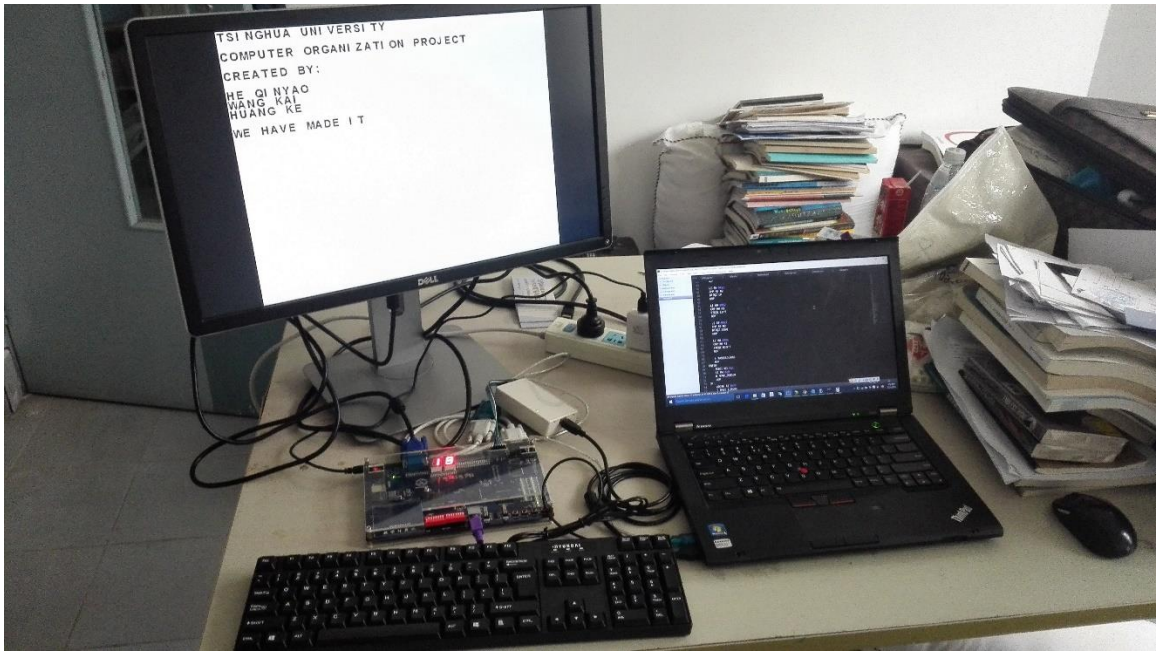
```
        SW R0 R5 0X0
        JR R7
        NOP
CLEAR_CURSOR:
        LI R0 0xF8
        SLL R0 R0 0x0   ;显存起始地址
        SLL R1 R3 0x5   ;行号 * 32
        SLL R5 R3 0X3   ;行号 * 8
        ADDU R1 R5 R1
        ADDU R0 R1 R0
        ADDU R0 R4 R0;偏移量 = 行号 * 40 + 列号
        LW R0 R5 0X0
        LI R1 0X0
        SUBU R1 R5 R5
        ADDIU R5 0XFF
        SW R0 R5 0X0
        JR R7
        NOP

        B WAIT          ;返回起始
        NOP
        JR R7
        NOP
```

8. 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：

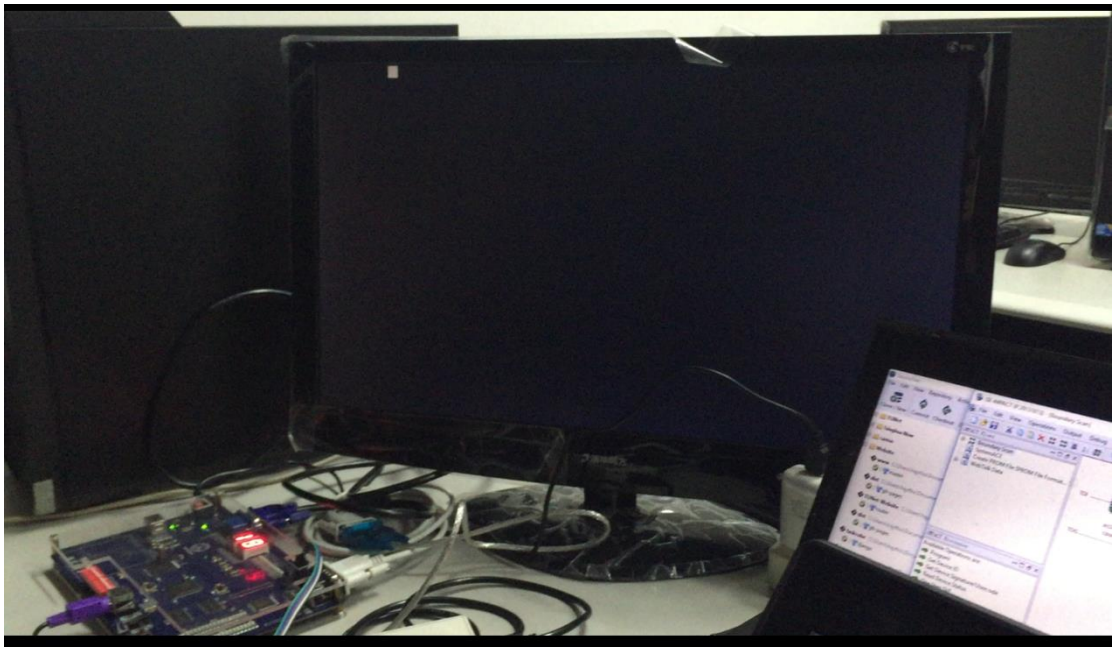
```
  >> G 4100
running time : 4.666 s

  >> G 4200
running time : 6.693 s

  >> G 4300
running time : 3.331 s

  >> G 4400
running time : 6.677 s

  >> G 4500
running time : 3.011 s

  >>
```

下图为我们的系统连接上 VGA 显示器和键盘，运行自己写的写字板程序。

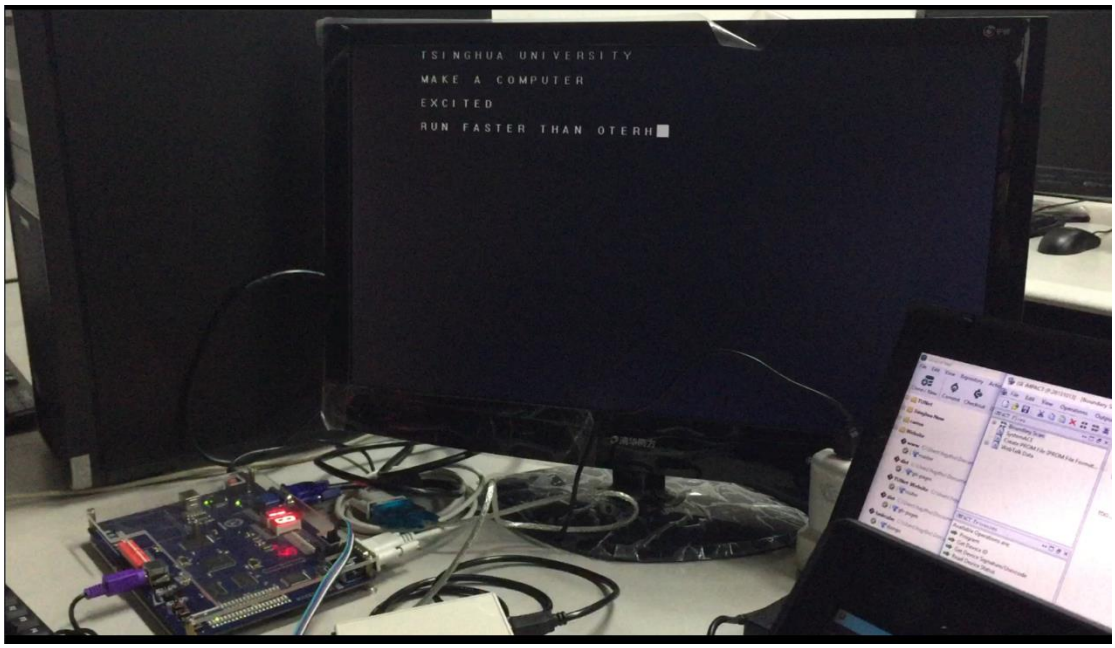


下面是运行打字板程序的照片

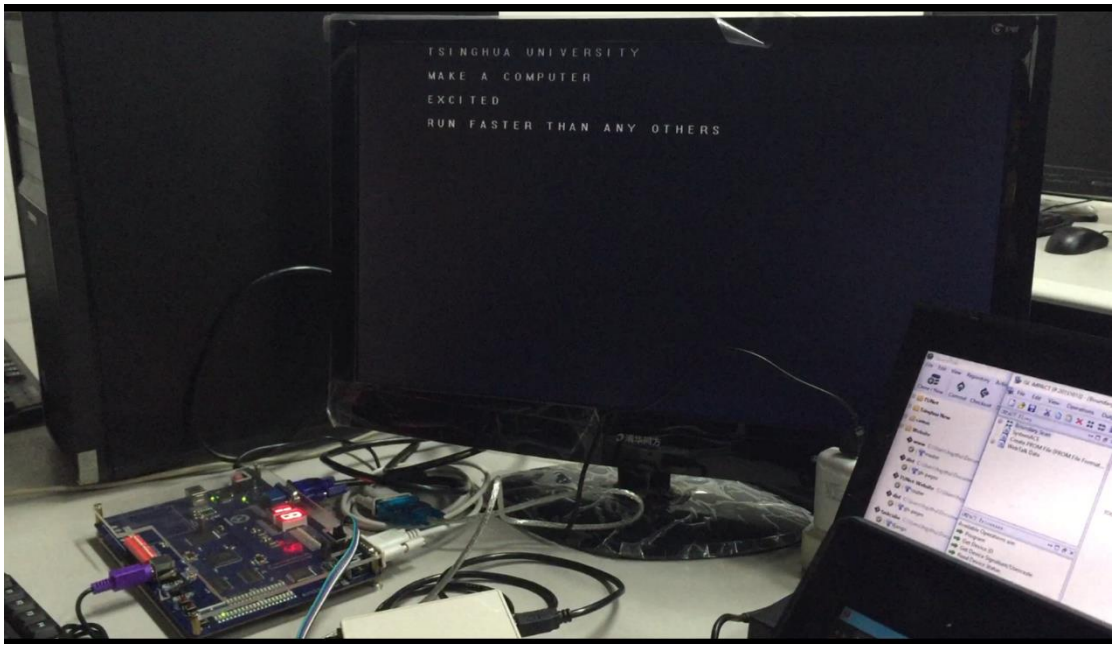
当不进行任何操作，打字板会显示一个不断闪烁的光标



之后打字板可以根据键盘的输入将不同的内容显示在屏幕上



也可以按 Backspace 进行回退，并修改其它输入的内容



9. 实验心得与体会

9.1. 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。我们经过了很长时间的思考，论证，确认这样的设计是没有问题了的以后，再开始编写具体的代码。而我们编写每一个分隔的组件，都会进行测试，确保其功能完全。最后将所有的组件总装到一起来以后，整个系统只经过了不到半个小时的调试就已经完全正常运行，节省了大量的工作。这样归功于之前的设计经过了仔细的论证，而不是随意的开始写，把调试的工作丢给最难调试的硬件。

9.2. 分工的重要性

整个 CPU 的顶层设计全部由一位同学完成，包括设计与验证。这样可以保证这个设计的思路的完整性不被打破。而在确定了设计，开始撰写每个组件的具体功能的文档，之后的编写的任务就可以完全的按照文档来进行，也方便于大家的分工，让没有参与到顶层设计的另外两名同学也能照着文档写出正确的代码。

9.3. 重视可扩展性

对于所有的外设，由于和 CPU 相对独立，都是并行的进行的编写和调试。虽然分立，但是由于一开始就做好了可扩展性的设想，实际编写完之后，只是在 IOBridge 当中做了一个简单的地址映射，就接入到系统当中来了，不管是 VGA 还是键盘的接入都没有遇到任何 bug。VGA 尤其是涉及为与 CPU 并行执行，这样 CPU 的操作和 VGA 完全不会互相干扰。

9.4. 尝试挑战

第一版我们完成了一个四分频，CPU 主频运行在 12.5MHz 的实现。当时使用的是四状态，内存控制器使用下降沿触发的方式。在这一版已经能够成功的运行监控程序之后，我们一直在尝试想要把这个提速得更快。于是去研究了 ISE 提供的 DCM 的使用方法。但是我们用 DCM 将时钟频率提高到了 14MHz 之后，运行就开始出现错误。我们一直在思考，不应该只能跑这么慢的速度，调试了若干天都没有找到原因。直到后来有一天我突然发现，在我的下降沿的实现方式中，数据写入寄存器之后，到达下一个阶段寄存器的时间只有半个周期。在 50MHz 的内存频率下，也就意味着只有 10ns。这 10ns 中需要完成寄存器的数据建立和保持时间，速度再稍微快一点可能就不够了。于是在读取指令的这一环就出现了错误，导致系统运行失效。

于是在思考了之后，调整为了上面所说的，上升沿触发的，四状态设计。这样寄存器的建立保持时间再 50MHz 的时候变为了 20ns，就能跑到了更高的速度。在这个条件下，我们运行到了 80MHz 的内存频率，也就是 20MHz 的 CPU 主频。

然后从这里开始就一发不可收拾，继续思考和探索内存读取的时序优化的可能性。于是有了后来的三分频和变频的实现。而且通过不同的时序配置的观察，发现了寄存器建立时间对内存读取速度这里的影响。

虽然实验的要求说的是运行的速度并不是评分点，但是还是着迷于这种通过自己仔细的研究，理解这里面细微的运行机理，进而对系统进行优化达到极限的性能的事情。这也加深了对于这一领域细节的知识的理解。