32 位 MIPS 流水线处理器加分申请报告

汤宸 无 62 2016011018
李云飞 无 62 2016011017
成大立 无 62 2016011029
2018 年 8 月 1 日

1 32 位 MIPS 流水线处理器概述

本次流水线 MIPS 处理器设计中，我们组设计的流水线经过 Vivado 时序分析，在 135MHz 的情况下，关键路径仍有 0.230ns 的建立时间余量。因此，实际时钟频率可以进一步达到 139.5MHz。实际使用串口收发助手进行验证时，输出结果正确，且未出现丢包、错包的情况，经老师验收通过。因此，我们组在此申请高频率流水线 MIPS 处理器加分。

2 32 位 MIPS 流水线处理器数据通路简介

我们组采用五级流水结构线结构设计，基本架构、各模块端口如下图所示。下图可能不够清晰，请参见附件中的“流水线框图.png”。

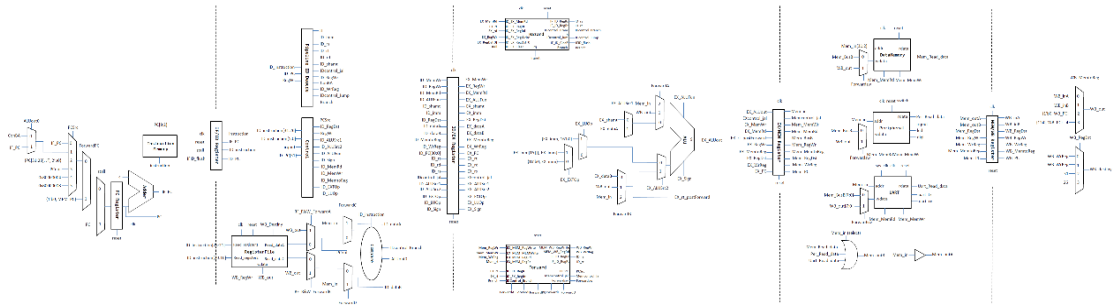

图 1 流水线 CPU 整体框图

五级流水线分别为 IF、ID、EX、MEM、WB，通过 PCSrc 及其它特殊控制信号来决定下一个周期的 PC 值，并读出与 PC 相对应的 instruction 信号一同传入 ID 段。除了对数据通路基本的划分之外，我们设计了冒险和转发单元，转发单元以尽量减少流水线的 stall 操作为目的，而冒险单元则是对必须的跳转等情况，对送入 ID 段的数据进行 flush 或 stall 操作等。按照要求，我们还设计了外设模块，读写 UART、leds、digi 等内容的模块。同时，我们设计了定时器中断处理模块，增加了保存断点进入 26 号寄存器的功能。

首先，我们简单介绍各个模块功能如下。

（1）pipeline.v：顶层模块，定义了各个模块间传递的变量，并把各个模块相连，以实现整体的流水线功能。

（2）pipeline_IF.v：在这里主要完成 PC 变化的操作，由 PCSrc 控制多路选择，正常情况下，PC 进行加四操作；stall 信号起作用时，PC 不变；中断处理入口、异常处理入口、分支目标地址、跳转目标地址等不再赘述。值得一提的是，我们发现当 jal 指令后紧跟着 jr $ra 指令时，会出现一次数据冒险，即从 31 号寄存器里读出来的值，是正确数据存进去之前的，因此我们添加了一条转发操作，进行 PC 的地址转发。

（3）InstructionMemory.v：通过 PC 读出对应的指令，准备送入 ID 段。

（4）IFID_reg.v：IF 与 ID 的段间寄存器，从 IF 段及 InstructionMemory 取出 ID 段需要用的 PC 值及指令，在正常情况下送入 ID 段。当 stall 控制信号处于高电平时，送入 ID 段的信号与上一周期保持一致，不进行变化，当 flush 信号处于高电平时，送出去的 PC 值与上一周期保持一致，而 instruction 置零。

（5）pipeline_ID.v：在这一段，我们对 instruction 进行了解析，生成了主要的控制信号、需要使用的寄存器编号、立即数、写入寄存器编号（如果需要当前指令需要写寄存器的话）；我们通过 RegisterFile.v 读出需要使用的寄存器内的值；我们对分支指令进行提前判断，并将相应生成的控制信号送入冒险单元及 IF 段，当发生 reg-beq 数据冒险时，通过转发来控制送入的数据；

（6）IDEX_reg.v：ID 与 EX 的段间寄存器，传递后续段需要用到的信号。

（7）pipeline_EX.v：这里主要进行 ALU 操作，ALU 部分已在实验报告中给出，这里不再赘述。由 ALUSrc 及 Forward 信号控制送入 ALU 的两条数据通路上的值。输出 ALU 的结果。值得一提的是，对于 sw 操作，虽然由 ALUSrc2 控制的多路选择选择了传递立即数，但是 rt 寄存器通路上的值在 MEM 阶段也需要用到，因此我们在这里额外添加了一个输出，以解决这个问题。

（8）EXMEM_reg.v：EX 与 MEM 间的段间寄存器，传递后续段需要用到的信号。

（9）pipeline_MEM.v：接入 DataMemory，Peripheral 及 UART 三个底层模块，进行数据存储器、外设数据的读写以及中断信号的生成等操作。通过多路选择器控制输出。

外设模块完成中断信号的生成，送入 ID 段及冒险单元来影响流水线。同时完成计时器功能，外设存储器写功能等。UART 模块负责 UART 的读写功能。

（10）MEMWB_reg.v：MEM 与 WB 的段间寄存器，传递 WB 段需要用的信号。

（11）pipeline_WB.v：进行写入数据、写入寄存器的选择。

（12）Forward_Unit.v：转发单元，涉及到四类转发。EX 阶段需要使用 MEM 阶段或者 WB 阶段正在传递的，还未写入寄存器堆的数据，先判断是否要从 MEM 阶段转发，再判断是否从 WB 阶段转发，以实现从最近的一条指令进行转发；提前判断分支的 reg-beq 转发；IF 段需要使用还未写入 31 号寄存器堆的 PC 值，这种情况带来的转发；load-use 型转发。

（13）Hazard_Unit.v：冒险单元，包括中断指令带来的冒险共有五类。跳转指令需要的对一行的 flush 操作；reg-beq 数据冒险，需要一次 stall；load-use 数据冒险，需要一次 stall；分支判断，需要进行一次 flush 置零指令；中断的到来，致使需要跳转到中断处理程序，这里需要一次 flush 操作。

如果不考虑 stall，以及在合理进行段间划分的情况下，五级 MIPS 流水线处理器的稳定速度是单周期处理器的五倍，一个处理器最多可以同时处理五条指令。而为了保证流水线的顺畅执行，转发模块、冒险检测模块以及 PC 的跳转都十分重要。下面我们就这些模块作出进一步分析。

（1）Forward 模块

如图所示，共有四种类型的 Forward 模块。ForwardA(B)，ForwardC(D)，ForwardPC 及 Forwardsw。

• 在 IF 段需要使用的有一种，通过 ForwardPC 来进行控制。当运行 jr $ra 指令时，若 31 号寄存器的值还没有更新成正确值，那么就需要将 PC 提前转发给 IF 段。当 ID 段解析出这是一个 jr 指令时，ID 段的周期内置 Memcontrol_jal 为 1 并一路传到 MEM 段，ForwardPC 在 PCSrc 等于 3（对应从 31 号寄存器读 PC），且 MEM 段的 Memcontrol_jal 为高电平时，开始起作用，从 MEM 段取出 PC 传入 IF 段。

• ID 段需要使用的有一种。在本段，由于需要提前判断分支，在发生 reg-beq 数据冒险时，要用到的数据值还未存入寄存器堆，这就需要从 MEM 阶段向 ID 阶段进行转发。当接到信号 IDControl_Branch 提示此时为分支指令，且正在传递的写入寄存器编号等于此时判断分支需要用的读出寄存器编号时，起作用并控制对数据进行转发。

```
always @(*) begin
  if (reset) begin
    ForwardA <= 2'b00;
    ForwardB <= 2'b00;
    ForwardC <= 0;
    ForwardD <= 0;
    Forwardsw <= 0;
    ForwardPC <= 0;
  end
  else begin
    if (EX_MEM_RegRd!=0 && EX_MEM_RegWrite && (EX_MEM_RegRd==ID_EX_RegRs)) ForwardA <= 2'b10;
    else if (MEM_WB_RegRd!=0 && MEM_WB_RegWrite && (MEM_WB_RegRd==ID_EX_RegRs)) ForwardA <= 2'b01;
    else ForwardA <= 2'b00;


    if (EX_MEM_RegRd!=0 && EX_MEM_RegWrite && (EX_MEM_RegRd==ID_EX_RegRt)) ForwardB <= 2'b10;
    else if (MEM_WB_RegRd!=0 && MEM_WB_RegWrite && (MEM_WB_RegRd==ID_EX_RegRt)) ForwardB <= 2'b01;
    else ForwardB <= 2'b00;

    ForwardC <= (IDControl_Branch && (EX_MEM_RegRd!=0)&&(EX_MEM_RegRd==IF_ID_RegRs));
    ForwardD <= (IDControl_Branch && (EX_MEM_RegRd!=0)&&(EX_MEM_RegRd==IF_ID_RegRt));
    ForwardPC <=(PCSrc==3 && Memcontrol_jal==1) ?1:0;
    Forwardsw <= EX_MEM_MEMWrite && MEM_WB_RegWrite && (EX_MEM_RegRt == MEM_WB_Reg);
  end
end

endmodule
```

图 2 转发模块的代码实现

• EX 段需要使用的有一种。在本段，输入到 ALU 的两条数据通路上的值，都存在数据冒险，需要进行判断。我们采取先判断就近的一条指令，再判断上上条指令的方法，当写入寄存器不为零，且与本 EX 段两条输入数据通路的寄存器编号相同时，forward 起作用，对参与 EX 段数据通路的多路选择，如下图所示。

```
assign Data1=(ForwardA==2'b0) ? EX_BusA :
             (ForwardA==2'b1) ? MEMWBdata :
             (ForwardA==2'b10) ? EXMEMdata : 32'b0;
assign Data2=(ForwardB==2'b0 || EX_ALUSrc2) ? EX_BusB :
             (ForwardB==2'b1) ? MEMWBdata :
             (ForwardB==2'b10) ? EXMEMdata : 32'b0;
```

图 3 EX 阶段使用的转发信号

值得一提的是，当指令为 sw 时，即会有 rt 寄存器的编号，立即数也会传来一个值并送入 ALU，此时对立即数不能转发，因此上图的 Data2 比之 Data1 略有不同。

• MEM 阶段有一种转发情况。当 MEM 阶段需要写寄存器，且写入数据的寄存器编号与上一条要写入寄存器堆的寄存器编号相同时，起作用。

这四条是最初的设计，但当我们的寄存器堆部分出问题后（问题已在实验报告中描述），就多了一条从 WB 阶段向 ID 段的转发，即当寄存器堆不能先写后读时，相隔两条指令也有可能存在数据冒险。这条转发只要稍作修改即可，并没有在 Forward 模块进行改动。

（2）Hazard 模块

在冒险检测单元中，主要存在两种冲突处理形式，一种是 flush，使 ID 段 PC 保持不变，

且清零 instruction（instruction 置零时，是对零号寄存器的值左移零格，相当于清空指令）；另一种是 stall，使 ID 段的 PC 和 instruction 保持不变，重复执行上一条指令的内容，并将送入 EX 段后面的写入使能归零，让上一行的执行无效，这样就相当于阻塞了一个周期。上图的 ID_EX_Clear 代表 stall 信号，IF_ID_clear 代表 flush 信号。

```
assign ID_EX_Clear=(reset | (IDcontrol_Jump)) ? 0:
(ID_EX_MemRd & ( (ID_EX_RegRt==IF_ID_RegRs) || (ID_EX_RegRt==IF_ID_RegRt) ) ) ? 1: //load-use
(Branch  & (ID_EX_RegWrite && (//reg-branch
          ((IF_ID_RegRs==ID_EX_RegRt) &&  ID_EX_RegDst_0) ||
          ((IF_ID_RegRs==ID_EX_RegRd) && ~ID_EX_RegDst_0) ||
          ((IF_ID_RegRt==ID_EX_RegRt) &&  ID_EX_RegDst_0) ||
          ((IF_ID_RegRt==ID_EX_RegRd) && ~ID_EX_RegDst_0) )
          ) ) ? 1:0;

assign IF_ID_Clear_temp=(reset | (ID_EX_MemRd && ( (ID_EX_RegRt==IF_ID_RegRs) || (ID_EX_RegRt==IF_ID_RegRt) ) )) ? 0:
(IDcontrol_Jump) ? 1://jump
( (IDcontrol_Branch) & ~(ID_EX_RegWrite && (//branch
          ((IF_ID_RegRs==ID_EX_RegRt) &&  ID_EX_RegDst_0) ||
          ((IF_ID_RegRs==ID_EX_RegRd) && ~ID_EX_RegDst_0) ||
          ((IF_ID_RegRt==ID_EX_RegRt) &&  ID_EX_RegDst_0) ||
          ((IF_ID_RegRt==ID_EX_RegRd) && ~ID_EX_RegDst_0) )
          ) ) ? 1:0;

assign cur_irq = irq;

assign irq_flush = (cur_irq && ~pre_irq);

assign IF_ID_Clear = IF_ID_Clear_temp || irq_flush;

always @(posedge clk or posedge reset) begin
  if(reset)  begin
    pre_irq <= 0;
  end
  else begin
    pre_irq <= cur_irq;
  end
end

endmodule
```

图 4 冒险检测模块的代码实现

• 需要 stall 的情况有两种，主要是数据冒险。一是 load-use 数据冒险，另一个是 reg-beq 数据冒险。都需要 stall 一个周期来等待转发。

• 需要 flush 的情况有三种，首先是跳转指令，需要对下一条指令进行清空；其次是分支指令，首先判断这不是一个 reg-beq 数据冒险，如果是的话需要先 stall 一个周期等待转发重新判断分支，然后得到分支判断成功的控制指令时，清空下一条指令；最后是中断操作，虽然中断进行的操作本质也是一个 jal 指令，但中断信号和表明这条是跳转指令的控制信号不一样，在我们的设计中，中断信号会持续高电平一段时间，那么为了让它只 flush 一次，我们使用了 irq 的现态和次态进行预判，只有当前一个周期中断信号为 0，本周期为 1 时，才需要进行一次 flush。

（3）PC 多路选择

PC 的正确选择对流水线处理器的正常运转至关重要，这里需要强调两点，一是 PC 高位置零，这在实验报告中详细阐述，这里就不过多赘述；第二点是 ForwardPC，由于 ForwardPC 的生成过程中就要求了 PCSrc 为 3，所以在这里的判断就没有加上 PCSrc。

```
assign PC_choose= (PCSrc==4) ?  32'h80000004:
                  stall? PC:
                  (PCSrc==0) ?  PC+4 :
                  (PCSrc==1 & ALUOut0==0) ?  ConBA: ,
                  (PCSrc==2) ?  {PC[31:28],JT,2'b0}:
                  (ForwardPC==1) ? {1'b0, MEM_PC}:
                  (PCSrc==3) ?  ID_BusA: //$ra
                  (PCSrc==5) ?  32'h80000008: PC+4;
```

图 5 PC 跳转地址的代码实现

3 时序分析与关键路径分析

在综合过程中，我们发现，受到 Vivado 算法的影响，在不同的时序约束下，甚至在不同的计算机上，关键路径都有所不同。在此，我们首先选取了在优化过程中出现最频繁的一条关进路径进行分析。注意为了方便起见，这里的时钟频率仅设置为 50MHz。

Summary

Name	Path 1
Slack	6.108ns
Source	reg_MEMWB/WB_MemtoReg_reg[1]/C (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@10.000ns period=20.000ns})
Destination	reg_EXMEM/Mem_in_reg[0]/D (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@10.000ns period=20.000ns})
Path Group	CLK
Path Type	Setup (Max at Slow Process Corner)
Requirement	20.000ns (CLK rise@20.000ns - CLK rise@0.000ns)
Data Path Delay	13.758ns (logic 5.397ns (39.227%) route 8.361ns (60.773%))
Logic Levels	18 (CARRY4=8 LUT3=1 LUT4=4 LUT5=2 LUT6=3)
Clock Path Skew	-0.130ns
Clock Un...rtainty	0.035ns

Data Path

Delay Type	Incr (ns)	Path (...	Location	Netlist Resource(s)
FDCE (Prop_fdce_C_Q)	(r) 0.456	5.574	Site: SLICE_X55Y49	reg_MEMWB/WB_MemtoReg_reg[1]/Q
net (fo=50, routed)	1.592	7.166		reg_MEMWB/WB_MemtoReg[1]
LUT4 (Prop_lut4_I1_O)	(r) 0.124	7.290	Site: SLICE_X45Y51	reg_MEMWB/RF_data[31][5]_i_6/O
net (fo=1, routed)	0.000	7.290		reg_MEMWB/RF_data[31][5]_i_6_n_0
CARRY4 (Prop_c...4_S[1]_CO[3])	(r) 0.550	7.840	Site: SLICE_X45Y51	reg_MEMWB/RF_data_reg[31][5]_i_2/CO[3]
net (fo=1, routed)	0.000	7.840		reg_MEMWB/RF_data_reg[31][5]_i_2_n_0
CARRY4 (Prop_carry4_CI_CO[3])	(r) 0.114	7.954	Site: SLICE_X45Y52	reg_MEMWB/RF_data_reg[31][9]_i_2/CO[3]
net (fo=1, routed)	0.000	7.954		reg_MEMWB/RF_data_reg[31][9]_i_2_n_0
CARRY4 (Prop_carry4_CI_O[3])	(f) 0.313	8.267	Site: SLICE_X45Y53	reg_MEMWB/RF_data_reg[31][13]_i_2/O[3]
net (fo=2, routed)	1.334	9.602		reg_MEMWB/WB_pipeline/WB_out1[13]
LUT5 (Prop_lut5_I0_O)	(f) 0.334	9.936	Site: SLICE_X49Y55	reg_MEMWB/RF_data[31][13]_i_1/O
net (fo=36, routed)	0.622	10.558		reg_EXMEM/WB_out[13]
LUT6 (Prop_lut6_I4_O)	(r) 0.332	10.890	Site: SLICE_X50Y56	reg_EXMEM/added_carry__2_i_12/O
net (fo=15, routed)	0.000	10.890		reg_EXMEM/EX_pipeline/p_0_in[13]

CARRY4 (Prop_c...4_S[0]_CO[3])	(r) 0.513	11.403	Site: SLICE_X50Y56	reg_EXMEM/added_carry__2_i_10/CO[3]
net (fo=1, routed)	0.000	11.403		reg_EXMEM/added_carry__2_i_10_n_0
CARRY4 (Prop_carry4_CI_O[0])	(r) 0.219	11.622	Site: SLICE_X50Y57	reg_EXMEM/added_carry__3_i_10/O[0]
net (fo=1, routed)	0.623	12.245		reg_EXMEM/EX_pipeline/b0[17]
LUT4 (Prop_lut4_I3_O)	(r) 0.295	12.540	Site: SLICE_X51Y57	reg_EXMEM/added_carry__3_i_7/O
net (fo=1, routed)	0.000	12.540		EX_pipeline/ALUs/adds/EX_ALUFun_reg[0]_2[1]
CARRY4 (Prop_c...4_S[1]_CO[3])	(r) 0.550	13.090	Site: SLICE_X51Y57	EX_pipeline/ALUs/adds/added_carry__3/CO[3]
net (fo=1, routed)	0.000	13.090		EX_pipeline/ALUs/adds/added_carry__3_n_0
CARRY4 (Prop_carry4_CI_CO[3])	(r) 0.114	13.204	Site: SLICE_X51Y58	EX_pipeline/ALUs/adds/added_carry__4/CO[3]
net (fo=1, routed)	0.000	13.204		EX_pipeline/ALUs/adds/added_carry__4_n_0
CARRY4 (Prop_carry4_CI_O[1])	(r) 0.334	13.538	Site: SLICE_X51Y59	EX_pipeline/ALUs/adds/added_carry__5/O[1]
net (fo=2, routed)	0.820	14.358		EX_pipeline/ALUs/adds/Mem_in_reg[27][1]
LUT4 (Prop_lut4_I0_O)	(r) 0.303	14.661	Site: SLICE_X52Y59	EX_pipeline/ALUs/adds/Mem_in[0]_i_24/O
net (fo=1, routed)	0.490	15.151		EX_pipeline/ALUs/adds/Mem_in[0]_i_24_n_0
LUT5 (Prop_lut5_I4_O)	(f) 0.124	15.275	Site: SLICE_X52Y60	EX_pipeline/ALUs/adds/Mem_in[0]_i_20/O
net (fo=1, routed)	0.951	16.226		EX_pipeline/ALUs/adds/Mem_in[0]_i_20_n_0
LUT4 (Prop_lut4_I3_O)	(r) 0.124	16.350	Site: SLICE_X52Y57	EX_pipeline/ALUs/adds/Mem_in[0]_i_14/O
net (fo=1, routed)	0.404	16.755		reg_IDEX/Z1
LUT6 (Prop_lut6_I5_O)	(r) 0.124	16.879	Site: SLICE_X53Y58	reg_IDEX/Mem_in[0]_i_7/O
net (fo=1, routed)	1.228	18.107		reg_IDEX/Mem_in[0]_i_7_n_0
LUT3 (Prop_lut3_I0_O)	(r) 0.146	18.253	Site: SLICE_X56Y53	reg_IDEX/Mem_in[0]_i_3/O
net (fo=1, routed)	0.296	18.549		reg_IDEX/Mem_in[0]_i_3_n_0
LUT6 (Prop_lut6_I1_O)	(r) 0.328	18.877	Site: SLICE_X57Y54	reg_IDEX/Mem_in[0]_i_1/O
net (fo=1, routed)	0.000	18.877		reg_EXMEM/EX_ALUFun_reg[4][0]
FDCE			Site: SLICE_X57Y54	reg_EXMEM/Mem_in_reg[0]/D
Arrival Time		18.877		

图 6 一条在综合过程中曾多次出现的关键路径

简单观察上述关键路径，可以看到上述路径的起始点位于 MEM/WB 寄存器内部；这条路径经过了由 MemtoReg 决定 WB_out 的操作之后，来到了 EX 阶段，并且进入了 ALU 内部；最后抵达了 EX/MEM 寄存器当中。虽然表面上看起来这条路径曾经经历过 ID/EX 寄存器，但我们发现，它经历的只是 ID/EX_reg 中的 LUT，而没有经过 FDCE；使用 Schematic->Netlist->Go to Source 操作后，发现 ID/EX 仍然对应 ALU 内部的源代码。（Vivado 似乎会把其他模块的功能放置在一些段间寄存器当中。）很明显，这是一条转发路径，是 ForwardA 和 ForwardB 对应的转发。在这条路径上，对延时贡献较大的包括导线延时（高扇出、路程长），以及 CARRY4 的串行延时。

在我们最终提交的代码版本中，关键路径如下图所示。这条路径是从 IF/ID 段间寄存器的 instruction 到 PC 的通路，中间经过了 RegisterFile，MEM/WB 段间寄存器，IF/ID 段间寄存器等通路。根据前述经验，虽然显示了 MEM/WB 和 IF/ID 寄存器，这条路径应该没有进入寄存器内部。（关键路径的首尾一定是寄存器，其中间不可能有别的寄存器！）既然这条路径从取指令开始，中间访问了寄存器堆，最后回到了 PC，这有可能是一条类似于 jr 指令的

数据通路。既然是正常指令的数据通路，我们认为，对这条路径进行进一步优化的余地可能不是很大。

另外值得注意的是，这条路径中的逻辑延时只占 30%，导线延时占到了 70%，因此在我们的处理器设计中，影响时序的更关键的因素或许是导线延时。

Name	Path 1
Slack	0.230ns
Source	reg_IFID/ID_instruction_reg[16]_rep__0/C (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@3.700ns period=7.400ns})
Destination	reg_IFID/ID_PC_reg[25]/CE (rising edge-triggered cell FDCE clocked by CLK {rise@0.000ns fall@3.700ns period=7.400ns})
Path Group	CLK
Path Type	Setup (Max at Slow Process Corner)
Requirement	7.400ns (CLK rise@7.400ns - CLK rise@0.000ns)
Data Path Delay	6.888ns (logic 2.097ns (30.443%) route 4.791ns (69.557%))
Logic Levels	11 (CARRY4=1 LUT5=1 LUT6=7 MUXF7=1 MUXF8=1)
Clock Path Skew	-0.097ns
Clock Un...rtainty	0.035ns

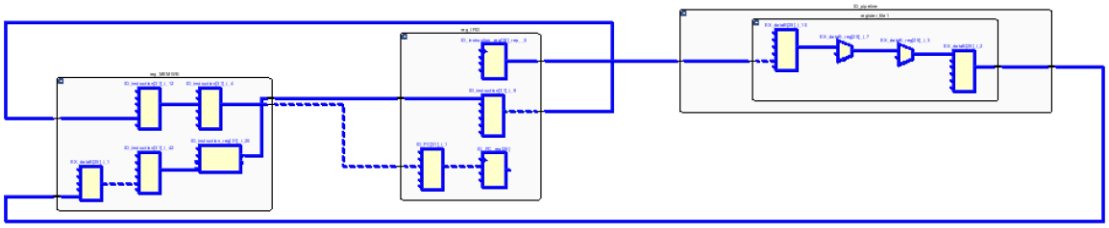

图 7 135MHz 流水线处理器的关键路径示意图

4 时序性能优化策略

为了尽可能提高流水线 CPU 的时钟主频，我们采取了一系列的优化措施，在此逐一介绍。

4.1 以空间换时间

在实验报告中我们已经看到，CPU 占用的硬件资源仅占到了 FPGA 总资源的 10%上下，因此硬件资源是非常充分的。如果我们对时序性能有更高的要求，可以考虑牺牲空间资源。这种策略有以下两个体现。

ALU 中移位的串行与并行的处理。按照实验指导书的要求，ALU 中的移位操作由移位 1，2，4，8，16 等几种操作拼合而成，因此理论上，完成一次移位至多需要 5 次串行操作（例如左移 31 位）。我们在分析关键路径的过程中，曾经发现关键路径中含有连续的 5 个 CARRY4，每个 CARRY4 的逻辑延时都相对比较长，它们串联起来造成了很大的延时。这 5 个 CARRY4 都在 EX_pipeline 阶段，并且使用 Go to Source 命令后，发现这些 CARRY4 都指向 Shift 模块，由此可以确定，ALU 当中移位的串行可能带来较长的延时。我们使用查找

表的方式，将每一种移位的结果都手动列写出来，希望可以解决这一问题；但在后续调试过程中发现，关键路径不再经过这几个 CARRY4 了，因此也就没有再做修改。但令人费解的是，CARRY4 本质上应该是 4 位并行的超前进位加法器，为什么移位操作（本质上就是导线的合并）需要用到 CARRY4 呢？或许也有可能是为了将 shamt 操作数分解为 1，2，4，8，16 的组合；还有一种可能，即移位操作和普通的加法操作复用了同一批 CARRY4，因此我们看到的关键路径，可能不仅仅是 Shift 模块导致的，也可能是普通的 32 位加法引起的延时。

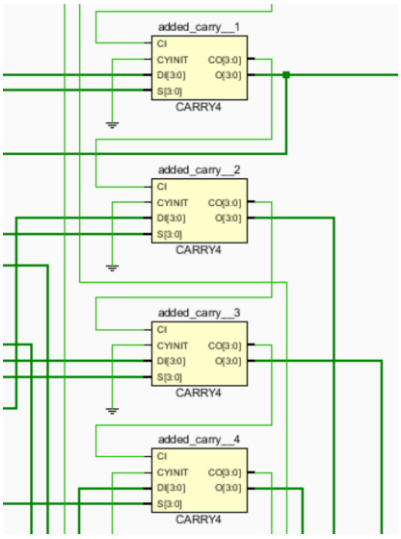

图 8 ALU 内部的串行的 CARRY4 加法器

对于$zero 零号寄存器的处理。在助教提供的 RegisterFile 版本中，我们看到，寄存器堆只有 31 个 32 位寄存器，0 号寄存器是不存在的。RF 的输出端有一个 MUX，如果源寄存器地址为 0，那么 RF 的输出由 MUX 选通为 32’b0。同样是在关键路径分析过程中，我们发现了这个 MUX 的存在。这个 MUX 的摆放位置距离其他硬件很远，带来了较大的导线延时。我们于是删掉了这个 MUX，在 RF 中实实在在地添加了一个 0 号寄存器，保证它的取值始终为 0。经过这次操作，建立时间余量增加了大约 0.1ns。

```
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[31:0];
    assign Read_data1 = RF_data[Read_register1];
    assign Read_data2 = RF_data[Read_register2];
    integer i;
    always @(posedge clk or posedge reset) begin
        if (reset)
            for (i = 0; i < 32; i = i + 1)
                if (i!=29) RF_data[i] <= 32'h00000000;
                else RF_data[29] <= 32'h0000_0080;
        else if (RegWrite && (Write_register != 5'b00000))
            RF_data[Write_register] <= Write_data;
    end
endmodule
```

图 9 加上零号寄存器后的寄存器堆（如图中橙色框所示）

4.2 想方设法减小导线延时

在实验报告中我们看到，关键路径的总延时中，只有大约 30%来自逻辑延时，剩下 70%都是导线延时（net delay）。因此，花大力气优化逻辑延时，不如想办法降低导线延时。我们认为，导线延时太大，主要是两种原因导致的：扇出太大或导线太长。

扇出系数（fanout）表征了某一元件驱动负载的多少。如果并联负载太多，下一级的输入电容就会增加，导致延时增长。整个设计中最大的扇出是全局时钟，总计 2000 余个触发器都需要时钟来驱动。但一方面，全局时钟使用了名叫 BUFG 的模块，这种缓冲器能有效提高扇出能力和时钟的稳定性；另一方面，源地点的时钟和目的地的时钟都要受到大扇出的干扰。因此，全局时钟的处理不在我们的优化范围内。我们仅尝试对某些扇出较大（大约 50 以上）的寄存器做了(*MAX_FANOUT*)规定，但从结果看来，没有明显优化效果，因此我们放弃了这种操作。

FDCE (Prop_fdce_C_Q)	(r) 0.518	5.630	Site: SLICE_X42Y45	reg_IFID/ID_instruction_reg[22]/Q
net (fo=263, routed)	1.152	6.782		ID_pipeline/register_file1/Q[6]

图 10 一个典型的高扇出的例子，导线延时长达 1.15ns，或许是由于扇出系数高达 263

导线太长也会引起导线延时过大。在 Device 图中，经常能看到关键路径跨越了很多个子区域。但这是 Vivado 的自动布线和绑定的结果，很难从源代码层级上修改。好在 Vivado 比较智能，能够根据时序要求调整布线策略。当我们缩短约束文件中的时钟周期时，可以明显看到，Vivado 把更多的使用到的硬件资源安排到了更近的位置，使得导线延时大大缩短，使得建立时间余量往往比我们预期的更大。至于在时序要求不严格的情况下，Vivado 会将硬件资源分散在 FPGA 芯片各处，我们认为，这样做可能更有利于芯片散热，进而提高硬件电路工作的稳定性。

4.3 优化代码风格

我们在后续调试过程中，发现了一些冗余代码和冗余变量。这些冗余主要包括两部分：一是有用但无连接的变量，二是无用但有连接的变量。

有用但无连接的变量主要是为了程序的完备性。例如，PC 在 InstructionMemory 中寻址时，低两位总是 00，并不需要作为索引。如果强行把 PC 的 32 位全部都接入 InstructionMemory，Vivado 会产生 warning，提示我们 PC[0]和 PC[1]是 unconnected port。我们认为这些有用但未连接的管脚，或多或少总会对时序产生影响，因此将取消了所有不必要的变量的连接。

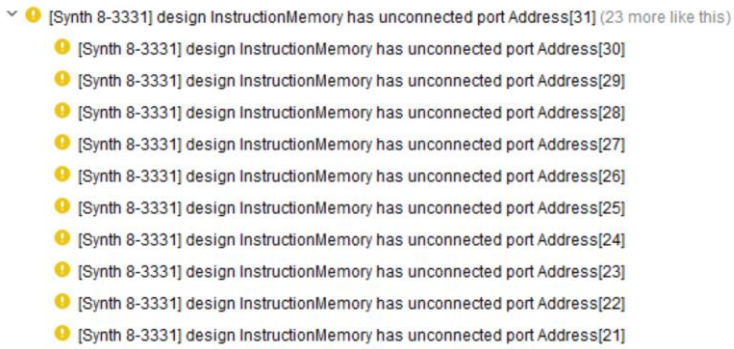

图 11 未连接的端口会引发 Vivado 的警告

无用但有连接的变量主要是代码风格的问题，例如 input A; output B; wire C; assign C = A; assign B = C;一类的代码，或者 assign notA = A ? 0 : 1;一类的不够简洁的代码。在第一个例子中，中间变量 C 必然引入一个缓冲器；在第二个例子中，一个非门可以完成的任务（而且往往 D 触发器就有反相输出端）由一个 MUX（甚至是 LUT2）来完成。这些冗余代码不仅造成了硬件资源的浪费，还造成了延时的增加。因此这些无用的变量也是要删除的。

```
assign ALUout0=(IDcontrol_Branch) ? 0:1;
```

图 12 一行可以进一步优化风格的代码

事实上，完成 CPU 的编写之后，通过和其他小组的交流，我们发现，初始编写时的设计，很可能比后期优化更为重要。我们的流水线 CPU 没有经过优化时就可以在 100MHz 时钟上运行了，但其他一些小组的流水线 CPU 即使经过了优化，也达不到 100MHz；还有一些小组，没有经过优化都可以达到 170MHz 或 180MHz。虽然不是很理解为什么差异如此巨大，但我们认为，一般而言，后续优化只能提升 30MHz 左右，不能指望通过修修补补实现时钟频率的巨大飞跃。因此初期的编写设计可能更为本质。流水线的各个阶段之间应该尽量平均，避免工作量分配不均匀；开发初期应该对整体需求有明确的认识、对整体架构有详细的规划，避免后期调试时给一个漏洞百出的数据通路打补丁；可能这些前期编写中的注意事项，反倒决定了最终 CPU 主频的基调。