

ExcitedCPU 实验报告

——支持 THCO-MIPS 指令集的 CPU

石伟男 2012011367 高越 2012011379 王轩 2012011369

TSINGHUA | 计 25

目录

一. 实验目标	1
二. 数据通路与系统设计	1
2.1 整体设计	1
2.2 数据通路	2
2.3 指令执行流程	3
2.4 扩展功能设计	4
2.4.1 冲突	4
2.4.2 中断	5
2.4.3 VGA 模块	6
2.4.4 PS2 键盘模块	8
2.4.5 监控程序修改	9
2.4.6 Flash 引导	9
三. 代码模块与接口设计	10
四. 成果展示	22
4.1 基本版本展示	22
4.2 VGA 版本展示	24
五. 实验中遇到的问题	26
5.1 栈顶指针初始位置设定不当	26
5.2 VGA 显示模块的时钟频率	27
5.3 外部中断	27
5.4 BTNEZ 跳转长度	28
5.5 串口 UART	28
5.6 键盘输入的冷却时间	29
六. 实验感想	29
6.1 设计容易操作难, 调试更难	29
6.2 对计算机组成原理和流水线工作理解更深入	30
6.3 培养了耐心、细心、熬夜能力	30
6.4 提高解决问题的能力	31
6.5 永不放弃	31
6.6 团队协作	32
6.7 乐观自信的态度	32

一. 实验目标

1. 完成五段流水线 CPU，支持 THCO-MIPS 指令集中的 25 条基本指令及 JRRA/JALR/BTNEZ/NOT/SLTU 这 5 条特殊指令。
2. 通过旁路和气泡解决数据冲突和控制冲突。
3. 正确实现软件中断和硬件中断。
4. 实现 VGA 显示和 PS/2 键盘输入。
5. 将 term 程序的部分功能移植到监控程序中，使得监控程序可以脱离 term 运行，并通过显示器和键盘实现与用户的交互，从而在开发板一台完整的计算机应有的功能。

二. 数据通路与设计

2.1 整体设计

我们设计并实现了经典的五级流水结构 CPU。每一条指令分为 IF（取指）、ID（译码）、EXE（计算）、MEM（访存）、WB（写回）五个阶段，每一阶段主要完成的工作如下：

IF：根据跳转信息计算出下一个 PC 值，并根据当前 PC 的值在指令寄存器中取出要执行的指令传递到下一阶段。

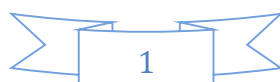
ID：根据 IF 阶段取出的指令生成控制信号并依次向下传递；根据生成的控制信号从寄存器堆中读取相应的寄存器的值一并传递到下一阶段。

EXE：根据 ID 阶段生成的控制信号和运算数进行运算，把结果传递到下一阶段。

MEM：根据 ID 阶段生成的控制信号和 EXE 阶段生成的运算结果判断是否需要访问内存并进行相应操作。

WB：根据控制信号和计算或者访存结果把数据写回到寄存器堆中。

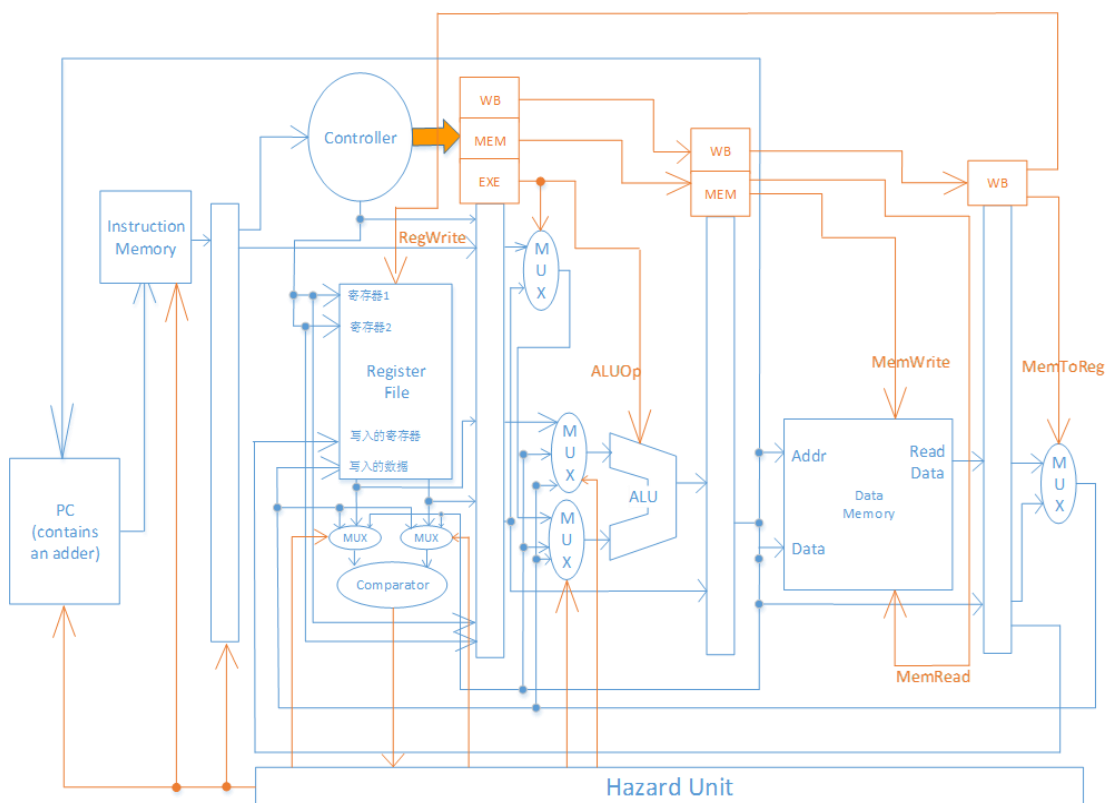
另外，每两个阶段之间均有一个段间寄存器以 ID_EXE_Register，



EXE_MEM_Register 的形式命名，主要功能是接收上一阶段的信号并在下一个时钟周期到来的时候传递到下一阶段。其中 IF_ID_Register 较特殊，除完成传递工作外，还需要对中断进行适当的处理，具体参见后面对中断的介绍。

2.2 数据通路

我们的整体数据通路设计如下（其中橙色的部分是控制信号相关）：



数据通路整体结构并未发生太大变化,然而我们将原本在 EXE 阶段的负责跳转的模块移到了 ID 阶段的比较器 **Comparator** 中,跳转地址仍由 EXE 阶段的 ALU 计算,这样在判断是否需要跳转和插入气泡时可以减少一个周期的延迟。

另外，我们把冲突控制和数据的转发统一放到了 Hazard Unit 中控制，这样使得整体结构更加简洁明了。

2.3 指令执行流程

我们列出了每个指令在每一阶段需要完成的工作

符号含义：Res:ALU 的运算结果；imm：根据需要进行符号扩展或零扩展的立即数；Rx、Ry、Rz：指令中的寄存器；SP、T、RA、IH、PC：特殊寄存器；Ram：存储器，用[]引用其地址；MEM：访存结果；ALUb：ALU 第二个操作数； \approx ：无符号数的大小比较结果

指令	IF	ID	EXE	MEM	WB
ADDIU	取指	译码	Res=Rx+imm	无	Rx=Res
ADDIU3	取指	译码	Res=Rx+imm	无	Ry=Res
ADDSP	取指	译码	Res=SP+imm	无	SP=Res
ADDU	取指	译码	Res=Rx+Ry	无	Rz=Res
AND	取指	译码	Res=Rx&Ry	无	Rx=Res
B	取指	译码	Res=PC+imm	无	无
BEQZ	取指	译码	Res=PC+imm	无	无
BNEZ	取指	译码	Res=PC+imm	无	无
BTEQZ	取指	译码	Res=PC+imm	无	无
CMP	取指	译码	Res=Rx!=Ry	无	T=Res
JR	取指	译码	Res=Rx+0	无	无
LI	取指	译码	Res=0+imm	无	Rx=Res
LW	取指	译码	Res=Rx+imm	Mem=Ram[Res]	Ry=Mem
LW_SP	取指	译码	Res=SP+imm	Mem=Ram[Res]	Rx=Mem
MFIH	取指	译码	Res=IH+0	无	Rx=Res
MFPC	取指	译码	Res=PC+0	无	Rx=Res
MTIH	取指	译码	Res=Rx+0	无	IH=Res
MTSP	取指	译码	Res=Rx+0	无	SP=Res
NOP	取指	译码	无	无	无
OR	取指	译码	Res=Rx Ry	无	Rx=Res
SLL	取指	译码	Res=Ry<<imm	无	Rx=Res

SRA	取指	译码	$\text{Res} = \text{Ry} \gg \text{imm}$	无	$\text{Rx} = \text{Res}$
SUBU	取指	译码	$\text{Res} = \text{Rx} - \text{Ry}$	无	$\text{Rz} = \text{Res}$
SW	取指	译码	$\text{Res} = \text{Rx} + \text{imm}$ $\text{ALUb} = \text{Ry}$	$\text{Ram}[\text{Res}] = \text{ALUb}$	无
SW_SP	取指	译码	$\text{Res} = \text{SP} + \text{imm}$ $\text{ALUb} = \text{Rx}$	$\text{Ram}[\text{Res}] = \text{ALUb}$	无
JRRA	取指	译码	$\text{Res} = \text{RA} + 0$	无	无
JALR	取指	译码	$\text{Res} = \text{PC} + 0$	无	$\text{RA} = \text{Res}$
BTNEZ	取指	译码	$\text{Res} = \text{PC} + \text{imm}$	无	无
NOT	取指	译码	$\text{Res} = \sim \text{Ry}$	无	$\text{Rx} = \text{Res}$
SLTU	取指	译码	$\text{Res} = \text{Rx} \approx \text{Ry}$	无	$\text{T} = \text{Res}$

2.4 扩展功能设计

2.4.1 冲突

2.4.1.1 结构冲突

通过 RAM1 和 RAM2 分别存储数据和指令来解决大部分的结构冲突。但在监控程序功能中有一个 A 指令需要向指令存储器中写入指令，这时需要暂停一个周期向 RAM2 中写入数据再继续之前的取指操作。

2.4.1.2 数据冲突

我们所需要解决的数据冲突主要是前一条指令写回到寄存器之后，后一条指令立即需要读取这个寄存器的情况。

对于除 lw 指令之外的这类冲突，后一条指令执行到 ID 完成之后前一条指令已经的 EXE 阶段已经结束，也就是写回到寄存器中的值已经确定，这时候可以通过旁路直接将数据送到后一条指令的 ALU 数据源上。

对于 lw 指令，因为后一条相关的指令至少需要在 lw 指令之后的第二个周期

才可以得到 `lw` 指令在 `MEM` 阶段取出的数据，因此不可避免的需要 `lw` 指令之后插一个气泡。

2.4.1.3 控制冲突

控制冲突指的是在写入寄存器相关的指令之后立刻使用了 `BEQZ` 之类的跳转指令。

对于这类冲突，我们的解决方法是首先在 `ID` 阶段，就确定是否需要跳转。因为确定跳转只需要判断寄存器中的值是否为零，这个判断通过组合逻辑来实现并不费时，所以可以在 `ID` 阶段进行。如果不需要跳转，则立刻执行之后的指令，如果需要跳转，则还需要在跳转指令之后插一个气泡，因为需要等 `ALU` 阶段才可以算出跳转到的地址，从而得到下一条指令具体是什么。

2.4.1.4 冲突检测

在代码中所有的冲突检测模块都集中到一个 `hazard` 模块中去。这个模块中都是组合逻辑，可以根据控制信号立即算出冲突的信号。在下一次跳变时，各寄存器、选择器就可以根据算出的冲突信号来确定数据。

2.4.2 中断

考虑到时钟中断和硬件中断的实现方式差不多，硬件中断时在检测到一个按键按下后发生中断，时钟中断是计时时钟到一定时间后发生中断，我们选择实现硬件中断而不实现时钟中断，另外实现软件中断。

软件中断和硬件中断的处理过程大部分相同，如下所述。

通过阅读监控程序的 `delint` 部分的代码，我们知道中断处理程序会用到中断时的 `PC` 值和中断号，所以需要在进入 `delint` 中断处理程序之前先保存好 `PC` 值和中断号。具体做法是在 `IF_ID_Register` 里加入一个状态机，如果没有中断就持续在状态 `0000` 运行，如果有中断发生就进入状态 `0001`，后面依次改变状态并执行以下命令：将当前的 `PC` 值取出放入 `R6` 寄存器、将栈顶寄存器 `SP` 的值加 1、将 `R6` 寄存器的值压栈、将中断号放入 `R6` 寄存器、再将栈顶寄存器 `SP` 的值加 1、

将 R6 寄存器的值压栈、将中断处理程序 `delint` 的起始地址放入 R6、跳转到 R6。这些命令依次执行完毕后回到状态 0。这时候 PC 已经在中断处理程序 `delint` 的地方，只要依次执行监控程序的指令即可，执行完毕后会跳回到中断发生时的 PC 处继续执行。

软件中断和硬件中断的不同之处如下分别介绍。

2.4.2.1 软件中断

在状态 0000 时，检测新得到的指令是不是 INT 指令，如果是进入状态 0001 开始处理中断，否则继续停留在状态 0000。

中断号为 0x00

2.4.2.2 硬件中断

在状态 0000 时，每经过一个时钟上升沿，检测此时外部中断的信号是否经历一个上升沿跳变，如果有跳变就进入状态 0001 开始处理中断，否则停留在状态 0000。

中断号为 0x10。

2.4.3 VGA 模块

显示模块大致可分为两部分，一部分为扫描每一个像素点，将其的颜色信息与同步信号输出至 VGA 接口。另一部分根据像素点的行列坐标算出像素点的颜色信息。

第一部分比较简单，只需要通过计数器实现循环行和列的坐标，并将坐标传给第二部分即可。

第二部分通过行列坐标算出像素点的颜色信息，并得到在显示器上直接显示与用户交互的类似命令行的界面的效果成为显示模块的最大难题。

按照传统，我们的项目采用 640*480 分辨率，一个字符占用 8*16 的像素点，即整个屏幕显示 30 行 80 列共 2400 个字符。要想正确显示字符，需要两块存储器分别存储每个字符对应的像素点以及每一个字符的模型（8*16 的像素点中哪

些点是黑哪些点是白)。然后程序依次扫描每个像素点,根据像素点的坐标算出属于哪一个字符块,再从两块存储器中分别读出字符和该字符的模型,从而得出像素点是黑还是白。这样就可以显示出类似命令行系统的文字界面。

这里首先明确表示字符的唯一标示符,作为程序内部的接口,在此我们使用了 7 位 ASCII 码。因为 ASCII 码中还有一些不可见字符,这给了我们修改的余地。我们从网络上下载了每个字符对应像素点的模型的字体文件,并根据我们的需要对其作了少量修改,增加了光标的字体,将其替代了一个不可见字符。

对于存储屏幕上显示的字符的存储器,方便起见,我们使用了 ISE 自带的 IP Generator 生成的 Block ROM。使用这个 Block Rom 的好处在于可以方便的定义初始化内存数据,这可以帮助我们方便的定义欢迎界面的信息。另外为了方便起见,该内存的地址位宽度可以设置为 12 位。其中前 5 位表示行,后 7 位表示列,这样虽然会有一点点空间的浪费,但是给编程带来了极大的便利。

在生成了内存之后,每次要修改屏幕上显示的字符,只需要修改这个存储器中的值就可以了。而无论是光标的显示,还是输入字符,还是删除字符(可以理解为输入退格字符),都只需要修改内存中的一个值,这给我们后续的编程提供了基本的工具。

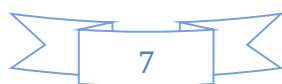
下面只需要根据程序的需求完成一些具体的模块。

2.4.3.1 光标显示

光标的显示比较简单,只需要在系统中维护当前光标所在的行和列的值,另外维护一个计数器,以及当前光标是否显示的状态。每次时钟信号跳变时,计数器加一;当计数器达到某一个和系统时钟频率相关的值时,改变当前光标是否显示的状态,并将当前字符块的值设为该状态。这样当下一次扫描到这一块时,就可以改变光标显示的状态。从整体上看,就可以造成光标一闪一闪的动画效果。

2.4.3.2 输出字符

如果是非换行符的普通字符,只需要在把当前光标所在的块的字符修改成要修改的字符,并将光标的列数加一即可(需注意不能超出屏幕最右)。如果是输出换行符,则需要将当前光标所在的块的字符修改成空,并且将光标的行数加一,



列初始化。如果输出的是退格字符，则直接将当前光标所在的块的字符修改成空（因为可能当前正在显示光标），然后列数减一即可。

2.4.3.3 换页

显示模块另外要处理一个换页的问题，即当输出换行时光标已经位于屏幕的底部时，这个时候需要将最上面一行删去，再将其它行上移一行，以空出最下面一行用于新的输出。若是直接这么去做，则需要耗费很多个时钟周期去更新内存中存储的值。若是在这些时钟周期中又有新的字符需要输出，则处理起来会比较棘手。在解决这个问题的时候我们用了一个小技巧，即当光标已经在底部时再需要输出时维护一个偏移量数值，表示在光标到底底部之后又换了多少行。在修改字符时，直接覆盖正好需要被删去的行。在显示根据像素点的标号计算当前属于哪一个块时，也加上这个偏移量，用于正确的显示。这样就可以在一个时钟周期内处理完光标在底部再换行的问题，即节省了时间又方便。

2.4.4 PS2 键盘模块

键盘模块需要完成的功能是从键盘读入字符，将是否读入字符的标志信号和读入字符的 ASCII 码传给系统。这个模块主要分为三部分，分别用来处理抖动、提取输入扫描码和转化 ASCII 码。

键盘模块首先要处理的问题是键盘时钟和数据的毛刺问题，可能会存在与实际不符合的抖动。因为键盘时钟远比提供的 50M 系统时钟慢的多，因此可以通过统计连续的几次系统时钟跳变时键盘时钟的值，来确定键盘时钟是否真的跳变，以此来确定是否有新的键盘输入。

在确定有新的键盘输入信号之后，就需要提取扫描码。依次提取 8 位数据位和 1 位校验位，在确认校验无误之后就可以得到键盘按下的按键。接下来需要将键盘的扫描码转化为 ASCII 码。因为扫描码中包括了 F0 这样的断码，以及 SHIFT 等控制型的按键，所以需要对按键的扫描码做一番处理之后才能得到传给系统的 ASCII 码。这里需要设计一个状态机，用于记录当前是否有键按下、是否有 SHIFT 按下等信息，由此得到真正的输入信息。

2.4.5 监控程序修改

为了能完全脱离 Term 程序，利用上面实现的 VGA 和 PS2 模块完成 Term 程序的功能，我们需要对提供的监控程序作一定的修改以满足我们的需求。首先我们确定 0xBF04 和 0xBF06 为键盘和显示器的接口。通过 lw 和 sw 指令即可实现监控程序和键盘/VGA 的交互。

修改监控程序的最大难题在于原本 Term 程序和监控程序作交互式用的字符编码在一些命令中用的是 ASCII 码，另一些命令中用的是 16 进制的数值。因为 Term 程序直接用 C++ 处理，所以会方便的多，但是要由 MIPS 指令直接完成 16 进制数值向 ASCII 字符的转化，就会麻烦的多。在编写这部分汇编代码的时候，我们先使用下发的 Simulator 对代码作了充分的测试，再将其结合到监控程序中，最大限度的减少了调试的麻烦。

监控程序主要修改体现在如下几点：

输入部分从原来的 0xBF00 串口读入改为从我们设计的键盘端口 0xBF04 读入，读入状态由 0xBF05 的值来决定，每读入一次数据后加入一个空的循环进行延时，以防止一次过多读入同一个键。

输出部分从原来的 0xBF00 串口写改为从我们设计的 VGA 显示器端口 0xBF06 写入，而无需状态，可随时写入。

输入数据处理的原来的直接使用改为两种方式处理：在等待命令时直接处理得到的 ASCII 码值，在命令中读入十六进制时需要将得到的 ASCII 码值从高位到低位转化成对应的二进制串进行操作。

输出数据统一转化成 ASCII 码值输出字符，如遇到需要输出十六进制数据或指令则需要将其从高到位每四位转化成一个字符输出。

2.4.6 Flash 引导

Flash 引导的好处是可以将监控程序每次自动地从 Flash 中加载到 Ram2 指令存储器中，这样可以免去每次写入监控程序的麻烦。

由于我们事先在 Flash 中写好监控程序的数据，因此我们在引导过程中要用到的操作有从 Flash 中读和向 Ram2 中写。

从 Flash 中自动读取的实现方法是设计了一个计数器，在按下 start 时初始化，然后在每个时钟周期里读取该计数器所对应的地址处的数据，并将计数器自动加 1，置“引导中”信号为 1，直到计数器的值变成 512（略大于监控程序的总指令条数），停止读取，引导结束，置“引导中”信号为 0。

由于 Ram2 本来有取指、读指、写入指令三种情况，所以 Ram2 的地址有 PC 模块输出值、ALU 计算结果两种，现在根据需要加入一个 Flash 引导写入的地址，修改原来的多路选择器，并将“引导中”信号加入控制信号。Ram2 的数据本来只可能从要写入内存的数据中来，现在加入一个 Flash 读出的数据，同样将“引导中”信号加入控制信号。这样便能实现对 Ram2 指令的写入功能。

对于其他部件，则将“引导中”信号的非和原重置信号做与运算作为新的重置信号，即可令引导状态中的其他部件一直处于重置状态，不发挥作用，这样便可以完整地完 Flash 引导功能。

三. 代码模块与接口设计

这里省略了一些不重要的多路选择器。

`common.vhd`

用于定义其它代码中用到的常数。

`freDivider.vhd`

用于对实验板提供的 50M 时钟信号进行分频处理。在不使用 PS/2 和 VGA 模块的情况下我们的 CPU 主频可以达到 25MHz，使用后可以达到 12.5MHz。

名称	类型	说明
clkin	IN	分频前时钟信号
clkout	OUT	分频后时钟信号

pc.vhd

对 PC 值进行处理的模块，包括的功能有将 PC 值加一，根据 ID 阶段 Comparator 的结果得到的是否跳转的信号来决定下一个 PC 值。

名称	类型	说明
clk	IN	时钟信号
rst	IN	重置信号
pc_in	IN	PC 值
jump	IN	是否跳转信号
jump_addr	IN	跳转地址
pc_plus_one	OUT	PC 加 1 的值，用于提供给 PC 寄存器
pc_out	OUT	用于取指的 PC 值

PCMUX

根据 Hazard 冒险检测单元得到的 PC 是否需要停留的结果来选择 PC 模块的下次输入仍是当前值还是 PC+1.

名称	类型	说明
pc_to_ram2	IN	用于取指的 PC 值
pc_plus_oneF	IN	PC 加 1 的值
pc_stay	IN	PC 是否需要停留
y	OUT	选择后的 PC 值

pcwriteMux2

读写 Ram2 的地址的选择器。Ram2 地址来源有 PC 模块中提供的 PC 值，SW 命令要写入程序指令时的地址(由 ALU 计算结果而来)，还有 Flash 引导时的地址，需要根据 Hazard 模块、boot 模块生成的控制信号决定。

名称	类型	说明
pc	IN	需要取指的指令地址
write	IN	需要写入指令存储器的地址，由 ALU 计算得到

flash_addr	IN	从 Flash 写入指令存储器的地址
booting	IN	是否在 Flash 引导的启动过程中
s	IN	控制信号，由 Hazard 模块生成，决定读指令还是写入指令
userlwstall	IN	根据是否需要读取指令存储器时判断是否需要暂停当前流水线的信号
y	OUT	输出，最后提供给指令存储器的地址

Boot

Flash 引导模块，可以自动从 Flash 中读取数据存入 Ram2 中。另外，本模块还提供了 **booting** 信号指示当前 Boot 模块是否在工作中以阻止其他模块的工作。

名称	类型	说明
clk	IN	时钟信号
enable	IN	是否允许 Flash 引导
start	IN	模块初始化，从 0 开始读取指令写入 Ram2
flash_byte	OUT	Flash 操作模式
flash_vpen	OUT	Flash 写保护
flash_rp	OUT	Flash 工作标志
flash_ce	OUT	Flash 使能信号
flash_oe	OUT	Flash 读使能
flash_we	OUT	Flash 写使能
flash_addr	OUT	Flash 地址总线
flash_data	OUT	Flash 数据总线
booting	OUT	当前 Flash 模块是否处于工作中的信号

IF_ID_Register

位于 IF 和 ID 阶段之间的段间寄存器。除了起传递作用外，还同时肩负处理中断的任务，将处理中断之前的一系列需要处理的指令插入到中断发生之后传递下去。

名称	类型	说明
----	----	----

clk	in	时钟信号
rst	in	重置信号
out_ter	in	是否有外部中断到来
stall	in	是否需要暂停一个周期
InInst	in	由指令存储器中读取的指令
InPC	in	由 PC 模块生成的 PC 寄存器的值
flush	in	是否需要清除当前指令对之后的影响，即插入气泡
PC_INT	out	在处理中断时通知 PC 暂停的信号
OutPC	out	输出到 ID 阶段的 PC 值
OutInst	out	输出到 ID 阶段的指令

Registerfile

寄存器堆。为了方便管理，我们把八个通用寄存器和五个特殊寄存器（IH、T、SP、RA、PC）统一用 4 位编码放到当前模块中，这样可以使得整体结构更简洁，少了很多控制信号。

名称	类型	说明
clk	in	时钟信号
rst	in	重置信号
REGF_SrcA	in	要读取的第一个寄存器的地址
REGF_SrcB	in	要读取的第二个寄存器的地址
REGF_InAddr	in	要写入的寄存器的地址
REGF_WE	in	寄存器堆写使能信号
REGF_InData	in	寄存器堆待写入数据
REGF_InPC	in	要写入的 PC 寄存器的值
REGF_OutA	out	输出的第一个寄存器的值
REGF_OutB	out	输出的第二个寄存器的值

Comparator

根据控制信号中的跳转类型比较寄存器的值来确定当前跳转指令是否会跳转，从

而决定下一个 PC 是否要跳转到 ALU 计算出的结果。

名称	类型	说明
rst	in	重置信号
JumpType	in	当前跳转类型
RegA	in	第一个寄存器的值
RegB	in	第二个寄存器的值
Jump	out	是否会跳转的输出

Controller

控制器。主要功能是译码，根据输入的指令得到各个控制信号的值。为了方便书写，这个模块采用了一个函数 `generate_control` 给一个 `record` 类型变量赋值（均定义在 `common` 中），然后统一把值赋到输出信号上，这样既保证了正确、简洁，又增强了可扩展性和易维护性，令控制信号的增加、修改都变得十分容易。

名称	类型	说明
CTRL_Inst	in	输入的要解析的指令
ALU_Op	out	输出的 ALU 操作信号
REGF_A	out	输出的寄存器堆的第一个寄存器的编号
REGF_B	out	输出的寄存器堆的第二个寄存器的编号
Imm	out	输出的进行扩展后的 16 位立即数
ALU_B_Src	out	输出的 ALU 第二个操作数是采用寄存器的值还是立即数的控制信号
MEMControl	out	输出的 MEM 阶段是否要访问内存以及访问方式的控制信号
RegWrite	out	输出的是否要在 WB 阶段写回寄存器堆的控制信号
RegWrite_Addr	out	输出的写回寄存器堆的寄存器的编号
MemtoReg	out	输出的写回寄存器堆的数据是 ALU 结果还是访存结果的控制信号
JumpType	out	输出的是否是跳转语句以及跳转类型的控制信号

ID_EXE_Register

位于 ID 和 EXE 阶段之间的段间寄存器。起数据传递功能，另外可以根据 Hazard 模块的控制信号来决定是否清空之前指令造成的影响。

名称	类型	说明
clk	in	时钟信号
rst	in	重置信号
flush	in	是否清空之前指令影响的信号
In_ALU_A	in	输入的 ALU 的第一个操作数
In_ALU_B	in	输入的 ALU 的第二个操作数
In_Imm	in	输入的拓展后的立即数
In_ALU_B_Src	in	输入的 ALU 第二个操作数的来源
In_ALU_Op	in	输入的 ALU 的运算
In_JumpType	in	输入的指令的跳转类型
In_Jump	in	输入的最终是否跳转的信号
In_Rs	in	输入的指令中的第一个寄存器
In_Rt	in	输入的指令中的第二个寄存器
In_RAM1_Control	in	输入的对存储器的控制信号
In_MemtoReg	in	输入的写回到寄存器堆中的数据来源的信号
In_RegWrite	in	输入的是否写回到寄存器堆中的信号
In_RegWriteAddr	in	输入的写回的寄存器的编号
Out_ALU_A	out	输出的 ALU 的第一个操作数
Out_ALU_B	out	输出的 ALU 的第二个操作数
Out_Imm	out	输出的拓展后的立即数
Out_ALU_B_Src	out	输出的 ALU 第二个操作数的来源
Out_ALU_Op	out	输出的 ALU 的运算
Out_JumpType	out	输出的指令跳转类型
Out_Jump	out	输出的指令是否跳转的信号
Out_Rs	out	输出的指令中的第一个寄存器
Out_Rt	out	输出的指令中的第二个寄存器

Out_RAM1_Control	out	输出的对存储器的控制信号
Out_MemtoReg	out	输出的写回到寄存器堆中的数据来源的信号
Out_RegWrite	out	输出的是否写回到寄存器堆中的信号
Out_RegWriteAddr	out	输出的写回的寄存器的编号

ALU

ALU 主要实现对两个操作数的运算功能。其中基本运算共有加、减、与、或、非、左移、右移、无符号减法、比较等适合于我们指令集的基本运算。

名称	类型	说明
srcA	in	ALU 的第一个操作数
srcB	in	ALU 的第二个操作数
op	in	ALU 的运算
result	out	ALU 运算的结果

Hazard

本模块主要用来处理冲突和数据的转发，根据 ID 和 EXE 阶段，即上一条指令和上上条指令的控制信号和指令中的寄存器的值来进行判断。

名称	类型	说明
pc_int	in	IF/ID 寄存器是否在处理中断的信号
rsD	in	ID 阶段指令使用的第一个寄存器
rtD	in	ID 阶段指令使用的第二个寄存器
rsE	in	EXE 阶段指令使用的第一个寄存器
rtE	in	EXE 阶段指令使用的第二个寄存器
writeregE	in	EXE 阶段指令写回的寄存器的编号
writeregM	in	MEM 阶段指令写回的寄存器的编号
writeregW	in	WB 阶段指令写回的寄存器的编号
RegWriteE	in	EXE 阶段是否要写回寄存器的控制信号
RegWriteM	in	MEM 阶段是否要写回寄存器的控制信号
RegWriteW	in	WB 阶段是否要写回寄存器的控制信号

memtoregE	in	EXE 阶段指令写回寄存器堆的数据源
memtoregM	in	MEM 阶段指令写回寄存器堆的数据源
jumpD	in	ID 阶段得到的是否要跳转的信号
jumpE	in	EXE 阶段得到的是否要跳转的信号
jumpType	in	当前编码的指令的跳转类型
memcontrolM	in	MEM 阶段指令对存储器访问的控制信号
writeAddr	in	要访问存储器的地址
forwardaD	out	ID 阶段对第一个寄存器的转发控制信号
forwardbD	out	ID 阶段对第二个寄存器的转发控制信号
forwardaE	out	IE 阶段对第一个 ALU 操作数的转发控制信号
forwardbE	out	IE 阶段对第二个 ALU 操作数的转发控制信号
stallF	out	IF 阶段是否需要暂停一个周期
flushD	out	ID 阶段是否需要清除影响，即插入 NOP
stallD	out	ID 阶段是否需要暂停一个周期
flushE	out	EXE 阶段是否需要清除影响
RAM2_Control	out	对 Ram2 存储的读写控制
userlwstall	out	是否是从 Ram2 读取数据使得流水线暂停

EXE_MEM_Register

在 EXE 和 MEM 阶段之间的段间寄存器，起数据传递作用。

名称	类型	说明
clk	in	时钟信号
rst	in	重置信号
In_RAM1_Control	in	输入的对 Ram1 读写的控制
In_RAM1_Addr	in	输入的对 Ram1 访问的地址
In_RAM1_InData	in	输入的要写入 Ram1 的数据
In_MemtoReg	in	输入的写回寄存器堆的数据来源
In_ALUResult	in	输入的 ALU 的计算结果
In_RegWrite	in	输入的是否要写回寄存器堆的控制信号

In_RegWriteAddr	in	输入的要写回的寄存器的编号
Out_RAM1_Control	out	输出的对 Ram1 读写的控制
Out_RAM1_Addr	out	输出的对 Ram1 访问的地址
Out_RAM1_InData	out	输出的要写入 Ram1 的数据
Out_MemtoReg	out	输出的写回寄存器堆的数据来源
Out_ALUResult	out	输出的 ALU 的计算结果
Out_RegWrite	out	输出的是否要写回寄存器堆的控制信号
Out_RegWriteAddr	out	输出的要写回的寄存器的编号

MEM_WB_Register

位于 MEM 和 WB 阶段之间的段间寄存器，起数据传递作用。

名称	类型	说明
clk	in	时钟信号
rst	in	重置信号
In_MemtoReg	in	输入的写回寄存器堆的数据来源
In_ALUResult	in	输入的 ALU 的计算结果
In_RegWrite	in	输入的是否要写回寄存器堆的控制信号
In_MemData	in	输入的访存的结果
In_RegWriteAddr	in	输入的写回的寄存器的编号
Out_MemtoReg	out	输出的写回寄存器堆的数据来源
Out_ALUResult	out	输出的 ALU 的计算结果
Out_RegWrite	out	输出的是否要写回寄存器堆的控制信号
Out_MemData	out	输出的访存的结果
Out_RegWriteAddr	out	输出的写回的寄存器的编号

RAM1

用于处理数据存储，串口访问，ps/2 键盘和 VGA 屏幕输出的模块。

名称	类型	说明
rst	in	复位信号

clk	in	分频后的时钟信号
clk_ori	in	原始输入的 50MHz 时钟信号
ps2_clk	in	给键盘的时钟信号
ps2_data	in	从键盘得到的数据
reg_addr	in	要写入 RAM1 的地址
reg_data	inout	要写入 RAM1 的数据
read_write	inout	读内存或是写内存
tsre	in	写内存时数据是否写入到暂存器的信号
tbre	in	写内存时数据是否从暂存器写出到串口的信号
data_ready	out	串口是否可读
ram1_en	out	RAM1 总使能
ram1_oe	out	RAM1 读使能
ram1_we	out	RAM1 写使能
port_oe	out	串口读使能
port_we	out	串口写使能
mem_addr	out	要读出或写入的内存地址
mem_data	inout	要读出或写入的内存数据
vga_h_sync	out	VGA 行同步信号
vha_v_sync	out	VGA 场同步信号
vga_r	out	VGA 红色输出
vga_g	out	VGA 绿色输出
vga_b	out	VGA 蓝色输出

RAM2

用于处理指令存储器的模块。

名称	类型	说明
clk	in	时钟信号
RAM2_Control	in	指令存储器读写控制信号
RAM2_InData	in	指令存储器的写入数据

RAM2_InAddr	in	指令存储器的访问地址
RAM2_EN	out	Ram2 的使能信号
RAM2_WE	out	Ram2 的写使能信号
RAM2_OE	out	Ram2 的输出使能
RAM2_Data	inout	Ram2 的数据总线
RAM2_Addr	out	Ram2 的地址总线

vga_test

显示模块的顶层结构

名称	类型	说明
clk	in	50M 时钟
clk_sys	in	系统运行时钟
rst	in	重置信号
write_ena	in	重写 vga 显示显存的使能
write_char	in	写入显存的字符，仅后 7 位表示 ASCII 码
h_sync	out	水平同步信号
v_sync	out	垂直同步信号
r	out	红色输出
g	out	绿色输出
b	out	蓝色输出

vga_controller

VGA 显示控制，循环扫描每个像素点，并将行列参数传给 text_generator

名称	类型	说明
pixel	in	VGA 时钟（25M）
reset	in	重置信号
h_sync	in	水平同步信号
v_sync	in	垂直同步信号
disp_ena	in	当前像素点是否显示

column	out	当前像素点所在行
row	out	当前像素点所在列

text_generator

根据行列像素点的坐标算出像素点的颜色，需要首先读出像素点对应的字符，再读出像素点在字符中的位置。另外这个模块一并处理外部修改显存的需求。

名称	类型	说明
clkr	in	VGA 时钟（25M）
clkw	in	写显存时钟（即系统运行时钟）
rst	in	重置信号
disp_ena	in	显示使能
row	in	当前像素所在行
column	in	当前像素所在列
write_ena	in	写入使能
write_char	in	写入字符
red	out	红色输出
green	out	绿色输出
blue	out	蓝色输出

debounce

用于处理键盘输入防抖动

名称	类型	说明
clk	in	50M 时钟
button	in	需要防抖动处理的信号
result	out	防抖动处理后的信号

ps2_keyboard

用于提取键盘的扫描码

名称	类型	说明
----	----	----

clk	in	50M 时钟
ps2_clk	in	ps2 时钟
ps2_data	in	ps2 数据
ps2_code_new	out	键盘是否有新输入标志位
ps2_core	out	键盘新输入的字符的键盘编码

ps2_keyboard_to_ascii

将键盘的扫描码转化成 ASCII 码

名称	类型	说明
clk	in	50M 时钟
ps2_clk	in	ps2 时钟
ps2_data	in	ps2 数据
ascii_new	in	是否有新 ASCII 码标志位
ascii_code	in	新输入的字符的 ASCII 码

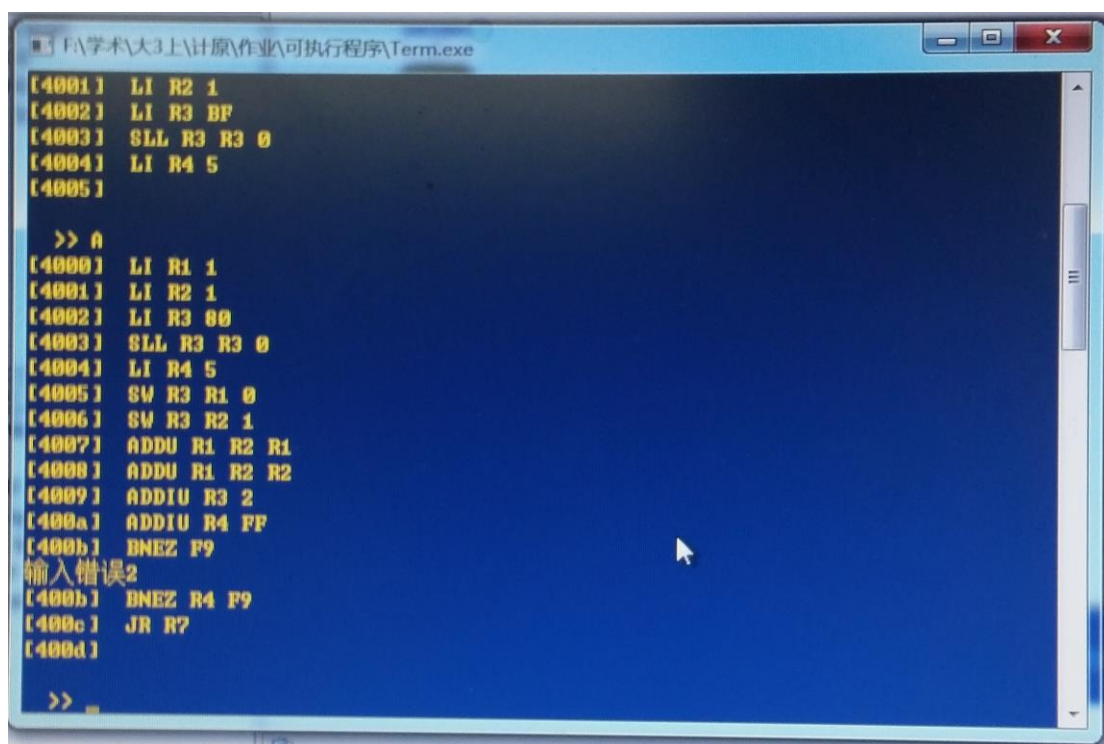
四. 成果展示

我们最终成果分为两个版本，基本版本和带 VGA 显示的版本。基本版本通过 term 程序与 CPU 交互，带 VGA 显示的版本通过键盘和显示器与 CPU 直接交互。

4.1 基本版本展示

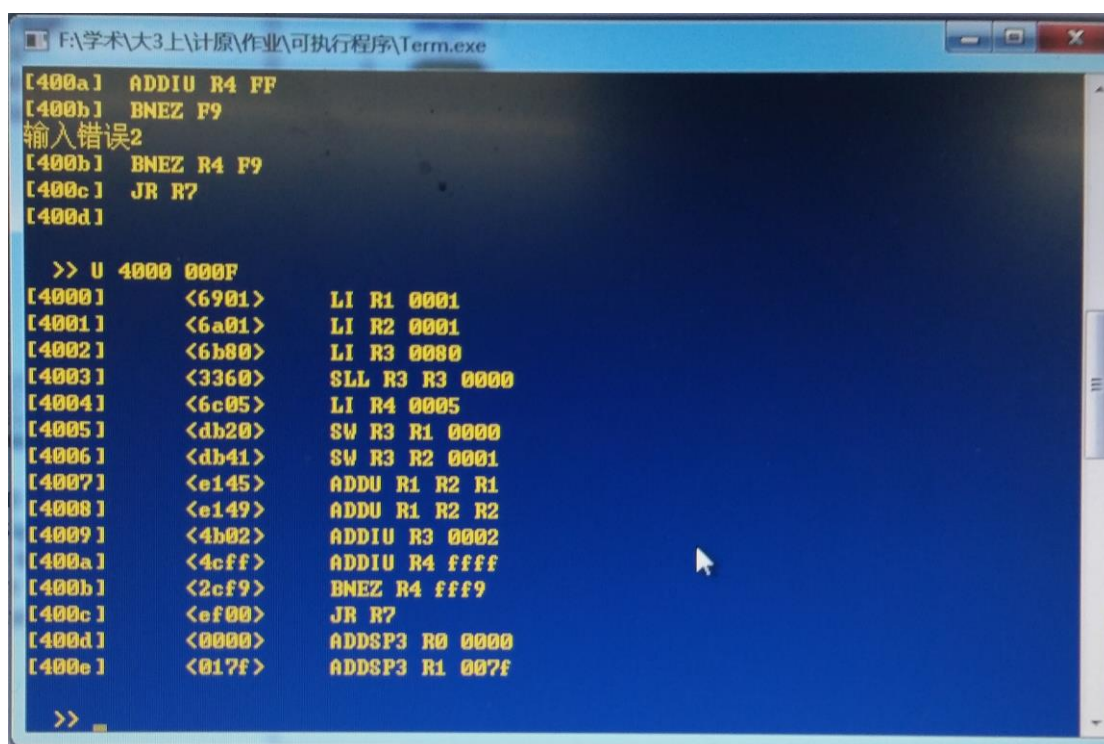
基本版本中由于已经存在了键盘模块，因此主频至多可以跑到 12.5M（在没有键盘模块的时候可以跑到 25M）。我们录制的视频“普通.mp4”展示了我们基本版本的运行情况，从中可以看到我们的基本版本可以正确的运行 A/D/G/U/R 等指令，并可以正确实现软件终端和硬件中断的功能。

通过 A 指令输入一个计算斐波那契数列的程序。



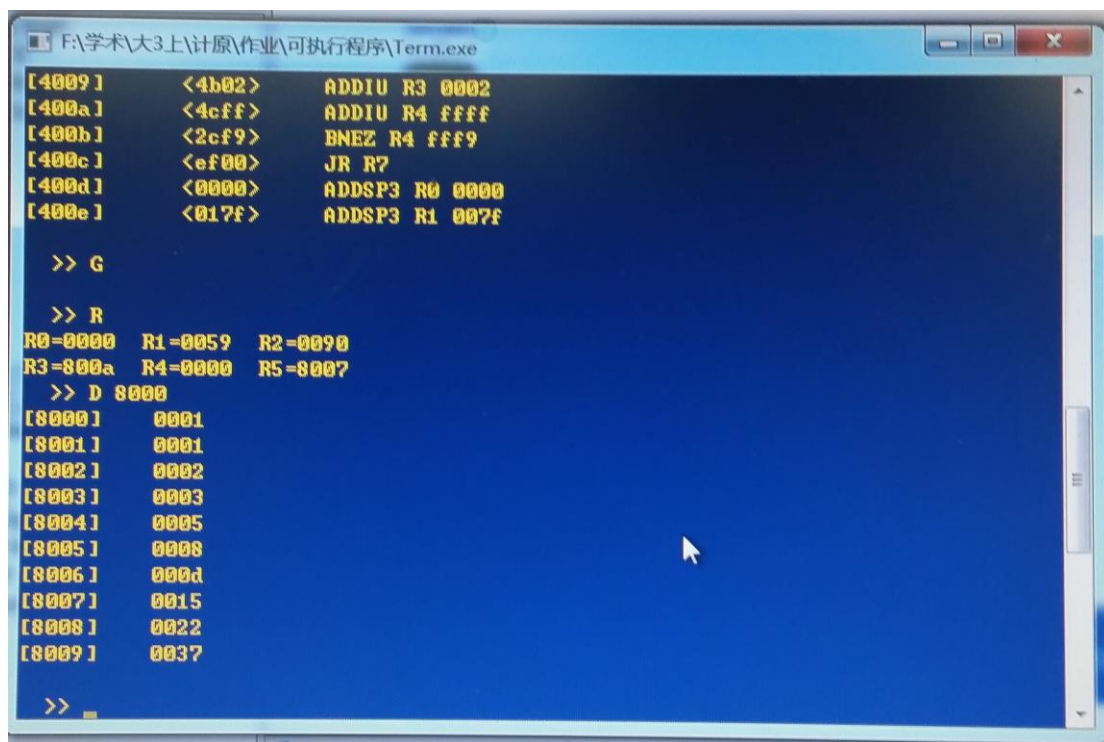
```
F:\学术\大3上\计原\作业\可执行程序\Term.exe
[4001] LI R2 1
[4002] LI R3 BF
[4003] SLL R3 R3 0
[4004] LI R4 5
[4005]
>> 0
[4000] LI R1 1
[4001] LI R2 1
[4002] LI R3 00
[4003] SLL R3 R3 0
[4004] LI R4 5
[4005] SW R3 R1 0
[4006] SW R3 R2 1
[4007] ADDU R1 R2 R1
[4008] ADDU R1 R2 R2
[4009] ADDIU R3 2
[400a] ADDIU R4 FF
[400b] BNEZ F9
输入错误2
[400b] BNEZ R4 F9
[400c] JR R7
[400d]
>>
```

通过 U 指令查看反汇编结果



```
F:\学术\大3上\计原\作业\可执行程序\Term.exe
[400a] ADDIU R4 FF
[400b] BNEZ F9
输入错误2
[400b] BNEZ R4 F9
[400c] JR R7
[400d]
>> U 4000 000F
[4000] <6901> LI R1 0001
[4001] <6a01> LI R2 0001
[4002] <6b00> LI R3 0000
[4003] <3360> SLL R3 R3 0000
[4004] <6c05> LI R4 0005
[4005] <db20> SW R3 R1 0000
[4006] <db41> SW R3 R2 0001
[4007] <e145> ADDU R1 R2 R1
[4008] <e149> ADDU R1 R2 R2
[4009] <4b02> ADDIU R3 0002
[400a] <4cff> ADDIU R4 ffff
[400b] <2cf9> BNEZ R4 fff9
[400c] <ef00> JR R7
[400d] <0000> ADDSP3 R0 0000
[400e] <017f> ADDSP3 R1 007f
>>
```

通过 G 指令运行，通过 R 指令查看寄存器的值，通过 D [地址]指令查看制定内存中的值。



```
F:\学术\大3上\计原\作业\可执行程序\Term.exe
[4009]    <4b02>    ADDIU R3 0002
[400a]    <4cff>    ADDIU R4 ffff
[400b]    <2cf9>    BNEZ R4 fff9
[400c]    <ef00>    JR R7
[400d]    <0000>    ADDSP3 R0 0000
[400e]    <017f>    ADDSP3 R1 007f

>> G

>> R
R0=0000 R1=0059 R2=0090
R3=800a R4=0000 R5=8007
>> D 8000
[8000]    0001
[8001]    0001
[8002]    0002
[8003]    0003
[8004]    0005
[8005]    0008
[8006]    000d
[8007]    0015
[8008]    0022
[8009]    0037

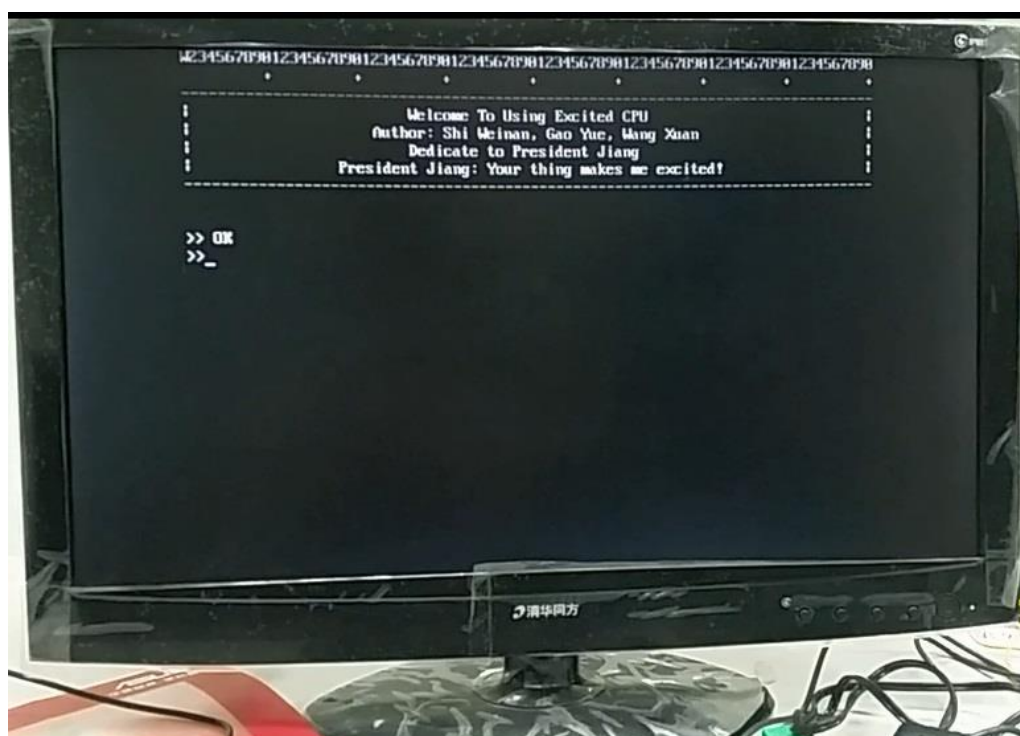
>>
```

可以看到正确计算出了斐波那契数列。

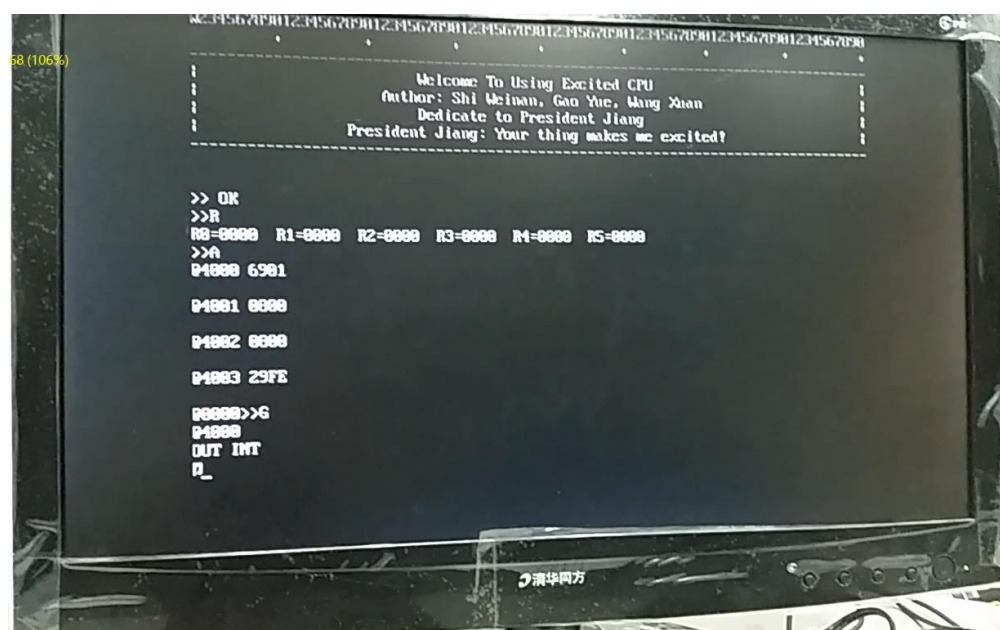
4.2 VGA 版本展示

带 VGA 显示的版本由于控制 VGA 显示的内容会产生大量时延，因此主频不能跑的非常快。我们录制的视频“VGA 模块说明.mp4”展示了这个版本的运行情况。从中可以看到这个版本可以正确运行监控程序中的 A/D/G/R 命令，可以在脱离 term 的情况下实现与用户的交互。这相当于是用开发板实现了一台冯诺依曼结构的计算机所需要的所有模块（存储器、运算器、控制器、输入设备、输出设备），使得开发板的名称“教学计算机”名副其实。

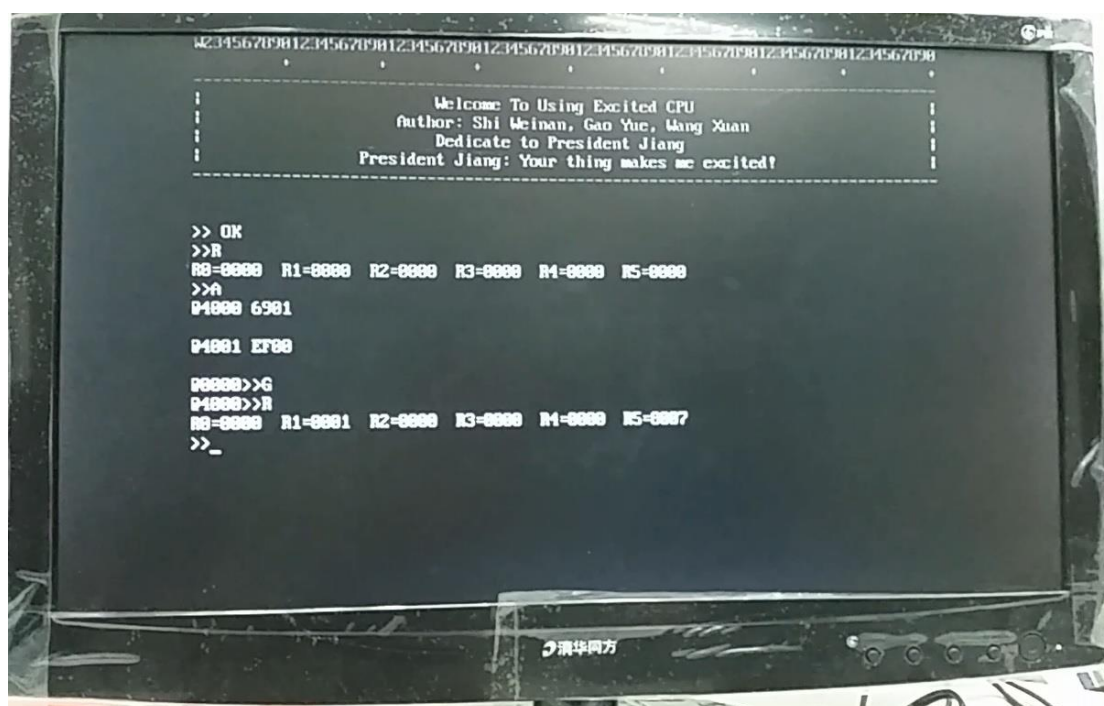
屏幕上部分是我们自己做的欢迎界面。下面是运行监控程序会输出的 OK



先通过 R 指令查看寄存器的值，可以看到初始时所有寄存器都是 0。然后通过 A[地址]输入自己编写的程序。此处我们为了测试外部中断编写了一个死循环程序。最后通过 G[地址]执行程序。可以看到当我们按下中断键后屏幕上打出了 OUT INT 表示出现了外部中断。



下面我们编写了一个给寄存器 R1 赋值为 1 的程序。先通过 R 指令查看所有寄存器的值，可以看到初始时都是 0，接着通过 A[地址]编写自己的程序，其中第一行 6901 是 LW R1 1，第二行 EF00 是 JR R7。通过 G[地址]运行程序，最后通过 R 指令查看寄存器的值，可以看到 R1 的值正确地变为了 1，R5 寄存器由于被监控程序执行用户程序时使用编程了 0087，这在 term 上运行时也是同样的结果。可以看到，我们的 VGA 版本的 CPU 可以摆脱 term，利用键盘输入将结果显示在屏幕上，运行的结果都是正确的。



五. 实验中遇到的问题

5.1 栈顶指针初始位置设定不当

监控程序中栈顶指针 SP 设定的位置在 BF11，我们处理中断时要保存现场，要执行将现在的 PC 和中断号压栈的操作，但是在经过几次压栈后，栈顶指针会逐渐减小，减到 BF00，这是串口的地址，所以我们压栈的数据没有正确地进入栈，而是输出到了串口。为此我们尝试修改 SP 的初始值，但是经过几次尝试后发现程序运行地仍然是十分不稳定，而此时距离检查只剩几个小时，我们没有耐

心地细想，只是在监控中尽量少使用压栈操作，压栈处理后迅速让其出栈，保证栈的地址不会到达串口地址，另外在使用寄存器的时候分外小心，已经有值的寄存器不能先压栈使用后再出栈，只能另选空闲的寄存器临时使用，导致我们使用寄存器的时候躲避已经有值的寄存器躲得手忙脚乱。

5.2 VGA 显示模块的时钟频率

在加入 VGA 显示模块后进行测试时我们并没有改变原来的时钟频率 12.5M，由于显示模块需要进行的计算比较复杂，一个时钟周期内并不能完成，这使得显示模块还没有来得及显示字符就收到了下一个字符，造成显示字符的不正确。

5.3 外部中断

在实现外部中断的时候，为了在按下一个键时看到屏幕上有“外部中断”的输出，我们写了一个死循环的程序，让屏幕上不断地输出“H”。这时我们按下按键后屏幕上有时会正确输出“外部中断”，有时候什么输出都没有。后来我们想到串口处理输出的速度并不快，而且听说别的组调试外部中断的时候只写了一个死循环的程序，没有让程序一直输出任何信息，所以我们认为是串口出现问题而不是我们程序的实现存在问题。将死循环内的输出去掉后，我们看到程序基本正常地输出“外部中断”，但有一定概率会自动跳出死循环。

后来我们猜想是在执行 B 指令时发生中断，这样程序会继续执行 B 指令后面的一条指令，就跳出了死循环。进一步分析，我们发现原先的实现中，一旦在执行某条指令时发生外部中断，我们会丢弃现在的这条指令，去处理中断，在中断结束后也没有继续执行中断前的这条指令，而是接着执行后面的一条指令，这样就由于发生中断而丢弃了一条指令。经过修改，我们让中断处理结束后继续执行中断前在执行的这条指令。最后看到程序正确地执行死循环，在外部发生后输出“外部中断”，之后仍然处在死循环内，无论重复发生多少次外部中断程序都能正确执行。

5.4 BTNEZ 跳转长度

我们在修改监控程序以适应 PS/2 键盘输入和 VGA 输出的时候，发现监控程序在等待指令时会比较输入的字母的值来进行跳转，而我们在输入 D 指令时却会跳转到一个输出 OK 的位置，即监控程序的开始部分，在输入其他命令的时候却不会有这个问题。

后来我们分析发现这些指令的不同之处在于跳转使用的是 BTNEZ 进行跳转，其他指令是跳到了一个 GOTO 模块，再用 B 指令跳转到应该跳到的位置，而 D 指令却是直接跳到应该跳转的位置。查询指令的格式发现 BTNEZ 跳转的立即数只有 8 位，B 跳转的立即数有 11 位，这样在 BTNEZ 的跳转行数超过 128 行时第八位立即数会变成 1，由于做符号扩展，这个跳转值就变成了负值，于是便跳到了程序的开始位置，而用 GOTO 模块中转一下便可以用 B 的超长跳转距离解决这个问题。

我们最后在可能会超出跳转长度的跳转都增加了一个 GOTO 模块解决了这个问题。

5.5 串口 UART

我们调试好基本指令后发现 Term 上可以输出 OK 字样，并且 R 指令也可以被正确运行，但 U、A、G 三个指令却始终无法正确运行。后来排除了所有可能后，我们采用了串口精灵进行调试，发现一个一个模拟终端输入数据可以得到正确的结果。这时我们所采用的时钟主频很低，小于 10Hz，将主频调高到 1MHz 以上时问题即得到解决。

在调试外部中断的时候我们开始时采用了一个不停输出字母到串口的死循环程序，然后通过按动微动开关来触发外部中断。可是这样却经常会得到不正常的结果，但在改成无任何输出的死循环后外部中断却可以正常工作。

以上两个问题都反映出了串口的处理问题，即在向串口写入数据时 UART 模块不会将一次收到的数据缓存下来供设备读取，而如果时钟频率过慢则会丢失数据，这样造成了监控程序一直在等待输入的情况（U、A、G 三个命令是需要再

次输入参数的)，而外部中断则因为串口的占用而导致无法将中断号正常输出到 Term 处，于是外部中断无法正常执行。

这个问题提示我们要对系统各个部分有全面而深入的了解，这样在遇到问题的时候才能从问题的根源找到原因加以解决。

5.6 键盘输入的冷却时间

在最后调试键盘输入的时候我们发现用自己定义的 0xBF04 端口作为键盘读入，用 0xBF05 端口作为指示是否可读的状态，如果按原来监控程序的写法会导致一次读入多个相同的字符。究其原因，我们的 0xBF05 是显示当前是否有键按下，而由于人手按键的停留时间和 CPU 高频的运行时间的巨大差异，这里会在一次读入后继续运行多条指令从而继续读入相同的键值，直到手指离开键盘为止。

这里的差异主要在读入键盘数据的子程序中，我们根据实际情况，考虑到真实的计算机中确实有一直按住同一个键可以输入一连串相同字符的情况，在读入键盘数据的子程序中加入了一个长度适当的空循环进行延时，以保证不会在正常的按键时间内多读入数据，且能响应一直按住一个键的重复输入。

最后，我们调整这个循环的长度达到了一个比较完美的效果，也体会到了理论和实践的差异，更明白了在很多地方不能直接照搬，而是要根据实际情况做出不同的修改的道理。

六. 实验感想

6.1 设计容易操作难，调试更难

我们一开始设计数据通路的时候以为课件上的通路已经十分完善，我们只要大体上以课件为指导，稍作改动就能作为自己的数据通路投入使用。但当我们真正实现起各个部分的时候才发现，原想的数据通路存在多处问题，尤其是在加入 forward 转发单元和 hazard detect 冲突控制单元的时候，发现很多之前的通路需

要重新实现。跳转部分也经过了多次修改，和最初的设想相去甚远。而调试起来更加麻烦，代码写好和真正能用完全是两回事，还需要经过无休止的调试。由此看来，李山山老师毕竟是身经百战，知道调试才是最费时间的阶段，在最初写代码的时候就一直催促我们早点动手早点写完早点开始调试。有了李老师传授的人生经验，我们的代码一早就都写完了，但也只是勉强有足够的时间进行调试，至今都剩了一点点小 bug 没有完全解决。

6.2 对计算机组成原理和流水线工作理解更深入

通过这次亲自动手做台 CPU 的实验，我深入地理解了一个实现流水的 CPU 有哪些结构，各个结构应该实现什么功能。对于我之前一直难以理解的冲突处理部分，在我们解决了数据冲突、控制冲突之后，我也有更透彻的理解，学习到了旁路转发是如何发挥作用的，冲突控制单元如何在检测到冲突之后让 PC 停止几个周期，让之前的指令执行完毕再继续执行。尤其是解决控制冲突之后，我彻底理解了对于各种类型的跳转 PC 应该停止几个周期，如何取消掉已经开始执行的指令的影响等等以前百思不得其解的问题。

在实验之前我们对制作一个 CPU 的难度没有直观的认识，甚至有同学夸下海口，能做出一个可以和 intel 媲美的主频达到 2GHz 的 CPU，现在看来他简直是一派胡言。通过一步步的实验，我们才逐渐了解到实现一个高效率流水 CPU 的难度。

6.3 培养了耐心、细心、熬夜能力

在控制冲突部分，很多问题我们都苦思冥想了很久，例如 PC 应该在什么时候停，停几个周期，要跳转的地址什么时候传给 PC，PC 的选择信号什么时候产生，要保持多久，已经开始执行的下一条指令的影响如何清除掉，如何给后续阶段传清除信号以取消已经产生的控制信号。我们一开始没有想清楚这些问题就急于动手写代码，但经过好多次实践发现程序始终运行不正确，我们一直处在拆东墙补西墙的状态，解决了一个 bug 另一个地方又会出现一个 bug。最终我们才决

定静下心来仔细想清楚这些问题再开始写代码。实验后我们深刻领悟到，写代码一定要细心，调试一定要耐心。

6.4 提高解决问题的能力

实验中我们遇到了大量的问题，有些问题十分离奇古怪，需要我们通过大胆的设想来猜测出现问题的地方。有些问题是别同学已经遇到过的，比如在最开始运行监控的时候，我们的 CPU 的频率很低，导致 U 指令从串口读入的时候始终错误。石伟男花费了好几个小时尝试解决这个问题，但一直无果，直到他请教了袁泰凌同学才恍然大悟，因为袁泰凌同学也遇到过类似错误。有些问题是我们之前遇到过的，但出现的问题症状不一样，所以我们一开始并不确定是否是由于之前的那个原因导致的，比如解决外部中断的时候，我们让程序在死循环内持续输出字符‘H’，程序始终不能正确运行。后来我们想到串口以前也出现过问题，这次可能也是由于在短时间内给串口传递了太多数据导致串口部分出现问题。

总之在这个实验中，出现的问题千奇百怪，原因也各种各样，我们需要动用各种资源，做出各种猜想来尝试解决问题，实验过后我们解决问题的能力得到了大大提高。

6.5 永不放弃

“不轻言放弃”，说来容易做来难。在检查作业的当天凌晨，我们小组尝试通宵刷夜赶工，但是在早上 5 点的时候高越率先顶不住了，回到宿舍睡觉，后来王轩也暂停工作回去睡觉。这时候，正印证了那句古话，“英雄总是诞生在危机时刻”，小组内的主力核心战将石伟男挺身而出，在队友都吹头丧气的时候，他沉着地念了两句诗“苟利国家生死以，岂因祸福避趋之”，之后就继续战斗。他将小组扛在肩上，没有丝毫放弃的念头，也没有任何沮丧的情绪，保持昂扬的斗志、永不服输的精神和胸有成竹的自信，以饱满的情绪埋头苦干，终于在清晨的第一缕阳光照进 ZJ2#408 的时候，在 VGA 上成功跑起了 R 指令，得知这个喜讯之后，我们都感到非常 Excited，双双从床上跳下继续赶工。Time flies very fast.

终于在下午两点钟检查之前，让 5 条指令都能脱离 term 在 VGA 上成功运行，完成了 VGA 模块的任务。事后，作为小组内最先放弃的败类，高越进行了深刻的忏悔，也表达了对石伟男永不放弃迎难而上刻苦拼搏精神的由衷的崇敬钦佩之情。

6.6 团队协作

在开始写代码之前，我们小组进行了明确的分工，但是苦于板子只有一块，我们不能同时利用，所以在一个人用板子进行调试的时候其他两人只能默默陪同，导致整个小组工作进展缓慢。后来组内的石伟男展现了他超越常人的伟大品质，他不顾自己身体可能遭受重大伤害，为了小组的利益，毅然决定改变自己的作息，利用夜晚和上午的时间进行工作，下午的时间进行短暂休息，而在他休息的片刻组内另两位成员利用板子进行调试。在使用了这种分时的流水线调试方式之后，我们组的进度飞快提升。石伟男秉着舍己为人的精神顾全大局而委屈小我，对于他做出的这些牺牲，我们都是很感动的。

6.7 乐观自信的态度

开始的时候我们班有很多组都决定做不用 term 的键盘和 VGA 模块，在最后一天晚上，另一组同学和我们一起熬夜，调试显示器。在遭遇百般挫折之后，凌晨两点他们放弃了这个功能。当时我们的调试也是持续了很久但进展缓慢，我们非常不确定自己能做出这个功能。在我们信念产生动摇的时候，石伟男展现了他的领袖气质，他鼓励我们，要做一个优秀的 CPU 就不能抱着应付作业的态度，我们做的 CPU，一定是敢同 intel 争高下，不向 mips 让寸分，就是要有这种精神。听了他的充满激情的演说，我们备受鼓舞，继续奋战，终于成为全班唯一做出不用 term 只靠键盘和显示器就能运行监控程序的小组。