2013 年 12 月 6 日

大作业实验报告

《计算机组成原理》

2011011237　计 13 班　张宏辉

2011011258　计 13 班　谭志鹏

目录

一、实验目标

1.1 基本目标

① 实现五级指令流水 CPU

② 实现 THCO MIPS 指令系统，能运行现有的监控程序

③ 实现在监控程序下运行应用程序

④ 完善监控程序等辅助软件系统

1.2 拓展目标

① 实现 VGA 模块，显示寄存器、PC 值与指令等，便于调试指令正确性

② 实现数据旁路、冒险控制单元等，更好地解决“冲突”问题

③ 实现软中断等

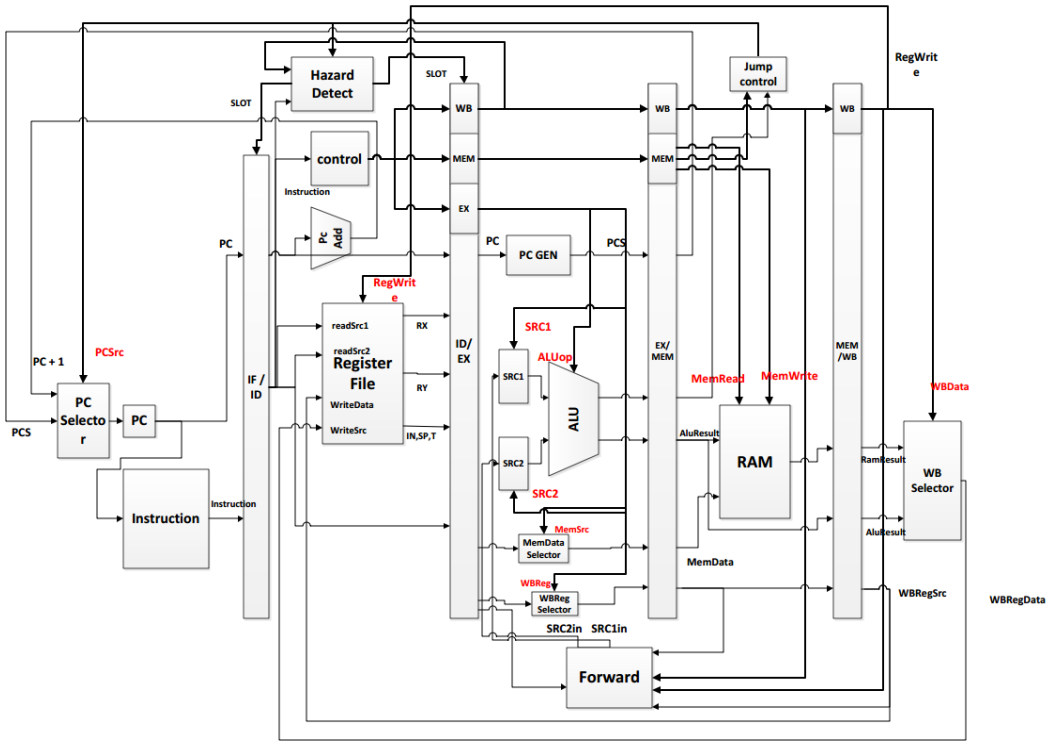
1.3 设计细节

① CPU 主频为 25MHz

② VGA 分辨率为 640 * 480

③ 按照要求，除 25 条基本指令外实现了 JALR、CMPI、ADDSP3、NOT 和 SLT

二、流水线数据通路



（注：大图请见“核心资料-数据通路.PNG”）

2.1 【IF 阶段】取指令

根据控制信号 PCSrc 与 ALU 输出的条形码 C 来选择新的 PC 值，并使用这个值从 Instruction Ram（即 RAM2）中读取指令，将读出的指令放入 IF/ID 流水线寄存器。同时，PC 中的地址也存入到 IF/ID 流水线寄存器中，以备后面的指令(例如 BEQZ)使用。

2.2 【ID 阶段】指令译码与寄存器堆的读写

取出 IF/ID 流水线寄存器中的指令并进行译码。因为 CPU 并不知道当前是哪条指令正在被译码，所以依据指令规格，我们通过两个寄存器号将 Register File 中对应的两个通用寄存器的值都读出；因为不知道当前指令是否会用到特殊的 IH、SP 和 T 寄存器，所以我们也都将其读出以备下一级流水使用；此外，将一个 11 位的立即数也一并读出。根据指令产生流水线后续阶段所需要的全部控制信号，并将这些控制信号、读出的寄存器值、立即数以及 PC 地址，一起存入 ID/EXE

流水线寄存器中。

因为是流水线工作，之前指令有可能在同一时刻执行到 WB 阶段需要写回寄存器堆，所以在这一时钟周期内，除上诉译码、读取寄存器外，还需要完成写回操作。设计寄存器堆的工作方式为：下降沿写入数据，持续输出数据，并在 RST 时所有寄存器清零。那么当时钟下降沿到来时，我们将可以根据 WriteReg 写入相应寄存器中。此外，该阶段还要将 PC 值加 1 后送回到 PC Selector。

2.3 【EXE 阶段】执行或计算地址

根据 ID/EXE 流水线寄存器中的各种控制信号与数据（其中的立即数需要进行对应的符号拓展或者零拓展），结合数据旁路正确选择出 ALU 的 Src1 与 Src2 并进行相应运算，将 ALU 的计算结果送入 EXE/MEM 流水线锁存。同时，这一阶段也要计算出所有可能的 PC 值，并根据控制信号 MemSrc 选择写回 Data Ram（即 RAM1）的数据，一并送入 EXE/MEM 流水线寄存器。

2.4 【MEM 阶段】存储器访问

根据 EXE/MEM 流水线寄存器中的控制信号、地址与数据，进行 Ram 的读/写操作，并将读出的数据存入到 MEM/WB 流水线寄存器中。此外，这一阶段还要将 EXE 阶段产生的所有下一条指令的可能 PC 取值都送回 PC Selector。

2.5 【WB 阶段】写回寄存器

从 MEM/WB 流水线寄存器中的控制信号与数据中选出要写回的寄存器标号以及要写回的数据，直接送回 Register File。

三、控制信号

3.1 控制信号分析

分析 THCO MIPS 指令集流水线各阶段所需的控制信号。具体如下：

3.1.1 【EXE 阶段】

① Src1：表示 ALU 的操作数 1 的来源。其可能取值如下：

取值	含义
rx	寄存器 x
IH	寄存器 IH
SP	寄存器 SP
T	寄存器 T
PC	寄存器 PC
ZERO	0
Z_imm3	将指令的低 3 位零拓展为 16 位数据
Z_imm8	将指令的低 8 位零拓展为 16 位数据

② Src2：表示 ALU 的操作数 2 的来源。其可能取值如下：

取值	含义
ry	寄存器 y
Zero	0
One	1
S_imm4	将指令的低 4 位符号拓展为 16 位
S_imm5	将指令的低 5 位符号拓展为 16 位
S_imm8	将指令的低 8 位符号拓展为 16 位

③ AluOp：表示 ALU 的操作符。其可能取值如下：

取值	含义
add	加法

sub	减法
and	按位与
or	按位或
not	非
sll	逻辑左移
sra	算数右移
eq	相等
ne	不相等
lessthan	小于
empty	无操作

④ MemSrc：表示写入 Ram 的数据来源。其可能取值如下：

取值	含义
rx	寄存器 x
ry	寄存器 y
empty	无

3.1.2 【MEM 阶段】

① PCSrc：表示寄存器 PC 值的来源，其可能的取值如下：

取值	含义
normal	PC+1
branch8	PC+1+s_imm8
branch11	PC+1+s_imm11
jump	寄存器 x
intjump	INT address

② MEMRead：表示 Mem 阶段是否要读取 Ram，其可能的取值如下：

取值	含义
true	读
false	不读

③ MEMWrite：表示 Mem 阶段是否要写 Ram，其可能的取值如下：

取值	含义
true	写
false	不写

3.1.3 【WB 阶段】

① WBReg：表示要写回的寄存器，其可能的取值如下：

取值	含义
rx	寄存器 x
ry	寄存器 y
rz	寄存器 z
IH	寄存器 IH
SP	寄存器 SP
T	寄存器 T
RA	寄存器 RA
empty	空

② WBData：表示要写回寄存器的数据来源，其可能的取值如下：

取值	含义
alu_res	ALU 的运算结果
mem_read	Mem 阶段从 Ram 中读取的值
empty	空

3.2 指令对应控制信号

实现了 30 条要求指令。除 25 条基本指令外，拓展指令为 JALR、CMPI、ADDSP3、NOT、SLT 共 5 条。请详见“核心资料-控制信号.xlsx”

四、流水线模块实现

4.1 主要模块的功能与实现

4.1.1 ALU 模块

输入输出：

```
module ALU(
    input [15:0] input1,
    input [15:0] input2,
    input [7:0] opcode,
    output reg[15:0] result,
    output zero
    );
```

功能：根据操作码执行相应的加减，移位，比较运算

实现：组合逻辑，依据操作吗分别计算结果

```
assign zero = (result == 0)? 1'b1: 1'b0;

always @(opcode)
begin
  case (opcode)
      ADD: result = input1 + input2;
      SUB: result = input1 - input2;
      AND: result = input1 & input2;
      OR:  result = input1 | input2;
      XOR: result = input1 ^ input2;
      NOT: result = ~input2;
      SLL: begin
          if(input1 == 0)   result = input2 << 4'b1000;
          else  result = input2 << input1;
        end
      SRL: begin
          if(input1 == 0)  result = input2 >> 4'b1000;
          else result = input2 >> input1;
        end
      SRA:  begin
          if(input1 == 0)  result = input2 >>> 4'b1000;
          else result = input2 >>> input1;
      end
      EQUAL:
          if(input1 == input2) result = 0;
          else  result = ONE;
      NEQUAL:
          if(input1 == input2) result = ONE;
          else  result = 0;
      LESSTHEN:
          if(input1 < input2) result = ONE;
          else  result = 0;
      EMPTY:
          result = 0;
      default: result = opcode;
   endcase
end
```

4.1.2 流水线寄存器模块

包括：IF_ID_Reg, ID_EXE_Reg, EXE_MEM_Reg, MEM_WB_Reg

功能：上升沿时存入前一阶段中被后续步骤所需要的数据，实现控制信号和数据的锁存功能

4.1.3 寄存器堆模块

输入输出：

```
module Registers(
    input CLK,
    input RST,
    input boot,
    input [3:0] readx,
    input [3:0] ready,
    input [3:0] writeReg,
    input [15:0] writeData,
    output reg[15:0] out_rx,
    output reg[15:0] out_ry,
    output [15:0]out_IN,
    output [15:0]out_SP,
    output [15:0]out_T,
```

功能：

保存保存通用寄存器 R0~R7 的，以及其他 IN、SP、T、RA 寄存器的值。

实现：

读取时采用组合逻辑，输出寄存器标号对应的寄存器的值。写入时采用时序逻辑，时钟下降沿时写入。

```
//上升沿读
always @ (readx, ready)
begin
    out_rx = registers[readx];
    out_ry = registers[ready];
end

assign   out_IN = registers[IN_index];
assign   out_SP = registers[SP_index];
assign   out_T = registers[T_index];

//寄存器堆，下降沿写入数据，RSTboot = 0时所有寄存器清零
always @ (negedge CLK, negedge RSTboot)
begin
    if(RSTboot == 0) begin
        for(index = 0; index < N; index = index + 1) begin
            registers[index] <= 16'h0000;
        end
    end
    else begin
        registers[writeReg] <= writeData;
    end
end
```

4.1.4 Ram 模块

输入输出：

```
module Ram(
    input CLK,
    input RST,
    input boot,
    input [15:0] addrin,
    input [15:0] datain,
    input MemRead,
    input MemWrite,
    input [15:0] pc,
    input tbre, tsre, data_ready,
    output rdn, wrn,
    output ram1_en, ram2_en,
    output reg ram1_oe, ram1_rw,
    output reg ram2_oe, ram2_rw,
    output [17:0] ram1_addr,
    output [17:0] ram2_addr,
    output [15:0] instruction,
    output [15:0] dataout,
    output reg RamSlot,
    inout [15:0] ram1_data,
    inout [15:0] ram2_data
    );
```

功能：

集成 Ram1, Ram2 及串口，通过分析地址空间来决定进行何种操作。

由于考虑到串口与 Ram1 共用一根总线，因此选择 Ram2 作为指令寄存器，Ram1 作为数据寄存器，同时负责与串口的通信。

启动或重启时所有设备不工作。时钟上升沿，根据MemRead、MemWrite与Addrin设置各设备的工作状态：

(a) 读写程序区：Ram1与串口不工作，Ram2进行相应的读写，此时无法进行取指操作，输出的NOP的指令使得流水线暂停一个周期。

(b) 读写数据区：Ram1进行相应的读写，Ram2正常取指，串口不工作。

(c) 读串口状态：Ram1不工作，Ram2正常取指，将tsre, tbre, dataReady按位拼接返回。

(d) 读写串口：Ram1不工作，Ram2正常取指，串口通过Ram1的总线进行读写。

实现：

Ram 这个模块在实现时遇到了比较多的困难。由于之前做的 Ram 和串口的实验时不要求在一个周期中完成读写操作，在实验中对于如何 一个周期中进行多个操作遇到了比较多困难。

对于 Ram 的写操作需要先设置好总线数据，之后拉低写使能信号维持一段写保持时间后再拉高才能使得数据正确写入。串口的写入类似。

对于 Ram 的读操作，则需要先设置好总线为高阻态，之后拉低读使能信号保持一段时间后在复原。串口的读入类似。

一开始实现思路不够清晰，使得模块存在比较大的时序问题，虽然单步调试时能够正确的进行读写操作，不过一旦加上高频率的时钟就会导致出现

错误。经过反复试验，最后的实现方法是使用时序逻辑实现 Ram 模块的状态选择，在时钟上升沿时依据读入的地址以及读写使能信号来设置 Ram1,Ram2 以及串口，总线的状态。而 Ram 以及串口的控制信号的改变则通过时序逻辑实现，依据 CLK = 0 以及 CLK = 1 在一个时钟周期中进行两步操作。经过试验,使用这种方法能够有效解决时序问题,使得 Ram 和串口都能够工作在 25M 的时钟周期下。

程序代码比较长，详见工程源文件。

4.1.5 数据选择器模块

包括:PCMux, RegMux, ALU_Src1Mux, ALU_Src2Mux, MemDataMux,RegDataMux

功能： 根据控制信号选择出需要的数据

实现： 采用组合逻辑可以很方便的实现多路选择器功能

4.2 冲突处理

4.2.1 前向通路模块

输入输出：

```
module ForwardUnit(
    input [15:0] rxdata_in,
    input [15:0] rydata_in,
    input [15:0] IN_in,
    input [15:0] SP_in,
    input [15:0] T_in,
    input [3:0] rx_index,
    input [3:0] ry_index,
    input [3:0] MEM_WBReg,
    input [3:0] WB_WBReg,
    input [15:0] MEM_ALUResult,
    input [15:0] WB_WBData,
    output reg [15:0] rxdata_out,
    output reg [15:0] rydata_out,
    output reg [15:0] IN_out,
    output reg [15:0] SP_out,
    output reg [15:0] T_out
    );
    //数据旁路，接收MEM段（上一条指令），WB段（上两条指令）的数据，选择是否通过旁路传递数据
```

功能： 解决数据冲突，发生冲突时通过旁路传递数据

根据 THCO MIPS 指令发生冲突的位置可以将数据冲突分为 EXE 段冲突和 MEM 段冲突。其冒险检测条件为：

1a. EXE/MEM.RegisterRz = ID/EXE.RegisterRx

1b. EXE/MEM.RegisterRz = ID/EXE.RegisterRy

2a. MEM/WB.WBIdx = ID/EXE.RegisterRx

2b. MEM/WB.WBIdx = ID/EXE.RegisterRy

设计数据旁路 Forward Unit，它接收 EXE/MEM 与 MEM/WB 流水线寄存器的数据与控制信号，根据控制信号检测上述 4 个冒险条件有无发生，若发生，则需通过旁路传递数据。所有可能需要通过旁路传递的数据，包括寄存器 x、寄存器 y 以及特殊寄存器 IH、SP、T，都需要通过 EXE/MEM 与 MEM/WB 流水线寄存器及时传递给 Forward Unit。在做冒险检测时都应先检查 EXE 段冲突，若有发生，则取 EXE/MEM 中的数据传递；否则，再检测 MEM 段。再则，正常取源操作数执行指令即可。

EXE 段发生冲突：

```
LI R0 1
LI R1 1
LI R2 5
ADDU R1 R2 R3
ADDU R3 R0 R4
```

MEM 段发生冲突：

```
LI R0 1
LI R1 2
ADDU R0 R1 R2
LI R3 R5
ADDU R2 R3 R5
LI R4 0
```

通过加入旁路后，上面两端程序均可以正确得出结果而不需要插入额外的 NOP。

实现：

```
parameter idx_IN = 4'b1001, idx_SP = 4'b1010, idx_T = 4'b1011;
reg [15:0] rxdata_else, rydata_else, IN_else, SP_else, T_else;

//rx_index
always @(MEM_WBReg, rx_index)
begin
   if (MEM_WBReg == rx_index)
      rxdata_out = MEM_ALUResult;
   else
      rxdata_out = rxdata_else;
end

always @(WB_WBReg, rx_index)
begin
   if (WB_WBReg == rx_index)
      rxdata_else = WB_WBData;
   else
      rxdata_else = rxdata_in;
end

//ry_index
always @(MEM_WBReg, ry_index)
begin
   if (MEM_WBReg == ry_index)
      rydata_out = MEM_ALUResult;
   else
      rydata_out = rydata_else;
end

always @(WB_WBReg, ry_index)
begin
   if (WB_WBReg == ry_index)
      rydata_else = WB_WBData;
   else
      rydata_else = rydata_in;
end
```

4.2.2 冒险检测模块

输入输出：

```
module BubbleUnit(
    input [7:0] ID_Src1,
    input [7:0] ID_Src2,
    input [7:0] ID_MemSrc,
    input [7:0] ID_PCSrc,
    input [3:0] ID_rx_index,
    input [3:0] ID_ry_index,
    input EXE_MemRead,
    input [3:0] EXE_WBReg,
    input MEM_zero,
    input [7:0] MEM_PCSrc,
    output LoadSlot,
    output BranchSlot
    );
```

功能：

检测是否发生冒险和控制冲突，若有则插入气泡暂停流水线。一方面根据上一条指令是否为读内存且写当前指令使用的寄存器，判断是否需要插入 Load 延

迟。另一方面根据前面第三条指令是否跳转且成功，判断是否需要清空上一条指令的影响（前面第二条指令默认为 NOP 且执行）

解决冒险：

Bubble Unit 根据 ID/EXE 与 EXE/MEM 流水线寄存器中的控制信号进行冒险检测，检测条件：ID/EXE.MemRead = =true && (ID/EXE.RegisterRx = = IF/ID.WBIdx || ID/EXE.RegisterRy == IF/ID.WBIdx)

若有冒险发生，则插入 LoadSlot，即：让当前指令的控制信号全部为 0（不进行任何写入操作），同时保持 PC 与 IF/ID 寄存器中的值不变。

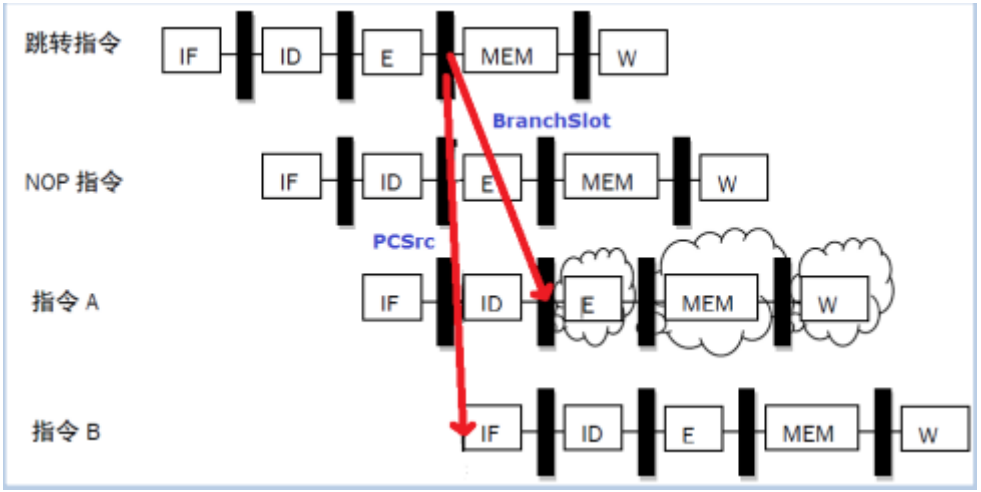
冒险实例：

```
LI R0 80
LI R5 0
SLL R0 R0 0
LI R1 60
SLL R1 R1 0
SW R0 R1 0
ADDIU R1 50
SW R0 R1 1
LI R2 50
LW R0 R3 0
ADDU R2 R3 R4
```

通过解决冒险插入气泡的方法，上面的程序可以真确执行。

控制冲突：

对于程序中的条件跳转指令，因为 THCO MIPS 跳转指令后的第一条指令默认为 NOP 且执行，所以我们需要合理的处理第二条指令的执行。



我们默认预测跳转失败，若确实不发生跳转，则指令流水仍按原来顺序正常执行，流水线不受影响；若预测失败，跳转发生，同时插入气泡，使得已经执行至 ID 译码阶段的控制信号全部置零。如上图，若需要跳转时，指令 A 已经执行到了 ID 阶段，这是我们通过检测清空 ID_EXE 寄存器堆，保证运行的正确。

实现：

```
wire both_SP, both_rx, both_ry;

assign   BranchSlot = ((MEM_zero == 1) && (MEM_PCSrc != NEXT));
assign   LoadSlot = (EXE_MemRead && (both_SP || both_rx || both_ry));
assign   both_SP = (ID_Src1 == SP) && (EXE_WBReg == idx_SP);
assign   both_rx = ((ID_Src1 == RX) || (ID_MemSrc == RX)
                  || (ID_PCSrc == JUMP)) && (EXE_WBReg == ID_rx_index);
assign   both_ry = ((ID_Src2 == RY) || (ID_MemSrc == RY))
                   && (EXE_WBReg == ID_ry_index);
/*
```

4.3 拓展功能与实现

4.3.1 软中断实现

实验中我们加入了对于软中断的支持，允许处理用户程序中的 INT 指令。不过由于没有具体的应用背景，我们目前在出现 INT 指令时简单的跳转至监控程序的 DelInt 部分，这部分的功能我们基本模仿了跳转指令的做法. 不过我们需要把放回地址存入寄存中以保证执行完中断部分后能够顺利放回原程序继续执行 0。

如下图是在监控程序中执行 INT 指令，可以看出程序转入了监控程序中指令中断的部分处理完后回到了源程序继续执行。

```
  >> A
[4000]  NOP
[4001]  LI R1 1
[4002]  LI R2 2
[4003]  INT 0
[4004]  ADDU R1 R2 R0
[4005]  LI R3 5
[4006]  LI R1 3
[4007]  LI R2 4
[4008]  NOP
[4009]  JR R7
[400a]  NOP
[400b]

  >> U
[4000]      <0800>      NOP
[4001]      <6901>      LI R1 0001
[4002]      <6a02>      LI R2 0002
[4003]      <f800>      INT 0000
[4004]      <e141>      ADDU R1 R2 R0
[4005]      <6b05>      LI R3 0005
[4006]      <6903>      LI R1 0003
[4007]      <6a04>      LI R2 0004
[4008]      <0800>      NOP
[4009]      <ef00>      JR R7

  >> A
[4000]

  >> G
int 指令中断

  >> R
R0=0003  R1=0003  R2=0004
R3=0005  R4=0000  R5=0000
  >>
```

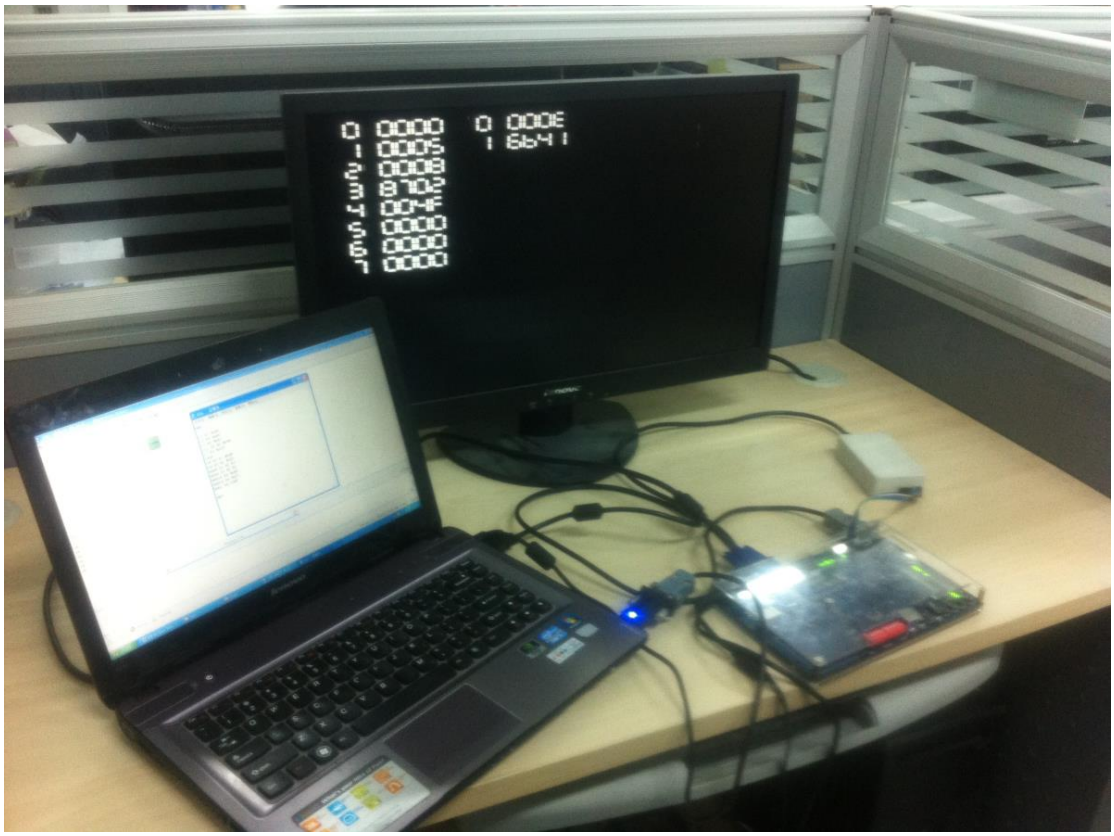
4.3.2 VGA 模块的实现

输入输出：

```
module VGATop (
  input clk50M, rst,
  input [159:0] registerVGA,
  input [15:0] pc,
  input [15:0] instruction,
  output vgaHs, vgaVs,
  output [2:0] vgaR, vgaG, vgaB
);
```

功能：

在 VGA 上显示 IF 阶段的寄存器值、PC 值与指令值，如下图所示（左侧由上到下依次是 R0 至 R7 寄存器的值，右侧由上到下依次是 PC 值、指令值），用于单步调试。



实现：

实现 VGA 的顶层文件为 VGATOP.v，首先解释一下其中的输入输出：clk50M 为频率为 50MHz 时钟,RST 为 reset 信号,registerVGA 记录寄存器信息(如[15:0]记录寄存器 R0 值;[31:16]记录寄存器 R1 值,依次类推。需要指出的是[159:128]暂定为空,便于以后有拓展需求),pc 为 PC 值,instruction 为当前指令值,vgaHs、vgaVs、vgaR、vgaG、vgaB 为 VGA 要求的输出信号。

VGA 要求的时钟频率为 25.18MHz 左右，所以先将 clk50M 分频为 clk25M。而实现主要依靠其下的两个模块：VGAControler 与 VGARenderer。

VGAControler 实现“扫描”控制，即根据 clk25M 和 RST 实现对坐标(x,y)和 vgaHs、vgaVs 的更新。VGARenderer 实现“渲染”控制，以信号 show 来表示当前扫描到的点(x, y)是否需要点亮（即 vgaR、vgaG、vgaB 均为 7，否则均为 0)。而我们根据寄存器、PC 值和指令值可以来预判断点亮的区域。

VGA 相关的各模块含义如下，具体实现详见工程代码：

模块	模块功能	需调用模块

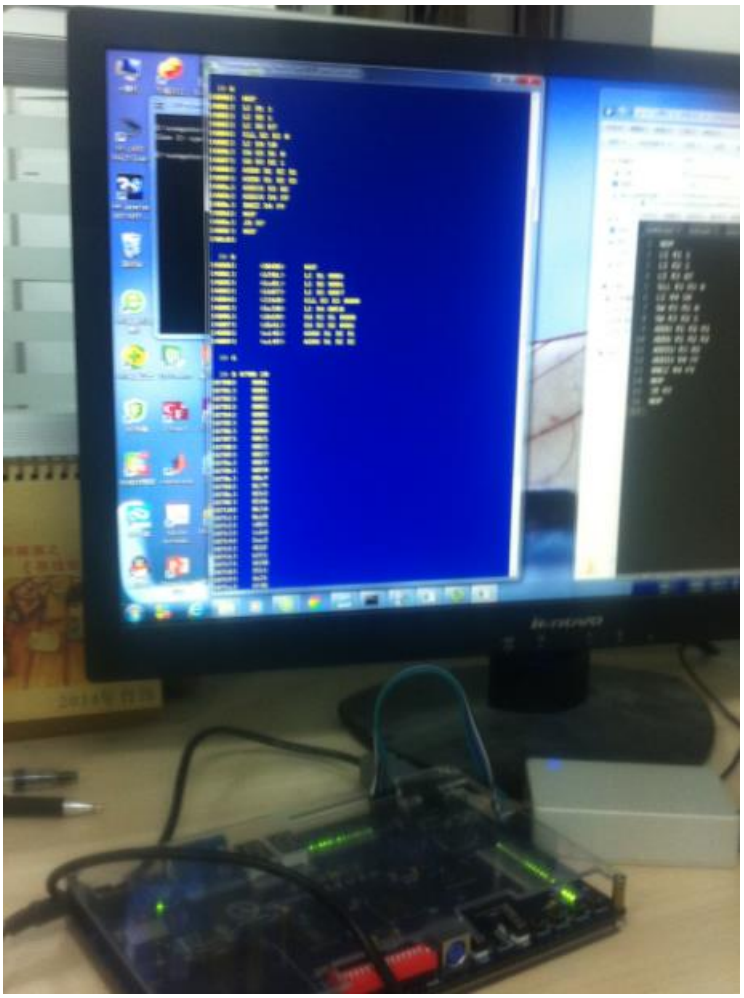
module VGATop (input clk50M, rst, input [159:0] registerVGA, input [15:0] pc, input [15:0] instruction, output vgaHs, vgaVs, output [2:0] vgaR, vgaG, vgaB);	VGA 模块实现的顶层模块	VGAControler VGARenderer
module VGAControler (input clk, rst, output reg hs, vs, output reg[10:0] x, y);	实现“扫描”控制	无
module VGARenderer (input [10:0] x, y, input [159:0] registers, input [15:0] pc, input [15:0] instruction, output reg [2:0] r, g, b);	实现“渲染”控制的顶层模块	VGARegisterRendererTop VGAOtherRendererTop
module VGARegisterRendererTop (input [10:0] x, y, input [10:0] center_x, center_y, input [159:0] registers, output wor show);	实现 R0-R7 渲染的顶层模块	VGARegisterRenderer
module VGAOtherRendererTop(input [10:0] x, y, input [10:0] center_x, center_y, input [15:0] pc, input [15:0] instruction, output wor show);	实现其他渲染的顶层模块	VGARegisterRenderer
module VGARegisterRenderer (input [10:0] x, y, input [10:0] center_x, center_y, input [3:0] registerIndex, input [15:0] registerValue, output wor show);	实现寄存器值显示	VGANumberRenderer VGADigitRenderer
module VGANumberRenderer (input [10:0] x, y, input [10:0] center_x, center_y, input [15:0] number, output wor show);	实现四位 16 进制数显示	VGADigitRenderer

module VGADigitRenderer (input [10:0] x, y, input [10:0] center_x, center_y, input [3:0] digit, output wor show);	实现单个数字显示	VGARectRenderer
module VGARectRenderer (input enable, input [10:0] x, y, input [10:0] center_x, center_y, input [10:0] width, height, output reg show);	判断(x, y)是否在该矩阵范围内显示	无

五、实验结果测试

5.1 基本运行测试

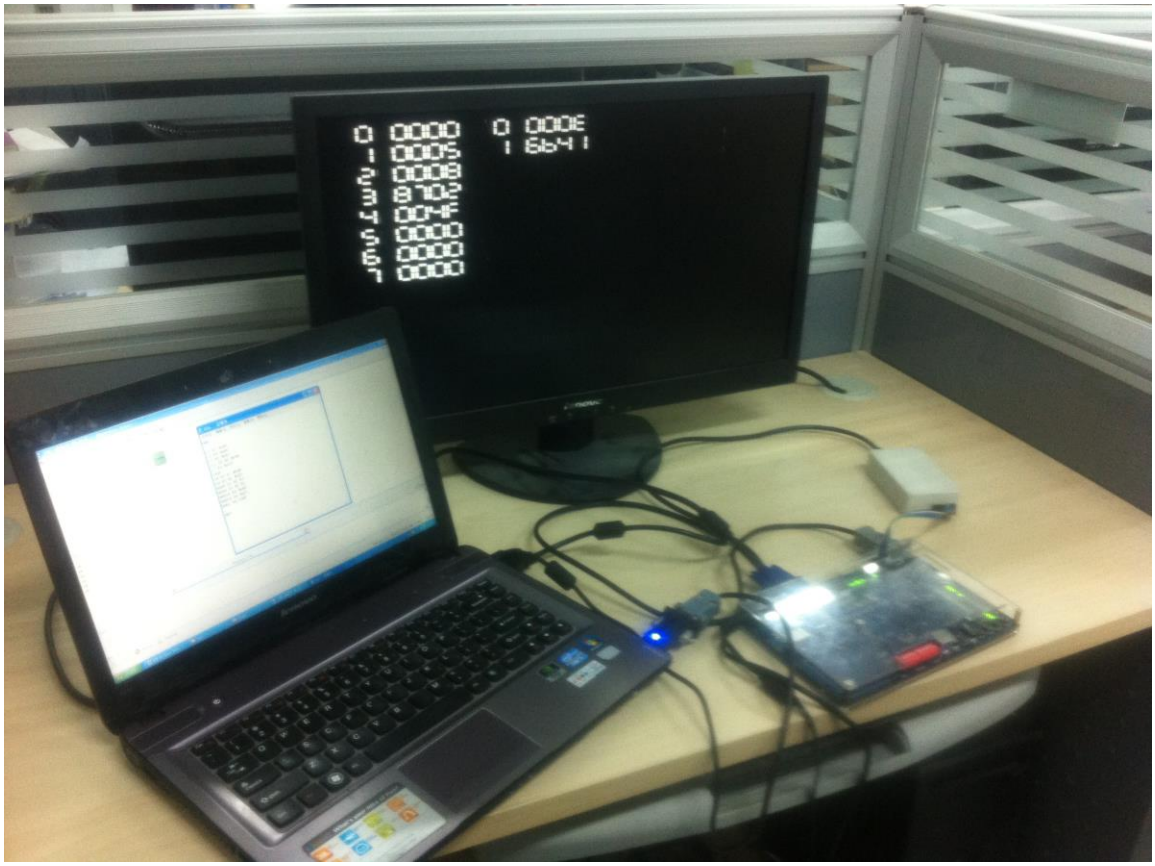
测试显示能够正常运行 kernel 的各项操作。



5.2 VGA 测试

VGA 显示各寄存器内容正常。

视频资料请见“测试资料—VGA 效果演示.MOV”



5.3 编程测试

5.3.1 测试斐波那契函数

```
>> A
[4000]  NOP
[4001]  LI R1 1
[4002]  LI R2 1
[4003]  LI R3 87
[4004]  SLL R3 R3 0
[4005]  LI R4 50
[4006]  SW R3 R1 0
[4007]  SW R3 R2 1
[4008]  ADDU R1 R2 R1
[4009]  ADDU R1 R2 R2
[400a]  ADDIU R3 02
[400b]  ADDIU R4 FF
[400c]  BNEZ R4 F9
[400d]  NOP
[400e]  JR R7
[400f]  NOP
[4010]

>> U
[4000]      <0800>      NOP
[4001]      <6901>      LI R1 0001
[4002]      <6a01>      LI R2 0001
[4003]      <6b87>      LI R3 0087
[4004]      <3360>      SLL R3 R3 0000
[4005]      <6c50>      LI R4 0050
[4006]      <db20>      SW R3 R1 0000
[4007]      <db41>      SW R3 R2 0001
[4008]      <e145>      ADDU R1 R2 R1
[4009]      <e149>      ADDU R1 R2 R2

>> G

>> D 8700 20
[8700]    0001
[8701]    0001
[8702]    0002
[8703]    0003
[8704]    0005
[8705]    0008
[8706]    000d
[8707]    0015
[8708]    0022
[8709]    0037
[870a]    0059
[870b]    0090
[870c]    00e9
[870d]    0179
[870e]    0262
[870f]    03db
[8710]    063d
[8711]    0a18
[8712]    1055
[8713]    1a6d
[8714]    2ac2
[8715]    452f
[8716]    6ff1
[8717]    b520
[8718]    2511
[8719]    da31
[871a]    ff42
```

5.3.2 测试跳转指令

```
  >> A
[4000]  NOP
[4001]  LI R1 23
[4002]  LI R2 1
[4003]  LI R3 3
[4004]  LI R4 4
[4005]  LI R6 82
[4006]  SLL R6 R6 0
[4007]  SW R6 R2 0
[4008]  SLT R1 R2
[4009]  BTEQZ 2
[400a]  NOP
[400b]  LI R3 33
[400c]  CMPI R2 1
[400d]  BTEQZ 2
[400e]  NOP
[400f]  LI R4 44
[4010]  JR R7
[4011]  NOP
[4012]

  >> U 4000 15
[4000]      <0800>      NOP
[4001]      <6923>      LI R1 0023
[4002]      <6a01>      LI R2 0001
[4003]      <6b03>      LI R3 0003
[4004]      <6c04>      LI R4 0004
[4005]      <6e82>      LI R6 0082
[4006]      <36c0>      SLL R6 R6 0000
[4007]      <de40>      SW R6 R2 0000
[4008]      <e942>      SLT R1 R2
[4009]      <6002>      BTEQZ 0002
[400a]      <0800>      NOP
[400b]      <6b33>      LI R3 0033
[400c]      <7201>      CMPI R2 0001
[400d]      <6002>      BTEQZ 0002
[400e]      <0800>      NOP
[400f]      <6c44>      LI R4 0044
[4010]      <ef00>      JR R7
[4011]      <0800>      NOP
[4012]      <442f>      ADDIU3 R4 R1 ffff
[4013]      <8323>      --- UNKNOWN ---
[4014]      <b850>      --- UNKNOWN ---

  >> G

  >> R
R0=0003  R1=0023  R2=0001
R3=0003  R4=0004  R5=0000
  >>
```

5.3.3 测试 SP 相关指令

```
R3-0005  R4-0000  R5-0000
  >> A
[4000]  NOP
[4001]  LI R1 23
[4002]  LI R2 2
[4003]  LI R3 3
[4004]  LI R4 81
[4005]  SLL R4 R4 0
[4006]  MTSP R4
[4007]  SW_SP R1 0
[4008]  LW_SP R2 0
[4009]  ADDSP 1
[400a]  ADDIU R1 15
[400b]  SW_SP R1 0
[400c]  LW_SP R3 0
[400d]  NOP
[400e]  JR R7
[400f]  NOP
[4010]

  >> U
[4000]      <0800>      NOP
[4001]      <6923>      LI R1 0023
[4002]      <6a02>      LI R2 0002
[4003]      <6b03>      LI R3 0003
[4004]      <6c81>      LI R4 0081
[4005]      <3480>      SLL R4 R4 0000
[4006]      <6480>      MTSP R4
[4007]      <d100>      SW_SP R1 0000
[4008]      <9200>      LW_SP R2 0000
[4009]      <6301>      ADDSP 0001

  >> G

  >> D 8100
[8100]    0023
[8101]    0038
[8102]    d462
[8103]    460d
```

5.3.4 测试 EXE 冲突

```
  >> A
[4000]  LI R0 1
[4001]  LI R1 1
[4002]  LI R2 5
[4003]  ADDU R1 R2 R3
[4004]  ADDU R3 R0 R4
[4005]  JR R7
[4006]  NOP
[4007]

  >> U
[4000]      <6801>      LI R0 0001
[4001]      <6901>      LI R1 0001
[4002]      <6a05>      LI R2 0005
[4003]      <e14d>      ADDU R1 R2 R3
[4004]      <e311>      ADDU R3 R0 R4
[4005]      <ef00>      JR R7
[4006]      <0800>      NOP
[4007]      <de40>      SW R6 R2 0000
[4008]      <e942>      SLT R1 R2
[4009]      <6002>      BTEQZ 0002

  >> G

  >> R
R0=0001   R1=0001   R2=0005
R3=0006   R4=0007   R5=0000
  >>
```

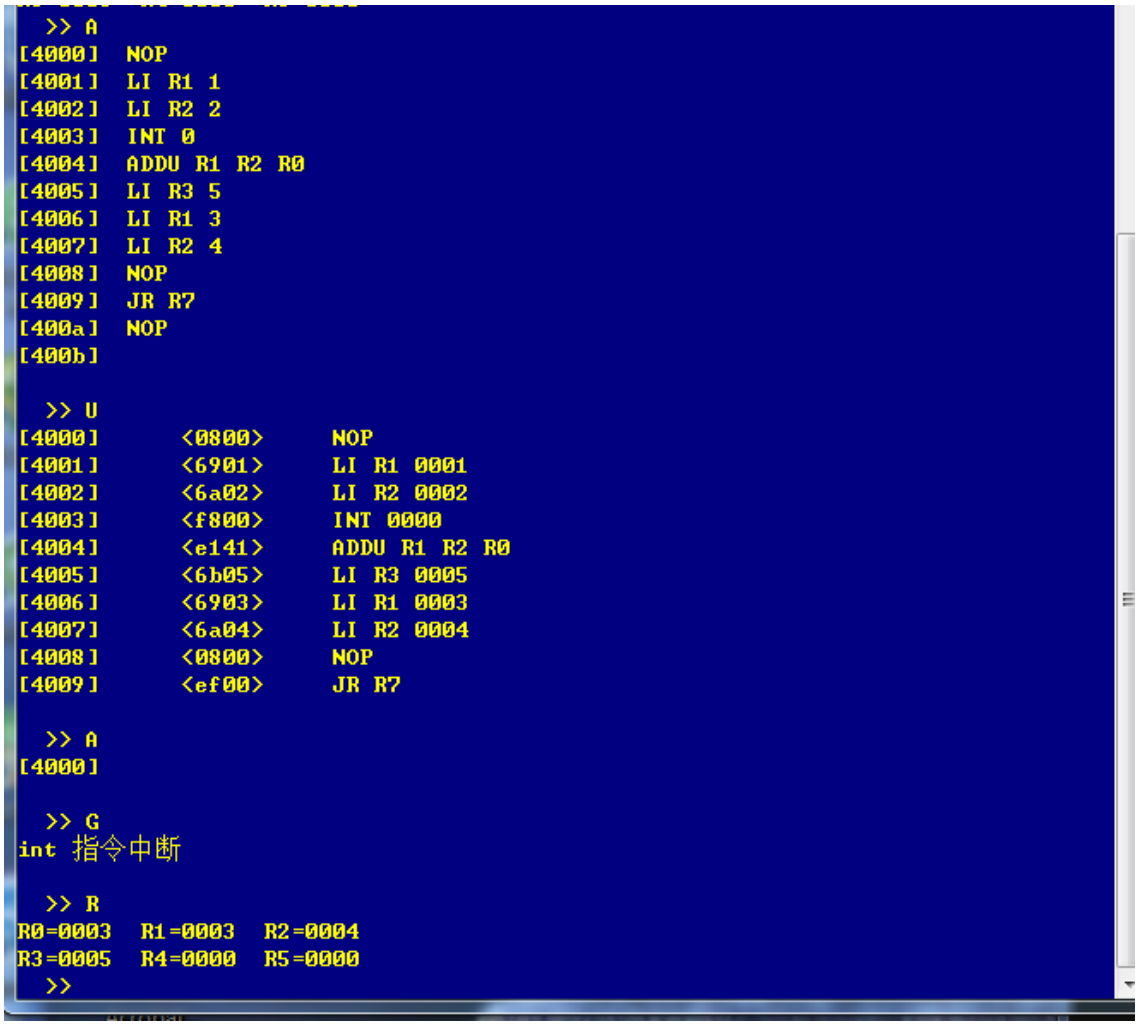
5.3.5 测试 MEM 冲突

```
R3=0005   R4=0007   R5=0008
  >> A
[4000]  LI R0 1
[4001]  LI R1 2
[4002]  ADDU R0 R1 R2
[4003]  LI R3 5
[4004]  ADDU R2 R3 R5
[4005]  LI R4 0
[4006]  JR R7
[4007]  NOP
[4008]

  >>

  >> G

  >> R
R0=0001   R1=0002   R2=0003
R3=0005   R4=0000   R5=0008
  >>
```

5.3.6 测试 LoadSlot

```
  >> A
[4000]  LI R0 80
[4001]  LI R5 0
[4002]  SLL R0 R0 0
[4003]  LI R1 60
[4004]  SLL R1 R1 0
[4005]  SW R0 R1 0
[4006]  ADDIU R1 50
[4007]  SW R0 R1 1
[4008]  LI R2 50
[4009]  LW R0 R3 0
[400a]  ADDU R2 R3 R4
[400b]  JR R7
[400c]  NOP
[400d]

  >> U
[4000]      <6880>      LI R0 0080
[4001]      <6d00>      LI R5 0000
[4002]      <3000>      SLL R0 R0 0000
[4003]      <6960>      LI R1 0060
[4004]      <3120>      SLL R1 R1 0000
[4005]      <d820>      SW R0 R1 0000
[4006]      <4950>      ADDIU R1 0050
[4007]      <d821>      SW R0 R1 0001
[4008]      <6a50>      LI R2 0050
[4009]      <9860>      LW R0 R3 0000

  >> G

  >> R
R0=8000  R1=6050  R2=0050
R3=6000  R4=6050  R5=0000
  >> D 8000
[8000]    6000
[8001]    6050
[8002]    2114
[8003]    0599
[8004]    24dd
[8005]    1a35
[8006]    5911
[8007]    4cba
[8008]    0083
[8009]    6119

  >>
```

5.3.7 测试 INT 软中断



六、实验总结与体会

6.1 实验心得概述

本次计原实验已经结束，回顾这次实验，我们组在完成基本要求的基础之外完成了自我设定的各项拓展，收获颇丰。

我们一致认为开始代码前的【前期准备】是实验中最重要的环节。因此我们花了大量的时间设计并讨论数据通路与控制信号，也与老师、助教和学长进行了交流，确保设计的科学性；两人商量分工，明确个人任务以提高合作效率；比对 VHDL 与 Verilog 的优缺点，并最终选择 Verilog 作为编程语言；明确【实验核心思想】，不打算纯粹为了加分而拓展（如某些组 VGA、键盘等外设的添加与本

次实验核心并没有关系），以更好地完成 CPU 为本次实验核心。

实验的中心环节自然是【代码编写】和【调试环节】。遵循我们的实验核心思想，我们选择用 VGA 显示各寄存器以及 PC、指令值，便于单步调试；花了大量时间在后期的各项调试上，尤其是编程调试，确保 30 条指令的正确性。过程中多次遇到了 Ram 与串口等外设的问题，通过反复调试和与同学们积极交流最终都得到了解决。实验整体上完成得也比较顺利，完成检查也比较早。

虽然我们做的附加项（如 Flash 自启动、键盘等）可能不是特别多，但我们确保尽可能好地完成了本次实验的核心部分，保证 30 条指令的正确性。检查时和李山山老师交流过我们的实验核心思想，并得到了老师的肯定，也给了我们莫大的鼓励。下面通过多方面的总结进一步说明我们的体会与心得。

6.2 合理分工提高效率

实验前期的数据通路以及控制信号的设计共同讨论完成，保证每个人对于整个 CPU 有一个大体框架认识。之后主要由一名同学负责搭建 CPU 的总体框架，主要包括根据数据流图指定的各个模块，定义好其输入输出端口，同时搭建好 CPU 顶层模块，完成各部分的连线工作。有了这个整体框架后，由另一名同学具体实现大部分的模块。对于 Ram 这个逻辑比较复杂的模块则是在后期一起讨论实现。

由一个人来搭建顶层模块，另一名同学实现具体模块的功能，这样一方面保证了设计得完整性，同时也利于两名同学互相分工合作。此外，实验中的一个经验教训就是变量名的定义一定要统一，因为由于几个变量名事先没有太明确的定义，使得另一名同学使用时理解错误。通常这样的问题比较不好发现，调试起来相当费时。

6.3 VeriLog 使用心得

这次的实验我们没有使用 VHDL 语言，而是采用了 Verilog 语言。通过对比两种语言可以发现，Verilog 语言在很大程度上更接近 C 语言习惯，同时语言风格比 VHDL 更简洁。通过这次实验我们在使用 Verilog 的过程中对于这种硬件描述语言有了比较多的体会：

① 使用 parameter 来定义常量

Verilog 中的 parameter 语句类型 c 语言中的 define 语句，通过对于各

种控制信号指定名称，不仅使得程序的可读性提高，同时也利于程序调试。这样写还能够大大减少程序修改的代价。例如我们要修改一个控制信号的取值，我们只需要修改 parameter 定义语句，而不需要修改内部程序。

```
parameter ADDU_SUBU = 5'b11100, ADDIU = 5'b01001, ADDIU3 = 5'b01000,
          ADDSP_BTEQZ_MTSP = 5'b01100, AND_OR_CMP_JALR_NOT_SLT_JR_MFPC = 5'b11101,
          B = 5'b00010, BEQZ = 5'b00100, BNEZ = 5'b00101,
          LI = 5'b01101, LW = 5'b10011, LW_SP = 5'b10010,
          SW = 5'b11011, SW_SP = 5'b11010, MFIN_MTIN = 5'b11110,
          SLL_SRA = 5'b00110,INT = 5'b11111,
          NOP = 5'b00001, CMPI = 5'b01110, ADDSP3 = 5'b00000;

parameter NEXT = 8'b00000001, BRANCH8 = 8'b00000010, BRANCH11 = 8'b00000011,
          JUMP = 8'b00000100, RX = 8'b00000101, RY = 8'b00000110, RZ = 8'b00000111,
          IN = 8'b00001000, SP = 8'b00001001, T = 8'b00001010, EMPTY = 8'b00001011,
          idx_IN = 8'b00001001, idx_SP = 8'b00001010, idx_T = 8'b00001011,
          idx_EMPTY = 8'b00001111, ZERO = 8'b00010000, Z_IMM3 = 8'b00010001,
          Z_IMM8 = 8'b00010010, PC = 8'b00010011, S_IMM4 = 8'b00010100,
          S_IMM5 = 8'b00010101, S_IMM8 = 8'b00010110, ALU_RES = 8'b00010111,
          MEM_READ = 8'b00011000, ADD = 8'b00011001, SUB = 8'b00011010,
          AND = 8'b00011011, OR = 8'b00011100, XOR = 8'b00011101, NOT = 8'b00011110,
          SLL = 8'b00011111, SRL = 8'b00100000, SRA = 8'b00100001, ROL = 8'b00100010,
          EQUAL = 8'b00100011, NEQUAL = 8'b00100100, LESSTHEN = 8'b00100101,
          ONE = 8'b00100110, RA = 8'b00100111, INTJUMP = 8'b00101000, R5 = 8'b00101001;
```

② 组合逻辑与时序逻辑的合理分解

通过这次实验，我们对于数字电路的认识有了很大的提高。一开始编写程序时我们往往直接按照软件编写的思路，只考虑电路逻辑，而不考虑电路能否综合实现。结果导致很多模块写完之后烧到 FPGA 上运行会出现各种奇怪的错误。反思下来便是我们在写 always 模块（类似 VHDL 的 process）时，其中的逻辑过于混杂，既包含时序逻辑又包含组合逻辑的功能，不符合目前的编译器能够综合的语言风格。因此我们后来编程时注意分析好电路功能，采用组合逻辑实现逻辑功能需求，采用时序逻辑控制状态的改变。

③ 大量使用状态机

对于如何编写 Verilog 使得电路综合效果最好，我们查阅了很多相关的教程，其中我们认识到最适合电路实现的模型是状态机模型。基于这个思想，比较复杂的模块，我们在混合时序逻辑和组合逻辑时采用的思路是：将模块的功能分为几个简单的组合逻辑电路，每个部分分配一个控制信号来控制状态机的状态。这样在时钟上升沿时我们改变各个状态值，状态值的改变可以相应的驱动不同的组合逻辑模块进行工作。这样么个组合逻辑模块就能够互不影响的工作。同时实验中也感受到这样编写程序综合出来的电路的时序效果很好。

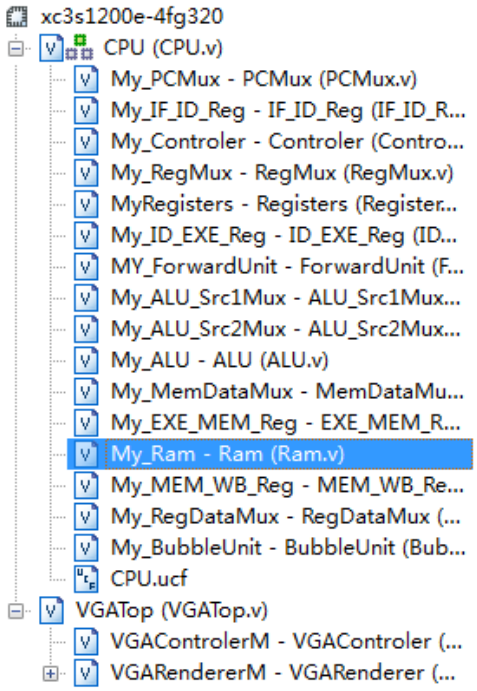
④ Assign 语句的使用

Verilog 中的 assign 语句针对的是线网类型 wire，这是一种实时驱动的方式。Assign 语句用来实现组合逻辑思路清晰，同时综合效果最好。因此我们在实现组合逻辑部分时尽可能的使用了 assign 语句，使得信号之间互不影响。在 Ram 模块中，我们就是通过大量修改 always 模块为一系列的 Assign 语句来解决我们遇到的时序问题。

```
assign ram1_data = (bus1_read)? 16'bzzzzzzzzzzzzzzzz: ((com_read)?16'b00000000zzzzzzzz: datain);
assign ram2_data = (bus2_read)? 16'bzzzzzzzzzzzzzzzz: datain;

assign com_readSign = (data_ready == 1'b1) ? 1'b1: 1'b0;
assign com_writeSign = (tbre == 1'b1 && tsre == 1'b1)? 1'b1: 1'b0;

assign ram1_addr = {2'b00, addrin};
assign ram2_addr = (ram2State == RAM2_NORMAL)? ({2'b00, pc}):({2'b00, addrin});
assign instruction = (insState == INS_NORMAL)?(ram2_data):(16'h0800);
assign dataout = (outState == OUT_RAM1DATA)?(ram1_data):
                 ((outState == OUT_RAM2DATA)?(ram2_data):
                 ((outState == OUT_PORTSIGN)?({14'b00000000000000, com_readSign, com_writeSign}):
                 (16'h0000)));
assign ram1_en = (ram1State == RAM1_DISABLE)? 1'b1:1'b0;
assign ram2_en = 1'b0;

assign rdn = (portState == PORT_READ)? 1'b0: 1'b1;
assign wrn = (portState == PORT_WRITE)? 1'b0: 1'b1;
```

⑤ 模块划分

实验中我们尽可能的将模块细分，使得每个 module 中的逻辑都相对比较简单。模块的细分有利于程序的编写，同时有利于分工编写程序之后再使用顶层 module 组装起来。



6.4 外设调试心得

实验中可以说调试外设花费的时间占的比重最大。我们花了 3 天左右完成了大部分程序的编写。不过在进入调试后却发现 Ram 和串口无法正常工作。并且外设的调试很多时候毫无头绪，往往一天就白白花费了。现在总结起来就是由于之前做的 Ram 和串口的实验时不要求在一个周期中完成读写操作，而在流水 CPU 中，需要保证在一个周期中完成外设的读写工作。不过正如前面说到的，通过改进程序的组织方式，合理划分组合逻辑和时序逻辑，使得模块的时序性能提高。

至于 VGA 部分，代码中碰到的问题不大，主要是调整显示数字的中心位置和大小，保证显示效果较好。此外，一开始显示器显示有零散的多余亮纹，纠结了很久，后来发现是显示器硬件本身的问题而不是代码的问题。

6.5 汇编程序测试心得

在完成实验的过程中，我们将 THCO MIPS 汇编语言进行了分组，对每组指令都编写了大量的汇编程序进行测试。除了测试指令的正确性，我们还编写了各种程序来进行 RAM，串口的调试，同时编写了程序检测冲突处理功能以及冒险检测功能的正确性。

6.6 沟通交流很重要

实验中我们遇到问题如果长时间无法解决，我们会主动同其他组的同学进行交流。实验中我们遇到的一些问题有些恰好已经被其他组同学解决了，这样可以节省比较多的时间。同时交流过程中我们也能帮助其他组的同学解决一些问题。这样的沟通讨论便能够取得双赢。

此外，与老师及助教的沟通交流也不可或缺。感谢老师和助教在实验过程中所给予的指导！

七、附录文件说明

文件夹	文件(夹)	功能
工程代码	*.v	Verilog 文件
	cpu.ucf	管脚绑定文件

测试资料	VGA 效果演示.mov	VGA 单步调试演示视频
	测试代码	编程测试源文件和生成文件，测试内容如文件名所示
	测试截图	各项测试的截图，测试内容如文件名所示
核心资料	控制信号.xlsx	各条指令的控制信号
	数据通路.png	数据通路图示
	cpuVGA.bit	用于单步调试的 bit 文件
	cpuFinal.bit	最终版本的 bit 文件
	term.cpp	修改的 Term.cpp
	term	生成的 Term 文件
	kernel.asm	修改的 kernel.asm
	kernel	生成的 kernel 文件