

支持指令流水的 THCO-MIPS 计算机系统的设计与实现

303 组

钱桥

许建林

计 13 班

计 13 班

2011011242

2011011238

2013.12.10

目录

支持指令流水的 THCO-MIPS 计算机系统的设计与实现	1
一、实验目标	4
二、主要模块设计	4
2.1 流水段设计	4
2.2 数据通路	5
2.3 指令流程	5
三、主要模块实现	7
3.1 CPU 总体概述	7
3.2 CLK MODULE	7
3.3 PC Register	8
3.4 多路选择器 MUX	8
3.5 RAM1 Visitor	8
3.6 RAM2 Visitor	9
3.7 PC Adder	9
3.8 IF/ID Register	9
3.9 Imm Extend	10
3.10 adder	10
3.11 Hazard Detector	10
3.12 Controller	11
3.13 Common Register	12
3.14 Comparator	12
3.15 ID/EXE Register	13
3.16 Alu	14
3.17 Special Register	14
3.18 Forward Unit	15
3.19 EXE/MEM Register	15
3.20 MEM/WB Register	16
3.21 各模块之间的接口、调用关系及组合	16
四、扩展功能	16

4.1 结构冲突	17
4.2 数据冲突	17
4.3 控制冲突	17
4.4 多种冲突汇聚	18
五、实验成果展示	18
六、实验心得体会	21

一、实验目标

计算机硬件系统由中央处理器（CPU）、存储器、输入输出设备等部件组成。在本次试验中，我们将分析计算机系统的基本组成、运行原理和协同工作机制，分析计算机组成对系统性能的影响，并设计一台简单的计算机，建立计算机整体系统的概念。我们将在 THINPAD 教学计算机上，利用硬件描述语言（VHDL），设计中央处理器，调度教学计算机的各种设备，并且最终运行监控程序，作为一个简单的操作系统，并且支持运行用户程序。

我们的最终实现目标是支持指令流水的计算机。而这一整体目标的实现分为一下几个阶段：

1. 分析 THCO-MIPS 指令系统的 25 条基本指令以及 5 条扩展指令，为每条指令划分流水段，分析其执行过程。
2. 确认在指令流水执行的过程中，可能存在的各种冲突，设计相应的控制信号合理调度各个部件，以使程序能够正确连续地运行。
3. 利用 VHDL 语言，实现支持指令流水的 CPU，并且运行监控程序。

二、主要模块设计

2.1 流水段设计

我们采用了经典的 MIPS 流水线，将一条指令的运行周期分为取指（ID）、译码（IF）、运算（EXE）、访存（MEM）和写回（WB）五个周期。

在取指（ID）阶段，取指器会通过 PC 的值取到相应的指令，同时加法器会将 PC 得值加 1，以便下一次取指可以访问到正确的地址。

在译码（IF）阶段，控制器会根据指令产生控制信号。之后的三个周期中，ALU、寄存器、访存器等组件都是根据控制信号来决定如何操作的。这样做的好处是，只需对指令进行一次解析，减小编码复杂度。

在运算（EXE）阶段，ALU 根据控制器产生的运算符 OP 和操作数 A、B 进行运算，并将运算结果输出。

在访存（MEM）阶段，访存器根据控制器产生的控制信号决定是读内存还是写内存，以及读、写地址、来源等。

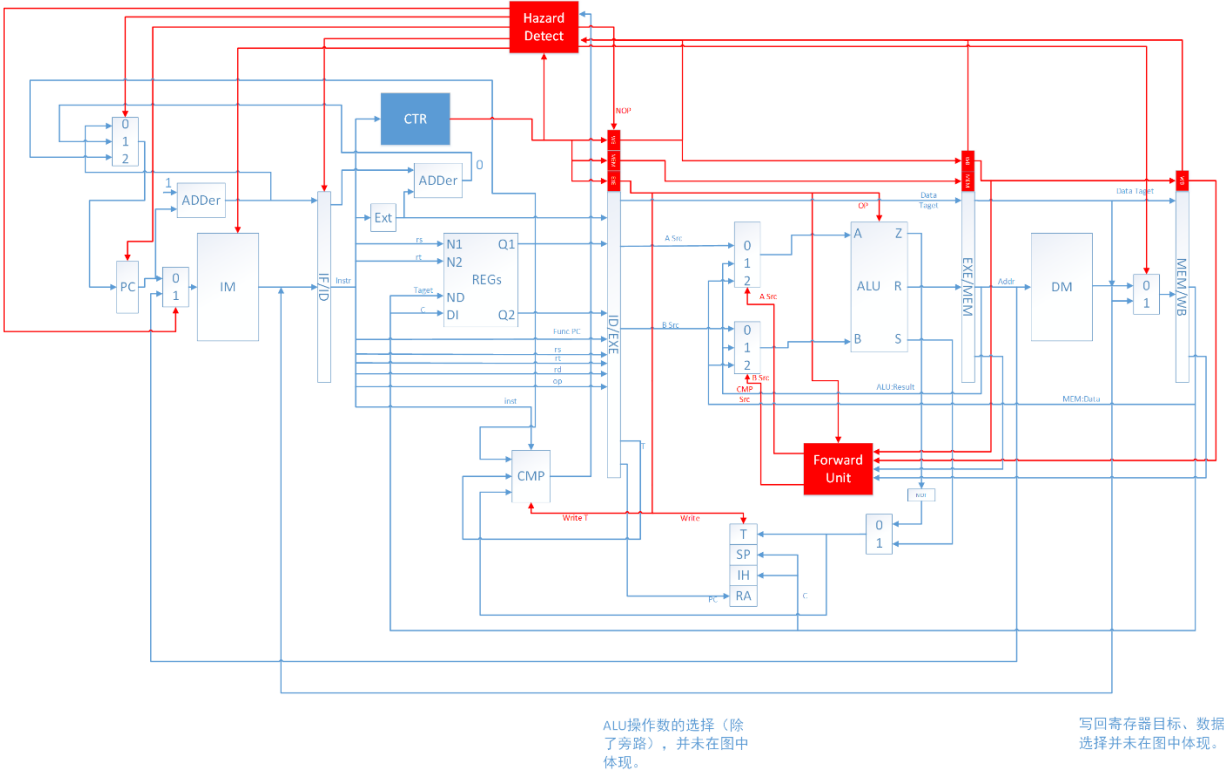
在写回（WB）阶段，寄存器堆根据控制信号决定将数据写回哪个寄存器，以及写回的数据来源。

对于一条指令，它会先后经历这五个周期，在每一个周期中都会产生一些临时变量供下一周期使用（例如 ALU 在运算周期算出结果，访存阶段会将该数据写回）。类似于这种数据传递的工作将由段寄存器来处理，具体的实现详见后面的章节。

此外，流水设计的缺陷是可能产生数据冲突。例如，前一条指令需要在第五周期将数据写入寄存器，而后一条指令在第三周期就会读取这个寄存器，此时读取到的数据是错误的。为此，我们专门设计了冲突解决方案。具体的解决方法详见后面的章节。

2.2 数据通路

我们的数据通路图设计如下：



和最初上交的版本有了很大的变化，最大的变化在于我们的实验目标改变了：由多周期 CPU 变为支持指令流水的 CPU，因此加入了各阶段之间的段寄存器。

在基础要求之上，我们进行了冲突的处理，为了解决冲突，就增加了相应的 Forward Unit 以支持旁路技术来解决数据冲突，当然对于无法通过旁路技术解决的数据冲突，通过 Hazard Detector 来进行检测和发出暂停流水线的信号。为了在跳转指令上减少暂停，我们将跳转目标地址的计算利用单独的加法器提前到了 ID 阶段，同时对于跳转条件的判断（寄存器值的比较）我们也利用单独的比较器提前到了 ID 阶段。对于条件跳转时可能存在的数据冲突，如果是 T 寄存器发生数据冲突，我们在比较器内根据指令进行解决。如果是寄存器 A 发生数据冲突，我们同样通过旁路技术进行解决，对于必须暂停流水线的数据冲突，我们给出了暂停流水线的信号。

而最终的版本也和开始编写 VHDL 代码时的设计有了一些变化，主要是在为了支持监控程序的 A 指令，能够向指令存储器内写入指令，因此为 IM 增加了地址选择、数据选择、读写使能，为 MEM/WB 段寄存器增加了数据选择。

更详细的介绍将在后面的章节中给出。

2.3 指令流程

我们分别对 30 条指令整理了每一周期需要做的事：

指令	IF	ID	EXE	MEM	WB
ADDSP3	取址	译码	$C = SP + \text{imm}$ 扩展		$rs = C$

NOP	取址	译码			
B	取址	译码	$C = PC + \text{imm}$ 扩展		$PC = C$
BEQZ	取址	译码	$C = PC + \text{imm}$ 扩展, 若 A 等于 0 生成 PC 的控制信号	(根据控制信号决定) 将 C 写回 PC	
BNEZ	取址	译码	$C = PC + \text{imm}$ 扩展, 若 A 不等于 0 生成 PC 的控制信号	(根据控制信号决定) 将 C 写回 PC	
SLL	取址	译码	$C = A \text{ sll } \text{imm}$ 扩展		$rs = C$
SRA	取址	译码	$C = A \text{ sra } \text{imm}$ 扩展		$rs = C$
ADDIU3	取址	译码	$C = A + \text{扩展 imm}$		$rt = C$
ADDIU	取址	译码	$C = A + \text{扩展 imm}$		$rs = C$
ADDSP	取址	译码	$C = SP + \text{扩展 imm}$		$SP = C$
BTEQZ	取址	译码	$C = PC + \text{imm}$ 扩展, 若 T 寄存器等于 0 生成 PC 的控制信号	(根据控制信号决定) 将 C 写回 PC	
MTSP	取址	译码	$C = 0 + B$		$SP = C$
LI	取址	译码	$C = 0 + \text{IMM}$		$rs = C$
CMPI	取址	译码	$C = A - \text{IMM}$ 扩展		若 $C = 0$, 则 $T = 0$, 否则 $T = 1$
LW_SP	取址	译码	$C = SP + \text{符号扩展 imm}$	$DR = \text{MEM}[C]$	$rs = DR$
LW	取址	译码	$C = A + \text{符号扩展 imm}$	$DR = \text{MEM}[C]$	$rt = DR$
SW_SP	取址	译码	$C = SP + \text{符号扩展 imm}$	$\text{MEM}[C] = A$	
SW	取址	译码	$C = A + \text{符号扩展 imm}$	$\text{MEM}[C] = B$	
ADDU	取址	译码	$C = A + B$		$rd = C$
SUBU	取址	译码	$C = A - B$		$rd = C$
AND	取址	译码	$C = A \& B$		$rs = C$
CMP	取址	译码	$C = A - B$		若 C 为 0, 则置 $T=0$, 否则置 $T=1$
JALR	取址	译码	$PC = A, RA = PC$		
JR	取址	译码	$PC = A$		
MFPC	取址	译码	$C = PC + 0$		$rs = PC$
NEG	取址	译码	$C = -A$		$rs = C$
OR	取址	译码	$C = A \mid B$		$rs = C$
SLT	取址	译码	$C = A - B$		若 $C < 0$, 则置 $T=1$, 否则置 $T=0$
MFIH	取址	译码	$C = IH + 0$		$rs = IH$
MTIH	取址	译码	$C = A + 0$		$IH = C$

三、主要模块实现

这一部分我们将主要介绍我们通过硬件描述语言来实现前面设计各个模块的功能,以及各个模块之间的调用关系及其接口。在我们的硬件代码部分中,我们主要增加了冲突检测、数据旁路、流水线暂停等扩展功能。这些扩展功能我们将在下一章节进行详细介绍。

3.1 CPU 总体概述

我们的 CPU 设计采用的模块化自底向上的设计方法,首先设计每一个底层功能子模块,当保证实现正确之后组合到顶层的 CPU CORE 中,进行综合调试。CPU CORE 是我们 CPU 的顶层部分,其对外接口如下:

信号名称	类型	功能
CLK_IN	in	输入时钟 50M
RAM1_Addr	out	RAM1 地址总线
RAM1_EN	out	RAM1 的使能信号
RAM1_WE	out	RAM1 的写使能信号
RAM1_OE	out	RAM1 的读使能信号
RAM1_Data	inout	RAM1 的数据总线
RAM2_Addr	out	RAM2 地址总线
RAM2_EN	out	RAM2 的使能信号
RAM2_WE	out	RAM2 的写使能信号
RAM2_OE	out	RAM2 的读使能信号
RAM2_Data	inout	RAM2 的数据总线
com_data_ready	in	串口的数据准备信号
com_rdn	out	串口的读使能信号
com_tbre	in	串口的写反馈信号
com_tsre	in	串口的写反馈信号
com_wrn	out	串口的写使能信号

其下的各个模块设计在下面的小节中将依次描述。

3.2 CLK MODULE

该模块主要负责将时钟分频。因为我们的串口访问及内存访问速度限制,CPU 主频无法达到 50M,因此需要有一个模块将输入的 50M 时钟信号分频,然后发送到各个时序模块。

该模块的对外接口如下:

信号名称	类型	功能
CLK_IN	in	输入时钟 50M
CLK	inout	输出分频之后的时钟

3.3 PC Register

PC 寄存器主要负责保存当前（下一条）指令的地址，接受输入，并且在时钟上升沿时更新输出，但是是否跟新 PC 值需要受到 Hazard Detector 控制信号的约束。因为其他模块使用 PC 寄存器的值都是在下降沿，所以此时更新正好。

该模块的对外接口如下：

信号名称	类型	功能
PC_IN	in	输入的新 PC 值
PC_OUT	out	输出的 PC 值
WRITE_OR_NOT	in	是否更新 PC 值的控制信号
CLK	in	时钟信号

3.4 多路选择器 MUX

在很多模块的数据来源中，都涉及到多路选择的问题，因此我们设计了 2、3、4、6 选 1 的多路选择器，共需要的模块使用。

该模块的对外接口如下（以 2 选 1 为例）：

信号名称	类型	功能
SELEC	in	选择控制信号
SRC_1	in	输入数据 1
SRC_2	in	输入数据 2
OUTPUT	out	输出数据

3.5 RAM1 Visitor

我们的数据存储器 and 串口共用 RAM1 的总线，在 RAM1 Visitor 内部根据当 LW/SW 指令的地址，来进行不同的物理介质访问操作，当地址是 BF01 时，返回串口的状态信息，当地址是 BF00 时，对串口进行读写操作，当地址不小于 8000 时，对 RAM1 进行读写操作，当地址小于 8000 时，不对 RAM1 和串口进行任何操作，而是对 RAM2（指令存储器）进行访问，具体的调度由 Hazard Detector 负责，详细的内容将在后面的小节进行介绍。

该模块的对外接口如下：

信号名称	类型	功能
clk	in	时钟信号
DMemReadWrite	in	RAM1 的读/写访问控制信号
EXandMEM_AluRes	in	访问 RAM1 时的地址输入
DataReady	in	串口的数据准备信号
WriteData	in	写操作时写入的数据
TSRE	in	串口的写反馈信号
TBRE	in	串口的写反馈信号
RAM1_Enable	out	RAM1 的使能信号
RAM1_ReadEnable	out	RAM1 的读使能信号

RAM1_WriteEnable	out	RAM1 的写使能信号
SPort_WriteEnable	out	串口的写使能信号
SPort_ReadEnable	out	串口的读使能信号
DMemData	inout	RAM1 的数据总线
DMemAddr	out	RAM1 地址总线

3.6 RAM2 Visitor

我们利用 RAM2 作为指令存储器，这样就保证了访问串口的时候不会和取指发生结构冲突，但是当监控程序执行 A 指令时，需要将指令写入到指令存储器，因此我们有相应的控制信号：访问 RAM2 地址选择、RAM2 读写控制信号、访问 RAM2 数据来源选择等。

该模块的对外接口如下：

信号名称	类型	功能
clk	in	时钟信号
DMemReadWrite	in	RAM2 的读/写访问控制信号
EXandMEM_AluRes	in	访问 RAM2 时的地址输入
WriteData	in	写操作时写入的数据
RAM2_Enable	out	RAM2 的使能信号
RAM2_ReadEnable	out	RAM2 的读使能信号
RAM2_WriteEnable	out	RAM2 的写使能信号
DMemData	inout	RAM2 的数据总线
DMemAddr	out	RAM2 地址总线

3.7 PC Adder

此模块负责每次取指之后计算 $PC + 1$ 的值，因为 RAM 的访问是以字为单位的，而每条指令大小刚好为 1 个字，因此只需将 $PC + 1$ 即可。

该模块的对外接口如下：

信号名称	类型	功能
OLD_PC	in	旧的 PC 值
NEW_PC	out	新的 PC 值 ($PC + 1$)

3.8 IF/ID Register

此模块的功能是保存每次取出的指令、以及加 1 之后的 PC 值供后面的阶段使用，每次在时钟上升沿更新输出的指令值和输出的 PC 值，但是需要受到 Hazard Detector 的控制信号控制。

该模块的对外接口如下：

信号名称	类型	功能
NEW_PC_IN	in	加 1 之后的 PC 值
WRITE_PC_OR_NOT	in	是否更新输出 PC 值的控制信号

NEW_PC_OUT	out	输出的 PC 值
CLK	in	时钟信号
INST_IN	in	取指阶段取出的指令
WRITE_IR_OR_NOT	in	是否更新输出指令的控制信号
WRITE_IR_SRC_SELEC	in	更新输出指令时的指令内容选择信号
INST_OUT_CODE	out	输出指令的 CODE 部分
INST_OUT_RS	out	输出指令的 RS 寄存器编号部分
INST_OUT_RT	out	输出指令的 RT 寄存器编号部分
INST_OUT_RD	out	输出指令的 RD 寄存器编号部分
INST_OUT_FUNC	out	输出指令的 FUNCTION 部分

3.9 Imm Extend

此模块的功能是将指令中的立即数进行扩展。特别地，对于不同类型的指令，立即数的扩展方式也不相同。由于这一过程与译码同时进行，所以它需要解析指令，根据指令内容输出对应的扩展后的立即数。

该模块的对外接口如下：

信号名称	类型	功能
CODE	in	指令的 CODE 字段（1-5 位）
RS	in	指令的 RS 字段（6-8 位）
RT	in	指令的 RT 字段（9-11 位）
RD	in	指令的 RD 字段（12-14 位）
FUNC	in	指令的 FUNC 字段（15-16 位）
IMM	out	扩展后的立即数（16 位）

3.10 adder

此模块的功能是将 pc 与立即数做加法，并将结果输出。

该模块的对外接口如下：

信号名称	类型	功能
PC	in	输入的 pc
IMM	in	输入的立即数
RES	out	输出的结果

3.11 Hazard Detector

此模块负责检测指令流水过程中的数据冲突、结构冲突、控制冲突等，并且生成相应控制信号：PC 寄存器是否更新、IF/ID 段寄存器是否更新、ID/EXE 段寄存器保存的控制信号内容、RAM2 访问时的地址数据选择、MEM/WB 段寄存器保存访问存储体的结果数据选择等。

该模块的对外接口如下：

信号名称	类型	功能
CUR_INST_CODE	in	本条指令（处在取指后期）的 CODE 字段
CUR_INST_RS	in	本条指令的 RS 字段
CUR_INST_RT	in	本条指令的 RT 字段
CUR_INST_RD	in	本条指令的 RD 字段
CUR_INST_FUNC	in	本条指令的 FUNCTION 字段
LAST_WRITE_REGS_OR_NOT	in	上条指令是否写通用寄存器的控制信号
LAST_WRITE_REGS_TARGET	in	上条指令写通用寄存器的目标寄存器编号
LAST_VISIT_DM_OR_NOT	in	上条指令是否访问数据存储器的控制信号
LAST_LAST_WRITE_REGS_OR_NOT	in	上上条指令是否写通用寄存器的控制信号
LAST_LAST_WRITE_REGS_TARGET	in	上上条指令写通用寄存器的目标寄存器编号
LAST_LAST_VISIT_DM_OR_NOT	in	上上条指令是否访问数据存储器的控制信号
LAST_LAST_DM_VISIT_ADDR	in	上上条指令访问数据存储器的地址
CUR_DM_READ_WRITE	in	本条指令访问数据存储器控制信号
CUR_DM_WRITE_DATA_SRC	in	本条指令写数据存储器数据选择控制信号
JUMP_OR_NOT	in	条件跳转指令是否跳转控制信号
WRITE_PC_OR_NOT	out	是否更新 PC 寄存器控制信号
NEW_PC_SRC_SELEC	out	更新 PC 寄存器数据选择控制信号
WRITE_IR_OR_NOT	out	是否更新 IF/ID 寄存器控制信号
WRITE_IR_SRC_SELEC	out	是否更新 IF/ID 寄存器 IR 数据选择控制信号
COMMAND_ORIGIN_OR_NOP	out	ID/EXE 内保存的控制信号是否置为 NOP
DM_DATA_RESULT_SELEC	out	MEM/WB 保存访存结果数据选择控制信号
IM_ADDR_SELEC	out	指令存储器访问地址选择控制信号
IM_DATA_SELEC	out	指令存储器访问数据选择控制信号
IM_READ_WRITE_SELEC	out	指令存储器访问控制信号

3.12 Controller

此模块的主要功能是根据指令生成对应的控制信号。

该模块的对外接口如下：

信号名称	类型	功能
INST_CODE	in	本条指令的 CODE 字段
INST_RS	in	本条指令的 RS 字段
INST_RT	in	本条指令的 RT 字段
INST_RD	in	本条指令的 RD 字段
INST_FUNC	in	本条指令的 FUNCTION 字段
ALU_OP	out	ALU 的运算种类
ALU_A_SRC	out	ALU 运算数 A 的来源
ALU_B_SRC	out	ALU 运算数 B 的来源
DATA_MEM_READ_WRITE	out	选择写内存或读内存的
WRITE_DM_DATA_SRC	out	写入内存数据来源
REGS_WRITE_OR_NOT	out	是否写寄存器

WRITE_REGS_DEST	out	写回寄存器的目标
REGS_WRITE_DATA_SRC	out	写寄存器数据来源
WRITE_RA_OR_NOT	out	是否写特殊寄存器 RA
WRITE_IH_OR_NOT	out	是否写特殊寄存器 IH
WRITE_SP_OR_NOT	out	是否写特殊寄存器 SP
WRITE_T_OR_NOT	out	是否写特殊寄存器 T
WRITE_T_SRC	out	写 T 寄存器数据来源

3.13 Common Register

此模块的功能是维护 8 个通用寄存器的数据，对外提供读写功能。

该模块的对外接口如下：

信号名称	类型	功能
CLK	in	时钟信号
RS	in	指令的 RS 字段
RT	in	指令的 RT 段
WRITE_FLAG	in	控制读/写寄存器
WRITE_REG	in	目标寄存器编号
WRITE_DATA	in	写入寄存器的数据
A	out	读出的结果 A（对应 RS 字段）
B	out	读出的结果 B（对应 RT 字段）

3.14 Comparator

此模块的功能是根据控制信号和 T 寄存器的内容，决定是否跳转。

该模块的对外接口如下：

信号名称	类型	功能
CODE	in	指令的 CODE 字段
WRITE_T	in	T 寄存器内容（通过数据旁路）
T	in	T 寄存器的内容
T_SRC_ZF	in	T 的比较对象 ZF
T_SRC_SF	in	T 的比较对象 SF
T_CMD_SRC	in	选 T 寄存器来源
A	In	操作数 A
JUMP	out	是否跳转

3.15 ID/EXE Register

此模块的主要功能是将 IF 阶段产生的控制信号传递给 EXE 阶段，并根据控制信号对操作数 A、B 进行初步的筛选。

该模块的对外接口如下：

信号名称	类型	功能
clk	in	时钟信号
command_origin_or_nop	in	控制器的控制信号，若为 1，则输出所有的默认控制信号
in_pc	in	输入 PC
out_pc	out	输出 PC
in_reg_a	in	输入操作数 A
out_reg_a	out	输出操作数 A
in_reg_b	in	输入操作数 B
out_reg_b	out	输出操作数 B
in_imm	in	输入立即数
out_imm	out	输出立即数
in_alu	in	输入 alu 运算符
out_alu	out	输出 alu 运算符
in_a_src	in	输入操作数 a 来源
out_a_src	out	输出操作数 a 来源
in_b_src	in	输入操作数 b 来源
out_b_src	out	输出操作数 b 来源
in_reg_result	in	写回寄存器控制信号
in_rs	in	指令 RS 字段
in_rt	in	指令 RT 字段
in_rd	in	指令 RD 字段
out_reg_result	out	写回寄存器编号
in_flag_RA	in	输入 RA 寄存器控制信号
out_flag_RA	out	输出 RA 寄存器控制信号
in_flag_IH	in	输入 IH 寄存器控制信号
out_flag_IH	out	输出 IH 寄存器控制信号
in_flag_SP	in	输入 SP 寄存器控制信号
out_flag_SP	out	输出 SP 寄存器控制信号
in_flag_T	in	输入 T 寄存器控制信号
out_flag_T	out	输出 T 寄存器控制信号
in_T_SRC	in	输入 T 寄存器来源
out_T_SRC	out	输出 T 寄存器来源
in_mem_cmd	in	输入读写内存控制信号
out_mem_cmd	out	输出读写内存控制信号
in_flag_reg	in	输入写回寄存器编号
out_flag_reg	out	输出写回寄存器编号

in_reg_src	in	输入写回寄存器来源
out_reg_src	out	输出写回寄存器来源

3.16 Alu

此模块的功能是对两个操作数进行计算。

该模块的对外接口如下：

信号名称	类型	功能
A	in	操作数 A
B	in	操作数 B
OP	in	运算种类，有加、减、与、或等
ZF	out	运算结果是否为 0 的标识位
SF	out	运算结果是否非负的标识位
C	out	运算结果

3.17 Special Register

此模块的功能是维护 4 个特殊寄存器的数据，对外提供读写功能。

该模块的对外接口如下：

信号名称	类型	功能
CLK	in	时钟信号
RA_CMD_WRITE	in	是否写 RA 的控制信号
RA_SRC	in	写 RA 寄存器的来源
RA_VALUE	out	输出 RA 寄存器的数据
IH_CMD_WRITE	in	是否写 IH 寄存器
IH_SRC	in	写 IH 寄存器的来源
IH_VALUE	out	输出 IH 寄存器的数据
SP_CMD_WRITE	in	是否写 SP 寄存器
SP_SRC	in	写 SP 寄存器的来源
SP_VALUE	out	输出 SP 寄存器的数据
T_CMD_WRITE	in	是否写 T 寄存器
T_CMD_SRC	in	控制写 T 寄存器的来源
T_SRC_ZF	in	ZF 的值（作为 T 的来源之一）
T_SRC_SF	in	SF 的值（作为 T 的来源之一）
T_VALUE	out	输出 T 寄存器的数据

3.18 Forward Unit

此模块负责检测是否存在数据冲突以及生成旁路选择控制信号。

该模块的对外接口如下：

信号名称	类型	功能
CUR_RS_REG_NUM	in	本条指令（处在译码后期）RS 寄存器编号
CUR_RT_REG_NUM	in	本条指令 RT 寄存器编号
CUR_ALU_A_SRC_SELECT	in	本条指令参与 ALU 运算的 A 操作数选择信号
CUR_ALU_B_SRC_SELECT	in	本条指令参与 ALU 运算的 B 操作数选择信号
LAST_WRITE_REGS_OR_NOT	in	上条指令是否写通用寄存器控制信号
LAST_WRITE_REGS_TARGET	in	上条指令写通用寄存器目的编号
LAST_DM_READ_WRITE	in	上条指令访问数据存储器控制信号
LAST_LAST_WRITE_REGS_OR_NOT	in	上上条指令是否写通用寄存器控制信号
LAST_LAST_WRITE_REGS_TARGET	in	上上条指令写通用寄存器目的编号
ALU_A_SRC_SELECT_FINAL	out	本条指令参与 ALU 运算 A 操作数最终选择信号
ALU_B_SRC_SELECT_FINAL	out	本条指令参与 ALU 运算 B 操作数最终选择信号

3.19 EXE/MEM Register

此模块负责保存 EXE 阶段的中间结果，供后面的阶段使用。

该模块的对外接口如下：

信号名称	类型	功能
CLK	in	时钟信号
NEW_PC_IN	in	PC 寄存器的值
WRITE_DM_DATA_IN	in	写数据存储器的数据
WRITE_REG_NUM_IN	in	写通用寄存器的目标编号
ALU_RESULT_IN	in	ALU 运算结果
IH_REG_IN	in	IH 寄存器的值
DATA_MEM_READ_WRITE_IN	in	数据存储器访问控制信号
REGS_READ_WRITE_IN	in	通用寄存器访问控制信号
REGS_WRITE_DATA_SRC_IN	in	写通用寄存器的数据选择控制信号
NEW_PC_OUT	out	PC 寄存器的值
WRITE_DM_DATA_OUT	out	写数据存储器的数据
WRITE_REG_NUM_OUT	out	写通用寄存器的目标编号
ALU_RESULT_OUT	out	ALU 运算结果
IH_REG_OUT	out	IH 寄存器的值
DATA_MEM_READ_WRITE_OUT	out	数据存储器访问控制信号
REGS_READ_WRITE_OUT	out	通用寄存器访问控制信号
REGS_WRITE_DATA_SRC_OUT	out	写通用寄存器的数据选择控制信号

3.20 MEM/WB Register

此模块负责保存 MEM 阶段的中间结果，供后面的阶段使用。

该模块的对外接口如下：

信号名称	类型	功能
CLK	in	时钟信号
NEW_PC_IN	in	PC 寄存器的值
WRITE_REG_NUM_IN	in	写通用寄存器的目标编号
ALU_RESULT_IN	in	ALU 运算结果
IH_REG_IN	in	IH 寄存器的值
DM_DATA_IN	in	访问数据存储器结果
REGS_READ_WRITE_IN	in	通用寄存器访问控制信号
REGS_WRITE_DATA_SRC_IN	in	写通用寄存器的数据选择控制信号
NEW_PC_OUT	out	PC 寄存器的值
WRITE_REG_NUM_OUT	out	写通用寄存器的目标编号
ALU_RESULT_OUT	out	ALU 运算结果
IH_REG_OUT	out	IH 寄存器的值
DM_DATA_OUT	out	访问数据存储器结果
REGS_READ_WRITE_OUT	out	通用寄存器访问控制信号
REGS_WRITE_DATA_SRC_OUT	out	写通用寄存器的数据选择控制信号

3.21 各模块之间的接口、调用关系及组合

我们各个模块之间根据每个模块的输入输出接口，按照数据通路图的连接关系，利用 signal 在 CPU CORE 里面把各个模块连接起来。例如 PC 寄存器的输出作为指令存储器访问地址二路选择器的一个输入，ALU 运算结果作为另一个输入，Hazard Detector 的指令存储器访问地址选择控制信号作为该二路选择器的控制信号，其输出作为指令存储器访问的地址，指令存储器的数据输出作为 IF/ID 段寄存器的指令输入等。如此把所有模块组合起来，构成一个有机的整体，各模块之间相互协调，共同完成整个 CPU 的功能。

有一点需要提到的是：关于各个模块之间数据的保持问题，因为各个关键时序模块均是在时钟下降沿使用输入数据，而段寄存器均是在上升沿更新输出数据，因此这样就不会因为段寄存器更新输出数据而影响其他模块的逻辑正确性。而 Hazard Detector、Forward Unit、ALU、多路选择器等模块均使用组合逻辑，这样就能保证一旦其敏感数据发生变化，输出结果就能立刻发生变化，而这些敏感数据的变化一定都是在时钟上升沿之后、下降沿之前稳定下来的，因此时序模块在时钟下降沿使用其生成的控制信号时，控制信号已经稳定下来了。

四、扩展功能

我们的扩展功能主要体现在指令流水过程中优美的冲突解决。

4.1 结构冲突

我们通过指令存储器、数据存储器分开的方式解决了大部分的冲突：每次访问数据存储器的时候并不会影响到对指令存储器的访问。但是由于监控程序的 A 指令，需要将用户从串口输入的数据写入到指令存储器中，因此会存在 SW 指令的地址位于指令区的冲突，对于这种冲突，我们是这样解决的：

当某条指令执行到 MEM 阶段时，我们的 RAM1 Visitor 模块检查其访问地址，如果位于指令区，则此次不访问 RAM1，同时 Hazard Detector 也在做同样的检查，当检查到本次访存是要访问指令区时，就发出暂停流水线的控制信号，具体来说就是：PC 寄存器停止更新，IF/ID 段寄存器内的指令置为 NOP，ID/EXE 段寄存器不受影响，同时让 RAM2 访问时的地址和数据分别选择 ALU 结果、EXE/MEM 段寄存器的输出。这样做的效果就是认为取出了一条 NOP 指令，而下个周期取指时，PC 仍然指向应该取出的那条指令的位置。

对于这种冲突，我们的流水线只需要暂停一个时钟周期，而这个暂停的时钟周期是必须的。

4.2 数据冲突

指令流水期间会发生写后读的数据冲突，我们采用数据旁路技术来解决这个问题。

我们的 Forward Unit 检测本条指令是否需要使用读取通用寄存器的结果作为 ALU 的运算操作数，同时检查上条指令、上上条指令是否改写了这个通用寄存器，如果是则存在冲突，我们通过旁路技术来进行解决，如果上条指令改写了该寄存器，我们让上条指令 ALU 的运算结果参与 ALU 运算，如果上上条指令改写了该寄存器，我们让上上条指令在 MEM/WB 的输出来参与 ALU 运算。

这样做的前提条件是上条指令不是 LW 指令，在这样的前提下，我们的流水线不需要暂停。

在 Forward Unit 进行检测的同时，我们的 Hazard Detector 也在做同样的检测，如果上条指令是 LW 指令，那么就会发出暂停流水线的信号：PC 寄存器停止更新，IF/ID 段寄存器内容停止更新并保持为原值，ID/EXE 段寄存器更新控制信号为 NOP 指令的控制信号。这样在下个周期的时候，访存的结果就可以在 MEM/WB 段寄存器里面拿到了，此时再通过旁路技术送到 ALU 操作数那里去。

这种情况下我们的流水线只需要暂停一个时钟周期，而这个暂停的时钟周期是必须的。

4.3 控制冲突

对于一定会跳转的指令，当我们得知这是一条跳转指令的时候已经是 ID 阶段了，此时下一条指令已经进入了流水线，但这条指令是不应该被执行的。因此我们的 Hazard Detector 在检测到这种情况时会发出相应的控制信号：PC 寄存器更新为跳转目标，IF/ID 段寄存器内的指令强制置为 NOP，ID/EXE 段寄存器内的控制信号强制置为 NOP 的控制信号。

对于条件跳转指令，我们把计算跳转目标的加法器、判断是否跳转的比较器都提前到了 ID 阶段进行，因此在 ID 阶段我们就可以得知是否应该跳转以及跳转目标了，对于不需要跳转的情形：已经进入流水线的指令正常执行，流水线不受到任何干扰；对于需要跳转的指令，控制信号和一定跳转的指令一样。

这样做的效果是：我们硬件实现了跳转指令延迟槽的填补，汇编代码里面在跳转和分支语句后面不需要任何 NOP 指令，我们硬件在延迟槽中加入了 NOP 指令！

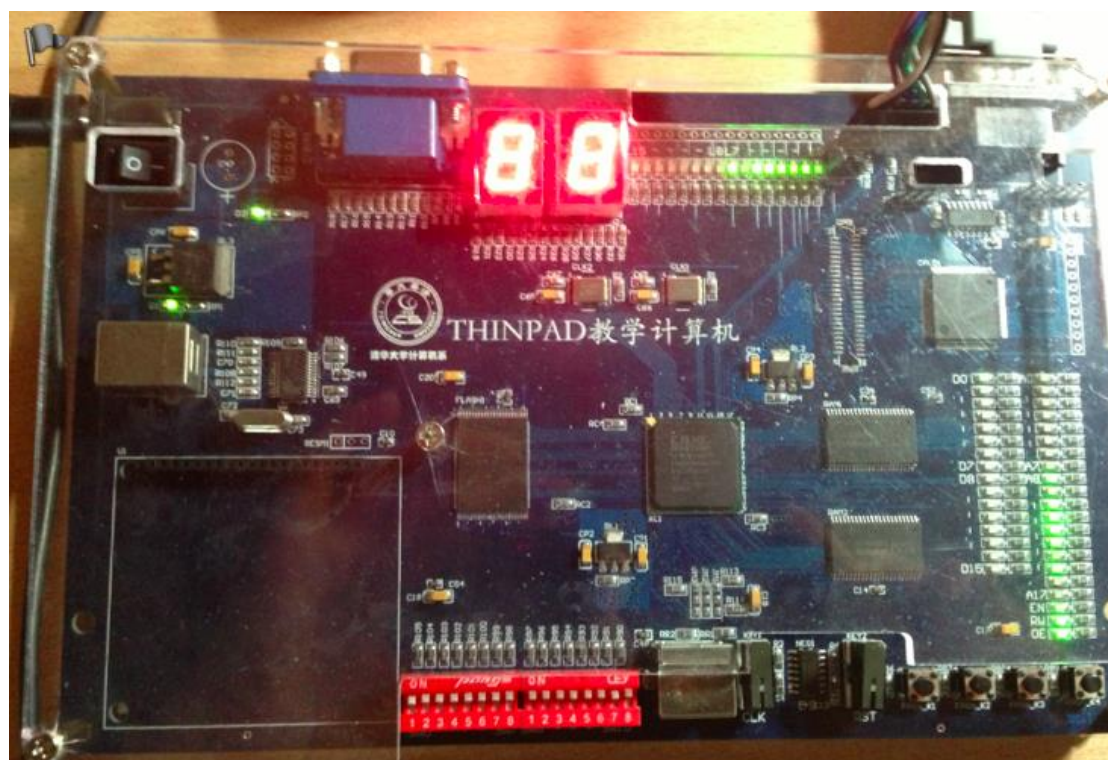
4.4 多种冲突汇聚

在指令流水的过程中，可能存在多种冲突并存的情形：对于条件跳转指令，可能其所使用的比较数据是在上条或者上上条指令被修改过的，因此存在写后读的数据冲突，这里我们同样采用旁路技术，将最新鲜的数据送到比较器中。

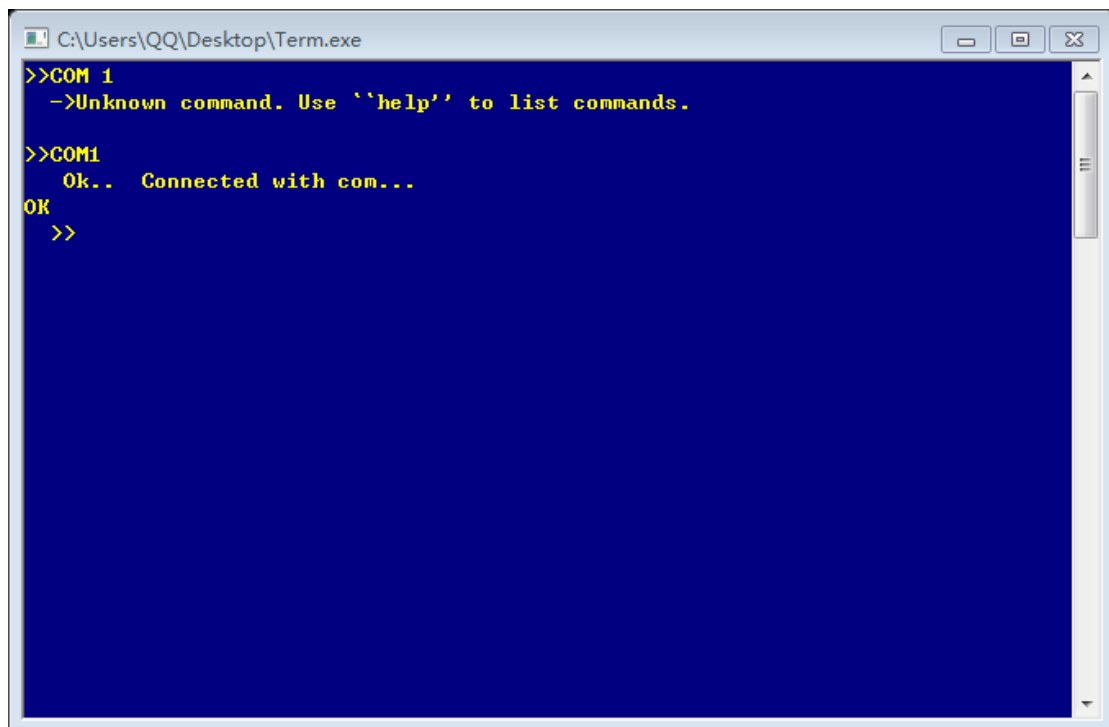
对于特殊寄存器，我们把对其写操作全部放到 EXE 阶段进行，因此对于特殊寄存器不存在写后读的数据冲突。

五、实验成果展示

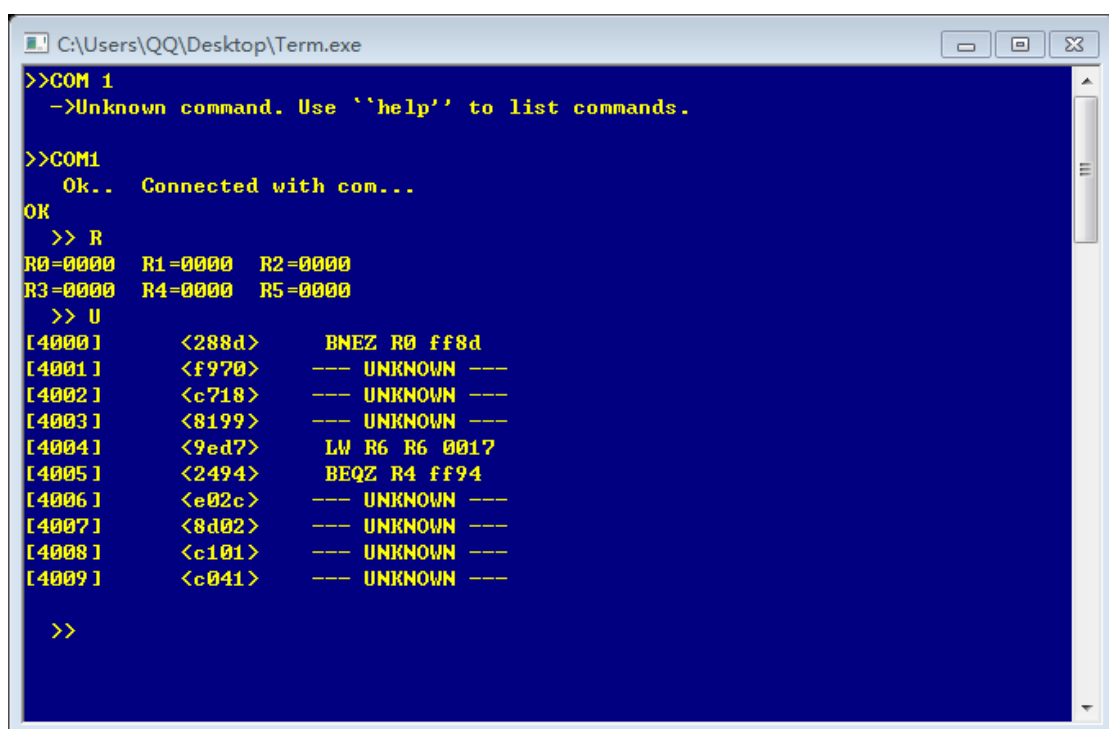
(1) 教学机运行图片



(2) 监控程序成功运行



(3) 使用 R 指令查看寄存器的值，使用 U 指令查案内存中的指令



(4) 使用 A 指令向教学机输入应用程序（图中显示的是计算斐波那契数列的程序）

```

C:\Users\QQ\Desktop\Term.exe
[8004] 8c5b
[8005] 6b5b
[8006] 4752
[8007] 3060
[8008] 1000
[8009] a940

>> A
[4000] LI R1 1
[4001] LI R2 1
[4002] LI R3 80
[4003] SLL R3 R3 0
[4004] LI R4 19
[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 R4 F9
[400c] JR R7
[400d] NOP
[400e]

>>

```

(5) 使用 U 指令查看应用程序代码，并使用 G 指令运行

```

C:\Users\QQ\Desktop\Term.exe
[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 R4 F9
[400c] JR R7
[400d] NOP
[400e]

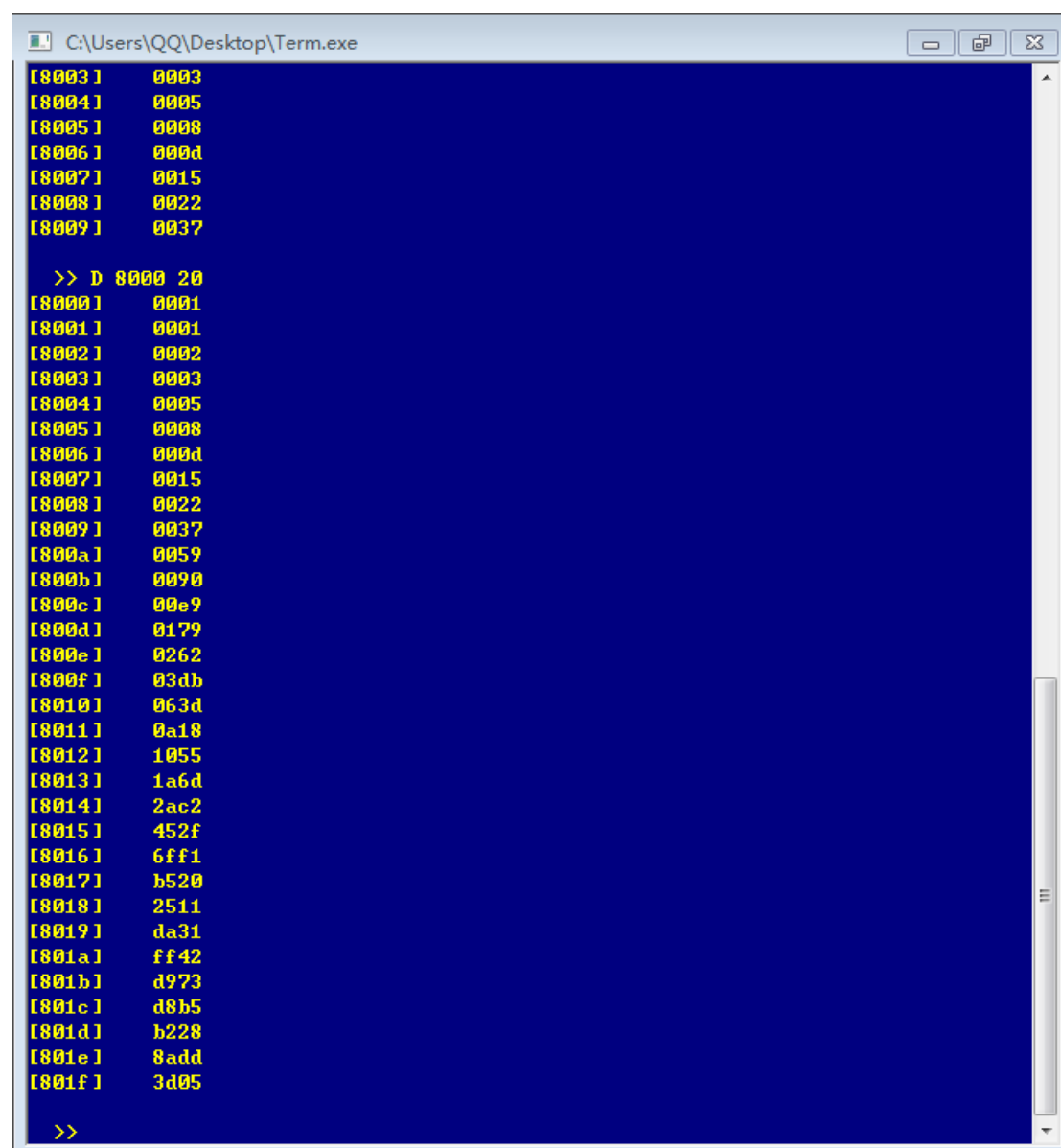
>> G

>> U
[4000] <6901> LI R1 0001
[4001] <6a01> LI R2 0001
[4002] <6b80> LI R3 0080
[4003] <3360> SLL R3 R3 0000
[4004] <6c19> LI R4 0019
[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

>>

```

(6) 使用 D 指令查看内存数据，验证程序运行成功



A screenshot of a terminal window titled "C:\Users\QQ\Desktop\Term.exe". The terminal displays a memory dump with addresses in hexadecimal brackets and corresponding values. The first block of data shows addresses from [8003] to [8009] with values 0003, 0005, 0008, 000d, 0015, 0022, and 0037. After a command ">> D 8000 20", a second block of data is shown, starting from [8000] and ending at [801f], with values ranging from 0001 to 3d05. The terminal ends with a ">>" prompt.

```
C:\Users\QQ\Desktop\Term.exe
[8003] 0003
[8004] 0005
[8005] 0008
[8006] 000d
[8007] 0015
[8008] 0022
[8009] 0037

>> D 8000 20
[8000] 0001
[8001] 0001
[8002] 0002
[8003] 0003
[8004] 0005
[8005] 0008
[8006] 000d
[8007] 0015
[8008] 0022
[8009] 0037
[800a] 0059
[800b] 0090
[800c] 00e9
[800d] 0179
[800e] 0262
[800f] 03db
[8010] 063d
[8011] 0a18
[8012] 1055
[8013] 1a6d
[8014] 2ac2
[8015] 452f
[8016] 6ff1
[8017] b520
[8018] 2511
[8019] da31
[801a] ff42
[801b] d973
[801c] d8b5
[801d] b228
[801e] 8add
[801f] 3d05

>>
```

六、实验心得体会

早在进入大三之前,就听说过“三座大山”的说法,计原就是其中之一,刚开始上课的时候还没怎么觉得,但是当大实验布置下来的时候,“奋战二十天,造台计算机”的口号瞬间让我们觉得压力山大,第一座大山就这样轰然压了下来!

刚开始构思大实验的时候,我们是准备做多周期的 CPU 的,相比于支持指令流水的 CPU,多周期还是要简单很多的,因此工作量就会相对小一些。但是在后来小班上课的时候,我们发现几乎所有同学都是做的指令流水的 CPU,而且助教和李山山老师的回答一个比一个吓人,先说成绩最多 80 分,后来又说顶多就一个及格分,我们立马决定调整目标,瞄准流水线 CPU。好在我们之前的工作没有白费,我们之前就已经讨论完了指令集里面每一条指令的工作流程以及每一部对每个部件的控制信号,虽然不能完全搬过来用,但是对于整体的构思还是很有帮助的。我们对于整体架构、指令执行过程的思考应该说是班里最早的小组之一,这对我们后期的编码工作有很大帮助:我们的整体架构一开始就定对了,没有出现代码重构的事件!

真正开始写代码是在布置下大实验之后的第二周，我们花了一个周末的时间就把各个基本模块及其组合的代码完成了，但是硬件的调试工作确是漫长与无奈的。首先我们在访存上面就遇到了问题，指令一直读不出来，调试了两天之后也还是没有太大的进展，只是发现在访存的代码里面，一旦出现为数据总线赋值的情况，那就无法读出数据，而不做任何赋值，就能正常读出指令（即便在读内存的时候需要先将数据总线置为高阻态，但是一旦我们有任何赋值，就无法读出数据）。无奈之余，我们请教了进度比较快的任勇他们小组，得出的结论就是我们的代码有一点细微差异，而就是这点差异造成了这样的后果：我们访存都是在一个周期内完成的，下降沿把使能信号拉低，上升沿把使能信号拉高，同时读出数据，我们进行了时钟的边沿检测，而任勇他们没有进行时钟边沿的检测。我想这是因为ISE在进行综合、实现、布线的时候，其实现方式和我们编写代码时的期望是有一定差异的，而由于我们不知道它在实现的时候干了些什么，所以让我们百思不得其解，当我们也采用了他们的实现方式之后，访存就得以正常进行了。所以说编写硬件描述语言代码的时候，一定不能以软件开发的思路去设计代码，因为如果我们的设计在硬件上实现起来是不行的，那么真正实现到硬件里面的就很可能与我们的设想出现巨大差异，这也是我们最大的教训！

我们在调试过程中最大的特点就是充分利用了仿真工具，通过给CPU的存储数据输入指定的指令，来观察CPU在运行的过程中是否会出现逻辑错误。通过仿真，我们查出了所有的逻辑错误：比如Forward Unit判断是否有数据冲突时的逻辑缺陷等。当我们存储和串口调试通过之后，我们立即就能运行监控程序了，所以仿真工具给我们带来了很大的帮助。

奋战了二十天，我们确实造出了一台计算机，虽然速度并不是很快，功能并不是很全，但是这也是我们熬夜、编码、调试得到的成果！在实验的过程中，我们没有参考任何学长的代码，从构思到设计，再到编码，调试，全是我们独立完成的，付出了无数的艰辛，在丰收的时刻，我们也收获了真正的成果！计原这三座大山之一，这造计算机的全部过程，将会是我们本科阶段难忘的一次回忆！

