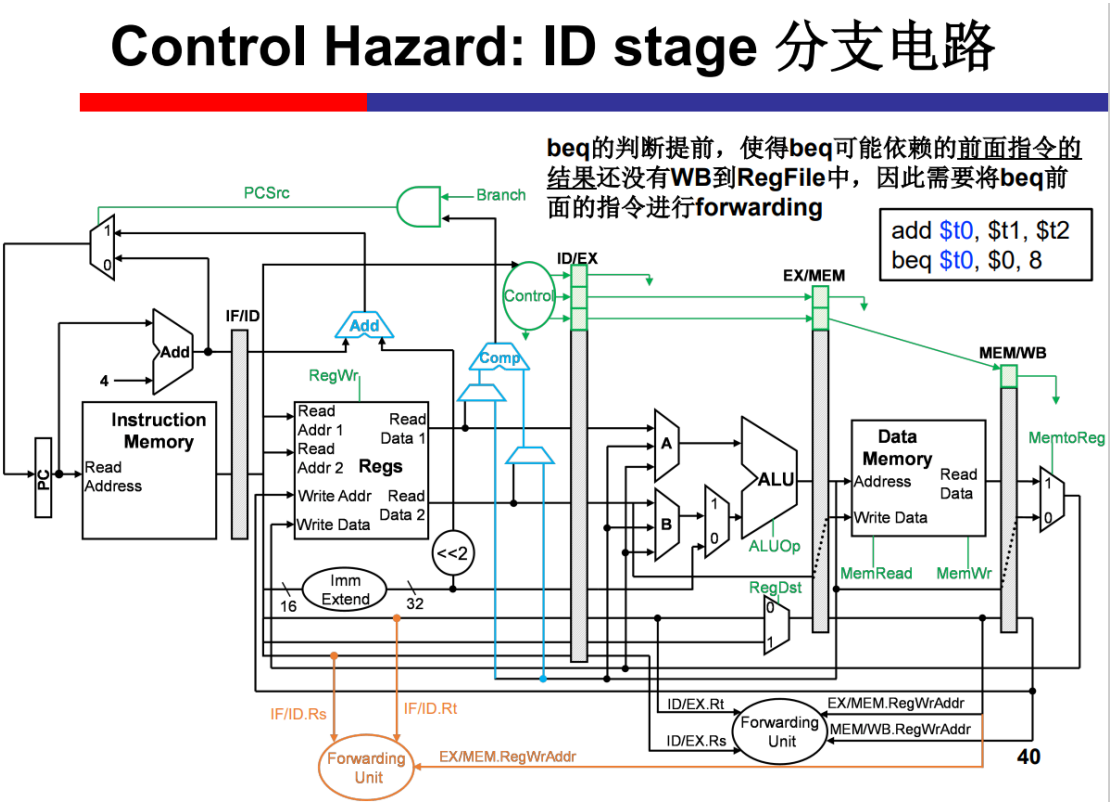
设计方案

参考《数字逻辑与处理器基础》课程的流水线处理器进行设计，主要依据的幻灯片如下。



设计上的主要思路是用 Verilog 语言按照幻灯片，分别实现各个部件，再在顶层模块中组装起来，在分别实现的过程中对一些模块进行了优化，将在关键代码中说明。

分模块的设计和关键代码

1. PC

PC 更新前，先判断是否处于 stall，然后判断是否应当跳转，分别将 PC 赋值为不同的地址。

```
always @ (posedge clk)
begin
    if (rst)
    begin
        PC <= 32'b0;
    end
    else
    if (stall)
    begin
        PC <= PC;
    end
    else
```

```
    begin
        PC <= newPC;
    end
end
```

2. 寄存器

和内存相似，为了提高时钟频率而放弃了初始化，对 0 作了特殊处理，支持先写后读。

```
    assign Read_data1 = (Read_register1 == 5'b0)? 32'b0: (RegWrite &&
Read_register1 == Write_register)? Write_data: RF_data[Read_register1];
    assign Read_data2 = (Read_register2 == 5'b0)? 32'b0: (RegWrite &&
Read_register2 == Write_register)? Write_data: RF_data[Read_register2];
```

3. 数据内存选择模块

CPU 的 lw 和 sw 指令不一定是取内存的数据，还有可能是 LED 的，因此在 CPU 中设计了选择模块，根据内存地址是否在规定的 LED 范围内，分别写入读取两个设备。

```
always @ (*) begin
    if (addr < 32'h40000000)
    begin
        DataMemoryaddr <= addr;
        DataMemorydatain <= data_in;
        DataMemoryWr <= MemoryWr;
        DataMemoryRd <= MemoryRd;
        data_out <= DataMemorydataout;

        LEDMemoryWr <= 1'b0;
    end
    else
    begin
        if (addr == 32'h40000010)
        begin
            LEDMemoryaddr <= addr;
            LEDMemorydatain <= data_in;
            LEDMemoryWr <= MemoryWr;
            LEDMemoryRd <= MemoryRd;
            data_out <= LEDMemorydataout;

            DataMemoryWr <= 1'b0;
        end
    end
end
```

4. 分支模块以及 Forwarding 模块

我把分支判断提前到了 ID 阶段，以获得更高的时钟频率，但是因此也需要增加一个对于分支指令的转发。在分支判断中，先并行判断两个操作数是否满足 lt gt ge 等条件，再根据指令取出对应的结果，能够节约时间。

```
wire eq, ne, lt, le, gt, ge;


assign eq = arg1 == arg2;
assign ne = !eq;
assign lt = (arg1[31] ^ arg2[31])? arg1[31] : arg1 < arg2;
assign le = !gt;
assign gt = (arg1[31] ^ arg2[31])? arg2[31] : arg1 > arg2;
assign ge = !lt;

/*
always @(*)
begin
    eq <= arg1 == arg2;
    ne <= !eq;
    lt <= arg1 < arg2;
    le <= !gt;
    gt <= arg1 > arg2;
end
*/

assign result = (OpCode == 6'h4)? eq: (OpCode == 6'h6)? le: (OpCode ==
6'h7)? gt: (OpCode == 6'h5)? ne: (OpCode == 6'h1)? lt: 1'b0;
```

Forwarding 有 2 个可能的来源，一是 MEM 阶段的 ALU 输出，二是 WB 阶段准备写入寄存器堆的值，分别判断一下，如果不需要 Forwarding，就正常地取寄存器堆读取的值。

```
assign BranchForwardingA =
(rs_ID != 0 && RegWrite_MEM && (RegWriteAddr_MEM == rs_ID))?
ALUOut_MEM:
                          (rs_ID != 0 && RegWrite_WB &&
(RegWriteAddr_WB == rs_ID))? RegWriteData_WB:
                          RFReadData1_ID;

assign BranchForwardingB =
(rt_ID != 0 && RegWrite_MEM && (RegWriteAddr_MEM == rt_ID))?
ALUOut_MEM:
                          (rt_ID != 0 && RegWrite_WB &&
(RegWriteAddr_WB == rt_ID))? RegWriteData_WB:
                          RFReadData2_ID;
```

5. 指令内存

我把指令内存和数据内存分开了，但结构上基本相同。

```
reg [31:0] InstructionMemoryData [511: 0];

assign Instruction = InstructionMemoryData[PC[10:2]];

always @(posedge rst) begin
InstructionMemoryData[9'h0] <= 32'h00000000;
InstructionMemoryData[9'h1] <= 32'h00000000;
InstructionMemoryData[9'h2] <= 32'h00000000;
InstructionMemoryData[9'h3] <= 32'h20080004;
InstructionMemoryData[9'h4] <= 32'hac080000;
InstructionMemoryData[9'h5] <= 32'h20080000;
InstructionMemoryData[9'h6] <= 32'hac080004;
InstructionMemoryData[9'h7] <= 32'h20080008;
```

6. 控制信号

在单周期处理器的基础上，加入了对新增指令的判断。

```
    assign PCSrc = (OpCode == 6'h2 || OpCode == 6'h3)? 2'b01: (OpCode ==
6'h0 && (Funct == 6'h8 || Funct == 6'h9))? 2'b10: 2'b00;
    assign Branch = (OpCode == 6'h1 || OpCode == 6'h4 || OpCode == 6'h5
|| OpCode == 6'h6 || OpCode == 6'h7)? 1'b1: 1'b0;
    assign RegWrite = (OpCode == 6'h2b || Branch || OpCode == 6'h2 ||
OpCode == 6'h0 && Funct == 6'h8)? 1'b0: 1'b1;
    assign RegDst = (OpCode == 6'h3) ? 2'b10 : (OpCode == 6'h23 ||
OpCode == 6'hf || OpCode >= 6'h8 && OpCode <= 6'hc) ? 2'b0 : 2'b1;
    assign MemRead = OpCode == 6'h23;
    assign MemWrite = OpCode == 6'h2b;
    assign MemtoReg = (OpCode == 6'h23)? 2'b1: (OpCode == 6'h3 || OpCode
== 6'h0 && Funct == 6'h9)? 2'b10: 2'b0;
    assign ALUSrc1 = (OpCode == 6'h0 && (Funct == 6'h0 || Funct == 6'h2
|| Funct == 6'h3))? 1'b1: 1'b0;
    assign ALUSrc2 = !(OpCode == 6'h0 || Branch || OpCode == 6'h1c);
    assign ExtOp = OpCode != 6'hc;
    assign LuOp = OpCode == 6'hf;
```

7. Stall 与 flush

在 ID 阶段如果发现是分支指令且有 load-use 冒险，那么 PC 需要阻塞一个周期，等待分支结果；如果分支是跳转的，那么需要 flush 清除 IF 阶段的指令。

```
always @ (*) begin
    if (Branch_ID &&
    (
        (RegWrite_EX &&
            (RegWriteAddr_EX == rs_ID || RegWriteAddr_EX == rt_ID)
```

```
        ) ||
        (MemRead_MEM &&
            (RegWriteAddr_MEM == rs_ID || RegWriteAddr_MEM == rt_ID)
        )
    )
    )
    begin
        stall <= 1'b1;
    end
    else
    begin
        stall <= 1'b0;
    end
end

always @ (*) begin
    flush[1] = 1'b0;
end

always @ (*) begin
    if (PCSrc_ID != 2'b00 || Branch_ID && BranchResult_ID) begin
        flush[0] = 1;
    end
    else
    begin
        flush[0] = 0;
    end
    // flush[1] = stall[0];
end
```

8. ALU 以及 Forwarding

ALU 基本没有改动，增加了 Forwarding 模块，从 MEM 或者 WB 阶段转发数据。

```
assign ALUInA =
(rs_EX == 0)? 32'b0:
(RegWrite_MEM && (RegWriteAddr_MEM == rs_EX))? ALUOut_MEM:
                (RegWrite_WB && (RegWriteAddr_WB == rs_EX))?
RegWriteData_WB:
                RFReadData1_EX;

assign ALUInB =
(rt_EX == 0)? 32'b0:
(RegWrite_MEM && (RegWriteAddr_MEM == rt_EX))? ALUOut_MEM:
                (RegWrite_WB && (RegWriteAddr_WB == rt_EX))?
RegWriteData_WB:
```

```
                RFReadData2_EX;
```

9. LED 的控制

对于如何显示出数字、字母，我采用了以下方法：对 LED 的每一根灯管做考量，比如 a 管，在数字是 14bcd 的情况下应当是灭的，对其他的数字都应当是亮的，于是就在程序中判断数字是否是 14bcd，是的话就将对应数位的 1 减去，以此类推。

综合情况

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1486	20800	7.14
LUTRAM	304	9600	3.17
FF	475	41600	1.14
IO	13	250	5.20
BUFG	3	32	9.38

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	LUT Flip Flop Pairs (20800)	Bonded IOB (250)	BUFGCTRL (32)
CPU	1486	572	174	77	468	1182	304	161	13	3
CPUALU (ALU)	0	0	0	0	12	0	0	0	0	0
CPUBranch (BranchU...	0	0	0	0	11	0	0	0	0	0
CPUDataMemory (Dat...	289	0	128	64	92	33	256	0	0	0
CPUInstructionMemo...	124	0	29	13	34	124	0	0	0	0
CPULEDMemory (LED...	0	23	0	0	12	0	0	0	0	0
CPUMemoryControl (M...	1	97	0	0	52	1	0	1	0	0
CPUProgramCounter (...	0	32	0	0	28	0	0	0	0	0
CPURF (RegisterFile)	48	0	0	0	12	0	48	0	0	0
EX_MEM (Reg_EX_ME...	53	111	0	0	113	53	0	1	0	0
ID_EX (Reg_ID_EX)	558	136	17	0	217	558	0	1	0	0
IF_ID (Reg_IF_ID)	164	63	0	0	66	164	0	45	0	0
MEM_WB (Reg_MEM_...	251	110	0	0	128	251	0	1	0	0

根据要求修改时钟为 100MHz，得到结果如下。

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.562 ns	Worst Hold Slack (WHS):	0.037 ns	Worst Pulse Width Slack (WPWS):	3.750 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:	1155	Total Number of Endpoints:	1155	Total Number of Endpoints:	828

根据 WNS 计算得到最高频率为 105.95MHz。

仿真中 reset 是在 10ns，出结果显示到 LED 是在 18054.370ns，仿真是以 10ns 为一周期的，因此总的时钟周期数约为 1804，因此算出 CPI。

$$\frac{1804}{1456} = 1.239$$

CPI 超过 1 应当主要因为 b 指令和 j 指令的跳转。