MIPS 流水线处理器设计

设计方案

一、总体设计

分为 IF、ID、EX、MEM、WB 五个流水阶段。j、jr、jal、jalr 指令在 ID 阶段检出并 flush IF，并可选是否使用延迟槽；beq 指令在 EX 阶段检出，采取静态分支预测（预测不分支，分支则 flush）。其中，jal 与 jalr 在 WB 阶段写回寄存器堆。

二、控制信号设计

单周期处理器的控制信号共 11 个，分别为：RegDst[1:0]，PCSrc[1:0]，Branch，MemRead，MemtoReg[1:0]，MemWrite，ALUSrc1，ALUSrc2，RegWrite，LuOp，ExtOp，可支持原单周期处理器的 26 条指令。由于字符串搜索算法中使用了 ori、bne、lbu 指令，增加该 3 条指令。针对 bne 新增 EqualOp 控制信号，针对 lbu 新增 LbOp 控制信号。引入的新指令不会产生新种类的冒险。

三、冒险解决设计

考虑以下几种冒险及解决方案：

①Read-after-Write 冒险，即计算指令（及 beq）在 EX 阶段依赖前条指令（MEM）/前前条指令（WB）的计算结果。使用 forwarding unit 解决。

②Save-after-Load 冒险，即 save 指令在 MEM 阶段依赖前条 load 指令（WB）的结果。使用 forwarding unit 解决。

③Load-use 冒险，即计算指令（及 beq）在 EX 阶段依赖前条 load 指令（WB）的结果或 jr/jalr 在 ID 阶段依赖前条 load 指令 (WB) /前前条 load 指令 (WB) 的结果。前者使用 harzard unit 进行 1 个周期的 stall + ①中 forwarding unit 解决，后者使用 harzard unit 进行 1 个/2 个周期的 stall（具体实现只需要进行 1 个周期的 stall，需要 2 个周期 stall 的情况则会退化为需要 1 个周期 stall 的情况） + 寄存器堆先写后读（用寄存器内转发实现）解决。

④Jump 指令的控制冒险，即 j/jal/jr/jalr 的控制冒险。使用 hazard unit 取消 IF 阶段指令解决。

⑤Branch 指令的控制冒险，即 beq/bne 的控制冒险。使用 hazard unit 在分支发生时刻取消 ID 和 IF 阶段的两条指令（即默认预测不分支）解决。

⑥jr/jalr-after-Write 冒险, 即 jr/jalr 在 ID 阶段依赖前条 jal/jalr（MEM, 中间有 1 个周期 stall）或依赖前条计算指令（EX）/前前条计算指令（MEM）的结果。使用 forwarding unit 解决。

四、数据通路设计

根据以上思路设计数据通路如下页图。一些说明：

1.①-⑥分别对应上文冒险解决设计中的各解决方案模块。

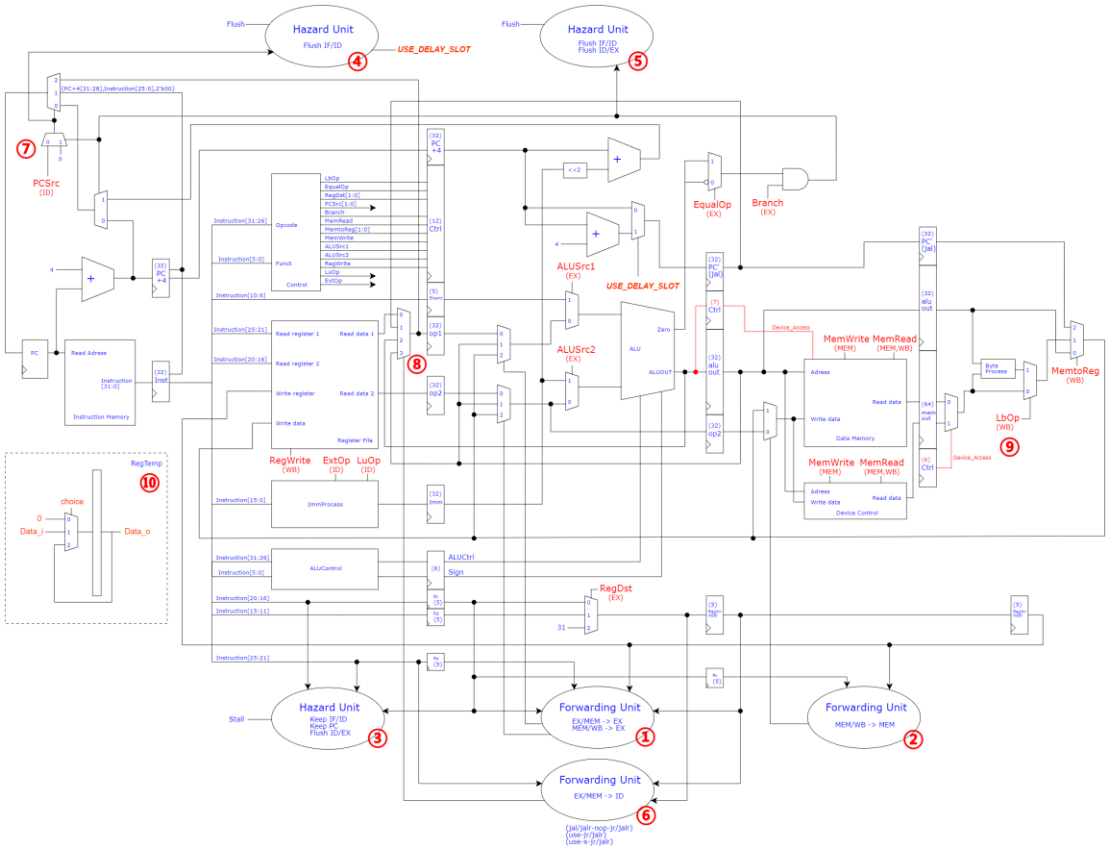
2.⑦为这种特殊情况设计：Jump 指令紧跟在 beq 指令之后，则 beq 进行到 EX 阶段时 Jump 指令进行到 ID, PCSrc 译码为 1 或 2, 若无该单元则 beq 不能正常跳转（需要 MUX 选为 0）。在 EX 阶段检出分支发生时，该 MUX 可将原 PCSrc 控制的 MUX 强制选为 0，而其他情况下 PCSrc 正常发挥作用。

3.⑧为将跳转指令扩充为 j、jal、jr、jalr 所需引入的 MUX，用于解决 jr/jalr-after-Write 冒险

问题。（与⑥搭配使用）

4.⑨为实现 lbu 指令所设计的单元，放在 WB 阶段防止 MEM 阶段关键路径过长，可根据 LbOp 取出存储器和外设中对应的字节或字。

5.⑩为各流水阶段间寄存器的设计，可控制寄存器的 flush、keep 与正常打拍。



关键代码

一、寄存器堆内部转发功能（即先写后读）实现——RegisterFile.v

```
// Write, then read
assign Read_data1=(Read_register1!=0 && RegWrite && Write_register==Read_register1)?
                  Write_data:RF_data[Read_register1];
assign Read_data2=(Read_register2!=0 && RegWrite && Write_register==Read_register2)?
                  Write_data:RF_data[Read_register2];
```

二、使用 IP 核 Block Memory Generator 构建例化的数据存储器——CPU.v

（详细配置见后文“优化历程”）

```
blk_mem_gen_0 RAM(
    .clka(clk),
    .ena(EX_MemRead|MEM_MemWrite),
    .wea(MEM_MemWrite & !MEM_Device_Access & !EX_MemRead),
    .addra(RAM_Address),
    .dina(MEM_Write_data),
    .douta(MEM_Read_data)
);
```

三、可进行阻塞、清空的各流水级间寄存器逻辑实现——RegTemp.v

```
always@(posedge reset or posedge clk) begin
    if (reset) begin
        Data_o <= 0;
    end else begin
        if(choice==2'b00) begin
            Data_o <= 0;
        end
        else if(choice==2'b01) begin
            Data_o <= Data_i;
        end
        else begin
            Data_o <= Data_o;
        end
    end
end
```

四、解决 Read-after-Write 冒险的转发单元实现——ForwardingUnit.v

```
// EX/MEM->EX
wire conditionA1=IDEX_Rs != 0 && EXMEM_RegWr && EXMEM_RegWrAddr==IDEX_Rs;
wire conditionB1=IDEX_Rt != 0 && EXMEM_RegWr && EXMEM_RegWrAddr==IDEX_Rt;
// MEM/WB->EX
wire conditionA2=IDEX_Rs != 0 && MEMWB_RegWr && MEMWB_RegWrAddr == IDEX_Rs;
wire conditionB2=IDEX_Rt != 0 && MEMWB_RegWr && MEMWB_RegWrAddr == IDEX_Rt;
assign ForwardA=conditionA1?2'b01:
                conditionA2?2'b10:
                2'b00;
assign ForwardB=conditionB1?2'b01:
                conditionB2?2'b10:
                2'b00;
```

五、解决 Save-after-Load 冒险的转发单元实现——ForwardingUnit.v

```
wire condition=EXMEM_Rt != 0 && EXMEM_MemWr && MEMWB_MemRead && MEMWB_RegWrAddr == EXMEM_Rt;
assign Forward=condition?1'b1:1'b0;
```

六、解决 jr/jalr-after-Write 冒险的转发单元实现——ForwardingUnit.v

```
// jr/jalr
wire conditionjr=ID_OpCode == 6'b0 && (ID_Funct == 6'b001000 || ID_Funct == 6'b001001);
// jal/jalr-nop-jr/jalr
wire condition1=(EXMEM_MemtoReg == 2'b10);
// use-x-jr/jalr
wire condition2=(ID_Rs != 0 && EXMEM_RegWr && EXMEM_RegWrAddr == ID_Rs);
// use-jr/jalr
wire condition3=(ID_Rs != 0 && IDEX_RegWr && IDEX_RegWrAddr == ID_Rs);
assign Forward=!conditionjr?2'b00:
                (condition1?2'b01:
                condition2?2'b11:
                condition3?2'b10:2'b00);
```

七、解决 Jump 指令的控制冒险的冒险检测单元（使用延迟槽时无效）——HazardUnit.v

```
wire condition=(USE_DELAY_SLOT == 0 && ID_willjump != 2'b00);
assign IFID_choice=condition?2'b00:// Flush IF/ID;
                    2'b01;
```

八、解决 Branch 指令的控制冒险的冒险检测单元——HazardUnit.v

```
wire condition=EX_willbranch;
assign IFID_choice=condition?2'b00:// Flush IF/ID;
                    2'b01;
assign IDEX_choice=condition?2'b00:// Flush ID/EX;
                    2'b01;
```

九、解决 Load-use 冒险的冒险检测单元——HazardUnit.v

```
wire condition=IDEX_MemRead && (IDEX_Rt==IFID_Rs || (!IFID_MemWr && IDEX_Rt==IFID_Rt));
assign PC_choice=condition?2'b10:// Keep PC;
                    2'b01;
assign IFID_choice=condition?2'b10:// Keep IF/ID;
                       2'b01;
assign IDEX_choice=condition?2'b00:// Flush ID/EX;
                       2'b01;
```

十、外设控制器实现——DeviceControl.v

```
always @(*)begin
    if(MemRead)begin
        case(Address)
            32'h4000000C: Read_data <= {24'b0,LED};
            32'h40000010: Read_data <= {20'b0,DIGI};
            default: Read_data <= 32'b0;
        endcase
    end
    else Read_data <= 32'b0;
end

integer i;
always @(posedge reset or posedge clk)begin
    if(reset)begin
        LED <= 8'b0;
        DIGI <= 12'b0;
    end
    else if(MemWrite)begin
        case(Address)
            32'h4000000C: LED <= Write_data[7:0];
            32'h40000010: DIGI <= Write_data[11:0];
        endcase
    end
end
```

文件清单

./asmcode/	存放相关汇编程序代码的目录
./Instructions/	存放相关机器码指令的目录
./test_delayslot/	存放使用延迟槽、使用 IP 核的处理器的项目代码目录
./test_nodelayslot/	存放无延迟槽、使用 IP 核的处理器的项目代码目录
./test_noip/	存放无延迟槽、不使用 IP 核的处理器的项目代码目录
./verification_delayslot/	存放使用延迟槽、使用 IP 核的可上板验证的项目代码目录
./verification_nodelayslot/	存放无延迟槽、使用 IP 核的可上板验证的项目代码目录

在项目代码目录中，各文件功能说明如下：

ALU.v	ALU（EX）
ALUControl.v	ALU 控制单元（ID）
ByteProcess.v	为 lbu 设计，从读取的 word 中抽取 byte（WB）
Control.v	控制信号生成模块（ID）
CPU.v	处理器顶层模块，数据通路，将各子模块和寄存器进行连接
DataMemory.v	不使用 IP 核时采用的数据存储器（MEM）
DeviceControl.v	外设控制器（MEM）
ForwardingUnit.v	各转发单元
HazardUnit.v	各冒险检测及阻塞单元
ImmProcess.v	立即数处理模块（ID）

RegisterFile.v	寄存器堆（ID）
RegTemp.v	流水级间寄存器（可用 parameter 改变容量）
pipeline.xdc	处理器 STA 用时序约束
InstructionMemory_delayslot.v	指令存储器（使用延迟槽）（IF）
InstructionMemory_nodelayslot.v	指令存储器（无延迟槽）（IF）
test_cpu.v	仿真使用 testbench
Computer_delayslot.v Computer_nodelayslot.v	上板验证用顶层模块，含分频器
Computer.xdc	上板验证管脚约束
InstructionMemory_varification_delayslot.v InstructionMemory_varification_nodelayslot.v	指令存储器（上板验证用）（IF）

仿真验证情况

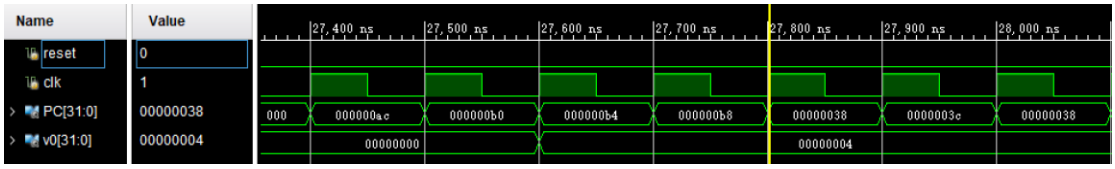
设计示例：模式串——ae，匹配字符串——eaeaeaeaea。结果应为 4，存在$v0 中。

（1）无延迟槽

使用 InstructionMemory_nodelayslot.v（包含字符串搜索算法本体部分，不包含后续上板验证所需的外设操作和循环延时相关程序，与 asm_bf_nodelayslot.asm 中的汇编程序及 Inst_bf_nodelayslot.txt 中的机器码程序对应）进行行为级仿真，注意 test_cpu.v 中 CPU 模块例化如下：

```
CPU #(.USE_DELAY_SLOT(0),.Inst_Num(46),.Inst_Num_BIT(8)) cpu(reset, clk, PC, v0);
```

波形及结果如下图：



对比可运行在 MARS 4.5（以下称 mars）上的 asm_bf_mars.asm 的运行情况：

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000001
$v0	2	0x00000004

可见结果一致，$v0=4。

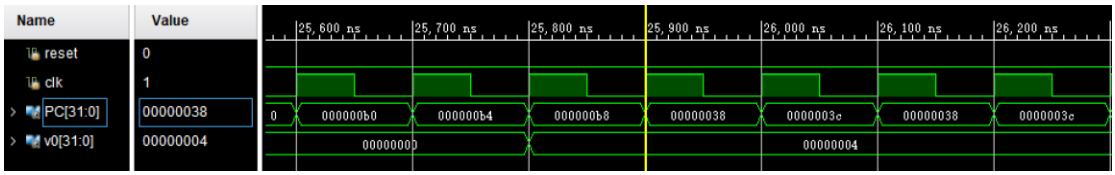
此外，由波形图可知，刚进入循环时（PC=38）仿真共执行 27800ns，换算后为 278 个周期。

（2）使用延迟槽

使用 InstructionMemory_delayslot.v（与不使用延迟槽的程序功能一致，但针对 Jump 指令的延迟槽进行了部分代码的顺序重排，与 asm_bf_delayslot.asm 中的汇编程序及 Inst_bf_delayslot.txt 中的机器码程序对应）进行行为级仿真，注意 test_cpu.v 中 CPU 模块例化如下：

```
CPU #(.USE_DELAY_SLOT(1),.Inst_Num(47),.Inst_Num_BIT(8)) cpu(reset, clk, PC, v0);
```

波形及结果如下图：

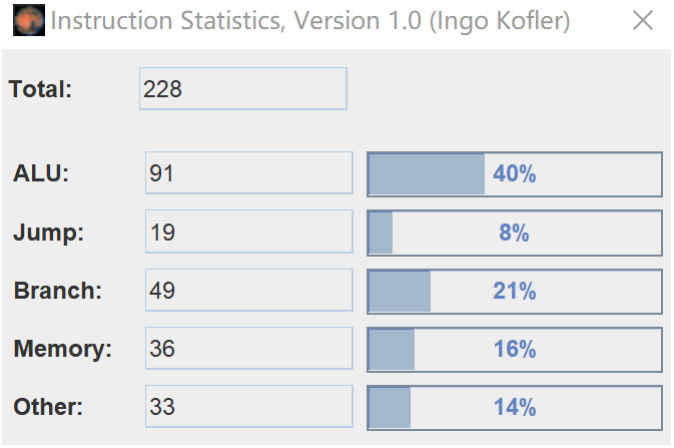


结果同样正确。

此外，由波形图可知，刚进入循环时（PC=38）仿真共执行 25900ns，换算后为 259 个周期。

（3）CPI 分析

由于本实验中数据存储地址从 0 起始，而 mars 中数据存储地址从 0x10010000 开始，故与前述指令对应的汇编程序 asm_bf_nodelayslot.asm 和 asm_bf_delayslot.asm 均不能在 mars 上正常运行。于是将数据地址前 16 位改为 0x1001 后存为可运行的 asm_bf_mars.asm。但这一修改使原来的 sw 指令需要 3 条指令才能完成（lui、addu、sw）（其余涉及前 16 位非零的地址的指令也是如此）。在 mars 运行程序，到进入循环时统计结果如下：



共 228 条指令，减去前述的额外多出来的指令（共 12 条）后，得到实验设计的处理器运行程序的总指令数 216。

故无延迟槽时$CPI = \frac{278}{216} = 1.287$，使用延迟槽时$CPI = \frac{259}{216} = 1.199$。

Implementation 情况

（1）资源占用

生成资源占用报告如下：（上图无延迟槽，下图有延迟槽）

Resource	Utilization	Available	Utilization %
LUT	1901	20800	9.14
FF	1631	41600	3.92
BRAM	0.50	50	1.00
IO	66	210	31.43

Resource	Utilization	Available	Utilization %
LUT	1894	20800	9.11
FF	1622	41600	3.90
BRAM	0.50	50	1.00
IO	66	210	31.43

由于使用了 IP 核，LUT 与 FF 的使用量极大减少，取而代之的是一部分 BRAM 的资源占用。
为了方便与单周期进行比较，使用./test_noip/中的代码构建项目并生成资源占用报告如下：

Resource	Utilization	Available	Utilization %
LUT	4492	20800	21.60
FF	9907	41600	23.81
IO	66	210	31.43

与先前实现的单周期处理器相比，单周期占用 4734 个 LUT，9248 个 FF，流水线使用了更少的 LUT 和更多的 FF。
（使用更多 FF 符合理论模型，流水线相比单周期插入了级间寄存器）

（2）时序性能

在 pipeline.xdc 中设置时序约束，create_clock 设定周期为 5.8ns。生成时序分析报告如下：
（上图无延迟槽，下图有延迟槽）

Statistics

Type	Worst Slack	Total Violation	Failing Endpoints	Total Endpoints
Setup	0.168 ns	0.000 ns	0	2503
Hold	0.102 ns	0.000 ns	0	2503
Pulse Width	2.400 ns	0.000 ns	0	1602

Statistics

Type	Worst Slack	Total Violation	Failing Endpoints	Total Endpoints
Setup	0.141 ns	0.000 ns	0	2485
Hold	0.134 ns	0.000 ns	0	2485
Pulse Width	2.400 ns	0.000 ns	0	1593

则无延迟槽时最短时钟周期为$T_{nd} = 5.800 - 0.168 = 5.632ns$，最高时钟频率为$f_{nd} = 177.56MHz$；使用延迟槽时最短时钟周期为$T_d = 5.800 - 0.141 = 5.659ns$，最高时钟频率为$f_d = 176.71MHz$。
由之前分析的 CPI，可知无延迟槽时平均单条指令执行时间为$t_{nd} = 1.287 \times 5.632 = 7.248ns$，使用延迟槽时平均单条指令执行时间为$t_d = 1.199 \times 5.659 = 6.785ns$。
与先前实现的单周期处理器相比，单周期平均单条指令执行时间为$6.868ns$，故使用延迟槽时流水线性能略优于单周期。

实际硬件验证情况

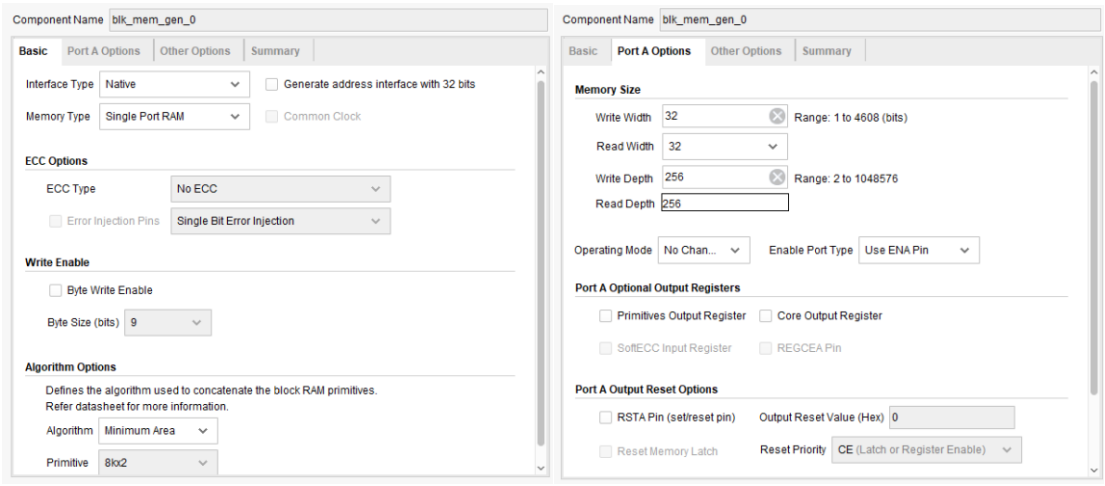
使用./verification_delayslot/和./verification_nodelayslot/中代码构建项目，分别对使用延迟槽和无延迟槽的处理器进行上板验证。实际结果均符合预期：完成搜索算法部分程序后，将 1 存入 0x4000000C，最左侧亮起 1 颗 LED；等待时间结束后，将 3 存入 0x4000000C，最左侧亮起 2 颗 LED，开始在数码管动态显示结果 0004。

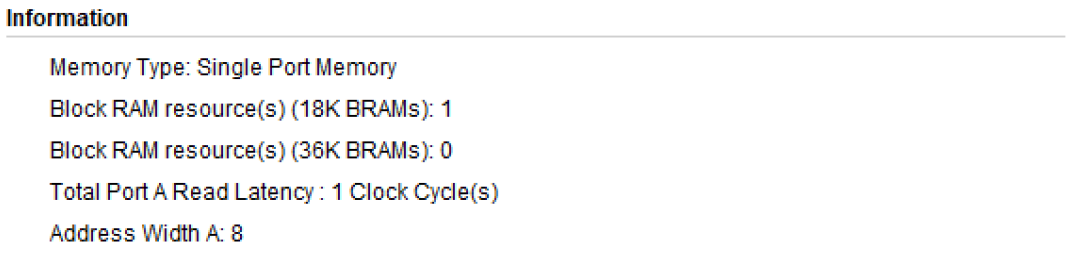
优化历程

（1）最初版本单纯根据数据通路图直接使用行为级描述，没有使用任何优化方法，综合实现后最高时钟频率只有 30MHz 左右。首先引入了静态分支预测（预测不分支）以及可选的延迟槽功能，降低了 CPI，但对时钟频率没有帮助。

（2）综合实现后的关键路径（起点为 EX/MEM 级间寄存器）显示，其 logic delay 其实很小（只有 1ns 左右），而 net delay 却有 20ns，占关键路径延时的绝大部分。该路径的 High Fanout 高达 1400，经查阅资料发现高扇出会拉低时序性能后，使用 max_fanout 对路径的起点（RegTemp）中的 reg 进行约束，并佐以手动的寄存器复制（降低扇出的另一种方法），同时调整数据通路使 MEM 阶段负载最小化（例如将取出字节或字的单元放到 WB 阶段而非 MEM 阶段）。经过一系列调整，High Fanout 降至 60，最高时钟频率上升至 70MHz。

（3）然而此时关键路径仍在 MEM 阶段，且 net delay 仍占延时的主要部分，约 90%。从此开始考虑数据存储器本身的问题，其描述中包含大量 reg 变量，使该部分所占面积偏大，导致布线困难。经查阅资料发现在 vivado 的 IP 核中 RAM 已有相应的实现——Block Memory Generator，其使用 BRAM 资源，使用它可以减少寄存器的使用。于是配置 IP 核如下：





使用 IP 核代替原本的 DataMemory 后，最高时钟频率上升至 90MHz。

（4）此时，在 xdc 的时序约束中 set-clock 的 period 值设置为 100ns。在与同学的交流讨论

中发现，可以通过减小该设置值使 vivado 综合实现工具尽力优化性能（如将层次化的设计打平，然后突破 HDL 的描述层级进行优化），以尽可能满足所设置的时序约束。受此启发，将 period 值不断调小多次综合实现，并佐以 max_fanout 的继续调整，最终将 period 调整为 5.8ns，最高时钟频率上升到现在的 170MHz 水平。（不过需要注意，打平层次化设计后，诸如 cpu.rf.RF_data[31]的写法就有可能出现问题，需要避免这种写法，转而将所需的信号写成 output）

参考资料

《数字逻辑与处理器基础》处理器部分课件与教材
高质量的 Verilog 描述方法-CSDN 博客
芯动力——硬件加速设计方法_西南交通大学_中国大学 MOOC
关于 Vivado 综合属性：MAX_FANOUT-CSDN 博客
vivado 常用 IP 调用配置——PLL、ROM、RAM-CSDN 博客
Vivado 中 Block Memory Generator v8.3 的使用-CSDN 博客
【正点原子 FPGA 连载】第十四章 IP 核之 RAM 实验-CSDN 博客