搭建流水线CPU完成字符串搜索

无04 2019012137 张鸿琳

一. 实验目的

了解5级流水线MIPS处理器原理，合理解决各种数据冒险和控制冒险，搭建流水线CPU，用该CPU完成字符串搜索，并利用外设显示结果。

二. 实验原理

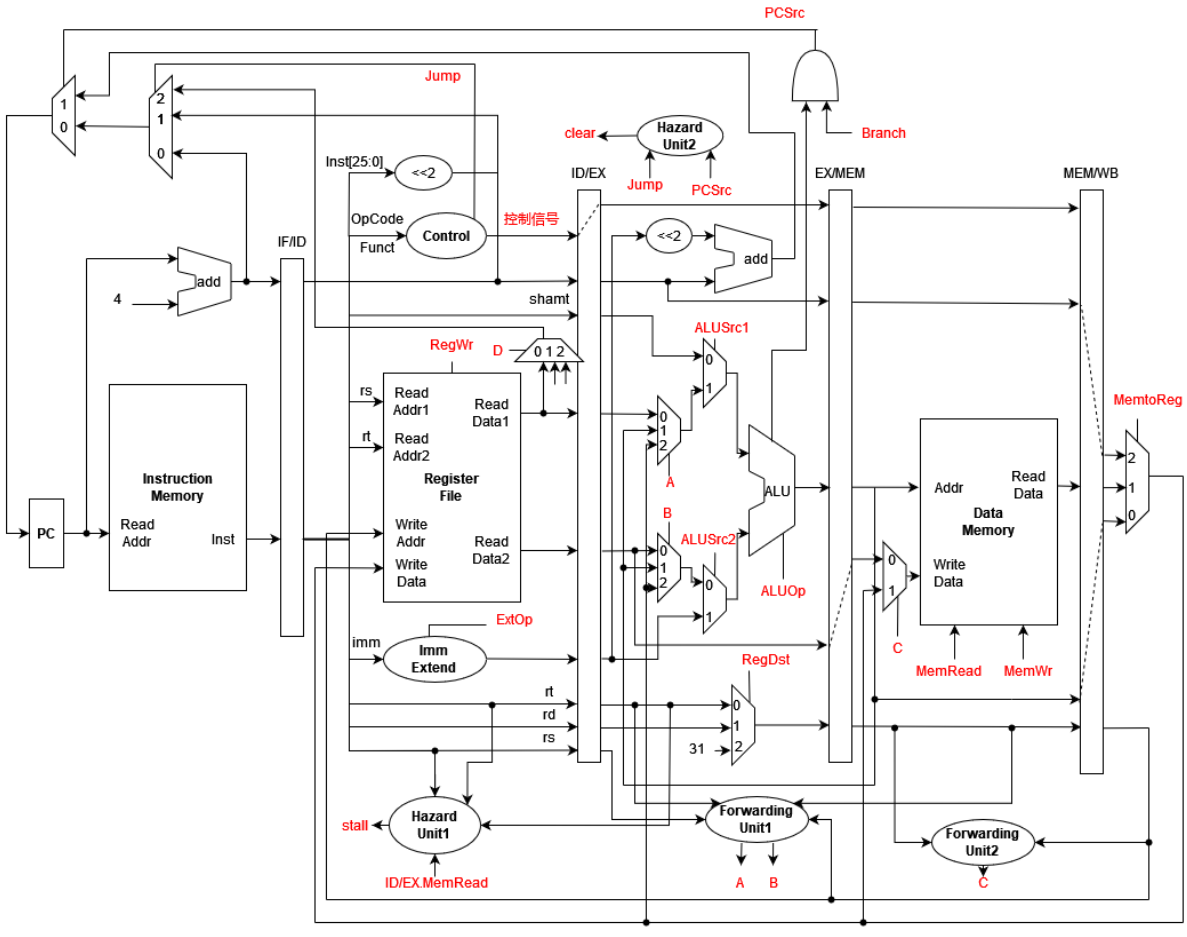
流水线CPU将单个指令拆分为5个阶段（IF、ID、EX、MEM、WB），在同一时间各个阶段对应的硬件资源可以执行不同的指令，从而提高硬件资源的利用效率，也降低了CPI，使得CPU性能得以提升。

三. 实验内容

设计一个5级流水线的MPIS处理器，其可以利用转发和阻塞操作正确处理数据冒险，同时可以利用转发和清空操作正确处理控制冒险。该MPIS处理器需要适当对指令集进行扩充，使其能正确执行字符串搜索程序，同时可以通过读取和写入外设寄存器，达到和外设通信的效果，便于输入测试数据和输出搜索结果。

四. 设计方案

在编写verilog代码前，为了能更好地理清楚思路，我绘制了如下5级流水线MIPS处理器的数据通路（该图中没有具体绘出外设寄存器的部分）：



我设计的5级流水线MIPS处理器的分支指令的判断发生在EX阶段，而跳转指令发生在ID阶段，支持beq、bne、blez、bgtz、bltz分支指令，支持j、jal、jr、jalr跳转指令，同时支持sb、lb指令。

图中红色的为控制信号，其功能分别如下：

- RegWr：用于判断是否需要写入寄存器堆
- ExtOp：用于控制立即数扩展方式，即有符号扩展、无符号扩展、lui扩展
- ALUSrc1：用于控制ALU的输入1是位移量还是寄存器读取出的数据
- ALUSrc2：用于控制ALU的输入2是扩展后的立即数还是寄存器读取出的数据
- ALUOp：用于控制ALU计算方式
- RegDst：用于控制寄存器堆输入地址
- Branch：用于判断是否为分支指令
- Jump：用于判断是否为跳转指令
- MemRead：用于判断是否读取数据存储器
- MemWr：用于判断是否需要写入数据存储器
- MemtoReg：用于控制写入寄存器堆的数据来源

图中的Forwarding Unit和Hazard Unit用于控制转发、阻塞、清空操作，不过在利用代码实现时，为了逻辑清晰，我将其整合为了ControlHazard和DataHazard两个模块，用于处理各种冒险。其他部分基本在单周期、多周期CPU部分有涉及，故而不再赘述。

在搭建上述CPU后，对指令存储器进行初始化，我采用了代码比较简便的暴力搜索算法，当执行完成后，CPU将进入死循环。此后又引入了外设寄存器的接口，从而可以将测试数据通过uart串口输入CPU，CPU将程序执行完成后，又可以通过uart串口输出搜索结果。最后我还实现了结果在数码管上的显示，数码管的刷新是由软件控制的，但数码管显示的数字是由硬件映射实现的。（根据老师的要求，实现了uart通信就可以无需显示结果在数码管上了，所以这部分没有完全严格按照软件输入数码管外设寄存器的思路实现，因为这样需要在程序上做较多改动）

五. 文件清单以及关键代码解读

文件清单

文件清单如下

- 实验报告
- PipelineCPU代码文件：
    - verilog源代码——UsePipelineCPU.v（顶层测试模块）
      ——PipelineCPU.v（流水线CPU模块）
      ——ALU.v（ALU运算模块）
      ——Control.v（控制信号生成模块）
      ——ControlHazard.v（控制冒险处理模块）
      ——DataHazard.v（数据冒险处理模块）
      ——DataMemory.v（数据存储器模块）
      ——EX_MEM_Reg.v（EX/MEM寄存器模块）
      ——ID_EX_Reg.v（ID/EX寄存器模块）
      ——IF_ID_Reg.v（IF/ID寄存器模块）
      ——ImmExtend.v（立即数扩展模块）
      ——InstMemory.v（指令存储器模块）
      ——MEM_WB_Reg.v（MEM/WB寄存器模块）
      ——PC.v（PC地址寄存器模块）

——RegisterFile.v（寄存器堆模块）

——uart_mem.v（uart外设寄存器模块）

——uart_rx.v（uart串口接收模块）

——uart_tx.v（uart串口发射模块）

  - 仿真文件——test.v
  - 约束文件——PipelineCPU.xdc
- 尝试进一步优化的PipelineCPU代码文件：（此处为"硬件调试情况、性能评估与进一步优化"部分尝试进一步优化的CPU的代码文件）
  - verilog源代码——UsePipelineCPU.v（顶层测试模块）

——PipelineCPU.v（流水线CPU模块）

——ALU.v（ALU运算模块）

——Control.v（控制信号生成模块）

——ControlHazard.v（控制冒险处理模块）

——DataHazard.v（数据冒险处理模块）

——DataMemory.v（数据存储器模块）

——EX_MEM_Reg.v（EX/MEM寄存器模块）

——ID_EX_Reg.v（ID/EX寄存器模块）

——IF_ID_Reg.v（IF/ID寄存器模块）

——ImmExtend.v（立即数扩展模块）

——InstMemory.v（指令存储器模块）

——MEM_WB_Reg.v（MEM/WB寄存器模块）

——PC.v（PC地址寄存器模块）

——RegisterFile.v（寄存器堆模块）

——uart_mem.v（uart外设寄存器模块）

——uart_rx.v（uart串口接收模块）

——uart_tx.v（uart串口发射模块）

  - 仿真文件——test.v
  - 约束文件——PipelineCPU.xdc

关键代码解读

最顶层的代码为UsePipelineCPU.v，其具体代码如下：

UsePipelineCPU代码

```
module UsePipelineCPU(
    input clk,      // 100MHz
    input rst,      // S1
    input mem2uart, // SW0, send message
    input Rx_Serial,
    output Tx_Serial,
    output [7:0] Cathodes,
    output reg [1:0] AN
```

```
    );

    parameter CLKS_PER_BIT = 16'd10417;  // 100M/9600

    reg send_done;
    reg [31:0] addr;
    reg rd_en;
    reg wr_en;
    wire [31:0] rdata;
    reg [31:0] wdata;


    PipelineCPU PipelineCPU(
        .reset(rst),
        .clk(clk),
        .out_wr(wr_en),
        .out_wdata(wdata),
        .out_address(addr),
        .out_read(rdata),
        .out_rd(rd_en)
        );

    /*--------------------------------UART RX-------------------------*/
    wire Rx_DV;
    wire [7:0] Rx_Byte;
    reg clear;

    uart_rx #(.CLKS_PER_BIT(CLKS_PER_BIT)) uart_rx_inst
        (.i_Clock(clk),
        .i_Rx_Serial(Rx_Serial),
        .o_Rx_DV(Rx_DV),
        .o_Rx_Byte(Rx_Byte),
        .clear(clear)
        );

    /*--------------------------------UART TX-------------------------*/
    reg Tx_DV;
    reg [7:0] Tx_Byte;
    wire Tx_Active;
    wire Tx_Done;


    uart_tx #(.CLKS_PER_BIT(CLKS_PER_BIT)) uart_tx_inst
        (.i_Clock(clk),
        .i_Tx_DV(Tx_DV),
        .i_Tx_Byte(Tx_Byte),
        .o_Tx_Active(Tx_Active),
        .o_Tx_Serial(Tx_Serial),
        .o_Tx_Done(Tx_Done)
        );


    reg [31:0] cntByteTime;

    always@(posedge clk or posedge rst)begin
        if(rst)begin
            addr <= 32'd0;
            rd_en <= 1'b0;
```

```
            wr_en <= 1'b0;
            wdata <= 32'd0;
            send_done <= 1'b0;
            cntByteTime <= 32'd0;
            Tx_DV <= 1'b0;
            Tx_Byte <= 8'd0;

            clear<=0;
            AN<=0;

        end
        else
        begin

            // uart to memory
            if(Rx_DV)begin
                // receive data
                addr <= 32'h4000001c;
                clear <= 1'd1;
                wr_en <= 1'b1;
                wdata <= {24'h0,Rx_Byte};
            end
            else begin
                wr_en <= 1'b0;
                clear <= 1'd0;
            end

            // memory to uart
            if(mem2uart==1'b1)begin
                // 1Byte time
                if(send_done == 0 && cntByteTime == CLKS_PER_BIT*20)begin
                    Tx_DV <= 1'b1;
                    send_done <= 1;
                end
                else if(send_done == 0 && cntByteTime !=
CLKS_PER_BIT*20)begin
                    rd_en<= 1'b1;
                    addr<=32'h40000018;
                    Tx_Byte <= rdata[7:0];
                    cntByteTime <= cntByteTime+1'b1;
                    Tx_DV <= 1'b0;
                end
                else begin
                    Tx_DV <= 1'b0;
                    rd_en<= 1'b1;
                    addr<=32'h40000010;
                    AN<= rdata[9:8];
                end
            end
            else begin
                AN<=0;
            end
        end
    end

    wire [7:0] temp;

    assign temp=(mem2uart==1'b0)?8'h00:Tx_Byte;
```

```
124
125     assign Cathodes=(AN==2'b01 && temp[3:0]==4'b0000)?8'b00111111://0
126                     (AN==2'b01 && temp[3:0]==4'b0001)?8'b00000110://1
127                     (AN==2'b01 && temp[3:0]==4'b0010)?8'b01011011://2
128                     (AN==2'b01 && temp[3:0]==4'b0011)?8'b01001111://3
129                     (AN==2'b01 && temp[3:0]==4'b0100)?8'b01100110://4
130                     (AN==2'b01 && temp[3:0]==4'b0101)?8'b01101101://5
131                     (AN==2'b01 && temp[3:0]==4'b0110)?8'b01111101://6
132                     (AN==2'b01 && temp[3:0]==4'b0111)?8'b00000111://7
133                     (AN==2'b01 && temp[3:0]==4'b1000)?8'b01111111://8
134                     (AN==2'b01 && temp[3:0]==4'b1001)?8'b01101111://9
135                     (AN==2'b01 && temp[3:0]==4'b1010)?8'b01110111://a
136                     (AN==2'b01 && temp[3:0]==4'b1011)?8'b01111100://b
137                     (AN==2'b01 && temp[3:0]==4'b1100)?8'b00111001://c
138                     (AN==2'b01 && temp[3:0]==4'b1101)?8'b01011110://d
139                     (AN==2'b01 && temp[3:0]==4'b1110)?8'b01111001://e
140                     (AN==2'b01 && temp[3:0]==4'b1111)?8'b01110001://f
141                     (AN==2'b10 && temp[7:4]==4'b0000)?8'b00111111://0
142                     (AN==2'b10 && temp[7:4]==4'b0001)?8'b00000110://1
143                     (AN==2'b10 && temp[7:4]==4'b0010)?8'b01011011://2
144                     (AN==2'b10 && temp[7:4]==4'b0011)?8'b01001111://3
145                     (AN==2'b10 && temp[7:4]==4'b0100)?8'b01100110://4
146                     (AN==2'b10 && temp[7:4]==4'b0101)?8'b01101101://5
147                     (AN==2'b10 && temp[7:4]==4'b0110)?8'b01111101://6
148                     (AN==2'b10 && temp[7:4]==4'b0111)?8'b00000111://7
149                     (AN==2'b10 && temp[7:4]==4'b1000)?8'b01111111://8
150                     (AN==2'b10 && temp[7:4]==4'b1001)?8'b01101111://9
151                     (AN==2'b10 && temp[7:4]==4'b1010)?8'b01110111://a
152                     (AN==2'b10 && temp[7:4]==4'b1011)?8'b01111100://b
153                     (AN==2'b10 && temp[7:4]==4'b1100)?8'b00111001://c
154                     (AN==2'b10 && temp[7:4]==4'b1101)?8'b01011110://d
155                     (AN==2'b10 && temp[7:4]==4'b1110)?8'b01111001://e
156                     (AN==2'b10 && temp[7:4]==4'b1111)?8'b01110001://f
157                     8'b0;
158
159 endmodule
```

该部分代码的输入Rx_Serial为uart串口输入，Tx_Serial为uart串口输出，Cathodes和AN则用于控制数码管显示结果，当mem2uart被拉高时，则会向uart串口输出结果，并将结果显示在数码管上。代码中实例化了PipelineCPU，其中输入的wr_en、wdata、addr、rdata、rd_en，分别用于控制CPU中的外设寄存器的写入使能、写入数据、写入(或读取)地址、读取数据、读取使能。并且实例了uart串口的接收和输出模块，分别为uart_rx_inst和uart_tx_inst。

PipelineCPU.v的代码则基本完全按照设计方案中的数据通路实现，需要补充的部分是额外增加了uart外设寄存器，用于和外设通信，该部分的代码如下：

PipelineCPU中关于数据存储器及外设寄存器部分的代码

```
1     wire [31:0] writedata;
2     wire [31:0] mem_readdata;
3     wire [31:0] mem_readdatax;
4     wire [31:0] mem_readdata1;
5     wire [31:0] mem_readdata2;
6     DataMemory DataMemory(
7         .reset(reset),
8         .clk(clk),
9         .Address(mem_aluout),
```

```
        .write_data(writedata),
        .MemWrite(mem_memwr),
        .Mem_data(mem_readdatax),
        .Loadbyte(mem_loadbyte)
        );

        assign mem_readdata1=mem_loadbyte? ((!(mem_aluout[1:0]^2'b11))?
    {{24{mem_readdatax[7]}},mem_readdatax[7:0]}:(!(mem_aluout[1:0]^2'b10))?
    {{24{mem_readdatax[15]}},mem_readdatax[15:8]}:(!
(mem_aluout[1:0]^2'b01))?
    {{24{mem_readdatax[23]}},mem_readdatax[23:16]}:
    {{24{mem_readdatax[31]}},mem_readdatax[31:24]}):mem_readdatax;

        wire read_cpu;
        wire [31:0] addr_cpu;
        uart_mem UARTMemory(
        .clk(clk),
        .reset(reset),
        .addr(out_address),
        .addr_cpu(addr_cpu),
        .rd_en(out_rd),
        .rd_en_cpu(read_cpu),
        .wr_en(out_wr),
        .wr_en_cpu(mem_memwr),
        .wdata(out_wdata),
        .wdata_cpu(writedata),
        .rdata(out_read),
        .rdata_cpu(mem_readdata2)
    );

    assign mem_readdata=(mem_aluout[31:28]==4'h4)?
mem_readdata2:mem_readdata1;
    assign addr_cpu=(mem_aluout[31:28]==4'h4)? mem_aluout:32'h0;
    assign read_cpu=(mem_aluout[31:28]==4'h4)? mem_memread:0;

    assign writedata=(C==0)?mem_regdata:writereg;
```

代码思路是，当CPU读取（或写入）地址为开头四位为0100（即0x4）时，就读取（或写入）外设寄存器中的相应位置，否则就正常使用数据存储器进行读写（我也根据这样的设计编写了相应的软件程序）。通过时刻检查0x40000020地址的外设寄存器的3bit，判断是否有数据输入uart串口，检测到有输入之后则读取0x4000001C地址的数据。最后输出结果也是将数据存入0x40000018地址的外设寄存器，并将0x40000020地址寄存器的相应位置拉高。同时外设寄存器应该具有检测读取操作，进而刷新0x40000020地址寄存器相应位置的功能。

由这样的设计思路，可以进一步具体写出外设寄存器uart_mem.v代码如下：

uart_mem代码

```
module uart_mem
(
    input clk,
    input reset,
    input [31:0] addr,
    input [31:0] addr_cpu,
    input rd_en,
    input rd_en_cpu,
    input wr_en,
```

```
    input wr_en_cpu,
    output wire [31:0] rdata,
    output wire [31:0] rdata_cpu,
    input [31:0] wdata,
    input [31:0] wdata_cpu
    );

    reg [7:0] UART_TX;
    reg [7:0] UART_RX;
    reg tx_done;//tx完成
    reg rx_done;//rx完成
    reg state;//表示模块状态，如果为1则模块处于发射状态
    reg [1:0] AN;

    assign rdata=rd_en? ((addr[7:0]==8'h18)?{{24{UART_TX[7]}},UART_TX}:
                        (addr[7:0]==8'h1c)?{{24{UART_RX[7]}},UART_RX}:
                        (addr[7:0]==8'h20)?
{27'h0,state,rx_done,tx_done,2'h0}:
                        (addr[7:0]==8'h10)?{22'h0,AN,8'h0}:32'h0):32'h0;

    assign rdata_cpu=rd_en_cpu? ((addr_cpu[7:0]==8'h18)?
{{24{UART_TX[7]}},UART_TX}:
                        (addr_cpu[7:0]==8'h1c)?{{24{UART_RX[7]}},UART_RX}:
                        (addr_cpu[7:0]==8'h20)?
{27'h0,state,rx_done,tx_done,2'h0}:32'h0):32'h0;

    always@(posedge clk or posedge reset)begin
    begin
        if(reset)begin
            UART_TX<=8'b0;
            UART_RX<=8'b0;
            tx_done<=1'b0;
            rx_done<=1'b0;
            state<=1'b0;
        end
        else begin
            case(addr[7:0])
                8'h18:begin//tx_info
                    if(rd_en)begin
                        tx_done<=0;
                    end
                end
                8'h1c:begin//rx_info
                    if(wr_en)begin
                        UART_RX<=wdata[7:0];
                        rx_done<=1;
                    end
                end
            endcase
            case(addr_cpu[7:0])
                8'h18:begin//tx_info
                    if(wr_en_cpu)begin
                        UART_TX<=wdata_cpu[7:0];
                        tx_done<=1;
                    end
                end
                8'h1c:begin//rx_info
                    if(rd_en_cpu)begin
```

```
                        rx_done<=0;
                    end
                end
                8'h20:begin
                    if(wr_en_cpu)begin
                        state<=wdata_cpu[4];
                    end
                end
                8'h10:begin
                    if(wr_en_cpu)begin
                        AN<=wdata_cpu[9:8];
                    end
                end
            endcase
            end
        end
    end

endmodule
```

为了减少硬件资源占用，该外设寄存器只含有六组寄存器，其中UART_TX低8位存储向外发送的数据，UART_RX低8位存储接收到的数据，tx_done表示UART_TX中数据是否已经可以发送，rx_done表示UART_RX中数据是否已经可以接收，state表示当前发送模块状态，而AN则用于刷新两个显示结果的数码管的使能信号。当UART_TX被写入时，tx_done被拉高，而其被读取时，tx_done被自动清零，rx_done也是类似机制。

此外，两个比较重要的模块是冒险检测处理单元：DataHazard以及ControlHazard。其代码分别如下：

DataHazard代码

```
module DataHazard(//solve load-use hazard and other data hazard
    input wire ex_mem_regwr,
    input wire mem_wb_regwr,
    input wire ex_mem_memwr,
    input wire [4:0] if_id_rt,
    input wire [4:0] if_id_rs,
    input wire [4:0] id_ex_rs,
    input wire [4:0] id_ex_rt,
    input wire id_ex_memread,
    input wire [4:0] ex_mem_regaddr,
    input wire [4:0] mem_wb_regaddr,
    output wire [1:0] A,
    output wire [1:0] B,
    output wire C,
    // output wire [1:0] D,
    output wire keep,
    output wire flush_id_ex
    );

    //load-use, keep pc, if/id and flush id/ex
    assign keep=(id_ex_memread && ((id_ex_rt==if_id_rs) ||
(id_ex_rt==if_id_rt)))?1:0;
    assign flush_id_ex=(id_ex_memread && ((id_ex_rt==if_id_rs) ||
(id_ex_rt==if_id_rt)))?1:0;

    //forward
```

```
    assign A=(ex_mem_regwr && ~ex_mem_regaddr &&
(ex_mem_regaddr==id_ex_rs))? 1:
                (mem_wb_regwr && ~mem_wb_regaddr &&
(mem_wb_regaddr==id_ex_rs))? 2:0;
    assign B=(ex_mem_regwr && ~ex_mem_regaddr &&
(ex_mem_regaddr==id_ex_rt))? 1:
                (mem_wb_regwr && ~mem_wb_regaddr &&
(mem_wb_regaddr==id_ex_rt))? 2:0;

    //lw-sw
    assign C=(ex_mem_memwr && (ex_mem_regaddr==mem_wb_regaddr))?1:0;

endmodule
```

ControlHazard代码

```
module ControlHazard(
    input [1:0] Jump,
    input PCSrc,
    output flush_if_id,
    output flush_id_ex
    );

    assign flush_if_id=(Jump!=0 || PCSrc!=0)?1:0;
    assign flush_id_ex=(PCSrc!=0)?1:0;

endmodule
```

可以看到代码思路还是比较清晰的，对于数据冒险，处理的方法是：

- 若是一般的数据冒险，将前一条或前前一条指令的ALU执行结果转发到当前EX阶段
- 若是load-use数据冒险，则先阻塞一个周期，再进行转发，阻塞时保持PC和IF/ID寄存器中数据不变，而将ID/EX寄存器中数据清除
- 若是lw-sw数据冒险，则将lw读取出的数据转发到MEM阶段

对于控制冒险，处理的方法是：

- 若是判断发生了跳转指令，则清除IF/ID寄存器中数据，因为跳转指令的判断都发生在ID阶段
- 若是判断发生了分支指令，则清除IF/ID寄存器和ID/EX寄存器中数据，因为分支指令的判断都发生在EX阶段

要特别说明的是，由于跳转指令判断的统一提前，导致jr和jalr指令和前一条指令的数据冒险不能再由原DataHazard模块处理，需要增加新的阻塞和转发机制，不过考虑到jr和jalr指令的使用有限，且发生数据冒险的概率也很低，所以为了保证CPU性能，没有再更改这部分代码，并尽量避免在软件中出现这样的特殊的数据冒险。

除上面这些代码外，其余代码在单周期、多周期CPU中均已经涉及或较为简单，不再展示并举例论述，只是做大致说明：

- ALU：ALU运算单元，可以进行基本的算术运算、逻辑运算和比较操作
- Control：根据OpCode以及Funct生成控制信号的模块，控制信号的生成发生在ID阶段
- DataMemory：数据存储器，该数据存储器增加了支持lb和sb指令的功能，由于按照字节存取比较节省空间，所以将大小减为128×32 bit
- InstMemory：指令存储器
- PC：PC指令地址存储器，存在保持开关
- EX_MEM_Reg：存储介于EX阶段和MEM阶段的数据和控制信号
- ID_EX_Reg：存储介于ID阶段和EX阶段的数据和控制信号，存在清除开关

- IF_ID_Reg：存储介于IF阶段和ID阶段的数据和控制信号，存在保持和清除开关
- MEM_WB_Reg：存储介于MEM阶段和WB阶段的数据和控制信号
- ImmExtend：立即数扩展单元
- RegisterFile：寄存器堆
- uart_rx：uart串口接收单元
- uart_tx：uart串口发送单元

最后展示汇编程序，如下：

汇编程序

```
.text
main:

li $s7, 1073741848 #存储TX地址
li $s6, 1073741852 #存储RX地址
li $s5, 1073741856 #状态存储地址
li $a1, 0          #str的地址存在$a1
li $a3, 256      #pattern的地址存在$a3

#先不断读取0x40000020外设寄存器中的第3位，若变为1，则进行读取
li $s0, 0 #len_str = 0
li $s1, 0 #len_pattern = 0
add $t4, $a1, $zero #暂存str地址
add $t3, $a3, $zero #暂存pattern地址
waitstr:
lw $t0, 0($s5)
sll $t0, $t0, 28
srl $t0, $t0, 31
beqz $t0, waitstr #发现了字节输入

#read str
read_str_entry:
slti $t0, $s0, 256 #如果读入数据充满全部空间则跳出
beqz $t0, waitpattern
lw $t0, 0($s6)
addi $t1, $zero, '!' #读到"!"换行则退出
beq $t0, $t1, waitpattern
sb $t0, 0($t4)
addi $t4, $t4, 1 #地址增加1位
addi $s0, $s0, 1 #记录总共读入了多少字节
j waitstr

waitpattern:
lw $t0, 0($s5)
sll $t0, $t0, 28
srl $t0, $t0, 31
beqz $t0, waitpattern #发现了字节输入

read_pattern_entry:
slti $t0, $s1, 256 #如果读入数据充满全部空间则跳出
beqz $t0, read_pattern_exit
lw $t0, 0($s6)
addi $t1, $zero, '!' #读到"!"换行则退出
beq $t0, $t1, read_pattern_exit
sb $t0, 0($t3)
addi $t3, $t3, 1 #地址增加1位
```

```
addi $s1, $s1, 1 #记录总共读入了多少字节
j waitpattern

read_pattern_exit:
#call brute_force
move $a0, $s0
move $a2, $s1
jal brute_force

loop:
beq $zero, $zero, loop

#str的地址存在$a1，其长度存在$a0，pattern的地址存在$a3，其长度存在$a2
brute_force:
li $t0, 0 #i=0
li $t2, 0 #cnt=0
sub $t5, $a0, $a2 #$t5存储len_str-len_pattern的值
move $t9, $a1 #把str地址存在$t9，防止改变$a1
move $t8, $a3 #把pattern地址存在$t8，防止改变$a3

loopout: #外循环，对i循环
li $t1, 0 #j=0
move $t7, $t9 #把str[i]地址暂存在$t7
move $t6, $t8 #把pattern地址暂存在$t6

loopin: #内循环，对j循环
lb $t3, 0($t7) #把str[i+j]存在$t3
lb $t4, 0($t6) #把pattern[j]存在$t4
bne $t4, $t3, if #判断str[i+j]与pattern[j]是否相等，不相等则跳出循环到if
addi $t1, $t1, 1 #j+=1
addi $t6, $t6, 1 #pattern地址加一
addi $t7, $t7, 1 #str地址加一
blt $t1, $a2, loopin #j<len_pattern，继续循环

if:
bne $t1, $a2, cnt_not_plus #如果j==len_pattern，cnt+=1
addi $t2, $t2, 1 #cnt+=1

cnt_not_plus:
addi $t0, $t0, 1 #i+=1
addi $t9, $t9, 1 #str[i]地址加一
ble $t0, $t5, loopout #i<=len_str-len_pattern，继续循环

move $v0, $t2 #返回cnt值
sw $v0, 0($s7) #更新状态

li $t0, 16
sw $t0, 0($s5)
li $s4, 256
li $s3, 768
li $v1, 1073741840

show:  #进入数码管刷新循环
li $t1, 10000
showin:
subi $t1, $t1, 4
bnez $t1, showin
xor $s4, $s4, $s3
```

```
sw $s4, 0($v1)
j show
jr $ra
```

汇编程序的思路为：不断读取地址为0x40000020外设寄存器中的数据，假如读取到uart串口接收标志位为1时，就读取1个字节，并将其存入数据存储器中，从str字节数组首地址开始不断存入（str为主字符串），若标志位为0，则进入等待死循环。当读取到作为换行符的“!”时，进入pattern（模式字符串）读取循环，和str的读取类似（不过是从pattern字节数组首地址开始不断存入），当读到"!"换行符时，跳转进入brute_force暴力搜索子程序，执行完毕后，将结果存入地址为0x40000018的外设寄存器，并调整外设寄存器中地址为0x40000020的相应标志位。此后进入数码管刷新循环，按照一定周期刷新两个用于显示结果的数码管。

六. 仿真结果及分析

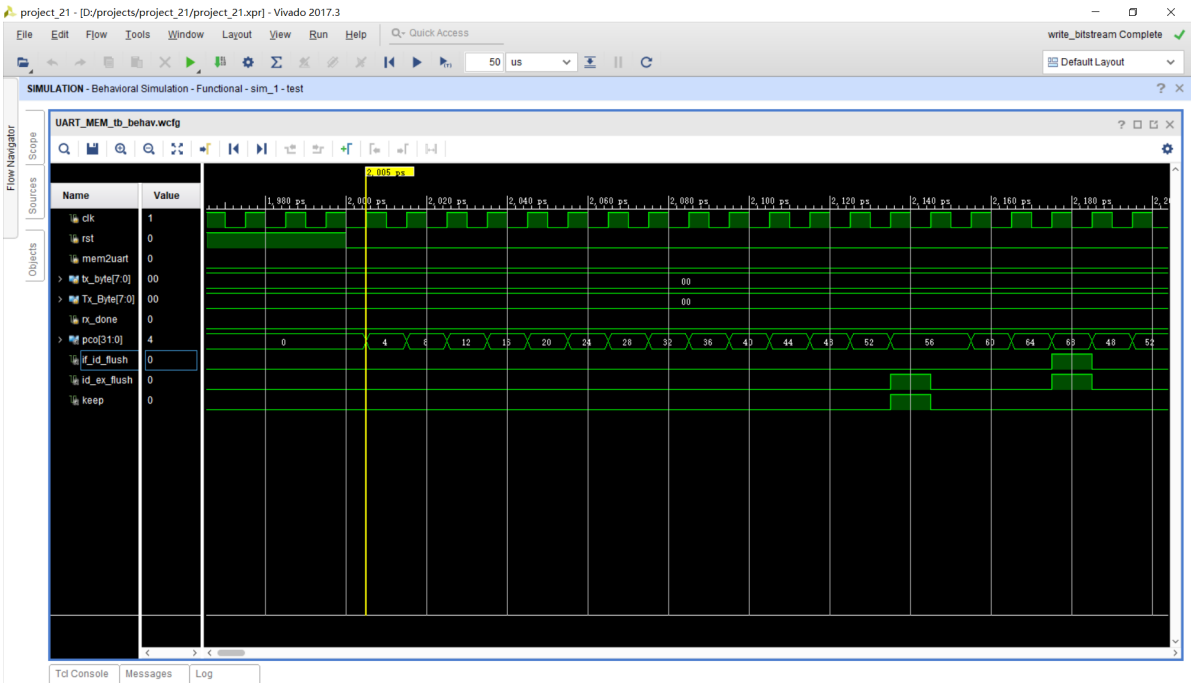
仿真代码如下：

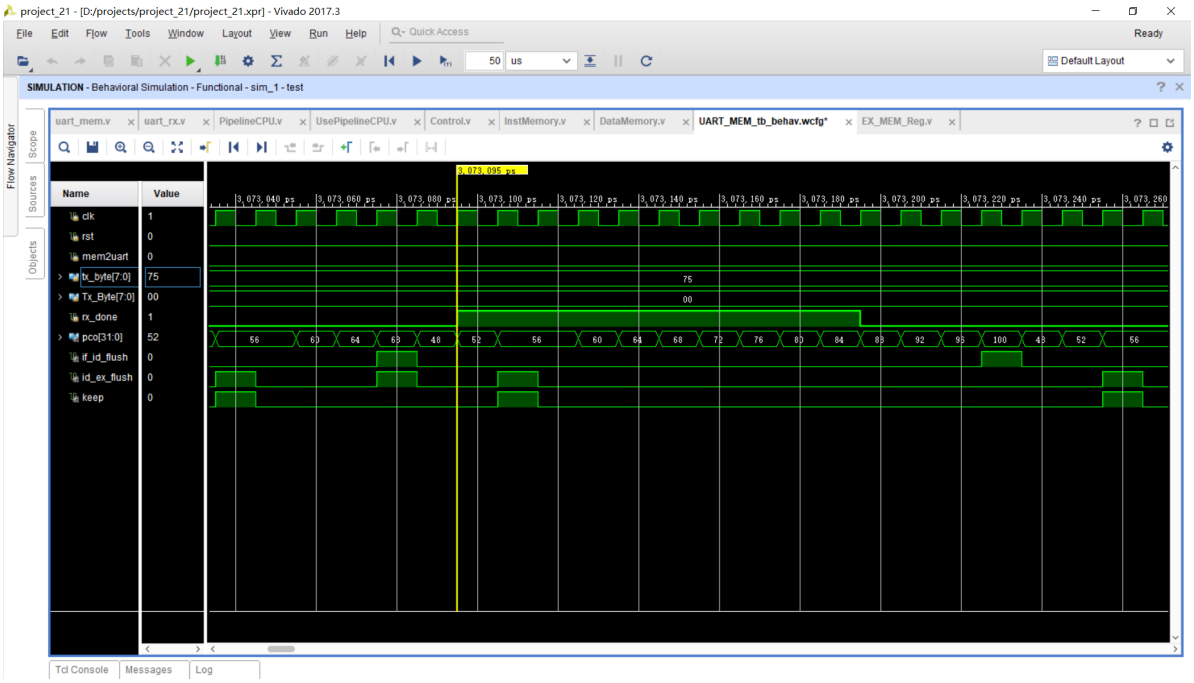
```
`timescale 1ps / 1ps
`define PERIOD 10

module test();

reg clk;
reg rst;
reg mem2uart;
reg tx_dv;
reg [7:0] tx_byte;
reg [31:0] cntByteTime;
reg [5:0] byte_cnt;
reg info_done;

parameter CLKS_PER_BIT=32'd10417;

wire rx;
wire tx;

UsePipelineCPU usepipelinecpu(
    .clk(clk),
    .rst(rst),
    .mem2uart(mem2uart),
    .Rx_Serial(rx),
    .Tx_Serial(tx)
    );

uart_tx sendinfo
  (
   .i_Clock(clk),
   .i_Tx_DV(tx_dv),
   .i_Tx_Byte(tx_byte),
   .o_Tx_Serial(rx)
   );

initial begin
    rst=0;
    mem2uart=0;
    clk=0;
    mem2uart=0;
```

```
    tx_dv=0;
    tx_byte=0;
    cntByteTime=0;
    byte_cnt=0;
    info_done=1;
    #(1000)rst=1;
    #(1000)rst=0;
    #(1000)info_done=0;
    #(50000000) mem2uart=1;
end

initial begin
    forever
      #(`PERIOD/2) clk=~clk;
end

always @(posedge clk)
begin
    if(cntByteTime == CLKS_PER_BIT*20 && info_done==0)begin  // 1Byte time
        cntByteTime <= 32'd0;
        tx_dv <= 1'b1;

        if(byte_cnt==6'd0) tx_byte <= "u";
        else if(byte_cnt==6'd1) tx_byte <= "n";
        else if(byte_cnt==6'd2) tx_byte <= "u";
        else if(byte_cnt==6'd3) tx_byte <= "i";
        else if(byte_cnt==6'd4) tx_byte <= "x";
        else if(byte_cnt==6'd5) tx_byte <= " ";
        else if(byte_cnt==6'd6) tx_byte <= "i";
        else if(byte_cnt==6'd7) tx_byte <= "s";
        else if(byte_cnt==6'd8) tx_byte <= " ";
        else if(byte_cnt==6'd9) tx_byte <= "p";
        else if(byte_cnt==6'd10) tx_byte <= "u";
        else if(byte_cnt==6'd11) tx_byte <= "n";
        else if(byte_cnt==6'd12) tx_byte <= "u";
        else if(byte_cnt==6'd13) tx_byte <= "n";
        else if(byte_cnt==6'd14) tx_byte <= "u";
        else if(byte_cnt==6'd15) tx_byte <= 8'h21;
        else if(byte_cnt==6'd16) tx_byte <= "u";
        else if(byte_cnt==6'd17) tx_byte <= "n";
        else if(byte_cnt==6'd18) tx_byte <= "u";
        else if(byte_cnt==6'd19)begin
            tx_byte <= 8'h21;
            info_done<=1;
        end
        else;

        if(byte_cnt == 6'd20)begin
            byte_cnt <= 3'd0;
        end
        else begin
            byte_cnt <= byte_cnt+1'b1;
        end
    end
    else begin
        cntByteTime <= cntByteTime+1'b1;
        tx_dv <= 1'b0;
    end
```

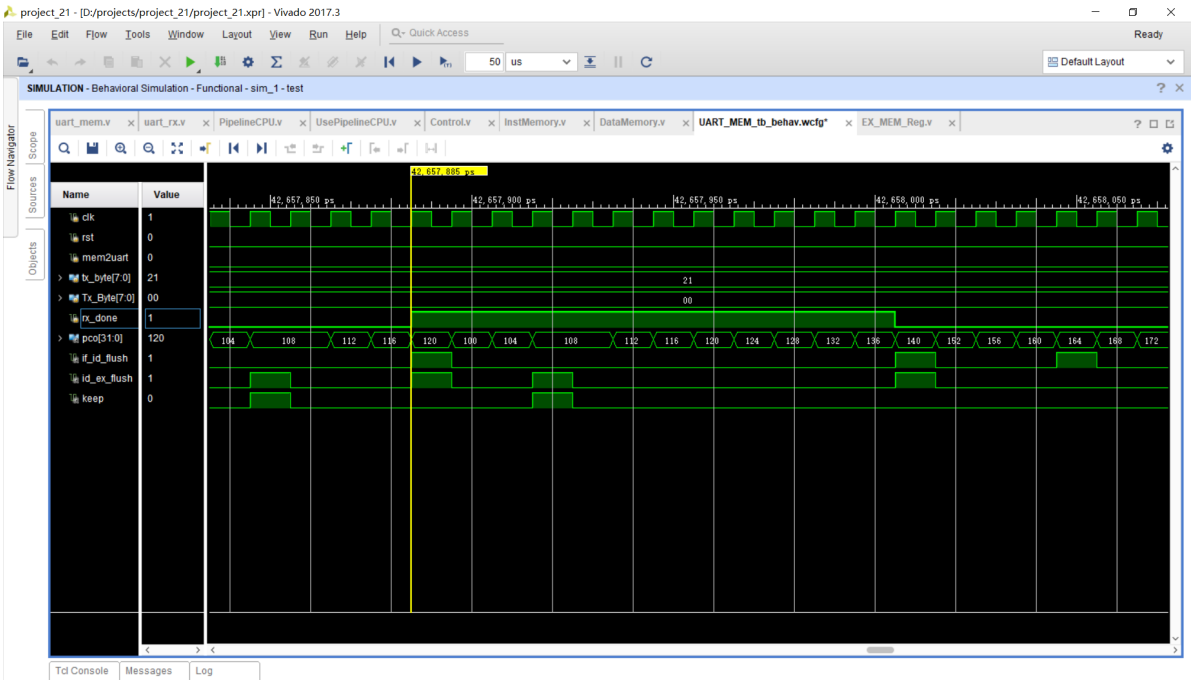
```
99  end
100
101 endmodule
```

该仿真程序，相当于通过uart串口向CPU输入了"unuix is pununu!unu!"，这样理论上搜索出的结果应为3（相当于主字符串为"unuix is pununu"，模式字符串为"unu"）。运行仿真，可以得到波形图，由于波形图跨度较长，难以完整展示全貌，所以先展示局部，检查程序运行是否正常：

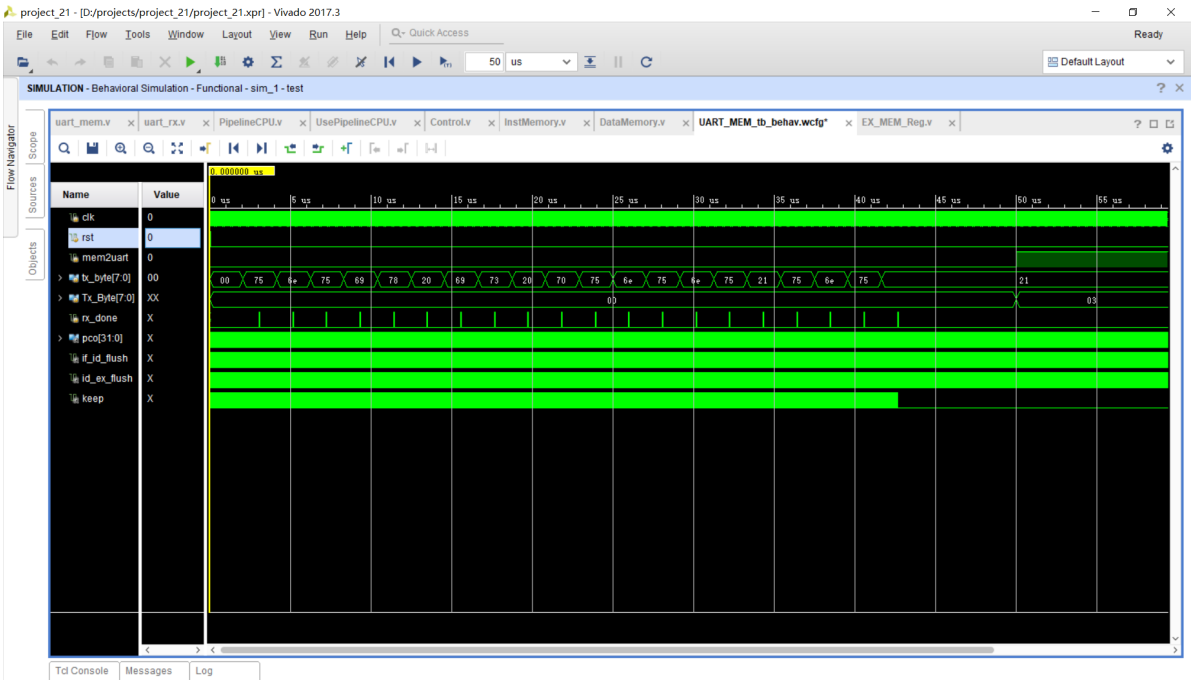




由上图可以看到，程序能够正常运行（pco为指令地址），在未检测到rx_done（表示串口接收到一个字节的数据）为1时持续循环，并且由flush和keep信号也可以看出，冒险处理单元也在正常工作，在检测到rx_done为1后，对其进行读取，且rx_done被自动清零，读取完毕后重新进入等待循环（因为尚未读取到作为换行符的"!"）。当str（主字符串）和pattern（模式字符串）都被正常读取到后，如下图：



可以看到程序跳出循环，开始进行字符串搜索，最后当mem2uart被拉高后，运算结果将经由TX_Byte被存入UART_TX，用于uart串口发送，如下图：



可以看到存入TX_Byte的结果是3，也就是程序运行结果是正确的。

七. 综合情况

面积情况

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Bonded IOB (210)	BUFGCTRL (32)
UsePipelineCPU	3689	5778	851	257	2424	3689	324	15	2
PipelineCPU (Pipeline...	3549	5659	850	257	2364	3549	259	0	1
ALU (ALU)	528	33	28	1	159	528	18	0	1
Control (Control)	23	18	0	0	8	23	15	0	0
ControlHazard (Con...	1	0	0	0	1	1	0	0	0
DataHazard (DataH...	22	0	0	0	10	22	0	0	0
DataMemory (Data...	1757	4096	544	256	1471	1757	0	0	0
EX_MEM_Reg (EX_...	0	106	0	0	69	0	0	0	0
ID_EX_Reg (ID_EX...	81	162	0	0	66	81	81	0	0
IF_ID_Reg (IF_ID_...	33	63	0	0	24	33	32	0	0
ImmExtend (ImmExt...	16	0	0	0	12	16	0	0	0
InstMemory (InstMe...	63	0	22	0	24	63	0	0	0
MEM_WB_Reg (ME...	0	104	0	0	52	0	0	0	0
PC (PC)	0	32	0	0	8	0	0	0	0
RegisterFile (Regist...	608	1024	256	0	640	608	0	0	0
UARTMemory (uart_...	35	21	0	0	16	35	4	0	0
uart_rx_inst (uart_rx)	51	33	0	0	22	51	31	0	0
uart_tx_inst (uart_tx)	29	29	1	0	15	29	21	0	0

时序性能

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.262 ns	Worst Hold Slack (WHS):	0.101 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	11012	Total Number of Endpoints:	11012	Total Number of Endpoints:	5728

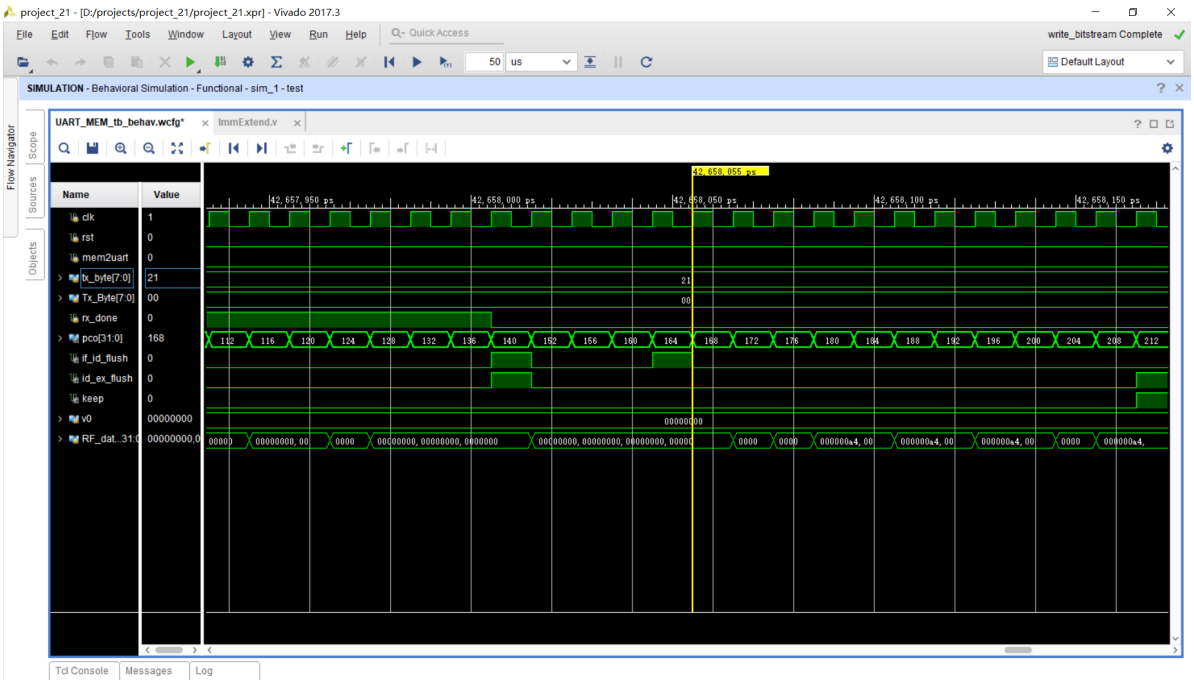
All user specified timing constraints are met.

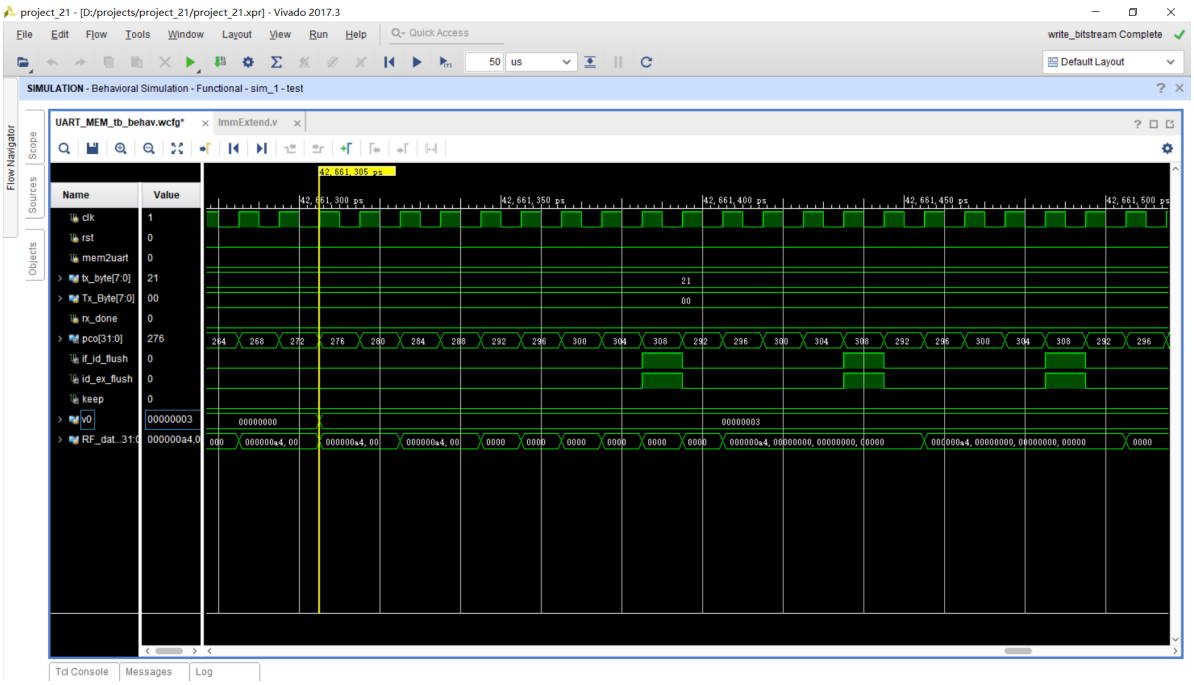
由于上面测试时序性能采用的时钟周期为10ns，故而由WNS可以计算得到理论最短时钟周期为10-1.262=8.738ns，由此得到理论最高时钟频率应为1/8.738=114.44MHz（**关闭将层次化的设计打平的选项**），可见该流水线CPU在保障正确运行的前提下具有不错的时序性能。

八. 硬件调试情况、性能评估与进一步优化

在进行调试时，由于有了此前调试单周期和多周期CPU的经验，且提前画好了数据通路故而整体较为顺利，前仿很快就成功了，但是一开始烧录到板子上确没有正常现象，经同学指出，是复位信号的写法存在问题，因为存在flush清空信号，所以我在写复位判定条件时将其和reset信号写在一起（即写为if(reset || flush)），这样的写法在前仿时没有问题，但是在实际电路中会产生问题，也是我没有好好检查warning带来的问题，该bug修改后，程序就可以正常在板子上运行了。最终实现了可以通过uart串口通信输入测试数据并将字符串搜索结果发送到uart串口的支持暴力搜索算法的流水线CPU，具体现象可见验收视频。

初步实现的流水线CPU的最高时钟频率为114.44MHz（关闭将层次化的设计打平的选项），再计算其CPI，由MARS的计数功能，可以计算得到运行输入字符串为“unuix is punu!unu!”的暴力搜索算法，需要执行的命令数量为224（指进入搜索子程序后到得到结果输入到$v0寄存器所用的指令数），而根据仿真结果，实际运行时间如下图：





共执行了(42661305-42658055)/10=325个周期，故而CPI=325/224=1.451，这是由于指令中存在一定比例的分支指令。

为了降低CPI，我想到了使用延迟槽技术，故而对汇编代码进行了修改，修改完成后如下：

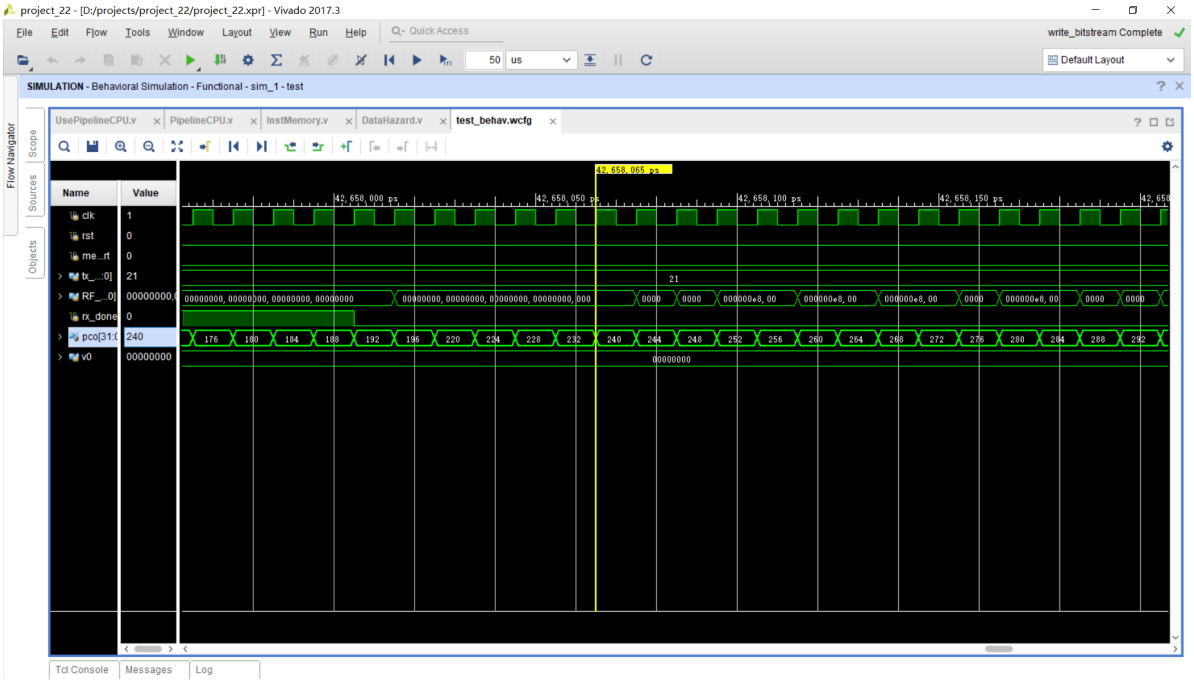
```
.text
main:

li $s7, 1073741848 #存储TX地址
li $s6, 1073741852 #存储RX地址
li $s5, 1073741856 #状态存储地址
li $a1, 0          #str的地址存在$a1
li $a3, 256      #pattern的地址存在$a3

#先不断读取状态中的第3位，若变为1，则进行读取
li $s0, 0 #len_str = 0
li $s1, 0 #len_pattern = 0
add $t4, $a1, $zero #暂存str地址
add $t3, $a3, $zero #暂存pattern地址
```

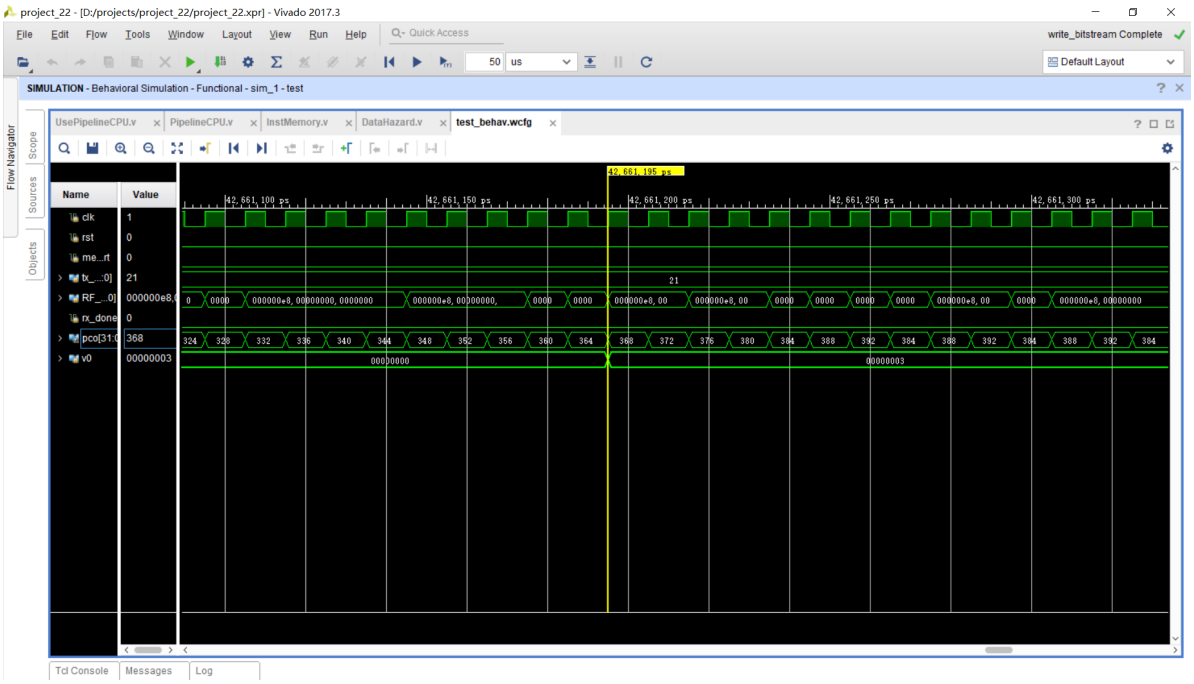
```
waitstr:
lw $t0, 0($s5)
nop
sll $t0, $t0, 28
srl $t0, $t0, 31
beqz $t0, waitstr #发现了字节输入
nop
nop

#read str
read_str_entry:
slti $t0, $s0, 256 #如果读入数据充满全部空间则跳出
beqz $t0, waitpattern
nop
nop
lw $t0, 0($s6)
addi $t1, $zero, '!' #读到换行则退出
beq $t0, $t1, waitpattern
nop
nop
sb $t0, 0($t4)
addi $t4, $t4, 1 #地址增加1位
addi $s0, $s0, 1 #记录总共读入了多少字节
j waitstr
nop

waitpattern:
lw $t0, 0($s5)
nop
sll $t0, $t0, 28
srl $t0, $t0, 31
beqz $t0, waitpattern #发现了字节输入
nop
nop

read_pattern_entry:
slti $t0, $s1, 256 #如果读入数据充满全部空间则跳出
beqz $t0, read_pattern_exit
nop
nop
lw $t0, 0($s6)
nop
addi $t1, $zero, '!' #读到换行则退出
beq $t0, $t1, read_pattern_exit
nop
nop
sb $t0, 0($t3)
addi $t3, $t3, 1 #地址增加1位
addi $s1, $s1, 1 #记录总共读入了多少字节
j waitpattern
nop

read_pattern_exit:
#call brute_force
move $a0, $s0
move $a2, $s1
jal brute_force
nop
```

```
73
74  loop:
75  beq $zero, $zero, loop
76
77  #str的地址存在$a1，其长度存在$a0，pattern的地址存在$a3，其长度存在$a2
78  brute_force:
79  li $t0, 0 #i=0
80  li $t2, 0 #cnt=0
81  sub $t5, $a0, $a2 #$t5存储len_str-len_pattern的值
82  move $t9, $a1 #把str地址存在$t9，防止改变$a1
83  move $t8, $a3 #把pattern地址存在$t8，防止改变$a3
84
85  loopout: #外循环，对i循环
86  li $t1, 0 #j=0
87  move $t7, $t9 #把str[i]地址暂存在$t7
88  move $t6, $t8 #把pattern地址暂存在$t6
89
90  loopin: #内循环，对j循环
91  lb $t3, 0($t7) #把str[i+j]存在$t3
92  lb $t4, 0($t6) #把pattern[j]存在$t4
93  nop
94  bne $t4, $t3, if #判断str[i+j]与pattern[j]是否相等，不相等则跳出循环到if
95  nop
96  nop
97  addi $t1, $t1, 1 #j+=1
98  blt $t1, $a2, loopin #j<len_pattern，继续循环
99  addi $t6, $t6, 1 #pattern地址加一
100 addi $t7, $t7, 1 #str地址加一
101
102 if:
103 bne $t1, $a2, cnt_not_plus #如果j==len_pattern，cnt+=1
104 nop
105 nop
106 addi $t2, $t2, 1 #cnt+=1
107
108 cnt_not_plus:
109 blt $t0, $t5, loopout #i<=len_str-len_pattern，继续循环
110 addi $t0, $t0, 1 #i+=1
111 addi $t9, $t9, 1 #str[i]地址加一
112
113 move $v0, $t2 #返回cnt值
114 sw $v0, 0($s7)
115 li $t0, 16
116 sw $t0, 0($s5)
117 li $s4, 256
118 li $s3, 768
119 li $v1, 1073741840
120
121 show:
122 li $t1, 10000
123 showin:
124 bnez $t1, showin
125 subi $t1, $t1, 4
126 nop
127 xor $s4, $s4, $s3
128 j show
129 sw $s4, 0($v1)
130 jr $ra
```

在会发生阻塞或清空处插入空指令nop（在分支指令后插入两个空指令，在跳转指令后插入一条空指令，在load-use数据冒险中插入一条空指令），再尽可能将空指令替换为必然会执行且不会产生新的阻塞或清空的指令，最后再从硬件层面，去除阻塞和清除功能。

测试发现修改完成后，流水线CPU仍可正确执行字符串搜索，且执行时间有所减少，仿真如下：





可以计算得到，优化后执行周期数为(42661195-42658065)/10=313，则CPI变为313/224=1.397（不将空指令视为有效指令），可以看到CPI有所下降，不过下降比例不大，这是因为由于修改前的代码较为精简，修改后很多空指令nop不能替换为有效指令，所以延迟槽技术没有很好地发挥作用。（不过在开启将层次化的设计打平的选项时，该优化后CPU的时序性能如下图，其最高时钟频率升至121.57MHz，尚不清楚原因。）

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.774 ns	Worst Hold Slack (WHS):	0.026 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	10926	Total Number of Endpoints:	10926	Total Number of Endpoints:	5787

All user specified timing constraints are met.

九. 实验小结

由于有了前面单周期和多周期CPU的铺垫，在实现流水线CPU时整体还是比较顺利的。在实现单周期、多周期CPU时，助教已经给出了一些基本模块的代码，只需要关注于控制信号的生成单元和数据通路的连接即可，但是在流水线CPU部分则没有助教提供的现成的代码，基本所有的模块都需要自行实现，不过在完成且能正常运行后还是非常有成就感的，且因为比较清楚各个模块的具体功能和代码实现，也为后续对CPU时钟频率进行优化提供了便利，比如我在进行CPU优化时就发现，读取数据存储器的耗时较长，基本可以确定是地址搜索带来的延时，所以我在保证程序正常运行的前提下适当降低了数据存储器的空间，使得地址的有效比特数减少，定位寄存器的速度得以提高，也确实使得CPU最高时钟频率由原本的一百零几兆赫兹提升至114.44MHz（关闭将层次化的设计打平的选项）。

最终实现的成果基本上将前几次实验都结合起来了，流水线CPU能够通过读取外设寄存器获取uart串口中接收的数据，也能通过写入外设寄存器向uart串口发送字符串搜索结果，且能不断刷新将结果显示在数码管上。