综合实验　流水线 MIPS 处理器

罗一夫 2018010811 无 85

目录

1 实验目的

将春季学期设计的单周期 MIPS 处理器改进为流水线结构，并利用此处理器和任意一种排序算法，编写汇编语言对 128 个 32bits 的无符号随机数进行排序。深入认识流水线处理器的工作原理，提高使用硬件描述语言设计硬件的能力，理解评价处理器性能的指标。

2 性能指标

表 1 处理器性能指标

性能指标	测试值
设计名称	基于 FPGA 的五级流水线 32 位 MIPS 处理器
指令集	MIPS 核心指令集（31 条）
最高主频	118.5MHz
CPI（基于冒泡排序程序测试）	1.187
每秒指令数	99.8Million
数据存储器容量	0.5KB
支持异常类型	未定义指令
支持中断类型	定时器中断
支持 forwarding 类型	MEM/WB→EX EX/MEM→EX
支持外设	定时器 4*七段数码管 8*LED 系统时钟计数器

设计中主要有如下优化点（详细说明见 4、5），使得处理器 CPI 低于 1.2 的同时主频提升至 100MHz 以上：

- 分离分支条件判断模块与 ALU 模块，降低组合逻辑复杂度；
- 转发控制单元提前到 ID 阶段，减少 EX 阶段流水线任务量，缩短涉及转发时的关键路径；
- 将 MemtoReg 控制信号对回写寄存器数据的选择提前至 MEM 阶段，同时在 MEM 阶段增加寄存器，使得 RegisterFile 能够在上升沿写入，为先写后读扩大时间裕度。

3 总体设计

该流水线 MIPS 处理器采用五级流水线设计，理想情况下流水线中同时有五条指令同时执行不同阶段的任务，相对于单周期处理器可以提高时钟频率，从而提高执行效率。

处理器中主要涉及以下模块：

表 2 处理器模块设计

模块名称	功能描述
InstructionMemory	指令存储器，存储指令机器码，根据对应的地址取出指令送入指令译码
RegisterFile	寄存器堆，包含 32 个 32bits 的数据寄存器，用于储存程序执行过程中临时变量、返回地址等数据的存储
Control 模块	指令译码控制模块，根据指令生成控制信号，控制流水线运行
ALU	计算单元，执行加减运算与逻辑运算，得到计算型指令的结果或得到数据传输型指令中的数据地址

Branch 模块	分支控制模块，对输入变量进行比较，判断 beq/bne/blez/bgtz/bltz 指令分支跳转条件是否成立
DataMemory & 外设	数据存储器，存储数据，同时采用总线设计，将外设对应为数据存储器地址空间，处理器与外设的交互转化为数据读写
Forwarding 模块	数据转发控制模块，判断流水线中是否存在数据依赖，并控制相应的 MEM/WB→EX，EX/MEM→EX 数据转发解决数据依赖
Hazard 模块	冒险综合控制模块，在流水线中存在 load-use 冒险时阻塞流水线，同时控制跳转指令执行后冲刷流水线以及出现异常/中断时冲刷流水线

各模块具体设计与功能见“4 模块设计与优化”。

处理器中包含以下五个流水级：

表 3 处理器流水级设计

流水级名称	功能描述
IF	取指令阶段，由 Control 模块、Branch 模块、Hazard 模块联合生成 PCSrc 控制信号后得到下一指令的地址存入 PC 寄存器，并从 Instruction Memory 中取相应指令
ID	指令译码阶段，根据指令生成控制信号，按照指令中 Rs、Rt 值从 Register File 中读出相应寄存器值，同时完成指令中立即数的有符号/无符号扩展，j/jr/jal/jalr 指令在该阶段被识别
EX	指令执行阶段，ALU 得到计算结果或数据交换地址，Branch 模块判断分支跳转指令分支条件是否成立
MEM	存储器访问阶段，数据交换型指令在该阶段完成对 Data Memory 的读写
WB	寄存器回写阶段，根据不同指令将 ALU 计算结果/Data Memory 访问结果/返回指令地址值写回 Register File

各流水级具体设计与功能见“5 流水级设计与优化”。

处理器整体设计图如下：

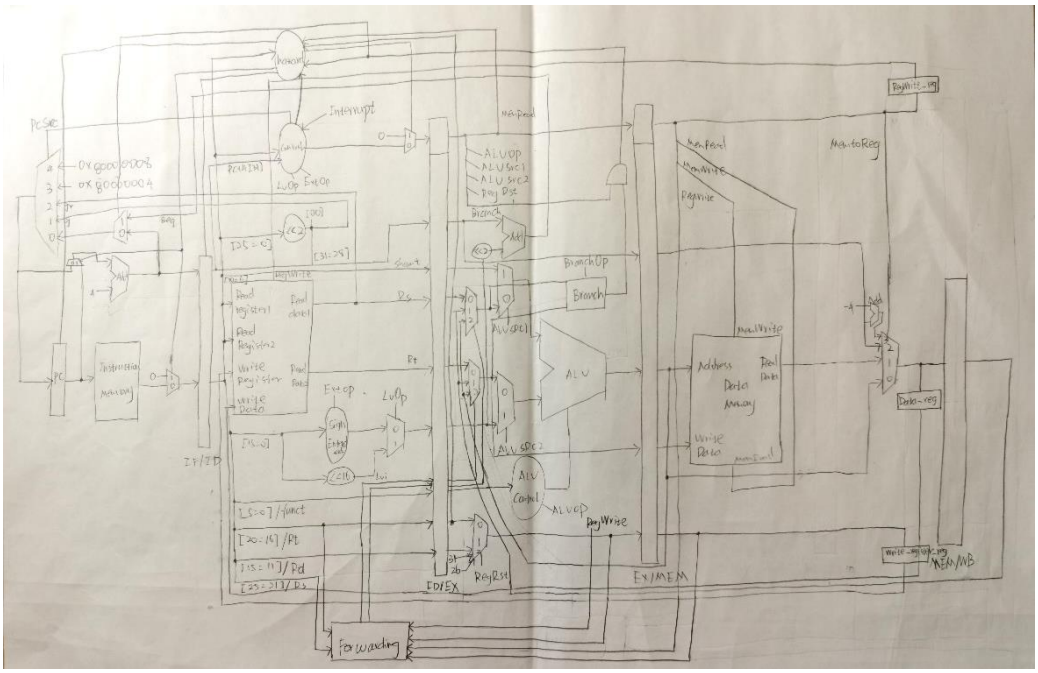


图 1 处理器整体设计图

高清晰度图片见 pipeline.jpg。

4 模块设计与优化

4.1 InstructionMemory

该模块为指令存储器，其输入输出定义如下：

表 4 InstructionMemory 模块输入输出定义

信号名称	位宽	输入/输出	功能描述
Address	*32bits*	*input*	待取指令的地址（以字节为单位）
Instruction	*32bits*	*output*	根据输入地址取出的指令机器码

由于地址的最高位为监督为，并不表示实际的地址值，该模块首先将输入端地址的最高位恒置为 0，后根据此地址取相应的指令。这一过程通过一个 case 块实现，输入端得到地址后经过一定延时，输出端可以得到对应的指令机器码。如果输入的地址值为非法值，则输出空指令（nop）。其核心代码如下，完整代码文件为 InstructionMemory.v。

```
module InstructionMemory(Address, Instruction);
    input [31:0] Address;
    output reg [31:0] Instruction;

    always @(*)
    begin
        case ({1'b0,Address[30:2]})
            30'd0:  Instruction <= 32'h08000003;
            30'd1:  Instruction <= 32'h08000134;
            30'd2:  Instruction <= 32'h080001c2;
            30'd3:  Instruction <= 32'h0c000004;
            ......
            default:  Instruction <= 32'h00000000;
        endcase
    end

endmodule
```

4.2 RegisterFile

该模块为寄存器堆，其输入输出定义如下：

表 5 RegisterFile 模块输入输出定义

信号名称	位宽	输入/输出	功能描述
reset	*1bit*	*input*	复位信号
clk	*1bit*	*input*	时钟信号
RegWrite	*1bit*	*input*	写使能信号
Read_register1	*5bits*	*input*	第一个读取寄存器的编号
Read_register2	*5bits*	*input*	第二个读取寄存器的编号
Write_register	*5bits*	*input*	写入寄存器的编号
Write_data	*32bits*	*input*	写入寄存器的数据
Read_data1	*32bits*	*output*	第一个读取寄存器中读出的数据
Read_data2	*32bits*	*output*	第二个读取寄存器中读取的数据

模块同时具有读寄存器和写寄存器的功能，其中读寄存器功能在输入端得到待读取寄存器的编号后经过一定延时即可在输出端得到待读取寄存器中的数值。写寄存器功能在时钟上升沿根据输入的待写入寄存器编号以及待写入数据，更新相应寄存器内容，通过 always 块实现。其核心代码如下，完整代码文件为 RegisterFile.v。

```
module RegisterFile(reset, clk, RegWrite, Read_register1, Read_register2, Write_register, Write_data
, Read_data1, Read_data2);
    input reset, clk;
    input RegWrite;
    input [4:0] Read_register1, Read_register2, Write_register;
```

```
    input [31:0] Write_data;
    output [31:0] Read_data1, Read_data2;

    reg [31:0] RF_data[31:1];

    assign Read_data1 = (Read_register1 == 5'b00000)? 32'h00000000: RF_data[Read_register1];
    assign Read_data2 = (Read_register2 == 5'b00000)? 32'h00000000: RF_data[Read_register2];

    integer i;
    always @(posedge reset or posedge clk)
        if (reset)
            for (i = 1; i < 32; i = i + 1)
                RF_data[i] <= 32'h00000000;
        else if (RegWrite && (Write_register != 5'b00000))
            RF_data[Write_register] <= Write_data;

endmodule
```

该寄存器支持先写后读的功能，有利于减少数据冒险。一般流水线设计中，寄存器堆使用下降沿写入，如此在先写后读中留给读寄存器的时间仅有小于 1/2 个时钟周期，不利于提高时钟频率。此处改用上升沿写入的寄存器堆，在写后读中给读寄存器提供更多的时间，提高时钟频率。配合上升沿写入的寄存器堆，流水线 MEM、WB 阶段有相应的优化设计以保证待写入数据、待写入寄存器编号的正确性，详细说明见“5 流水级设计与优化”。

4.3 Control 模块

该模块为控制模块，通过解析指令，生成控制信号控制流水线运行。模块内部具备检测未定义指令的功能，如果出现未定义指令则抛出异常。同时该模块接收 DataMemory 中外设定时器传来的中断信号，将该中断信号与自身异常信号取或运算得到总中断/异常信号，传递给 Hazard 模块。其输入输出定义如下：

表 6 Control 模块输入输出定义

信号名称	位宽	输入/输出	功能描述
OpCode	*6bits*	*input*	指令 OpCode 部分
Funct	*6bits*	*input*	指令 Funct 部分
Interrupt	*1bit*	*input*	DataMemory 中定时器传来的中断信号
PC_sign	*1bit*	*input*	指令地址最高位（监督位）
PCSrc	*3bits*	*output*	选择下一条指令地址
Branch	*1bit*	*output*	分支指令标记
RegWrite	*1bit*	*output*	寄存器写使能
RegDst	*2bits*	*output*	选择待写入寄存器编号
MemRead	*1bit*	*output*	数据存储器读使能
MemWrite	*1bit*	*output*	数据存储器写使能
MemtoReg	*2bits*	*output*	选择写入寄存器的数据
ALUSrc1	*1bit*	*output*	选择 ALU 第一个输入端的数据
ALUSrc2	*1bit*	*output*	选择 ALU 第二个输入端的数据
ExtOp	*1bit*	*output*	控制立即数有/无符号扩展
LuOp	*1bit*	*output*	选择扩展后立即数
ALUOp	*4bits*	*output*	控制 ALU 运算类型
BranchOp	*3bits*	*output*	控制 Branch 模块运算类型
Exception	*1bit*	*output*	传入 Hazard 的异常/中断信号

各条指令对应的各控制信号值如下表（MUX 控制信号与选通数据对应关系见图 1 标注），其中 inter 表示计时器中断情况，exc 表示未定义指令异常情况。

表 7 指令与控制信号对应关系

指令	PCSrc	Branch	RegWrite	RegDst	MemRead	MemWrite	MemtoReg	ALUSrc1	ALUSrc2	ExtOp	LuOp	Exception
nop	0	0	x	x	0	0	x	x	x	x	x	0
lw	0	0	1	0	1	0	1	0	1	1	0	0
sw	0	0	0	x	0	1	x	0	1	1	0	0
lui	0	0	1	0	0	0	0	0	1	x	1	0
add	0	0	1	1	0	0	0	0	0	x	x	0
addu	0	0	1	1	0	0	0	0	0	x	x	0
sub	0	0	1	1	0	0	0	0	0	x	x	0
subu	0	0	1	1	0	0	0	0	0	x	x	0
addi	0	0	1	0	0	0	0	0	1	1	0	0
addiu	0	0	1	0	0	0	0	0	1	1	0	0
and	0	0	1	1	0	0	0	0	0	x	x	0
or	0	0	1	1	0	0	0	0	0	x	x	0
xor	0	0	1	1	0	0	0	0	0	x	x	0
nor	0	0	1	1	0	0	0	0	0	x	x	0
andi	0	0	1	0	0	0	0	0	1	0	0	0
ori	0	0	1	0	0	0	0	0	1	0	0	0
sll	0	0	1	1	0	0	0	1	0	x	x	0
srl	0	0	1	1	0	0	0	1	0	x	x	0
sra	0	0	1	1	0	0	0	1	0	x	x	0
slt	0	0	1	1	0	0	0	0	0	x	x	0
slti	0	0	1	0	0	0	0	0	1	1	0	0
sltiu	0	0	1	0	0	0	0	0	1	1	0	0
beq	0	1	0	x	0	0	x	0	0	1	0	0
bne	0	1	0	x	0	0	x	0	0	1	0	0
blez	0	1	0	x	0	0	x	0	0	1	0	0
bgtz	0	1	0	x	0	0	x	0	0	1	0	0
bltz	0	1	0	x	0	0	x	0	0	1	0	0
j	1	x	0	x	0	0	x	x	x	x	x	0
jal	1	x	1	2	0	0	2	x	x	x	x	0
jr	2	x	0	x	0	0	x	x	x	x	x	0
jalr	2	x	1	1	0	0	2	x	x	x	x	0
inter	3	0	1	3	0	0	3	x	x	x	x	1
exc	4	0	1	3	0	0	2	x	x	x	x	1

该模块中控制信号生成全部由三目运算符对应组合逻辑，在设计与异常信号相关的控制信号时，异常信号应当被优先处理，三目运算符串的最前面。核心代码如下，完整代码文件见 Control.v。

```
    assign RegWrite =
        (Exception == 1'b1)? 1'b1:
        (OpCode == 6'h2b || OpCode == 6'h01 || OpCode == 6'h04 || OpCode == 6'h05 || OpCode == 6'h06
|| OpCode == 6'h07 || OpCode == 6'h02 || (OpCode == 6'h00 && Funct == 6'h08))? 1'b0:
        1'b1;
```

4.4 ALU & ALU 控制

该模块为运算模块，运算产生计算型指令的计算结果或数据传输型指令的数据地址。其输入输出定义如下：

表 8 ALU 模块输入输出定义

信号名称	位宽	输入/输出	功能描述
in1	*32bits*	*input*	输入数据 1
in2	*32bits*	*input*	输入数据 2
ALUCtl	*5bit*	*input*	控制信号
Sign	*1bit*	*input*	符号运算控制
out	*32bits*	*output*	运算结果

模块内部根据控制信号选择运算类型，经过一定时间后输出运算结果，核心代码如下，完整代码文件见 ALU.v。

```
module ALU(in1, in2, ALUCtl, Sign, out);
    input [31:0] in1, in2;
    input [4:0] ALUCtl;
    input Sign;
    output reg [31:0] out;

    wire ss;
    assign ss = {in1[31], in2[31]};

    wire lt_31;
    assign lt_31 = (in1[30:0] < in2[30:0]);

    wire lt_signed;
    assign lt_signed = (in1[31] ^ in2[31])?
        ((ss == 2'b01)? 0: 1): lt_31;

    always @(*)
        case (ALUCtl)
            5'b00000: out <= in1 & in2;
            5'b00001: out <= in1 | in2;
            5'b00010: out <= in1 + in2;
            5'b00110: out <= in1 - in2;
            5'b00111: out <= {31'h00000000, Sign? lt_signed: (in1 < in2)};
            5'b01100: out <= ~(in1 | in2);
            5'b01101: out <= in1 ^ in2;
            5'b10000: out <= (in2 << in1[4:0]);
            5'b11000: out <= (in2 >> in1[4:0]);
            5'b11001: out <= ({{32{in2[31]}}, in2} >> in1[4:0]);
            default: out <= 32'h00000000;
        endcase

endmodule
```

ALU 模块运算种类多，控制信号生成的组合逻辑比较复杂，延时较长。本设计中使用两级逻辑控制，在 ID 阶段先由 Control 模块根据 OpCode 代表的指令类型生成一级控制信号 ALUOp，在 EX 阶段再由 ALUControl 模块根据 ALUOp 以及 R 型指令的 Funct 生成二级控制信号 ALUCtl 和 Sign。

这一两级控制结构图如下：

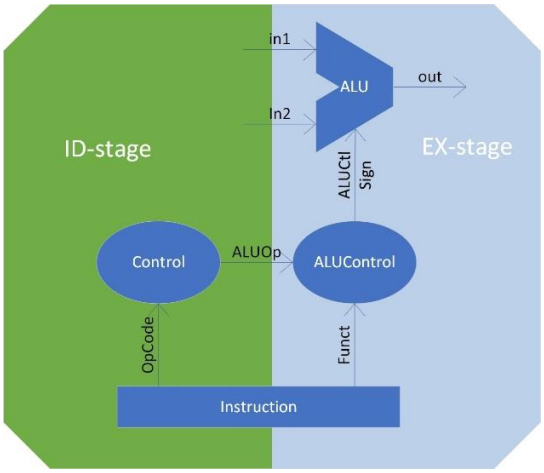


图 2 ALU 两级控制结构图

这一设计将组合逻辑延时分布在两个流水级，有利于提高时钟频率。ALUControl 模块输入输出定义如下：

表 9 ALUControl 模块输入输出定义

信号名称	位宽	输入/输出	功能描述
ALUOp	*4bits*	*input*	Control 模块产生的一级控制信号
Funct	*6bits*	*input*	R 型指令 Funct
ALUCtl	*5bit*	*output*	ALU 二级控制信号
Sign	*1bit*	*output*	符号运算控制

ALUControl 模块内部首先定义加减、各类逻辑运算法 ALUCtl 参数，再通过 always 块根据 ALUOp 和 Funct 将参数赋予 ALUCtl。核心代码如下，完整代码文件为 ALUControl.v。

```
module ALUControl(ALUOp, Funct, ALUCtl, Sign);
    input [3:0] ALUOp;
    input [5:0] Funct;
    output reg [4:0] ALUCtl;
    output Sign;

    parameter aluAND = 5'b00000;
    parameter aluOR  = 5'b00001;
    parameter aluADD = 5'b00010;
    parameter aluSUB = 5'b00110;
    parameter aluSLT = 5'b00111;
    parameter aluNOR = 5'b01100;
    parameter aluXOR = 5'b01101;
    parameter aluSLL = 5'b10000;
    parameter aluSRL = 5'b11000;
    parameter aluSRA = 5'b11001;

    assign Sign = (ALUOp[2:0] == 3'b010)? ~Funct[0]: ~ALUOp[3];

    reg [4:0] aluFunct;
    always @(*)
        case (Funct)
            6'b00_0000: aluFunct <= aluSLL;
            6'b00_0010: aluFunct <= aluSRL;
            6'b00_0011: aluFunct <= aluSRA;
            6'b10_0000: aluFunct <= aluADD;
            6'b10_0001: aluFunct <= aluADD;
            6'b10_0010: aluFunct <= aluSUB;
            6'b10_0011: aluFunct <= aluSUB;
            6'b10_0100: aluFunct <= aluAND;
            6'b10_0101: aluFunct <= aluOR;
            6'b10_0110: aluFunct <= aluXOR;
```

```
            6'b10_0111: aluFunct <= aluNOR;
            6'b10_1010: aluFunct <= aluSLT;
            6'b10_1011: aluFunct <= aluSLT;
            default: aluFunct <= aluADD;
        endcase

    always @(*)
        case (ALUOp[2:0])
            3'b000: ALUCtl <= aluADD;
            3'b100: ALUCtl <= aluAND;
            3'b101: ALUCtl <= aluSLT;
            3'b010: ALUCtl <= aluFunct;
            default: ALUCtl <= aluADD;
        endcase

endmodule
```

4.5 Branch 模块

该模块为分支跳转控制模块，判断 beq/bne/blez/bgtz/bltz 指令分支跳转条件是否成立。原流水线设计中仅支持 beq 一条分支跳转指令，分支条件是否成立的判断集成在 ALU 模块中。

本设计中开始时将 beq/bne/blez/bgtz/bltz 全部集成在 ALU 模块中，同时扩展 ALU 的两级控制信号，实现对这五条指令分支条件是否成立的判断。综合、实现后发现该路径逻辑过于复杂，延时较长，阻碍时钟频率的提升，故将分支跳转控制的功能分离，由 Control 模块在 ID 阶段直接生成控制信号 BranchOp 简化了相应的控制逻辑，加快执行速度。同时，ALU 模块的输入还可能来自立即数扩展或指令中的偏移量字段，而 Branch 模块无此来源。因此，分离 Branch 模块还可以简化输入选择逻辑，进一步缩短路径。

模块输入输出定义如下：

表 10 Branch 模块输入输出定义

信号名称	位宽	输入/输出	功能描述
in1	*32bits*	*input*	输入数据 1
in2	*32bits*	*input*	输入数据 2
BranchOp	*3bit*	*input*	控制信号
zero	*1bits*	*output*	判断结果

内部判断分支条件是否成立通过组合逻辑实现，对于 32 位有符号数，判断其正负可以由最高位（符号位）为 0/1 实现。核心代码如下，完整代码文件为 Branch.v。

```
module Branch(in1, in2, BranchOp, zero);
    input [31:0] in1, in2;
    input [2:0] BranchOp;
    output zero;

    assign zero = (BranchOp == 3'b001)? (in1 == in2):
                  (BranchOp == 3'b010)? (in1 != in2):
                  (BranchOp == 3'b011)? ((in1 == 32'd0) || (in1[31] == 1'b1)):
                  (BranchOp == 3'b100)? ((in1 != 32'd0) || (in1[31] == 1'b0)):
                  (BranchOp == 3'b101)? (in1[31] == 1'b1):
                  1'b0;

endmodule
```

4.6 DataMemory & 外设

该模块为数据存储器模块，提供 1KB 的数据存储空间。同时利用总线设计，给 LED、七段数码管、定时器三件外设分配专用的地址空间。处理器与这些外设进行交互时只需要对相应的地址空间上数据进行读写。地址空间分配如下：

表 11 DataMemory 地址空间分配

名称	地址空间
数据存储	0x00000000~0x000001FF
定时器	0x40000000~0x4000000B
LED	0x4000000C
七段数码管	0x40000010
系统时钟计数器	0x40000014

其中，当定时器使能 TCON[0]为 1 时，定时器外设通过 TH、TL 两个变量不断循环计数，计数值达到设定值后，如果中断使能 TCON[1]为 1，定时器中断标志 TCON[2]置为 1，同时生成长度为一个时钟周期的中断信号 Interrupt，该部分通过 always 块实现，其代码为：

```
if (TCON[0])
begin
    if(TL==32'hffffffff)
    begin
        TL <= TH;
        if(TCON[1])
        begin
            TCON[2] <= 1'b1;
            Interrupt <= 1'b1;
        end
    end
    else
    begin
        TL <=TL +1;
        Interrupt <= 1'b0;
    end
end
```

模块整体输入输出定义如下：

表 12 DataMemory 模块输入输出定义

信号名称	位宽	输入/输出	功能描述
reset	*1bit*	*input*	复位信号
clk	*1bit*	*input*	时钟信号
Address	*32bits*	*input*	数据地址
Write_data	*32bits*	*input*	待写入数据
MemRead	*1bit*	*input*	存储器读使能
MemWrite	*1bit*	*input*	存储器写使能
Read_data	*32bits*	*output*	读出数据
Interrupt	*1bit*	*output*	定时器中断信号
digi_out	*12bits*	*output*	七段数码管控制
led_out	*8bits*	*output*	该模块 LED 控制

该模块读存储器通过组合逻辑实现，当读使能有效时，输入端获得地址后经过一定延时，输出端会得到相应数据。写存储器通过 always 块实现，在写使能有效时，时钟上升沿时根据输入地址将数据写入存储器。核心代码如下，完整代码文件为 DataMemory.v。

```
    assign Read_data = (MemRead)? ((Address == 32'h40000000)? TH:
                                   (Address == 32'h40000004)? TL:
                                   (Address == 32'h40000008)? {29'b0,TCON}:
                                   (Address == 32'h4000000C)? {24'b0,led}:
                                   (Address == 32'h40000010)? {20'b0,digi}:
```

```
                                (Address == 32'h40000014)? systick:
                                RAM_data[Address[RAM_SIZE_BIT + 1:2]]):
                        32'h00000000;
    assign digi_out = digi[11:0];
    assign led_out = led[7:0];

    integer i;
    always@(posedge reset or posedge clk)
    begin
        if (reset)
        begin
            for (i = 0; i < RAM_SIZE; i = i + 1)
                RAM_data[i] <= 32'h00000000;
            TH <= 32'h00000000;
            TL <= 32'h00000000;
            TCON <= 3'd0;
            led <= 8'd0;
            digi <= 12'hfff;
            systick <= 32'd0;
            Interrupt <= 1'd0;
        end
        else
        begin
            systick <= systick + 1;
            if(MemWrite)
            begin
                case(Address)
                    32'h40000000:TH <= Write_data;
                    32'h40000004:TL <= Write_data;
                    32'h40000008:TCON <= {29'b0,Write_data[2:0]};
                    32'h4000000C:led <= {24'b0,Write_data[7:0]};
                    32'h40000010:digi <= {20'b0,Write_data[11:0]};
                    default:     RAM_data[Address[RAM_SIZE_BIT + 1:2]] <= Write_data;
                endcase
            end
        ......
        end
     end
```

4.7 Forwarding 模块

该模块控制，EX/MEM→EX 数据转发解决数据依赖。其控制逻辑为：

表 13 Forwarding 模块控制逻辑

转发类型	控制逻辑
EX/MEM→EX	if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRs)) ForwardA = 10 if (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd == ID/EX.RegisterRt)) ForwardB = 10
MEM/WB→EX	if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and (MEM/WB.RegisterRd == ID/EX.RegisterRs) and (EX/MEM.RegisterRd != ID/EX.RegisterRs || ~ EX/MEM.RegWrite)) ForwardA = 01 if (MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0) and (MEM/WB.RegisterRd == ID/EX.RegisterRt) and (EX/MEM.RegisterRd != ID/EX.RegisterRt || ~ EX/MEM.RegWrite))) ForwardB = 01

模块输入输出定义如下：

表 14 Forwarding 模块输入输出定义[1]

信号名称	位宽	输入/输出	功能描述
IDEX_Reg_Rs	*5bits*	*input*	ID/EX 阶段 Rs 编号
IDEX_Reg_Rt	*5bits*	*input*	ID/EX 阶段 Rt 编号
EXMEM_Reg_Rd	*5bits*	*input*	EX/MEM 阶段 Rd 编号
EXMEM_RegWrite	*1bit*	*input*	EX/MEM 阶段寄存器写使能
MEMWB_Reg_Rd	*5bits*	*input*	MEM/WB 阶段 Rd 编号
MEMWB_RegWrite	*1bit*	*input*	MEM/WB 阶段寄存器写使能
Forwarding_Ctrl1	*2bit*	*output*	选择 ALU 输入数据 1 端口的数据来源
Forwarding_Ctrl2	*2bit*	*output*	选择 ALU 输入数据 2 端口的数据来源

模块内部用组合逻辑实现，核心代码如下，完整代码文件为 Forwarding.v。

```
    assign Forwarding_Ctrl1 = (EXMEM_RegWrite && (EXMEM_Reg_Rd != 0) && (EXMEM_Reg_Rd == IDEX_Reg_Rs
))? 2'b10:(MEMWB_RegWrite && (MEMWB_Reg_Rd != 0) && (MEMWB_Reg_Rd == IDEX_Reg_Rs) && ((EXMEM_Reg_Rd
!= IDEX_Reg_Rs)||(!EXMEM_RegWrite)))? 2'b01: 2'b00;
    assign Forwarding_Ctrl2 = (EXMEM_RegWrite && (EXMEM_Reg_Rd != 0) && (EXMEM_Reg_Rd == IDEX_Reg_Rt
))? 2'b10:(MEMWB_RegWrite && (MEMWB_Reg_Rd != 0) && (MEMWB_Reg_Rd == IDEX_Reg_Rt) && ((EXMEM_Reg_Rd
!= IDEX_Reg_Rt)||(!EXMEM_RegWrite)))? 2'b01: 2'b00;
```

4.8 Hazard 模块

该模块为冒险综合控制模块，在流水线中存在 load-use 冒险时阻塞流水线，同时控制跳转指令执行后冲刷流水线以及出现异常/中断时冲刷流水线。模块共控制四种操作：保持 PC 寄存器值不变、保持 IF/ID 寄存器值不变、冲刷 IF/ID 寄存器中的指令、冲刷 ID/EX 寄存器中的控制信号。在各情形下的操作为：

表 15 Hazard 模块操作

情形	操作
Load-Use Hazard	保持 PC 寄存器值不变 保持 IF/ID 寄存器值不变 冲刷 ID/EX 寄存器中的控制信号
beq/bne/blez/bgtz/bltz 指令跳转	冲刷 IF/ID 寄存器中的指令 冲刷 ID/EX 寄存器中的控制信号
j/jr/jal/jalr 指令跳转	冲刷 IF/ID 寄存器中的指令
异常/中断	冲刷 IF/ID 寄存器中的指令 同时结合 Control 模块控制保存中断位置至$k0

模块中 Load-Use Hazard 判断逻辑为：

```
if (ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
or (ID/EX.RegisterRt = IF/ID.RegisterRt)))
stall the pipeline
```

模块输入输出定义如下：

表 16 Hazard 模块输入输出定义

信号名称	位宽	输入/输出	功能描述
OpCode	*6bits*	*input*	指令 OpCode
Funct	*6bits*	*input*	指令 Funct
Exception	*1bit*	*input*	Control 模块传来的异常/中断信号
Branch_and_Zero	*1bit*	*input*	EX 阶段 beq/bne/blez/bgtz/bltz 指令跳转条件判断结果

[1] 此处输入输出定义为理论课所学，本设计中针对此处有优化，见“5.2 ID-stage & ID/EX-register”。

IDEX_MemRead	*1bit*	*input*	ID_EX 阶段数据存储器读使能
IDEX_Reg_Rt	*5bits*	*input*	ID_EX 阶段 Rt 编号
IFID_Reg_Rs	*5bits*	*input*	IF_ID 阶段 Rs 编号
IFID_Reg_Rt	*5bits*	*input*	IF_ID 阶段 Rt 编号
pause	*1bit*	*input*	处理器暂停信号
Flush_Instruction	*1bit*	*output*	冲刷 IF/ID 寄存器中的指令
Flush_Control	*1bit*	*output*	冲刷 ID/EX 寄存器中的控制信号
Hold_PC	*1bit*	*output*	保持 PC 寄存器值不变
Hold_IF_ID	*1bit*	*output*	保持 IF/ID 寄存器值不变

模块内部用组合逻辑实现，核心代码如下，完整代码文件为 Hazard.v。

```
    assign Flush_Instruction = pause || ((Exception || Branch_and_Zero || (OpCode == 6'h02) || (OpCo
de == 6'h03) || (OpCode == 6'h00 && ((Funct == 6'h08) || (Funct == 6'h09))))? 1'b1: 1'b0);
    assign Flush_Control = ((IDEX_MemRead && ((IDEX_Reg_Rt == IFID_Reg_Rs) || (IDEX_Reg_Rt == IFID_R
eg_Rt))) || Branch_and_Zero)? 1'b1: 1'b0;
    assign Hold_PC = pause || ((IDEX_MemRead && ((IDEX_Reg_Rt == IFID_Reg_Rs) || (IDEX_Reg_Rt == IFI
D_Reg_Rt)))? 1'b1: 1'b0);
    assign Hold_IF_ID = (IDEX_MemRead && ((IDEX_Reg_Rt == IFID_Reg_Rs) || (IDEX_Reg_Rt == IFID_Reg_R
t)))? 1'b1: 1'b0;
```

流水线中的跳转、异常中断、load-use 冒险控制由 DataMemory（定时器外设）、Control 模块、Hazard 模块三者联合完成，其间层次结构如图：

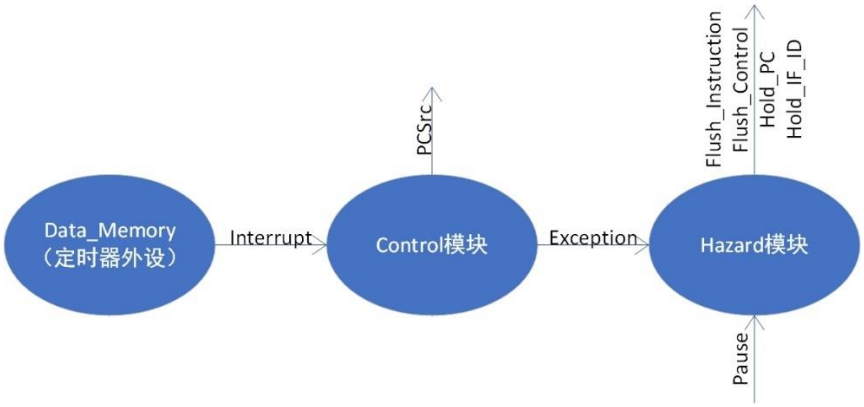


图 3 跳转、异常中断、load-use 冒险层次结构

5 流水级设计与优化

*各级流水线结构见总体设计图

5.1 IF-stage & IF/ID-register

IF 阶段为取指令阶段，该级流水线主要根据 PCSrc 信号更新 PC 值，而后更加 PC 值取对应指令。PC 的最高位 PC[31]为监督位。当该位为‘1’时，处理器处于内核态，此时异常和中断被禁止；当该位为‘0’时，处理器处于普通态，此时允许发生中断和异常。程序开始执行时，通过软件方法将 PC 的最高位由 1 置为 0，使得处理器进入普通态。具体操作见“6 汇编代码设计”。

PC 选取逻辑如下：

表 17 PC 选取逻辑

情形	PC next
正常执行	PC+4
beq/bne/blez/bgtz/bltz 指令跳转	PC+4+指令中立即数偏移量
j/jal 指令跳转	{PC[31:28]，指令中 26 位偏移量，2’b0}
jr/jalr 指令跳转	$ra 中地址

异常	0x00000008 异常处理程序入口
中断	0x00000004 中断服务程序入口

IF/ID 寄存器中保存内容如下：

表 18 IF/ID 寄存器设计

变量名	位宽	说明
IF_ID_PC_plus_4	*32bits*	下一指令地址，一般情况下为当前 PC+4，如果当前 ID/IF 中的指令正被冲刷则为 PC next+4
IF_ID_Instruction	*32bits*	当前指令
总计	*64bits*	

5.2 ID-stage & ID/EX-register

ID 阶段为指令译码阶段，Control 模块根据指令各字段产生控制信号。同时指令中 Rs、Rt 字段被送入寄存器堆地址输入端，读取相应的寄存器值。立即数扩展单元完成对 I 型指令中 16 位立即数的有符号/无符号扩展。特别地，在处理 lui 指令时 16 位立即数不经过立即数扩展单元，而是直接左移 16 位。

j/jr/jal/jalr 指令在这一阶段被识别出，Control 模块产生相应的 PCSrc 信号，Hazard 模块产生相应的冲刷控制信号。

初始设计中，Forwarding 模块按照理论课知识设计在 EX 阶段，但通过分析时序报告发现涉及转发操作时，EX 阶段需要执行如下步骤：根据各寄存器由组合逻辑生成转发控制信号；由 MEM 或 WB 阶段转发数据；ALU 或 Branch 模块对来的数据进行运算处理。这一系列操作耗时很长，极大限制了处理器的时钟频率。

因此在此处做优化设计：将 Forwarding 模块提前至 ID 阶段，其输入各信号均对应提前一个流水级；将 Forwarding 模块生成的控制信号存入 ID/EX 寄存器，在 EX 阶段控制转发。由此使得 ID 阶段分担 EX 阶段的工作负担，提高流水线时钟频率。

ID/EX 寄存器中保存内容如下：

表 19 ID/EX 寄存器设计

变量名	位宽	说明
ID_EX_Control	*17bits*	控制信号
ID_EX_PC_plus_4	*32bits*	下一指令地址
ID_EX_Instruction_Shamt	*5bits*	指令中偏移量（sll/srl/sra 指令）
ID_EX_Databus1	*32bits*	读出的寄存器数据 1
ID_EX_Databus2	*32bits*	读出的寄存器数据 2
ID_EX_16to32	*32bits*	扩展/左移后的立即数
ID_EX_Instruction_Funct	*6bits*	指令中 Funct
ID_EX_Instruction_Rt	*5bits*	指令中 Rt
ID_EX_Instruction_Rd	*5bits*	指令中 Rd
ID_EX_Forwarding_Ctrl	*4bits*	Forwarding 模块输出转发控制信号
总计	*170bits*	

5.3 EX-stage & EX/MEM-register

EX 阶段为指令执行阶段，该阶段由 ALUControl 模块产生 ALU 二级控制信号，决定 ALU 运算类型。同时由 Forwarding 模块产生的控制信号和 ALUSrc1、ALUSrc2 共同控制送入 ALU 的两路数据，计算得到计算型指令的结果或数据交换型指令的数据地址。同时，对于 beq/bne/blez/bgtz/bltz 指令，Branch 单元在 BranchOp 的控制下判断其分支条件是否成立，得到 zero 信号，zero 信号与 Branch 信号取与运算，得到最终的分支判断。

同时该阶段由 RegDst 信号控制选择出回写寄存器的编号，选择逻辑如下：

表 20 回写寄存器编号选择逻辑

情形	回写寄存器选择
正常执行，需要回写寄存器的 I 型指令	指令中 Rt
正常执行，需要回写寄存器的 R 型指令	指令中 Rd
jal/jalr 指令	$ra
异常/中断	$k0

EX/MEM 寄存器中保存内容如下：

表 21 EX/MEM 寄存器设计

变量名	位宽	说明
EX_MEM_Control	*5bits*	控制信号
EX_MEM_PC_plus_4	*32bits*	下一条指令地址
EX_MEM_ALU_out	*32bits*	ALU 计算结果
EX_MEM_Databus2	*32bits*	读出的寄存器数据 2
EX_MEM_Write_register	*5bits*	回写寄存器编号
总计	*106bits*	

5.4 MEM-stage & MEM/WB-register & WB-stage

MEM 阶段为存储器访问阶段，指令 lw 根据数据地址读取数据存储器中的数据，指令 sw 根据数据地址向数据存储器中写入数据。WB 阶段为回写寄存器阶段，根据回写寄存器编号将数据写入指定寄存器。

一般流水线设计中，在 WB 阶段首先由 MemtoReg 信号控制选出要回写的数据，再于时钟下降沿将数据写回寄存器中。这种方式存在弊端，由于寄存器堆需要支持先写后读，时钟下降沿写入数据后，留给数据读取操作的时间小于 1/2 个时钟周期，不利于提高时钟频率。

本设计中针对此问题进行优化：在 MEM 阶段提前由 MemtoReg 信号选取要回写的数据，选择逻辑如下：

表 22 回写数据选择逻辑

情形	回写数据选择
一般计算型指令	ALU 计算结果
lw 指令	数据存储器读出数据
异常	PC+4
中断	(PC+4)-4

特别指出，当中断发生时，当前指令没有执行完就被冲刷了，因此返回地址需要再减去 4，返回后重新执行当前指令。

选出要回写的数据后，本设计在 MEM 阶段增加三个寄存器，分别存储寄存器写使能信号 RegWrite、回写寄存器编号、回写数据。由此，下一个时钟周期开始时，指令进入 WB 阶段，回写所需数据已经全部准备完毕，可以在时钟上升沿直接写入寄存器堆。这一设计对提升时钟频率很有帮助。

MEM/WB 寄存器中保存内容如下：

表 23 MEM/WB 寄存器设计

变量名	位宽	说明
MEM_WB_Databus	*32bits*	回写数据
总计	*32bits*	

5.5 流水级组装与顶层模块

按照上述设计，连接组装各流水级，形成顶层模块 CPU，其输入输出定义如下：

表 24 顶层模块输入输出定义

信号名称	位宽	输入/输出	功能描述
reset	*1bit*	*input*	复位信号
pause	*1bit*	*input*	暂停信号
clk	*1bit*	*input*	时钟信号
DIGI	*12bits*	*output*	七段数码管控制
LED	*8bits*	*output*	该模块 LED 控制

顶层模块完整代码文件为 CPU.v。

6 汇编代码设计

6.1 PC 监督位设置代码

处理器复位后 PC 监督位为 1，处理器处于内核态，程序开始执行前需要先将其置为 0，使得处理器进入普通态。本设计中通过 jal 指令获取 PC 值，再操作$ra 寄存器实现 PC 监督位置零，最后通过 jr $ra 指令将监督位置零后的$ra 设为新的 PC，汇编代码如下：

```
j Main
j Interrupt
j Exception
Main:   jal start
start:  li $t0,0x7fffffff
    and $ra,$ra,$t0
    addi $ra,$ra,28
    li $s0,0x40000000
    jr $ra
```

6.2 数据生成与载入

本设计中通过 lw 指令将待排序的数据载入 DataMemory。为便于操作，编写 Python 脚本 randomint.py，随机生成 128 个数字整数，并得到对应的载入指令。部分载入指令如下：

```
#load data
li $a0,0x00000000
li $t0,266
sw $t0,0($a0)
li $t0,834
sw $t0,4($a0)
li $t0,424
sw $t0,8($a0)
```

6.3 冒泡排序

使用冒泡排序算法对 128 个随机数进行排序，核心代码如下：

```
#bubsort
li $s1,128
li $s2,0 #$s2=i,a0=v[]
li $s3,0 #$s3=j
for1:   slt $t2,$s2,$s1
    beq $t2,$zero,end
    addi $s3,$s2,-1
for2:   bltz $s3,out2
    sll $t2,$s3,2
    add $t2,$a0,$t2
    lw $t3,0($t2)
    lw $t4,4($t2)
    slt $t5,$t4,$t3
    beq $t5,$zero,out2
    sw $t4,0($t2)
    sw $t3,4($t2)
    addi $s3,$s3,-1
    j for2
    out2:   addi $s2,$s2,1
```

```
    j for1
end:
```

程序中在排序开始前和完成后分别记录系统时钟计数器的值，相减得到排序消耗的时钟周期数。同时设计排序完成后将 8 个 LED 点亮以作为标志。

这一程序中包含了多种指令组合情形，涵盖了多种指令类型，能够充分地测试出设计处理器的功能与性能。

6.4 中断服务（七段数码管显示）

利用定时器中断信号对七段数码管进行扫描显示，扫描频率为 1kHz，设置扫描一定次数后关闭七段数码管并熄灭 8 个 LED 灯（为方便仿真测试，此处仅设置扫描五轮）。进入中断服务程序后定时器中断使能禁止，同时中断状态清除，中断服务程序中首先利用逻辑运算指令取出待显示数字的每一位，通过 srl 指令将取出的数位移动到最低位。再通过查表的方法获得七段数码管上对应的控制信号予以显示。退出中断服务函数前恢复定时器中断使能。数字 0-F 对应七段数码管控制信号（8bits）为：

表 25 数字 0-F 对应七段数码管控制信号

数字	0	1	2	3	4	5	6	7
控制信号	0x0c	0xf9	0xa4	0xb0	0x99	0x92	0x82	0xf8
数字	8	9	A	b	C	d	E	F
控制信号	0x80	0x90	0x88	0x83	0xc6	0xa1	0x86	0x8e

中断服务函数核心代码为：

```
    Interrupt:  li $t0,0x00000001 #set TCON
       sw $t0,8($s0)
       beq $s5,$zero,pos_zero
       li $at,1
       beq $s5,$at,pos_one
       li $at,2
       beq $s5,$at,pos_two
       li $at,3
       beq $s5,$at,pos_three
pos_zero:   li $s4,0x00000100
       andi $t1,$s7,0x000f
       j number
pos_one:    li $s4,0x00000200
       andi $t1,$s7,0x00f0
       srl $t1,$t1,4
       j number
pos_two:    li $s4,0x00000400
       andi $t1,$s7,0x0f00
       srl $t1,$t1,8
       j number
pos_three:  li $s4,0x00000800
       andi $t1,$s7,0xf000
       srl $t1,$t1,12
number:     li $at,0x00000000
       beq $t1,$at,number0
       li $at,0x00000001
       beq $t1,$at,number1
       li $at,0x00000002
       beq $t1,$at,number2
       li $at,0x00000003
       beq $t1,$at,number3
       li $at,0x00000004
       beq $t1,$at,number4
       li $at,0x00000005
       beq $t1,$at,number5
       li $at,0x00000006
       beq $t1,$at,number6
       li $at,0x00000007
       beq $t1,$at,number7
       li $at,0x00000008
       beq $t1,$at,number8
```

```
        li $at,0x00000009
        beq $t1,$at,number9
        li $at,0x0000000a
        beq $t1,$at,numberA
        li $at,0x0000000b
        beq $t1,$at,numberB
        li $at,0x0000000c
        beq $t1,$at,numberC
        li $at,0x0000000d
        beq $t1,$at,numberD
        li $at,0x0000000e
        beq $t1,$at,numberE
        li $at,0x0000000f
        beq $t1,$at,numberF
number0:    ori $s4,$s4,0x00c0
        j show
number1:    ori $s4,$s4,0x00f9
        j show
number2:    ori $s4,$s4,0x00a4
        j show
number3:    ori $s4,$s4,0x00b0
        j show
number4:    ori $s4,$s4,0x0099
        j show
number5:    ori $s4,$s4,0x0092
        j show
number6:    ori $s4,$s4,0x0082
        j show
number7:    ori $s4,$s4,0x00f8
        j show
number8:    ori $s4,$s4,0x0080
        j show
number9:    ori $s4,$s4,0x0090
        j show
numberA:    ori $s4,$s4,0x0088
        j show
numberB:    ori $s4,$s4,0x0083
        j show
numberC:    ori $s4,$s4,0x00c6
        j show
numberD:    ori $s4,$s4,0x00a1
        j show
numberE:    ori $s4,$s4,0x0086
        j show
numberF:    ori $s4,$s4,0x008e
show:       li $at,4
        addi $s5,$s5,1
        sw $s4,16($s0)
        bne $s5,$at,continue
        addi $s6,$s6,1
        li $at,5
        li $s5,0
        bne $s6,$at,continue
        sw $zero,8($s0)
        sw $zero,12($s0)
        li $t0,0x00000fff
        sw $t0,16($s0)
        jr $k0
continue:   li $t0,0x00000003
        sw $t0,8($s0)
        jr $k0
```

6.5 异常处理程序

本设计中对于未定义指令异常将其简单跳过，代码如下：

```
Exception:  nop
        nop
        nop
        jr $k0
```

6.6 软件性能优化

除硬件优化外，在软件中可以通过交换指令顺序以减少转发需求，实例如下：

表 26 软件优化前后对比

原代码	优化后代码

addi $s5,$s5,1
sw $s4,16($s0)
li $at,4
bne $s5,$at,continue
	li $at,4
	sw $s4,16($s0)
	addi $s5,$s5,1
	bne $s5,$at,continue

优化后将载入立即数操作提前，在执行 bne 指令时立即数已经载入完成，不再需要依赖转发。

完整代码文件为 mips1.asm。利用 MARS 导出十六进制机器码后，插入一条 0xffffffff 未知指令以测试处理器异常设计，编写 Python 脚本 convert.py 将其转化为 Verilog 语句，写入 InstructionMemory.v 中。

6.7 指令数测试

使用与 mips1.asm 中相同的数据，去除其他部分而仅保留冒泡排序部分，同时去除全部 nop 空指令，得到 mips2.asm 文件，利用 MARS 统计指令数目，结果如下，共 45849 条指令。

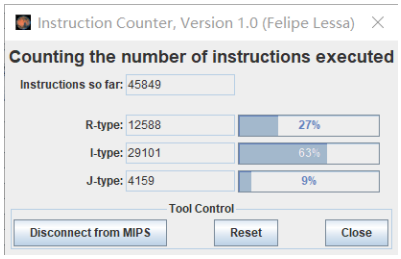


图 4 指令数目统计

7 行为级仿真验证[2]

7.1 testbench 设计

设计 testbench 文件 test_cpu.v，例化流水线处理器后先将其复位，再接入 100MHz 的时钟，使流水线处理器开始运行，其代码如下：

```
`timescale 1ns / 1ps
module test_cpu();

    reg reset;
    reg pause;
    reg clk;
    wire [11:0] DIGI;
    wire [7:0] LED;

    CPU cpu1(reset, pause, clk, DIGI, LED);

    initial begin
        reset = 0;
        clk = 0;
        pause = 0;
        #5 reset = 1;
        #8 reset = 0;
        #60000 $finish;
    end

    always #5 clk = ~clk;

endmodule
```

[2] 考虑到板上时钟周期为 100MHz，行为级仿真和实现后仿真均使用 10ns 周期的时钟信号。由于此时仍存在时间裕度，获取最高主频时进一步减小时钟周期，得到本文中所述的主频。

7.2 分支跳转功能验证

仿真结果如下：

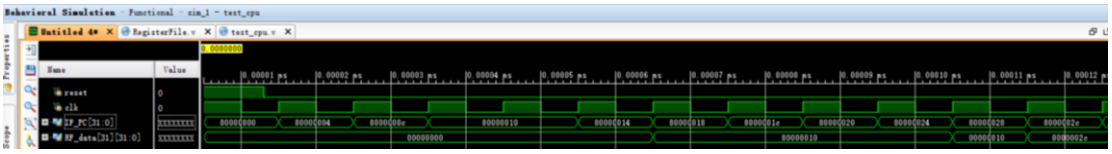


图 5 行为级仿真分支跳转功能

可见 0x80000000 处 j Main 指令控制流水线冲刷掉下一条指令 0x80000004 后跳转至 0x8000000c 处的 jal start 指令；执行 jal start 指令流水线跳转至 0x80000010 处的 start 标签，同时一段时间后 jal start 指令的下一条指令地址正确存入$ra。执行至 0x80000028 处 jr $ra 指令后正确跳转至此时经过运算的$ra 中的地址 0x0000002c，实现 PC 监督位置零。分支跳转功能无误。

7.3 异常处理功能验证

仿真结果如下：

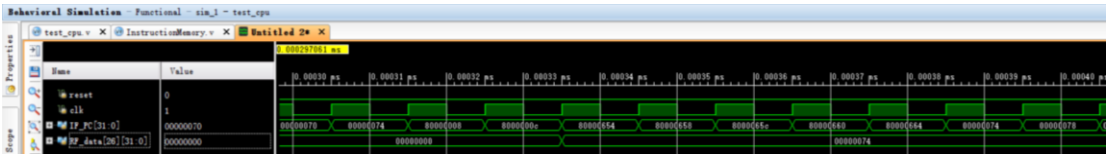
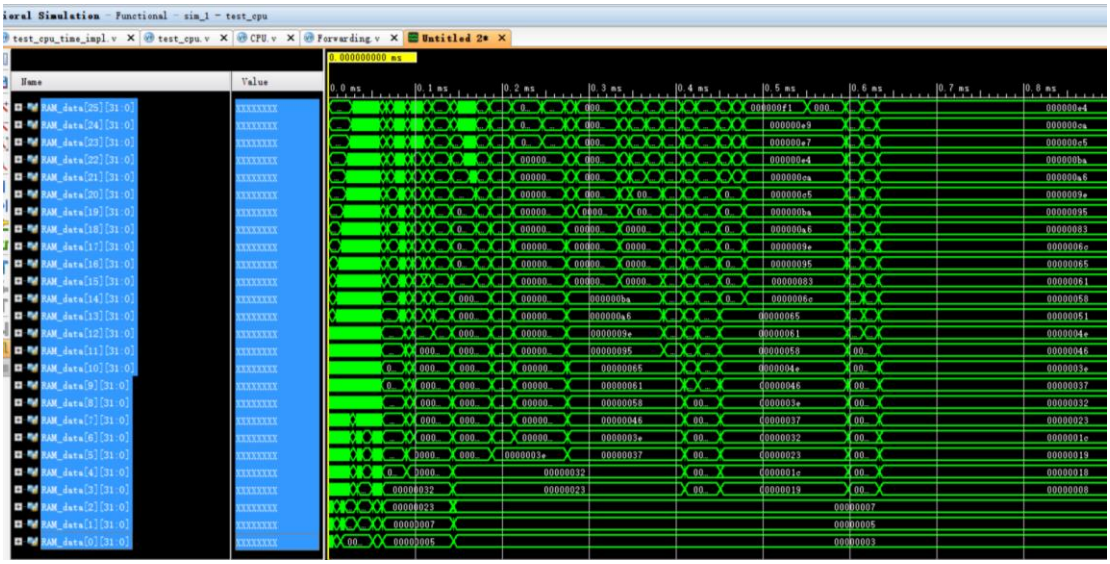


图 6 行为级仿真异常处理功能

0x00000070 处为未定义指令 0xffffffff[3]，可见执行到此处时下一条指令被冲刷后流水线跳转到异常处理程序入口 0x80000008，流水线进入内核态，同时下一条指令地址 0x00000074 一段时间后被写入$k0。流水线由异常处理程序入口跳转至异常处理程序 0x80000654，执行完毕异常处理程序后由 0x80000660 处 jr $k0 冲刷掉下一条指令 0x800000664，跳转回到 0x00000074 继续执行原程序，流水线恢复普通态。异常处理功能无误。

7.4 冒泡排序综合功能验证

部分仿真结果如下：



3 此指令仅用于测试，所附.asm 文件中不包含该异常指令

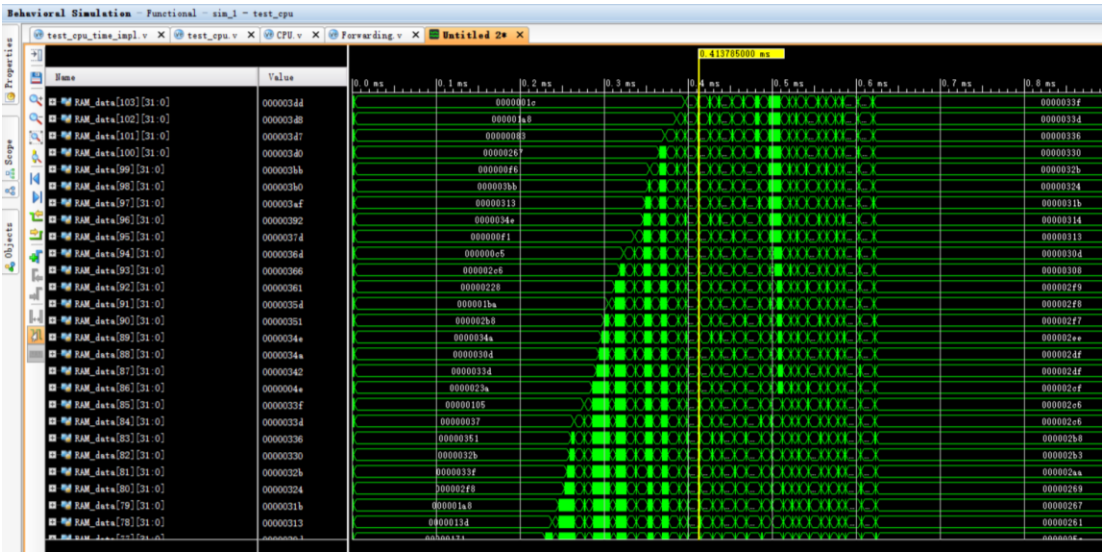


图 7 行为级仿真冒泡排序功能

经过一段时间的执行，由多次“冒泡”操作后数据存储器前 128 个单元中的数字被正确排成有序数字（上图仅展示部分单元），冒泡排序功能无误。

7.5 中断服务（七段数码管显示）功能验证

仿真结果如下：

图 8 行为级仿真中断服务功能

定时器外设每 1ms 发送一次中断信号，程序进入中断服务，$k0 寄存器中正确保存了返回地址 0x000004b8，控制七段数码管的 12 位二进制数变换一次，实现扫描显示。扫描五轮后控制信号变为 fff，七段数码管关闭。中断服务功能无误。

8 实验验证（综合、后仿真情况及分析）

8.1 管脚约束

设置如下管脚绑定：

表 27 管脚绑定

信号	绑定硬件
DIGI[7:0]	七段数码管 DP、CG、……CA
DIGI[11:8]	七段数码管 AN3、……AN0
LED[7:0]	LED7、……LED0
clk	板上 100MHz 时钟
reset	按键 BTNL
pause	按键 BTNR

编写约束文件 CPU.xdc 内容如下：

```
#reset&pause
set_property -dict {PACKAGE_PIN W19 IOSTANDARD LVCMOS33} [get_ports {reset}]
set_property -dict {PACKAGE_PIN T17 IOSTANDARD LVCMOS33} [get_ports {pause}]

#leds
set_property -dict {PACKAGE_PIN U16 IOSTANDARD LVCMOS33} [get_ports {LED[0]}]
set_property -dict {PACKAGE_PIN E19 IOSTANDARD LVCMOS33} [get_ports {LED[1]}]
set_property -dict {PACKAGE_PIN U19 IOSTANDARD LVCMOS33} [get_ports {LED[2]}]
set_property -dict {PACKAGE_PIN V19 IOSTANDARD LVCMOS33} [get_ports {LED[3]}]
set_property -dict {PACKAGE_PIN W18 IOSTANDARD LVCMOS33} [get_ports {LED[4]}]
set_property -dict {PACKAGE_PIN U15 IOSTANDARD LVCMOS33} [get_ports {LED[5]}]
set_property -dict {PACKAGE_PIN U14 IOSTANDARD LVCMOS33} [get_ports {LED[6]}]
set_property -dict {PACKAGE_PIN V14 IOSTANDARD LVCMOS33} [get_ports {LED[7]}]

#digits
set_property -dict {PACKAGE_PIN W7 IOSTANDARD LVCMOS33} [get_ports {DIGI[0]}]
set_property -dict {PACKAGE_PIN W6 IOSTANDARD LVCMOS33} [get_ports {DIGI[1]}]
set_property -dict {PACKAGE_PIN U8 IOSTANDARD LVCMOS33} [get_ports {DIGI[2]}]
set_property -dict {PACKAGE_PIN V8 IOSTANDARD LVCMOS33} [get_ports {DIGI[3]}]
set_property -dict {PACKAGE_PIN U5 IOSTANDARD LVCMOS33} [get_ports {DIGI[4]}]
set_property -dict {PACKAGE_PIN V5 IOSTANDARD LVCMOS33} [get_ports {DIGI[5]}]
set_property -dict {PACKAGE_PIN U7 IOSTANDARD LVCMOS33} [get_ports {DIGI[6]}]
set_property -dict {PACKAGE_PIN V7 IOSTANDARD LVCMOS33} [get_ports {DIGI[7]}]
set_property -dict {PACKAGE_PIN U2 IOSTANDARD LVCMOS33} [get_ports {DIGI[8]}]
set_property -dict {PACKAGE_PIN U4 IOSTANDARD LVCMOS33} [get_ports {DIGI[9]}]
set_property -dict {PACKAGE_PIN V4 IOSTANDARD LVCMOS33} [get_ports {DIGI[10]}]
set_property -dict {PACKAGE_PIN W4 IOSTANDARD LVCMOS33} [get_ports {DIGI[11]}]

#clk
set_property -dict {PACKAGE_PIN W5 IOSTANDARD LVCMOS33} [get_ports {clk}]
create_clock -name clk -period 8.500 [get_ports clk]
```

[4]

8.2 资源占用情况

板上资源占用情况如下：

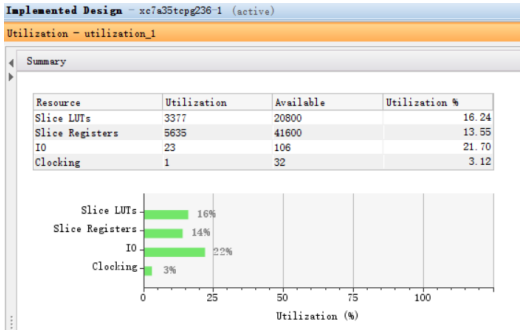


图 9 资源占用情况

本设计中有寄存器堆、PC 寄存器、流水线寄存器等多处寄存器设计，因此占用了较多的板上 Registers 资源。同时各处组合逻辑需要通过查找表实现，也占用了较多的 LUT 资源。

8.3 时序性能

设置时钟周期为 8.5ns，静态时序分析结果如下：

图 10 静态时序分析

[4] 由于设定时钟周期为 10ns 时仍存在时间裕度，获取最高主频时进一步减小时钟周期至 8.5ns，得到本文中所述的主频。如要烧写到硬件上，此处应配合硬件改为 10ns 时钟周期。

由此可以得到最高时钟频率

$$f_0 = \frac{1}{(8.5 - 0.062) \times 10^{-9}} = 118.5MHz$$

关键路径中主要涉及访问数据寄存器，因此数据存储器的访存耗时成为制约处理器主频的重要因素。

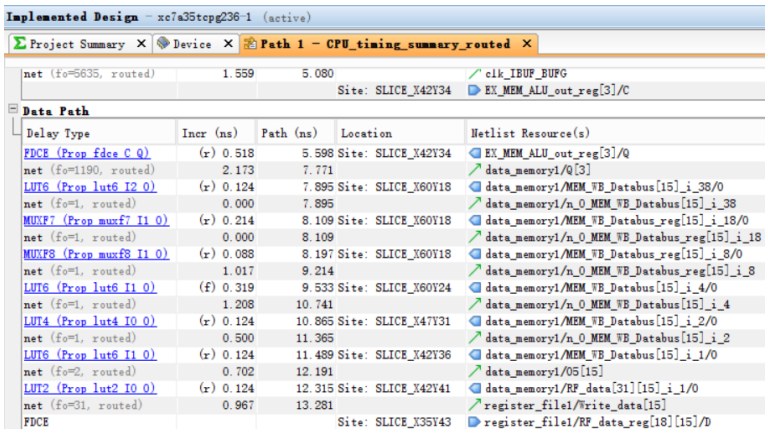


图 11 关键路径延时

8.4 Post-Implementation Timing Simulation

综合并实现后再次仿真[5]，验证设计功能正确如下：

（1）异常处理功能

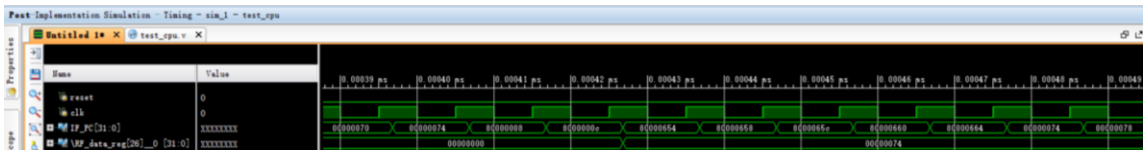


图 12 Post-Implementation Timing Simulation 异常处理功能

0x00000070 处为未定义指令 0xffffffff，可见执行到此处时下一条指令被冲刷后流水线跳转到异常处理程序入口 0x80000008，流水线进入内核态，同时下一条指令地址 0x00000074 一段时间后被写入$k0。流水线由异常处理程序入口跳转至异常处理程序 0x80000654，执行完毕异常处理程序后由 0x80000660 处 jr $k0 冲刷掉下一条指令 0x800000664，跳转回到 0x00000074 继续执行原程序，流水线恢复普通态。异常处理功能无误。

（2）冒泡排序功能

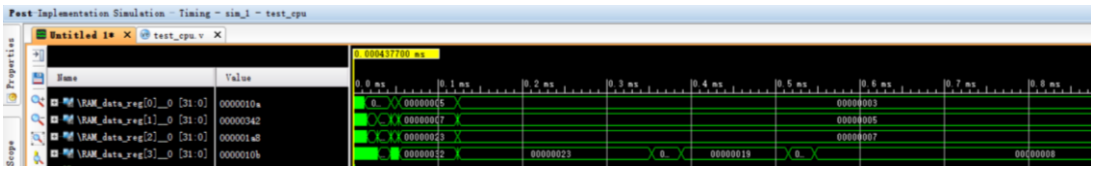


图 13 Post-Implementation Timing Simulation 异常处理冒泡排序功能

为缩短后仿真时间，此处仅选取了数据存储器中前四个单元的内容进行仿真。由于 128 个数字随机生成，最后位于前四个单元的四个最小数字初始时分布在数据存储器各处，只有整个程序正确执行，才能够得到正确的排序结果。因此前四个单元的内容足以验证整个冒泡排序功能的正确性。

经过一段时间的执行，由多次“冒泡”操作后数据存储器前 4 个单元中的数字被正确排序，且排序结果与 behavior simulation 中完全一致，冒泡排序功能无误。

[5] 前面的时序性能为综合时开启 flatten-hierarchy 选项得到的结果，此处后仿为便于跟踪信号，关闭了 flatten-hierarchy 选项进行综合和实现

（3）中断服务（七段数码管显示）功能

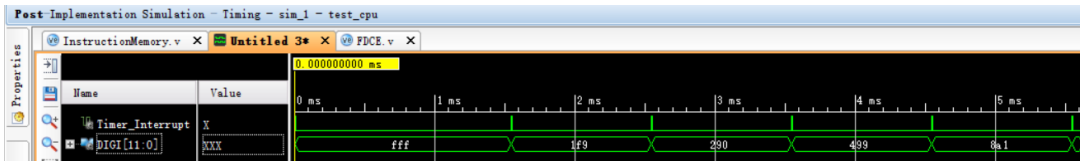


图 14 Post-Implementation Timing Simulation 中断服务功能

定时器外设每 1ms 发送一次中断信号，程序进入中断服务，控制七段数码管的 12 位二进制数变换一次，实现扫描显示。控制信号依次为 0x1f9-0x290-0x499-0x8a1，与 behavior simulation 中完全一致，中断服务功能无误。

显示内容为十六进制数 D491，即十进制数 54417，前面测得全部指令数目为 45849，由此得到

$$CPI = \frac{54417}{45849} = 1.187$$

8.5 与单周期处理器对比

该处理器与单周期 32 位 MIPS 处理器时序性能对比如下：

表 28 单周期处理器对比

处理器	主频/MHz	CPI	指令执行数目/s^{-1}
单周期处理器	54.7	1	54.7M
流水线处理器	118.5	1.187	99.8M

可见流水线处理器牺牲了一部分 CPI，但极大提高了时钟频率，使得指令执行效率提升了近一倍。同时，流水线可以使用转发、动态分支预测、编译器优化等多种手段，使得 CPI 非常接近 1，能够有效提高处理器执行效率。

该处理器与单周期 32 位 MIPS 处理器占用资源对比如下：

表 29 单周期处理器对比

处理器	占用 LUTs（%）	占用 Registers（%）
单周期处理器	18	21
流水线处理器	16	14

可见两者占用 LUT 资源基本相等，由于该流水线处理器设计中 DataMemory 部分容量设计较小，因此占用 Registers 数目小于单周期处理器。

9 经验体会

在设计、验证、改进流水线处理器设计的过程中，我有如下经验与体会：

首先，要整体把握系统结构。开始时，我并未画出完整的设计图，仅参考理论课上各功能的局部讲解图，凭借脑海中的印象开始编写设计代码。这时遇到较为复杂的逻辑常常无从下手，难以设计正确。因此我转而先画出完整、详细的设计图，再根据设计图编写代码，这时设计变得十分清晰明了，代码书写只需要根据设计图转化即可。把握系统整体结构是大量工程问题中都需要的，只有放眼整体才能够明确各个局部的需求。

其次，在较为复杂的系统设计中，要注意保留设计、修改过程的记录。在调试初期，我往往直接在代码上修改，常常出现新的修改不起作用而忘记原设计无法回退，造成了很多麻烦。因此，设计中一定要保留每一版本的设计文件，并记录相应的改动，以备查阅。在这方面可以使用 github 等工具，十分便捷。

第三，在解决一切问题的过程中都要抓住主要矛盾进行突破。在初步设计出流水线，进行提高时钟频率的优化时，我最先使用的方法是根据理论课知识和自身经验，优化“可能比较耗时”的环节。一番尝试后发现并没有很明显的效果，分析后发现这些“可能比较耗时”

的环节并非制约时钟频率的关键路径，优化之只能起到“长板更长而短板未补”的效果。因此，我转而根据时序报告中关键路径进行针对性优化，效果明显了很多。

最后，在反复修改、调试的过程中，我认识到解决一个工程问题是不可能一蹴而就的，需要充分运用所学知识、仔细思考、反复尝试才有可能取得进展。

10 文件清单

表 30 文件清单

文件（夹）名	说明
实验报告.pdf（本文件）	实验报告
pipeline.jpg	整体设计图
/project_1	vivado 工程文件（为避免文件过大，工程中只进行了综合、实现，未保存仿真结果）
/asm code	汇编代码
/asm code/mips1.asm	用于指令存储器中的汇编代码
/asm code/mips2.asm	用于统计排序指令数的汇编代码
/Python code	辅助 Python 脚本
/Python code/convert.py	用于将机器码转化为 Verilog 代码的脚本
/Python code/randomint.py	用于生成随机整数的脚本
/Verilog code	Verilog 设计代码
/Verilog code/CPU.v	流水线组装与顶层模块
/Verilog code/ALU.v	ALU 模块
/Verilog code/ALUControl.v	ALU 控制模块
/Verilog code/Branch.v	分支条件判断模块
/Verilog code/Control.v	控制模块
/Verilog code/DataMemory.v	数据存储器模块
/Verilog code/Forwarding.v	转发控制模块
/Verilog code/Hazard.v	冒险综合控制模块
/Verilog code/InstructionMemory.v	指令存储器模块
/Verilog code/RegisterFile.v	寄存器堆模块
/Verilog code/test_cpu.v	测试向量
/Verilog code/CPU.xdc	管脚约束文件