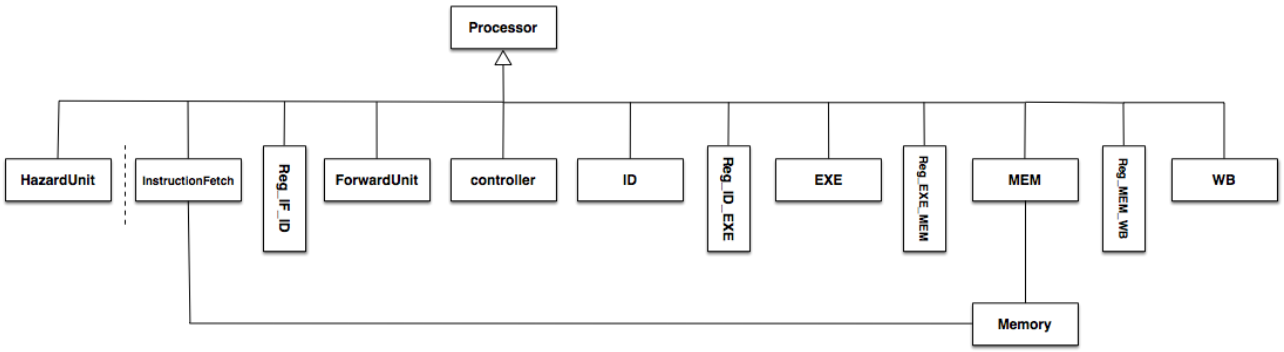
计算机组成原理 大实验实验报告

黄予 2013011363
茹逸中 2013011375
杨晓成 2013011383

一，整体结构

我们实现了 **25M** 的流水线处理器。

每条命令的处理由 4 个阶段构成，它们依次是 **IF**（取指阶段），**ID**（寄存器取值阶段），**EXE**（计算阶段），**MEM**（访问内存阶段），**WB**（写回寄存器阶段）。立即数扩展同样在 ID 阶段处理。IF 和 MEM 段段内存访问和串口访问由一个统一模块 Memory 控制。使用模块 HazardUnit 实现加气泡和中断，使用模块 FowardUnit 实现数据旁路。



结构图如下：（数据流向为自左向右）

功能简述：

Reg_IF_ID & Reg_ID_EXE & Reg_EXE_MEM & Reg_MEM_WB 模块间锁存器

HazardUnit 出现结构冲突或读内存时，增加一个气泡；处理中断

———IF 段—————

InstructionFetch 保存 PC 值，处理跳转，解决了控制冲突

———ID 段—————

ForwardUnit 控制数据旁路，提供数据选择的信号，解决了数据冲突

controller 分析指令，为其他模块提供控制信号

ID 寄存器读写，立即数扩展，计算跳转指令的跳转地址

———EXE 段—————

EXE ALU 计算

———IF & MEM 段—————

为了处理结构冲突，我们将对串口，SRAM1 和 SRAM2 的读写统一由一个模块 Memory 控制

Memory 串口和 SRAM 读写

———WB 段—————

WB 数据写回寄存器

基础指令：

ADDIU ADDIU3 ADDSP ADDU AND B BEQZ BNEZ BTEQZ CMP JR LI LW LW_SP MFIH MFPC MTIH MTSP NOP OR SLL SRA SUBU SW SW_SP

扩展指令：

JRRA JALR CMPI MOVE SLTUI

二，团队分工

数据通路设计**********杨晓成
控制指令设计**********茹逸中
————
Processor 综合**********杨晓成
Processor 数据旁路**********黄予
————
Reg_IF_ID&Reg_ID_EXE&Reg_EXE_MEM&Reg_MEM_WB****黄予
HazardUnit**********黄予
———IF 段————
InstructionFetch**********杨晓成
———ID 段————
ForwardUnit**********黄予
controller**********茹逸中
ID**********黄予
———EXE 段————
EXE**********茹逸中
———IF & MEM 段————
Memory**********杨晓成
串口控制**********茹逸中
———其他功能————
中断**********黄予
Flash**********茹逸中
键盘**********杨晓成
VGA**********茹逸中
打字小游戏**********杨晓成

三，详细功能

1. 数据通路设计 文件名 DataPath.png & DataPath.graffle & DataPath.vdx

1) 基础部分 **数据流（黑色，部分紫色） 指令流（蓝色）**

① PC 经由锁存器，当**上升沿**来临时+1; 根据 PC 值取出指令

②根据指令产生控制信号

③立即数扩展

符号扩展：

指令 3-0 位

指令 4-0 位

指令 4-2 位

指令 7-0 位

指令 10-0 位：仅跳转指令 B 使用

零扩展：

指令 7-0 位

④指令 10-8 位，寄存器 rs 取值，记为 RD1

⑤指令 7-5 位，寄存器 rt 取值，记为 RD2

（注：指令 4-2 位，rd 的取值，由②进行处理，表现为控制信号 RegDst，4bits）

⑥ALU 的操作数 1Src1 选取：

RD1

RD2

IH

SP

PC+1(即 RPC)

⑦ALU 的操作数 2Src2 选取：

RD2

立即数选取：

由指令 3-0 位**符号扩展**得来：ADDIU3

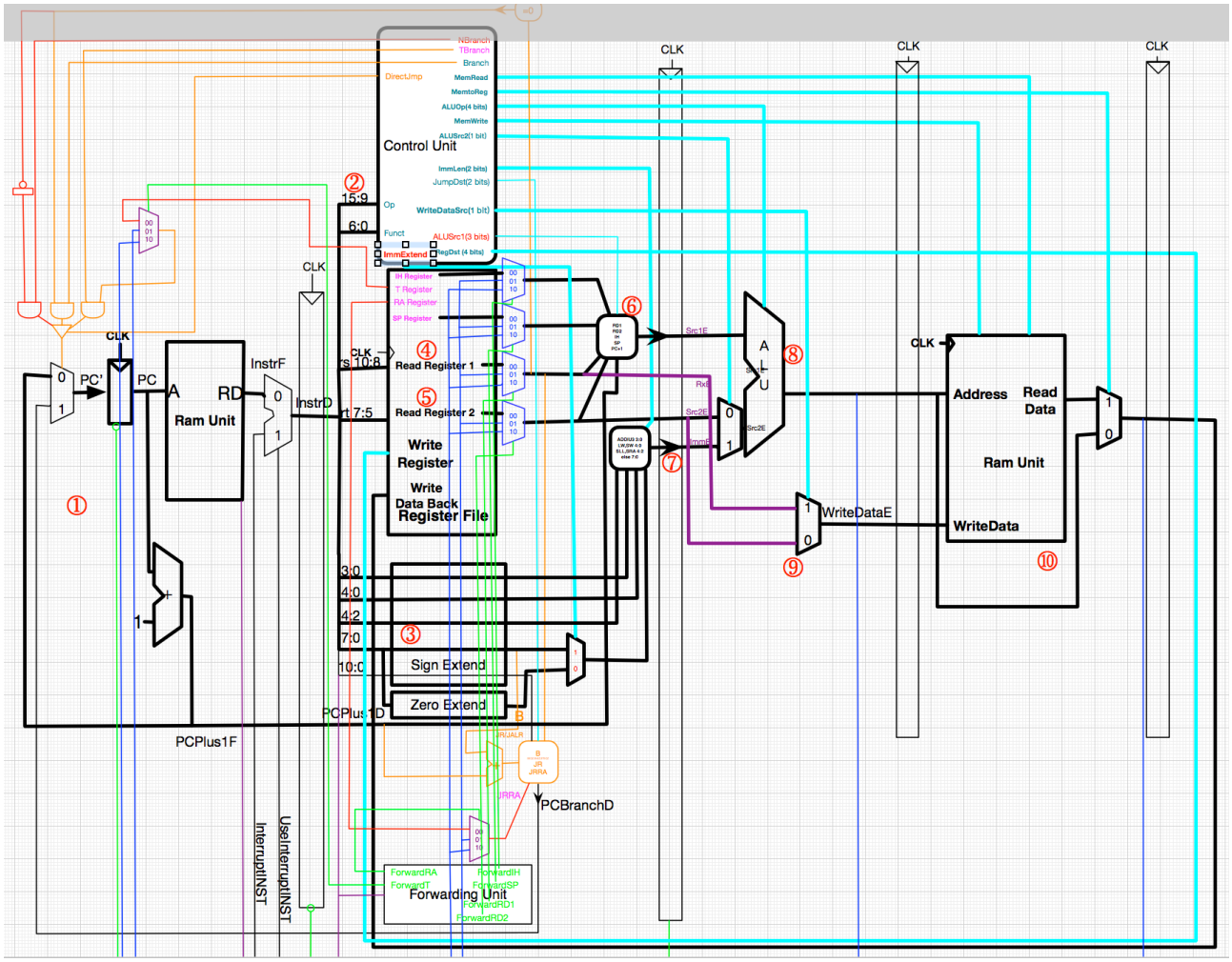
由指令 4-0 位**符号扩展**得来：LW，SW

由指令 4-2 位**符号扩展**得来：SLL，SRA

其余由指令 7-0 位**扩展**得来，扩展方式由控制信号 ImmExtend 控制

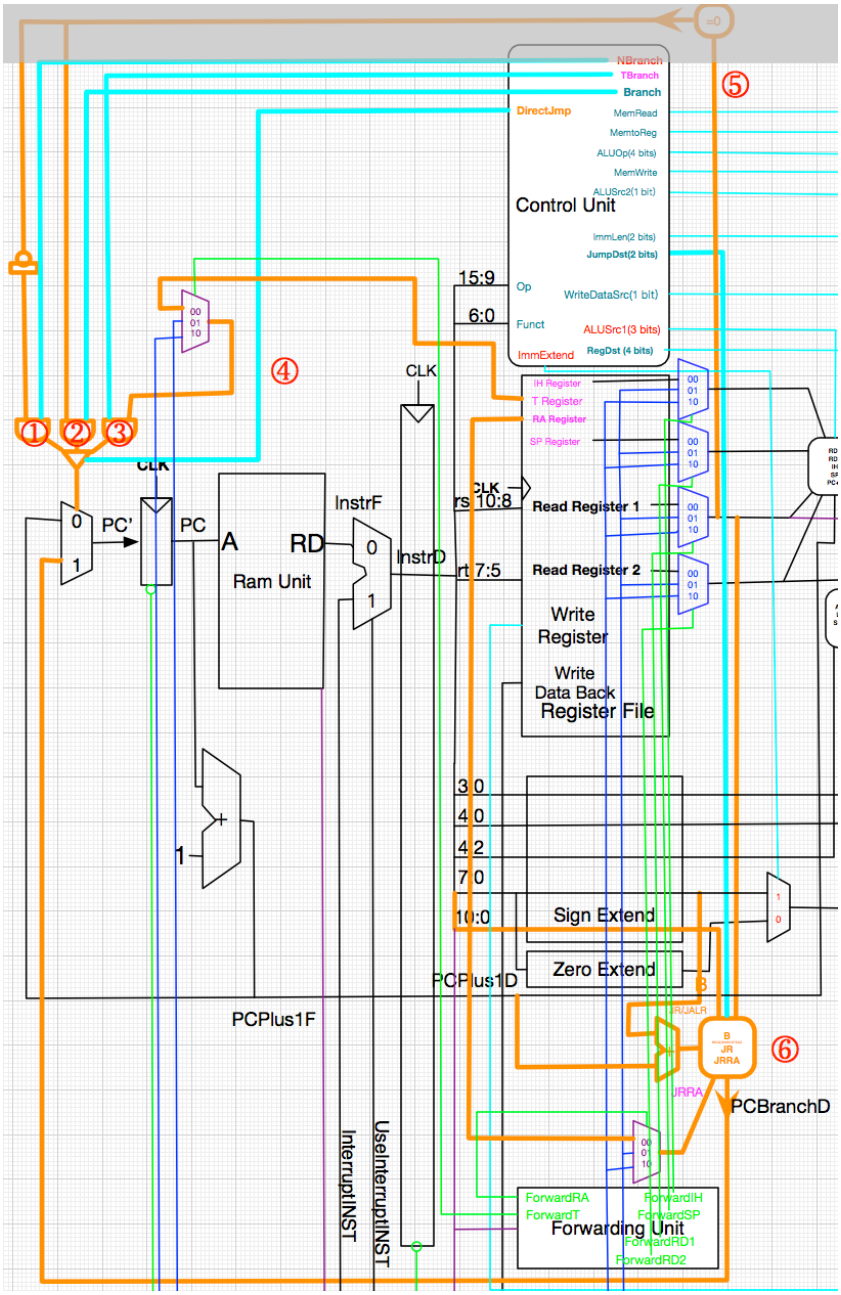
⑧ALU 计算

⑨写入内存的数据选取:



RD1
Src2
由控制信号 WriteDataSrc 控制
⑩内存访问部分
内存读写控制信号 MemRead，MemWrite
最后写回的数据选取：
从内存中读出的数据
ALU 结果
由控制信号 Mem2Reg 控制

2) 跳转部分 数据流（橙色） 控制信号（蓝色）



①指令 BNEZ
(1). RD1 != 0
(2). NBranch == 1

②指令 BEQZ
(1). RD1 == 0
(2). Branch == 1

③指令 BTEQZ
(1). RD1 == 0
(2). TBranch == 1

④指令 B, JR, JRRA
控制信号 DirectJmp

当满足以上 4 个条件之一时，修改 PC 值，执行跳转

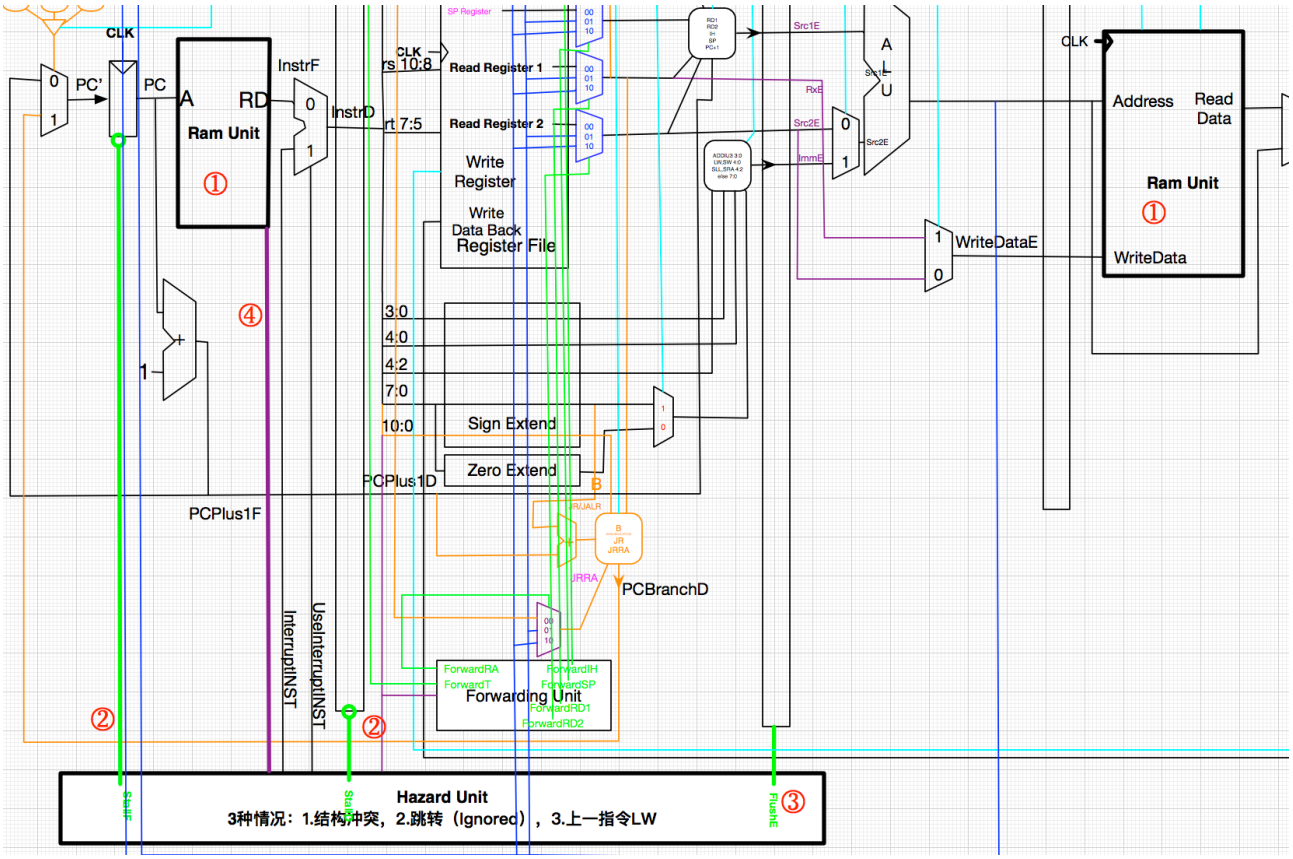
⑤判断 RD1 是否为 0

⑥跳转的目的地址选取：
指令 10-0 位**符号扩展**：B 指令
指令 7-0 位**符号扩展**+(PC+1)：BEQZ，BNEZ，BTEQZ 指令
RD1 值：JR 指令
RA 寄存器值：JRRA 指令

总结：我们的跳转模块集中于 IF 和 ID 阶段，保证了在跳转完成前最多只额外执行 1 条指令，该指令在跳转指令的延迟槽内执行。

3) 结构冲突处理 数据流（黑色） 控制信号（紫色或绿色）

解决方式为增加一个气泡。此时，②处 Stall 信号和③处 Flush 信号均置为 1。



① 我们以 SRAM1 为数据区（地址范围 8000~FFFF），SRAM2 为指令区（地址范围 0000~7FFF）。由于串口与 SRAM1 共享同一总线，如此可以保证在访问串口和数据时，对指令的按时钟依次访问不会被打扰。但是，当需要对 0000~7FFF 的内存进行读写时，就必须终止在本周期内对指令的读取。解决方式为增加一个气泡。

我们将 IF 段的从 SRAM2 中取指和 MEM 段的内存访问统一由模块 Memory 处理。

②Stall 信号，上升沿来临时，若 Stall 为 1，锁存器的值不会改变。

③Flush 信号，上升沿来临时，若 Flush 为 1，锁存器的值均置为 0。设计中，我们保证了当信号数据全为 0 时，不会进行任何操作。

②③即为我们加气泡的方式。我们通过加气泡，将之后的指令延迟一个时钟周期执行，从而解决了结构冲突问题和数据旁路来不及的问题。

模块 Hazard Unit 输入有两处，一处为当前指令，对应情形 2；另一处④为当 Memory 模块（即图中 Ram Unit）发现 InstructionFetch 和 MEM 需要同时访问同一片内存时，所发送的信号。

4) 数据旁路 数据流（深蓝色） 控制信号（绿色）

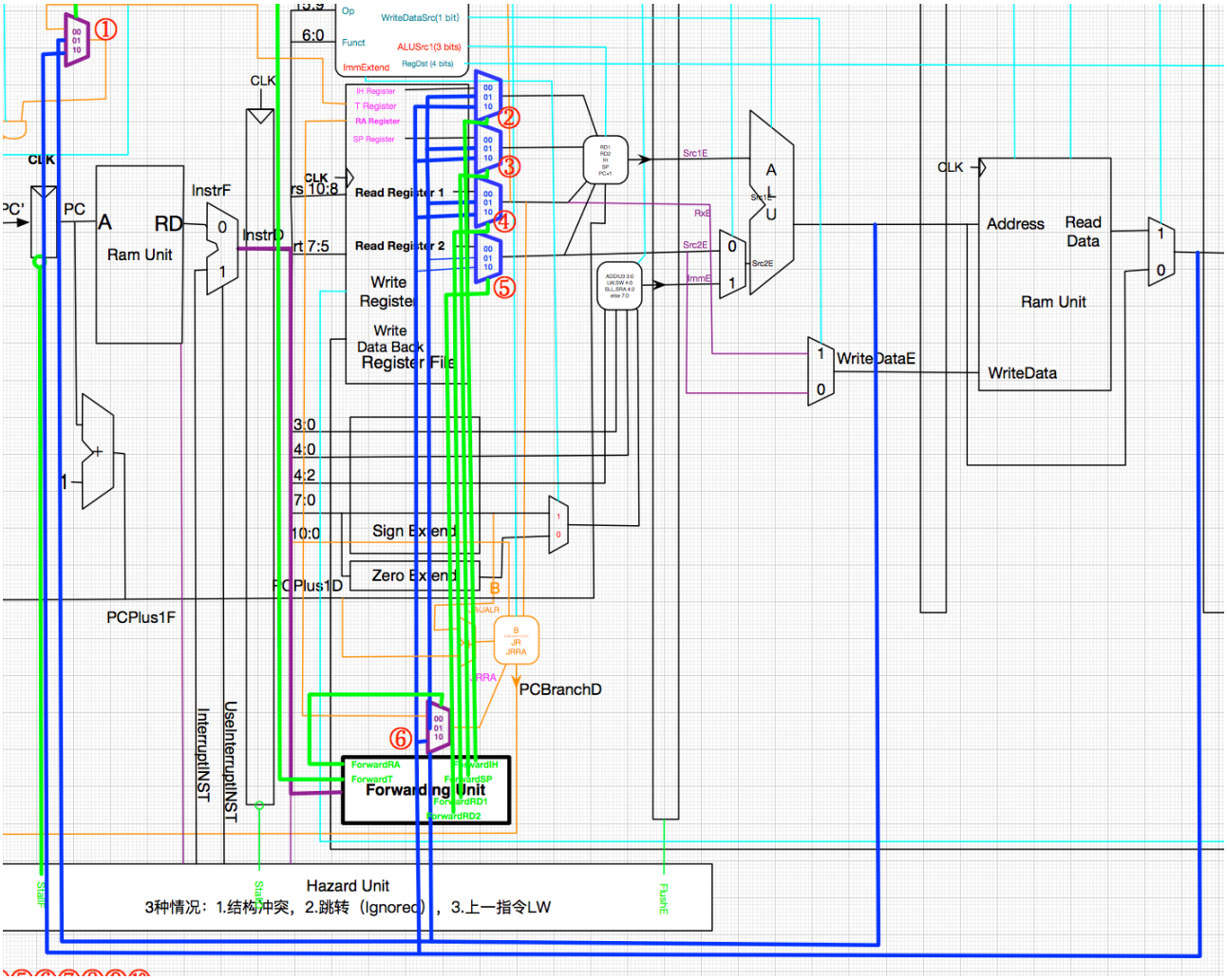
由于并行，在下一条指令取出寄存器的值（ID 段）可能是错误的，若上一条指令改变了寄存器的值而没有写回（未经过 WB 段）。正确的寄存器数据可能在以下位置：

1. 恰为从寄存器堆（Register File）中取出的值。即前两条指令不会对该寄存器的值进行修改。
2. EXE 段作为 ALU 的结果。
3. MEM 段二选一之后的值，将在下一上升沿来临时传给 WB 段作为写回到寄存器的值。

由此我们设计了数据旁路。为以上数据的三选一。控制信号由 Foward Unit 控制。

① T 寄存器数据旁路

② IH 寄存器数据旁路



③ SP 寄存器数据旁路
④ RD1 寄存器数据旁路
⑤ RD2 寄存器数据旁路
⑥ RA 寄存器数据旁路

2. 控制指令设计
详见文件 InstructionAnalysis.xlsx Sheet1~4

———————

3. Processor 综合 文件 Processor.vhd
详见“一，整体结构”中数据流图

4. 数据旁路

a) 旁路控制器（ForwardUnit）：

旁路控制器位于 ID 阶段，负责根据 ID 阶段指令与 EXE 阶段指令、MEM 阶段指令的读写寄存器冲突情况，给出对于寄存器值的三选一选择信号，分别选择寄存器堆中的原值、位于 EXE 阶段的待写回寄存器的值、位于 MEM 阶段的待写回寄存器的值。

由于所有阶段锁存器为时钟上升沿写入，寄存器堆为时钟下降沿写入，故在当前时钟周期的中间时刻（时钟下降沿）寄存器堆将得到更新，选择信号无需选择 WB 阶段的待写回寄存器的值。

b) 旁路选择器：

位于 ID 阶段，负责根据旁路控制器给出的控制信号选择正确的寄存器值。由于计算跳转指令的跳转地址以及判断是否跳转位于 ID 阶段，要求 ID 阶段取出的寄存器值应为正确值，而 EXE 阶段的计算速度与 MEM 阶段的读取内存速度足够快，所以我们将所有的旁路选择器置于 ID 阶段统一管理，逻辑简单而高效。

5. 阶段锁存器（REG_IF_ID & REG_ID_EXE & REG_EXE_MEM & REG_MEM_WB）：

阶段寄存器负责暂存数据与控制信号，在时钟的上升沿进行更新，在检测到异步清零信号时将所有锁存器清零。此外，IF 阶段与 ID 阶段之间的锁存器（REG_IF_ID）接入 stall 信号，当检测到 stall 信号时所有锁存器的值保持不变；ID 阶段与 MEM 阶段之间的锁存器（REG_ID_EXE）接入 flush 信号，当检测到 flush 信号时将所有锁存器的值清零。

6. HazardUnit

1) 暂停流水线

HazardUnit 负责根据需要插入气泡以暂停流水线，进行的操作为：

1. 向 PC 锁存器发送 stall 信号，使其在下一个时钟上升沿保持原值。
2. 向 REG_IF_ID 阶段锁存器发送 stall 信号，使其在下一个时钟上升沿保持原值。
3. 向 REG_EXE_MEM 阶段锁存器发送 flush 信号，使其在下一个时钟上升沿清零。

这样当下一个时钟上升沿来临，位于 ID 阶段之后的指令可继续执行，而位于 ID 阶段以及 ID 阶段之前的指令将暂时停留于所在的阶段。

触发暂停流水线的条件为：

1. 当存储指令的内存需要外部读写，发生结构冲突。
2. 当前位于 EXE 阶段的指令需要读取内存到寄存器，而 ID 阶段的指令需要使用相同的寄存器。

2) 中断处理

详见“14. 中断”

———IF 段—————

7. InstructionFetch 文件 InstructionFetch.vhd

1. 保存着一个 PC 值，在每个时钟周期，根据 PC 值访问 SRAM1 中对应的地址，将其中的指令取出。有一个锁存器，在上升沿更新 PC 值。详见“三，详细功能 1.数据通路设计 1)基础部分①”。

2. 跳转的处理，解决控制冲突。详见“三，详细功能 1.数据通路设计 2)跳转部分 ①~④”

———ID 段—————

8. ForwardUnit

详见“4.数据旁路 a)旁路控制器”

9. Controller

控制器(CU)

控制器主要用来分析指令，发出控制信号来控制各部件协作执行该指令。

控制器为纯组合逻辑，其输入为一条 16 位指令，输出为个控制信号的值。

以下表格列出了个控制信号以及它们的功能和取值。

信号名	位数	功能和取值
Branch	1	当前指令为 BEQZ 时取 1，用于控制跳转条件。
NBranch	1	当前指令为 BNQZ 时取 1，用于控制跳转条件。
TBranch	1	当前指令为 BTEQZ 时取 1，用于控制跳转条件。
DirectJmp	1	当前指令为 B,JR,JRRA 时取 1，用于无条件跳转。
MemRead	1	当前指令需读内存时取 1，用于控制 RAM 读写。
MemWrite	1	当前指令需写内存时取 1，用于控制 RAM 读写。
MemtoReg	1	写回寄存器的来源为内存时取 1，为 ALU 输出时取 0，用于控制寄存器的写回值。
ALUSrc2	1	用于控制 ALU 第二操作数的选取，为 ry 时取 0，为立即数时取 1。
ALUSrc1	3	用于控制 ALU 第一操作数的选取，RX 取 0，SP 取 1，IH 取 2，PC 取 3，RY 取 4。
ALUOp	4	用于控制 ALU 操作符，取值在 ALU 部分介绍。
ImmExtend	1	用于控制立即数的扩展方式，零扩展取 0，符号扩展取 1。
ImmLen	2	用于控制立即数在指令中的位置，[0,3]取 0，[0,4]取 1，[2,4]取 2，[0,8]取 3。
JumpDst	2	用于控制跳转地址，B 取 0，BNQZ/BTNQZ/BEQZ 取 1，JR 取 2，JRRA 取 3。
RegDst	4	用于控制写回的寄存器，8 个通用寄存器取[8,15]，不写回取 0，SP 取 1，IH 取 2，T 取 3。

10. ID 模块

1) 寄存器堆（reg_controller）：

寄存器堆包含 8 个通用寄存器 R0~R7，4 个特殊寄存器 SP、IH、T、RA，在时钟下降沿写回相应的寄存器，在检测到异步清零信号时将所有寄存器清零。

2) 立即数扩展（SignedExtend & ZeroExtend）：

指令集的立即数共分为 5 种，分别为指令的第 3 位到第 0 位、第 4 位到第 0 位、第 4 位到第 2 位、第 7 位到第 0 位、第 10 位到第 0 位，扩展器同时对这 5 种立即数进行扩展。为保证速度，立即数扩展与指令译码并行进行，最后再由指令译码产生的控制信号对扩展结果进行选取。

3) 计算跳转地址

位于 ID 阶段，根据控制信号进行选择。跳转指令的地址计算可分为 4 种：

a) B 指令：指令的第 10 位到第 0 位立即数扩展与（PC+1）相加作为跳转地址。
b) BEQZ、BNEQ、BTEQZ 指令：指令的第 7 位至第 0 位立即数扩展与（PC+1）相加作为跳转地址。
c) JR 指令：读出的普通寄存器中的值作为跳转地址。
d) JRRA 指令：读出的特殊寄存器 RA 中的值作为跳转地址。

计算跳转地址与指令译码同时进行，最后再由指令译码产生的控制信号对结果进行选取。

－－－EXE 段－－－－

11. EXE

算术逻辑单元(ALU)

算术逻辑单元主要用来进行算术和逻辑的运算，除跳转指令以外的所有指令都需要算术逻辑单元的运算结果。

算术逻辑单元是纯组合逻辑。输入为经过选择器选择后的两个运算数（其中可能有一个运算数是无效的）以及由控制器提供的 ALU 操作符。输出为相应的运算结果，用来写回寄存器或者内存。

以下表格列出了 ALUOp 的各个取值时 ALU 的输出。

ALUOp	输出
0	0
1	无意义
2	Src1+Src2
3	Src1-Src2
4	Src1=Src2 时输出 1，否则输出 0
5	Src1!=Src2 时输出 1，否则输出 0
6	Src1<Src2 时输出 1，否则输出 0
7	无意义
8	Src1 & Src2
9	Src1 | Src2
10	Src1 << Src2
11	Src1 >> Src2
12	Src1

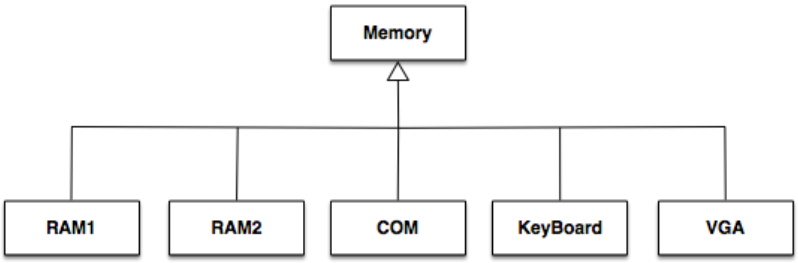
13	Src2
14	Src1 + 1
15	无意义

－－－IF & MEM 段－－－

12. Memory

文件 Memory.vhd

功能如下：



处理读写信号 ReadSignal，WriteSignal，使能信号 ramEN，地址数据 addr，输入数据 dataIn，输出数据 dataOut。将这些信号定位到不同的子模块或局部变量。

子模块 Ram：

ramEN：使能时为 0，否则为 1。

ramOE：控制读取 RAM，需要低电平才能读取。当 ReadSignal 为 1 时，ramOE<=clk。

ramWE：控制写入 RAM，需要先高电平数据准备时间，后低电平数据写入时间。当 WriteSignal 为 1 时，ramWE<=clk。

13. 串口控制

串口是实验用计算机上最基础的输入输出设备，计算机可以通过串口读入或输出数据。为控制串口，将串口数据映射到内存上的 BF00 地址的后 8 位，将串口状态映射到内存上的 BF01 地址的后 2 位。

内存中的 BF01 地址上保存串口的当前状态。当它最后一位为 1 时，表示此时串口可写；当它倒数第二位为 1 时，表示此时串口可读。

当串口可写时，向 BF00 地址中写入数据即可写入到串口。写入的数据只有低 8 位有效，高 8 位数据会被忽略。当串口可读时，从 BF00 地址读取数据即可读取串口，读取的数据只有低 8 位有效，其它的会被置为 0。

－－－其他功能－－－－

14. 中断

中断处理集成于 HazardUnit 中。当检测到中断信号时，HazardUnit 向 PC 锁存器发送 stall 信号，使 IF 阶段暂停取指令操作，还将插入一系列新指令取而代之，使得 CPU 能顺利跳转至中断处理程序继续执行。插入的指令为：

1. 暂存寄存器 R6 的值，以便在接下来的指令中使用 R6。
2. 将中断号压栈
3. 将中断处理程序执行完后需要跳转回的地址压栈
4. 跳转至中断处理程序（由中断处理程序负责恢复 R6 的值）

15. Flash

Flash 引导开机

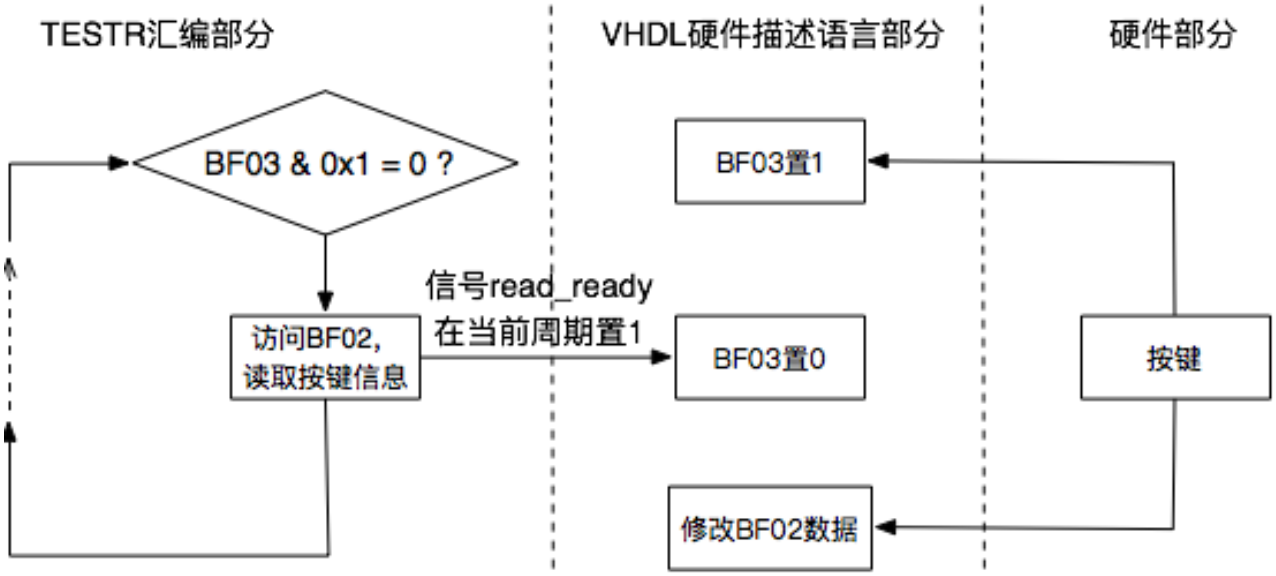
Flash 是实验用计算机上唯一一个断电后仍然能保存数据的设备，因此我们需要将监控程序的机器代码预先存入 Flash 中，在每次开机时计算机可以自动从 Flash 中载入监控程序。

Flash 的读取速度较慢，其时钟频率被设置为 12.5MHz。

开机时，开机信号置为 1，其它设备均停止运作，并将 Flash 中前 2048 字节的数据依次写入 Ram2 中。写入完毕后，开机信号置为 0，此时 Ram2 中已经存有监控程序的所有代码，计算机可以正常运行。

16. 键盘 文件 KeyBoardDriver.vhd

作为 Memory 的子模块，按键信息存于 Memory 的局部变量 BF02，BF03 中。访问时内存映射地址亦分别为 0xBF02，0xBF03。



17. VGA

(a) 像素显示

计算机以 25MHz 的频率对屏幕进行扫描。包括消隐区在内，屏幕的总大小为 800*525。因此 VGA 显示大约能达到 59~60Hz，人眼无法分辨。

每个像素的颜色由 R,G,B 三个分量表示，每个分量用 3 位存储，共有 8 种取值。因此一个像素点的颜色需要用 9 位存储，共可以表示 512 种颜色。

(b) 显示字符存储

屏幕的分辨率为 640*480，每个字符的大小为 8*16 个像素，因此在整个屏幕上共可以显示 80*30=2400 个字符。为了能让 CPU 管理这 2400 个字符，将 A000~AFFF 这 4096 个内存地址分配给屏幕上字符的存储，共计 12 位。其中高 5 位表示行号，低 7 位表示列号。行号超过 30 或者列号超过 80 的地址是无效地址。这些字符存储在一块 FPGA 上的板载块状内存上。这些数据只能写，读取时会返回无效值。

每个字符显示内存地址中保存了 16 位的数据，其中低 7 位表示字符的 7 位 ASCII 码，高 9 位表示颜色。在这 9 位中，高 3 位表示颜色的 R 分量，中 3 位表示 G 分量，低 3 位表示 B 分量。

(c) 字符显示

计算机以 25MHz 的频率对屏幕进行扫描，每一个时钟周期扫描一个像素。在这个时钟周期内，计算机需要计算出该像素点的颜色值并通过 RGB 三个端口输出。在计算像素时，首先需要确定当前

像素点落在哪一个字符坐标上，然后读出该字符坐标的字符及其颜色。之后查询字符格点表，判断该格点是否存在。若不存在，则 R,G,B 均为 0，否则为该字符的颜色。

18 打字小游戏
原始文件 typingame.s
嵌入系统文件 kernel_new_board.s

运行方式：打开 Term，连接串口，VGA 和键盘，输入指令 G 5000 开始游戏。按键盘 ESC 键退出。

游戏方式：初始时屏幕上会显示白色的 a-h 字母。按字母对应按键，对应的字符消失，同时在显示屏随机位置产生一个随机颜色的随机字符。

原理：使用寄存器 R4，R5 用于随机位置，字符和颜色。每当程序执行一次 TESTR（检测是否有键盘输入），R4，R5 分别加上一个质数。

0xC080 存储循环开始地址，用于返回主循环。

0xC000~0xC07f 存储 128 个 ASCII 码是否存在及对应的位置信息。

0xD000~0xEFFF 为映射到 VGA 的信息，包括当前位置是否有字符及字符颜色。

四，问题反馈

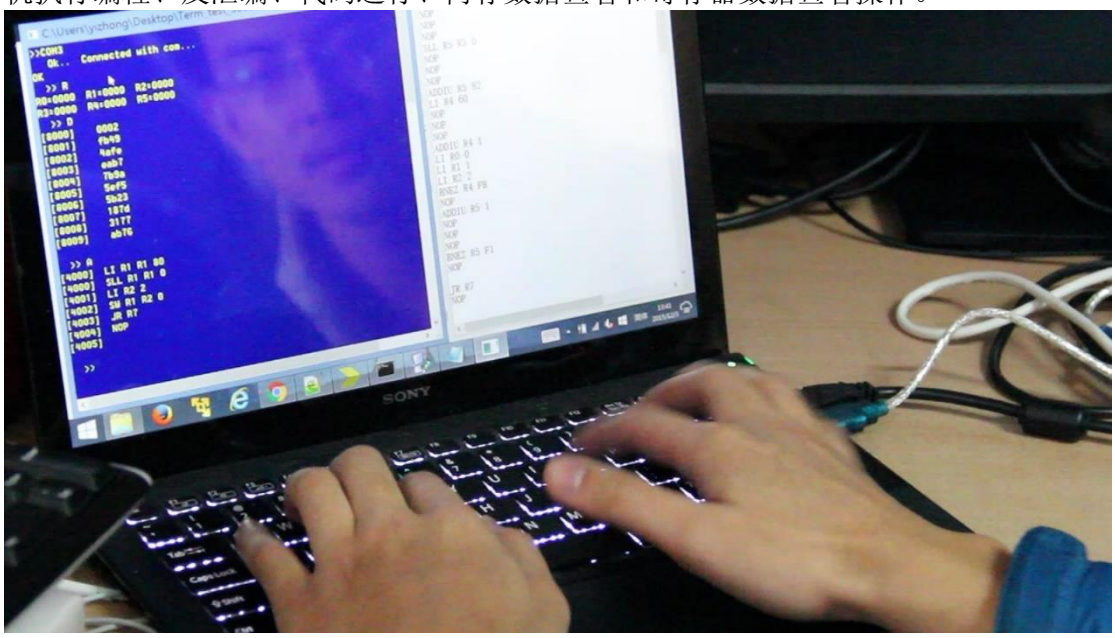
对 kernel 进行的改动

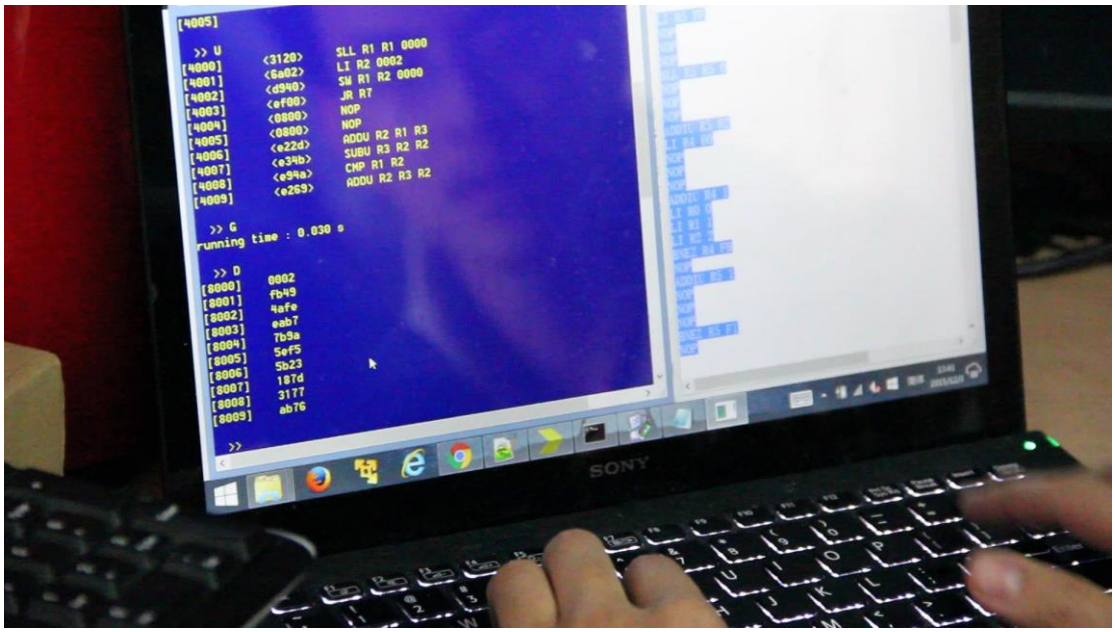
将栈顶指针 SP 由初始化为 BF10 改成初始化为 BF70。

此处应为 kernel 自身的错误，系统堆栈区应为 BF10 至 BFFF，而栈是向低地址延伸的。若初始化为 BF10，则栈使用的空间为地址低于 BF10 的空间，当栈不断增长，将使用 BF01 以及 BF00 两处地址，与串口读写冲突。故将 SP 初始化为 BF70，这样预留给栈的空间为 BF10 至 BF70

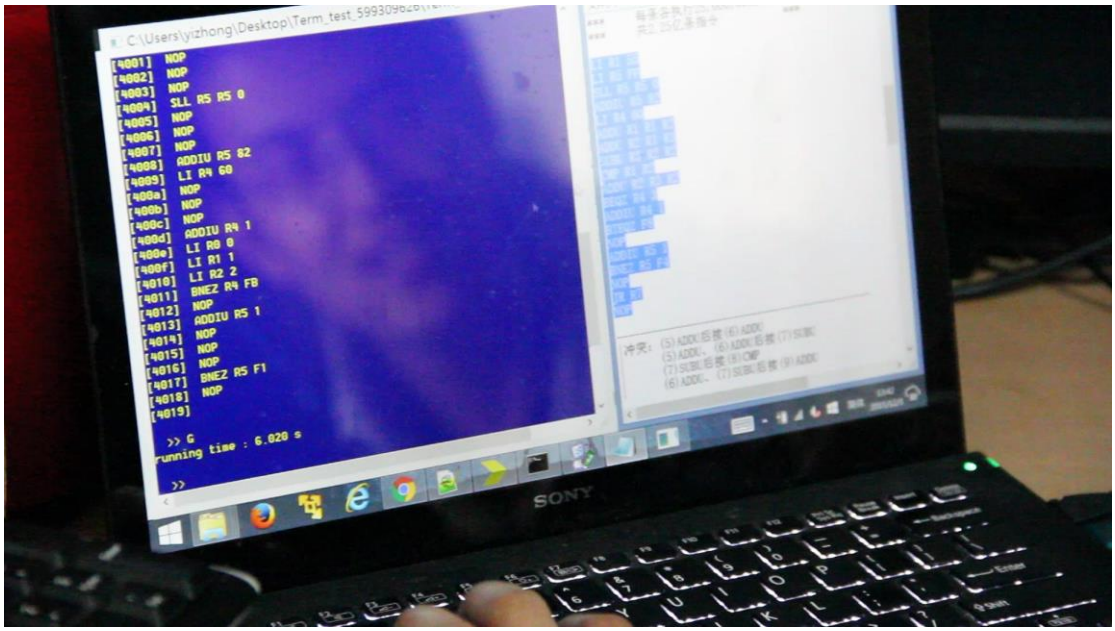
五，成果展示

1. 将串口连接到 PC 机，打开 term 软件与实验用计算机进行交互。使用 term 软件控制实验用计算机执行编程、反汇编、代码运行、内存数据查看和寄存器数据查看操作。





2. 运行 5 个效率测试程序，运行时间分别为 6s，9s，4s，7s，4s。图中为运行第一个效率测试程序的结果。

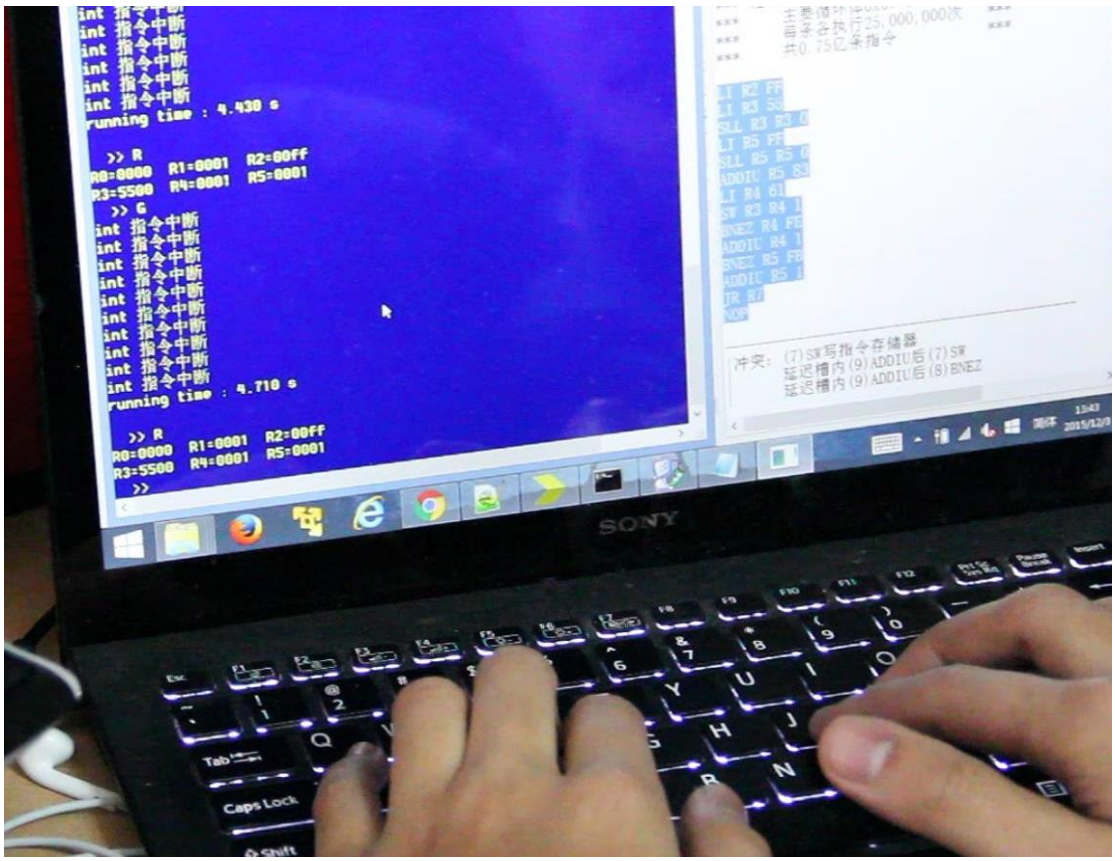


3. 中断测试

在单击 clk 按钮时会产生中断，在 term 中会显示。产生中断不会对程序运行的结果造成任何影响。

手按 clk 键实现硬中断

1. 运行程序时中断，不会影响结果
2. 打字游戏时中断，仍能正常运行



4. 打字游戏

在开始游戏时，系统会固定在屏幕上生成 8 个字符。在用键盘输入屏幕上的字符时，该字符会消失，同时在随机位置生成一个随机颜色的随机字符。键盘的输入能够正确响应，屏幕也能够正确显示字符。

六，总结

（杨晓成）

对于数据通路的设计。整个造计算机的流程中我们多次修改了流程设计。其中一部分是我们在优化我们的计算机结构，如将数据通路的三选一统一由 EXE 段移动至 ID 段，但是还有一大部分是由于预期设计的不足引起的，主要集中于 ID 的寄存器取值和 ALU 的操作数选区。一方面是我在设计时没有完整的一条条过指令，另一方面指令分析时也没有完全的对照图进行分析，以至于后期多做了很多修改图的工作。不过看着图一点点被完善，感觉还是不错的。

Processor 代码综合我认为做的很不错。首先是对于不同阶段的变量我分别用 FDEMW 做为后缀，以便于区分。同时将每一模块中输入输出的信号和数据都做了详细的注释，这些都非常有利于代码的再修改。

Memory 阶段中，遇到的最大困难就是解决 RAM 和 COM 之间的访问冲突。这里要注意两点，一是不访问 COM 时要把串口开关 rdn，wrn 关掉，二是信号在 RAM 和 COM 间的再分配要仔细分析，最好列表。将 RAM 的 OE 和 WE 信号与时钟信号同步是一个非常优秀的 idea。

键盘的驱动在于流程图的设计，如何保证同一个数据不会重复被读取，以及将内存 BF02，BF03 的重新映射，详见键盘部分。

打字小游戏为了展示键盘和 VGA 而设计，随机数的生成做的很好。

总体来看，由于软工大作业的影响，我们组开始的时间比其他组要慢一些，同时组内交流不足，这也是最后一些想做的扩展并没有完成的原因。最后完成的效果还是很满意的。

（茹逸中）

遇到的困难

(a) 串口输入输出

在刚写完串口输入输出时，串口无法工作。串口无法发送数据，也接收不到发来的数据。仔细检查和调试后发现原因是串口和 Ram1 冲突了。由于串口和 Ram1 共用同一条总线，因此在使用串口时需要将 Ram1 的使能全部置“1”（关闭），在使用 Ram1 时需要将串口的使能全部置“1”（关闭）。在修改完后串口即能正常工作了。

(b) Flash 读取问题

在刚写完 Flash 时，发现每次从 Flash 载入到 Ram2 的数据总会出现随机的错误（大约有 10%的数据时错误的），经过多次调试、降低频率等方法仍然没有办法解决。起初我以为是 Flash 本身的问题，但是更换了其它组的板子后仍然存在此问题。后来我才发现问题的所在。在开机过程中，CPU 仍在运行。由于此时 Ram2 中还是随机数据，因此 CPU 会运行随机指令，导致 Ram2 中的数据出错。我在开机过程中暂停了 CPU 的运行，问题得到解决。

（黄予）

1. 数据通路

   一开始绘制好数据通路之后自以为已经十分完善，今后只需稍作改动就能投入使用。但是当开始实现起各个部分的时候，才知道原有的数据通路存在多处问题，尤其在加入旁路控制器（ForwardUnit）与暂停流水线控制器（HazardUnit）的时候，发现之前的通路存在一些问题，例如没有考虑到跳转指令使用的寄存器值还没有写回等等。

   经验教训是，在数据通路绘制完成后应该系统地将所有指令的执行流程过一遍，在动手打代码之前尽量考虑到所有的设计细节，磨刀不误砍柴工。这样可以避免不断修改已有的代码，从而从整体上提高开发效率。

2. 单元测试

   当写好代码后我们对每条指令进行了单元测试，解决了非常多的位于 controller 模块的 bug。在基本确定每条指令都能正确执行之后，我们才尝试运行 kernel 程序，因此我们在调试运行 kernel 的时候并没有太多困难，半个小时之内就能让 kernel 打出 OK 字样，在排除了几个之前没有发现的小 BUG 之后又迅速调通了 R、D、A、U、G 五条指令。由此可见单元测试的重要性，若是略过这一阶段直接运行 kernel 程序，那么我们需要先在 kernel 程序中一步步定位出错的位置再去解决 bug，耗费的时间远多于单元测试所占的时间，得不偿失。

3. 团队协作

在开始写代码之前，我们小组进行了明确的分工，使用 git 进行代码管理，由于 git 只能管理文本，因此我们还使用了有道云平台共享其他文件。

在写好代码之后的调试阶段，由于板子只有一块，三个人一起调试的工作效率并不会比一个人高太多，因此我们组内一个人进行 CPU 的调试，另外两个人开始写扩展功能，当 CPU 正常可以工作后，扩展功能部分也基本完成。

由此可见良好的团队协作可以节约大量的时间，充分提高团队的工作效率，避免出现 1+1 < 2 的合作悲剧。

不足之处在于，在调试初期，当发现了问题之后，我们的关注点在于是谁的代码出了问题，而不是问题本身。人无完人，码无完码，代码出错是一件非常正常的事情，与其将时间浪费在团队成员间的互相责备上，不如将集中精力把问题解决。

4. 对流水线的工作原理理解的更加深入

“纸上得来终觉浅，绝知此事要躬行”。通过亲自动手写出一个 CPU，能够更加深入地理解一个 CPU 应该有哪些结构，各个结构应该实现什么功能。之前在课上一直难于系统地理解控制信号与数据如何通过阶段寄存器进行传递并在下一个阶段发挥作用，各种冲突如何解决，数据旁路如何发挥作用，如何暂停流水线，如何在中断来临时跳转至中断处理程序等等，在经过这次大实验之后都对其有了宏观以及微观的了解。