流水线 MIPS 处理器的设计

一、实验目的

将之前设计的 MIPS 处理器改进为流水线结构，重点在实现外设、解决数据关联冒险并利用此处理器求解字符串搜索问题

二、设计方案（原理说明及框图）

1） 控制信号设置：

PCSrc [1:0]：00 PC+4
01 指令是分支类型，由 BranchCtrl 决定 PC 更新值来源
10 jal 和 j 地址
11 jr 地址(从寄存器读)
BranchType[2:0]：000 不是分支指令
001 beq
010 bne
011 blez
100 bgtz
101 bltz
Branch：0 不是分支指令或分支不成立
1 分支成立
RegWrite：0 不写入寄存器
1 写入寄存器
RegDst [1:0]：00 rt
01 rd jalr 也用这个
10 ra jal 指令用
MemRead：0 不读 MEM
1 读 MEM
MemWrite：0 不写 MEM
1 写 MEM
MemtoReg [1:0]：00 ALU 运算结果
01 MEM 读取数据
10 PC+4，用于 jalr（写入 rd）和 jal（写入 ra）
ALUSrc1：0 rs
1 移位指令的无符号扩展 Shamt
ALUSrc2：0 rt
1 Imm
ExtOp：0 无符号
1 有符号扩展
LuiOp：0 不是 Lui
1 是 Lui，要把 16 位立即数放入目标寄存器 rt 高 16 位
LbOp：0 不是 Lb 指令

1 是 Lb 指令，根据地址后两位决定写回数据

Flush：0 不清除指令

1 清除 IF 阶段的指令

Stall：0 不暂停前面指令运行

1 本周期保持 IFID 寄存器和 IDEX 寄存器的指令，同时 PC 不更新

ID_ForwardA[1:0]：00 rs 数据

01 EX 阶段转发数据

10 MEM 阶段转发数据

11 WB 阶段转发数据

ID_ForwardB[1:0]：00 rt 数据

01 EX 阶段转发数据

10 MEM 阶段转发数据

11 WB 阶段转发数据

EX_ForwardA[1:0]：00 RF 读取数据

01 移位指令的无符号扩展 Shamt

10 MEM 阶段转发数据

11 WB 阶段转发数据

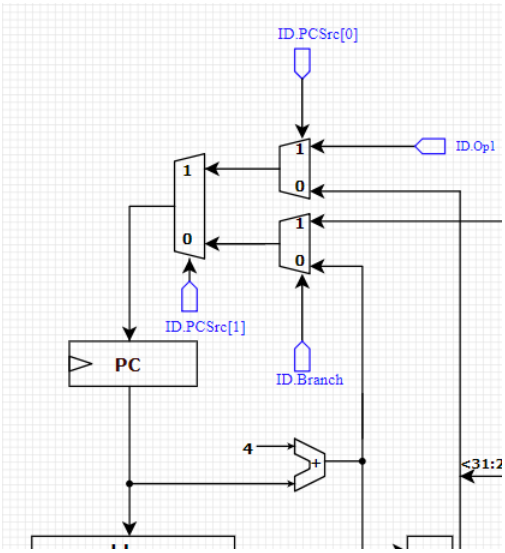
EX_ForwardB[1:0]：00 RF 读取数据

10 MEM 阶段转发数据

11 WB 阶段转发数据

2） 设计原理

```
assign IF_PCplus4=PC_o+32'd4;
assign
PC_i=(ID_PCSrc[1]==1'b0)?((ID_Branch==1'b1)?ID_PCplus4+(ID_ExtOut<<2):I
F_PCplus4):((ID_PCSrc[0]==1'b1)?ID_Op1:{ID_PCplus4[31:28],{ID_rs,ID_rt,
ID_rd,ID_Shamt,ID_Funct}<<2});
```



对应于原理图中的 PC 更新部分，PC 寄存器的输入来源由 ID 阶段的 PCSrc[1:0]和 Branch 信号决定

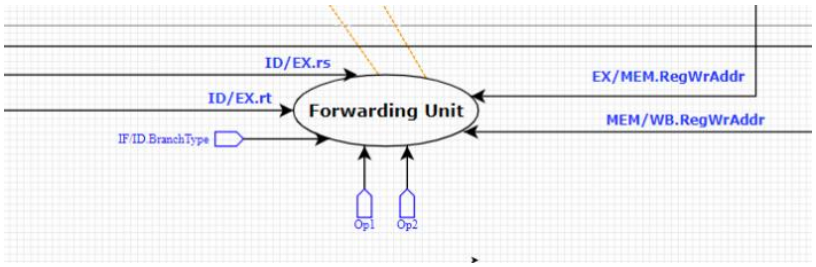
```
assign IF_Flush=(ID_Branch&~Stall||ID_PCSrc==2'b10||ID_PCSrc==2'b11)?
1:0;
```

仅当 ID 阶段的指令是j、jal，jr，$jalr$或 ID 阶段判断分支成立时 flush 掉 IF 阶段的指令

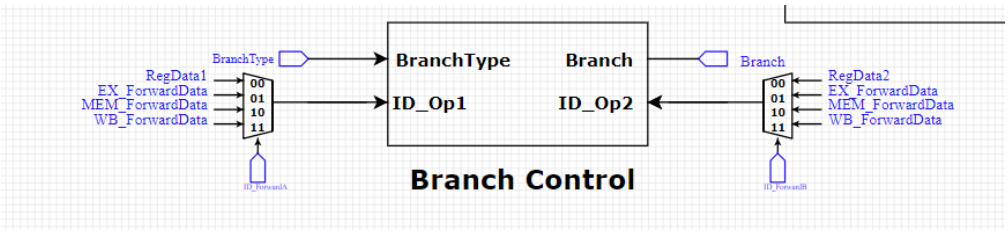
```
always@(*)
begin
  if((BranchType!=3'b000||ID_PCSrc==2'b11)&&EX_Rw!=0&&ID_rs==EX_Rw&&EX_RegWrite)
     ID_ForwardA=2'b01;
  else if((BranchType!=3'b000||ID_PCSrc==2'b11)&&MEM_Rw!=0&&ID_rs==MEM_Rw&&MEM_RegWrite)
     ID_ForwardA=2'b10;
  else if((BranchType!=3'b000||ID_PCSrc==2'b11)&&WB_Rw!=0&&ID_rs==WB_Rw&&WB_RegWrite)
     ID_ForwardA=2'b11;
  else
     ID_ForwardA=2'b00;

  if(BranchType!=3'b000&&EX_Rw!=0&&ID_rt==EX_Rw&&EX_RegWrite)
     ID_ForwardB=2'b01;
  else if(BranchType!=3'b000&&MEM_Rw!=0&&ID_rt==MEM_Rw&&MEM_RegWrite)
     ID_ForwardB=2'b10;
  else if(BranchType!=3'b000&&WB_Rw!=0&&ID_rt==WB_Rw&&WB_RegWrite)
     ID_ForwardB=2'b11;
  else
     ID_ForwardB=2'b00;

 if (MEM_RegWrite&&MEM_Rw!=0&&MEM_Rw==EX_rs)
          EX_ForwardA=2'b10;
  else if (WB_RegWrite&&WB_Rw!=0&&WB_Rw==EX_rs)
     EX_ForwardA=2'b11;
  else
     EX_ForwardA={1'b0,EX_ALUSrc1};
if (MEM_RegWrite&&MEM_Rw!=0&&MEM_Rw==EX_rt)
          EX_ForwardB=2'b10;
else if (WB_RegWrite&&WB_Rw!=0&&WB_Rw==EX_rt)
     EX_ForwardB=2'b11;
  else
     EX_ForwardB=2'b00;
```

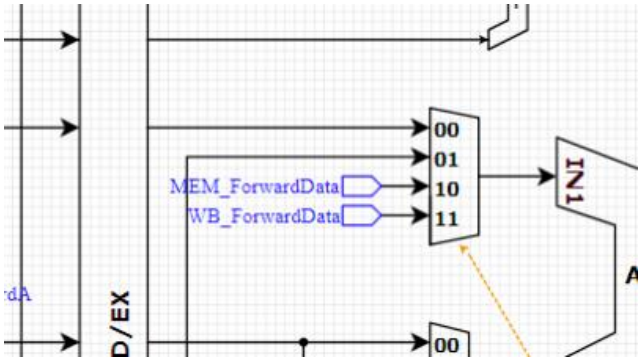


对应于图中的转发单元，ID 阶段 rs 的转发条件：ID 阶段的指令是分支指令或者$jr/jalr$，EX/MEM/WB 阶段的写回寄存器是 rs 且不是 0 号寄存器，写回使能信号有效；由 ifelse 语句顺序保证了 EX 转发优先于 MEM 转发优先于 WB 转发，此转发单元转发的数据供 BranchCtrl 模块使用，rs 和 rt 的转发区别仅在指令是$jr/jalr$需不需要转发。实际上，存入级间寄存器的数据转发只需要 WB 阶段的数据来代替 RF 先写后存，当前周期 EX 和 MEM 阶段的数据即使不转发在下一个周期的 EX 阶段也会被转发回来，因此代码在非分支指令的 ID 阶段转发只判断 WB 阶段是否需要转发。



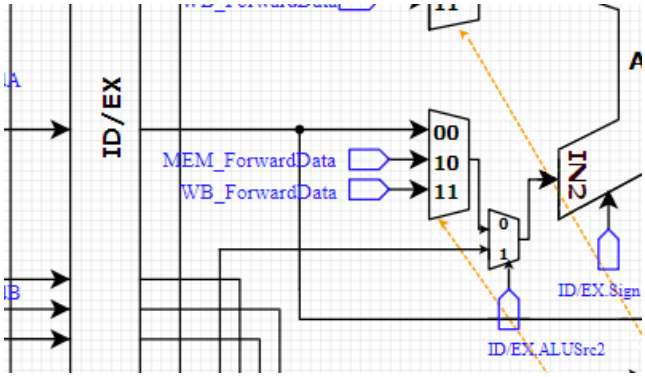
使用 ID 阶段转发数据的 BranchCtrl 模块

EX 阶段 rs 的转发条件和 ID 阶段类似，转发来源只有 MEM 和 WB 阶段，当两个来源都不需要转发时，说明 ALU 输入 1 要么是上一级读出的 rs 数据，要么是上一级扩展的 Shamt，因此用 1'b0 和 ALUSrc1 拼成 EX_ForwardA。



EX 阶段控制 IN1 的 MUX，控制信号是 EX_ForwardA

EX 阶段 ALU 输入 2 由于包括扩展后的立即数，而立即数不需要转发，因此分出 ALUSrc2 控制输入是 ExtImm 或者上一级寄存器数据/转发数据，其他和 rs 的转发基本一致。

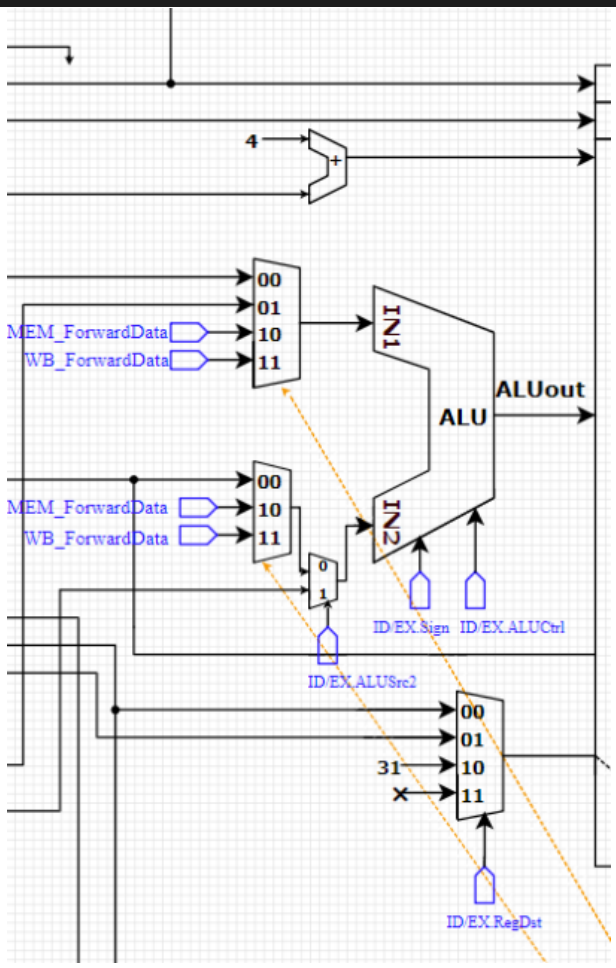


EX 阶段控制 IN2 的两个 MUX，第一级控制信号是 EX_ForwardB,第二级控制信号是 ALUSrc2

```
//ID
assign ID_ReadData1=(WB_Rw!=0&&ID_rs==WB_Rw&&WB_RegWrite)?WB_WriteData:RegData1;
assign ID_ReadData2=(WB_Rw!=0&&ID_rt==WB_Rw&&WB_RegWrite)?WB_WriteData:RegData2;
assign ID_Op1=(ID_ForwardA[1]==1'b1)?((ID_ForwardA[0]==1'b1)?WB_WriteData:MEM_ForwardData):((ID_ForwardA[0]==1'b1)?EX_ALUOut:ID_ReadData1);
assign ID_Op2=(ID_ForwardB[1]==1'b1)?((ID_ForwardB[0]==1'b1)?WB_WriteData:MEM_ForwardData):((ID_ForwardB[0]==1'b1)?EX_ALUOut:ID_ReadData2);
assign ID_ShamtOut={27'b0,ID_Shamt};
assign ID_ExtOut=(ID_LuiOp==1'b1)?{ID_rd,ID_Shamt,ID_Funct,16'b0}:((ID_ExtOp==1'b1)?{{16{ID_rd[4]}},ID_rd,ID_Shamt,ID_Funct}:{16'b0,ID_rd,ID_Shamt,ID_Funct});
```

如上分析 ID 阶段转发时所说，传入级间寄存器的 ID_ReadData1 和 ID_ReadData2 只判断 WB 是否需要转发，而供 BranchCtrl 使用的 ID_Op1 和 ID_Op2 由 ID_ForwardA，ID_ForwardB 控制四个数据来源；位移量 Shamt 使用无符号扩展，拼接上 27 比特 0 即可；ExtImm 需要根据是不是 Lui 指令和是否符号扩展决定输出。

```
//EX
assign EX_In1=(EX_ForwardA[1]==1'b0)?((EX_ForwardA[0]==1'b0)?EX_Op1:EX_ShamtOut):((EX_ForwardA[0]==1'b0)? MEM_ForwardData:WB_WriteData);
assign EX_In2=(EX_ALUSrc2==1'b1)?EX_ExtOut:((EX_ForwardB[1]==1'b1)?((EX_ForwardB[0]==1'b0)? MEM_ForwardData:WB_WriteData):EX_Op2);
assign EX_Rw=(EX_RegDst[1]==1'b0)?((EX_RegDst[0]==1'b0)?EX_rt:EX_rd):32'd31;
assign EX_PCplus8=EX_PCplus4+32'd4;
```



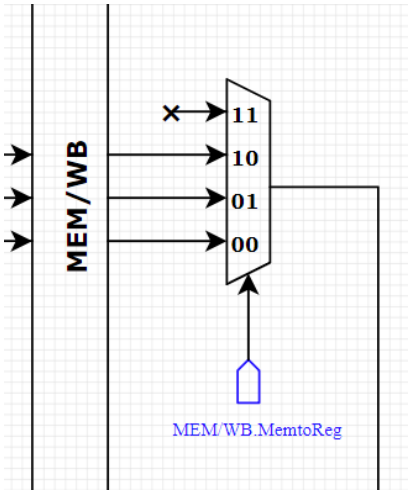
上述代码实现了如图的 ALU 输入选择和 EX 阶段 Rw 选择。

```
assign
MEM_MEMData=(MEM_LbOp==1'b1)?((MEM_ALUOut[1]==1'b1)?((MEM_ALUOut[0]==1'
b1)?{24'b0,MEM_ReadData[31:24]}:{24'b0,MEM_ReadData[23:16]}):((MEM_ALUO
ut[0]==1'b1)?{24'b0,MEM_ReadData[15:8]}:{24'b0,MEM_ReadData[7:0]})):MEM
_ReadData;
assign
MEM_ForwardData=(MEM_MemtoReg[1]==1'b0)?((MEM_MemtoReg[0]==1'b0)?MEM_AL
UOut:MEM_MEMData):MEM_PCplus8;
```

在 DM 外实现 lb 指令，根据输入 DM 地址的后两位判断输出读取 DM 数据的哪 1/4；同时 MEM 阶段可供转发的有三组数据，上一周期 ALU 输出结果、这周期读取的 DM 数据和 MEM 阶段指令的 PC+8，具体转发哪个数据需要根据 MemtoReg 决定。

```
assign
WB_MEMData=(WB_LbOp==1'b1)?((WB_ALUOut[1]==1'b1)?((WB_ALUOut[0]==1'b1)?
{24'b0,WB_ReadData[31:24]}:{24'b0,WB_ReadData[23:16]}):((WB_ALUOut[0]==
1'b1)?{24'b0,WB_ReadData[15:8]}:{24'b0,WB_ReadData[7:0]})):WB_ReadData;
```

```
assign
WB_WriteData=(WB_MemtoReg[1]==1'b0)?((WB_MemtoReg[0]==1'b0)?WB_ALUOut:W
B_MEMData):WB_PCplus8;
```



上述代码实现了 lb 指令和写回数据的选择。

```
if(EX_MemRead&&(EX_Rw==ID_rs||EX_Rw==ID_rt))
  begin Stall=1'b1;
  end
//else if((Branchtype!=3'b000||ID_PCSrc==2'b11)&&EX_Rw==ID_rs&&EX_RegWrite)
//  begin Stall=1'b1;
//  end
//else if(WB_RegWrite&&(WB_Rw==ID_rs||WB_Rw==ID_rt))
    //     begin Stall=1'b1;
    //     end

else begin
  Stall=1'b0;
end

end

endmodule
```

冒险检测模块，可以检测 MEMData 相关的先写后读冒险，注释掉的分支和 jr、jalr 的数据冒险被转发单元解决，不能先写后读也不需要 Stall 一个周期，只要引旁路即可解决。

级间寄存器只是普通的寄存器，只是 IFID 和 IDEX 寄存器可以由 Stall 控制数据“停止流动”一个时钟周期；其他元件沿用数逻大作业单周期处理器设计，不再赘述。

3)：框图

见附件 schematic.pdf。

三、关键代码及文件清单

代码清单：

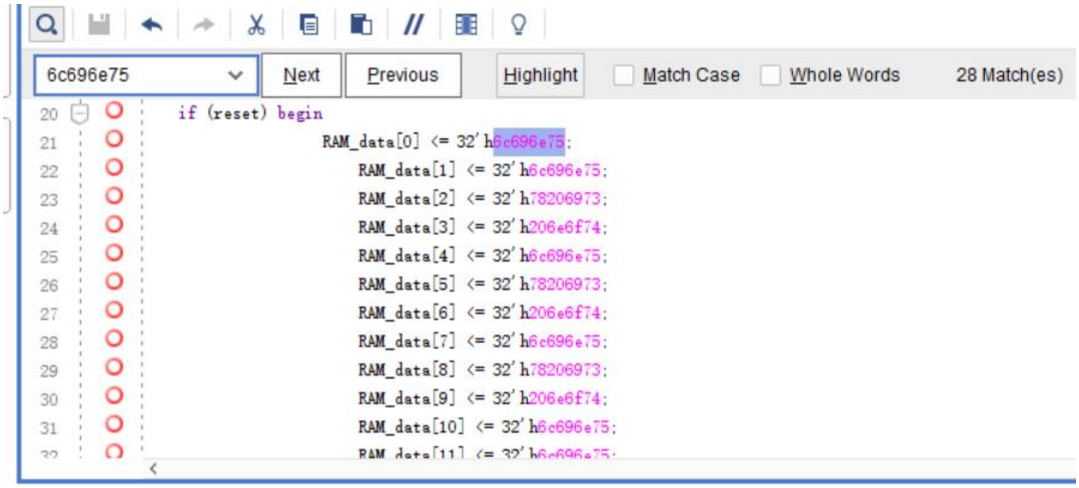
CPU_pipeline.v	顶层文件，连接各个模块
clk_gen.v	分频模块，时序分析和仿真时没有添加该模块
PC.v	程序计数器
InstructionMemory.v	指令储存器，存有指令的机器码
IF_IDreg.v	IF 和 ID 阶段的级间寄存器

RegisterFile.v	寄存器堆
Controller.v	控制信号产生模块
BranchCtrl.v	在 ID 阶段判断是否产生分支的模块
HazardUnit.v	判断是否需要 Stall 一个周期的模块
ForwardingUnit.v	输出 ID，EX 阶段控制转发的信号
ID_EXreg.v	ID 和 EX 阶段的级间寄存器
ALU.v	ALU 模块
ALUCtrl.v	输入 Controller 的 ALUOp，输出 ALU 控制信号
EX_MEMreg.v	EX 和 MEM 阶段的级间寄存器
DataMemory.v	存有 4 字节模式串，和 249 字节测试数据
MEM_WBreg.v	ID 和 EX 阶段的级间寄存器
CPU_pipeline.xdc	约束文件，为了在后仿时监视寄存器输出除七段数码管还有其他管脚绑定
INSTRUCTION.asm	汇编指令
test_cpu.v	测试用 testbench

文件清单：

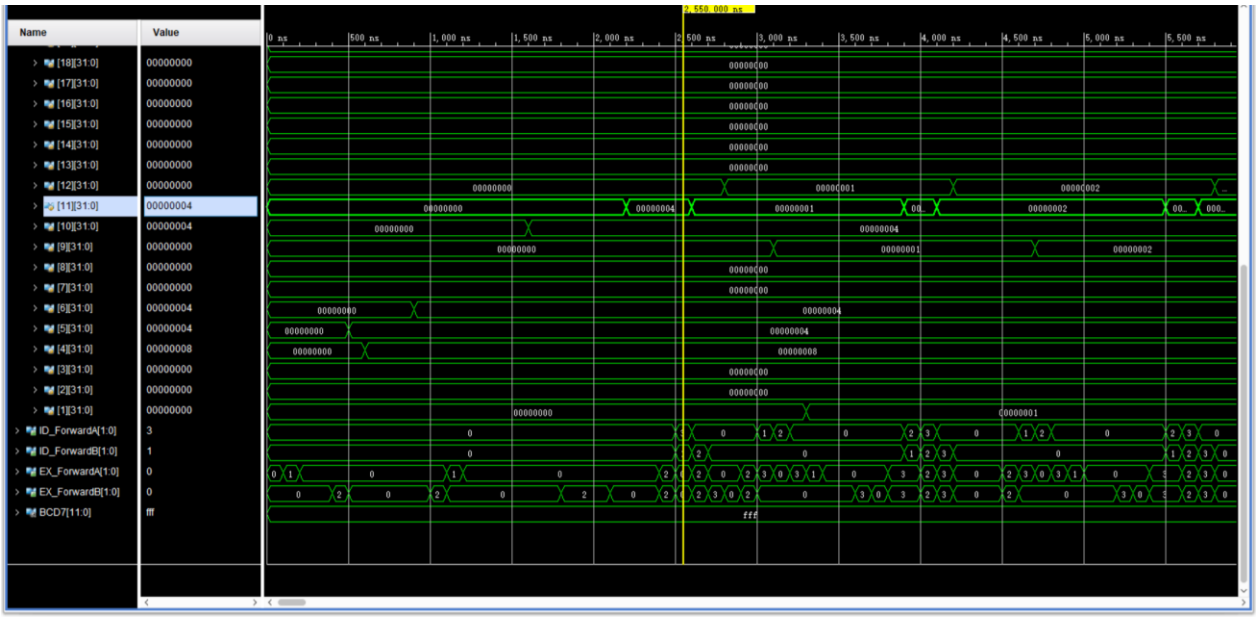
数据通路.pdf	流水线 CPU 数据通路
schematic.pdf	原理图
指令集.pdf	此流水线指令的格式以及相应指令 Opcode 和 Funct 的设置

四、仿真结果及分析

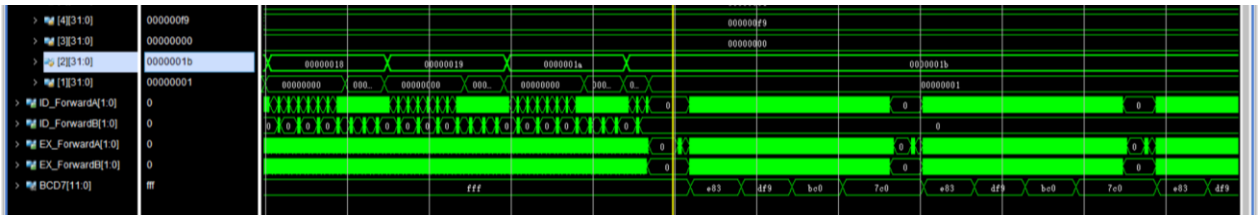


字符串中共 27 个匹配项（数据设计时不存在会在两个 RAM_data 数据首位拼接成模式串，因此直接搜索得到匹配模式串的数目，而且 bf 算法外层循环每次移动一字节，这样的数据设计并不影响 CPU 功能的验证）。因此理论上程序执行完成后$v0$的值为 **0x0000001b**，BCD7 应该在 **0xe83(**最低位显示 b**)，0xdf9(**次低位显示 b**)，0xbc0(**次高位显示 0**)，0x7c0(**最高位显示 0**)**之间循环变化。

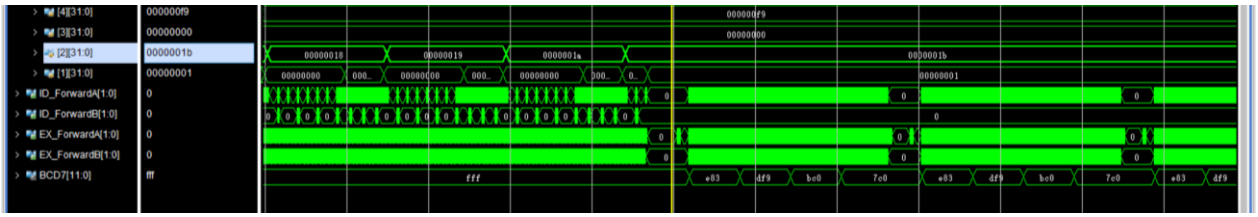
1）前仿：



如图，第一次分支成立发生在找到第一个模式串后$j = len_pattern$时，转发单元也在正常工作，ID_ForwardA=2'b11 说明某个寄存器在这个周期同时存在读和写，由于 RF 不能先写后读从旁路转发要写入的寄存器数据供下一级使用。



bf 算法执行结束后$v0$的值为 **0x0000001b，**与理论一致。



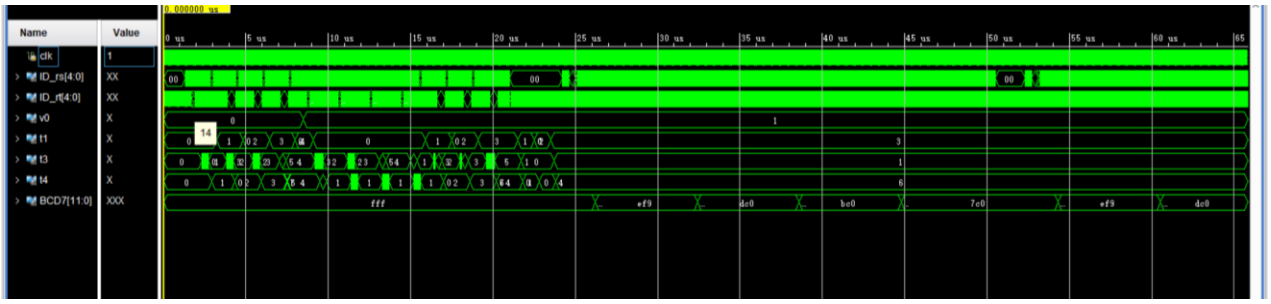
七段数码管对应数据，在 **0xe83，0xdf9，0xbc0，0x7c0** 之间循环变化，与理论预测一致。

2）后仿：

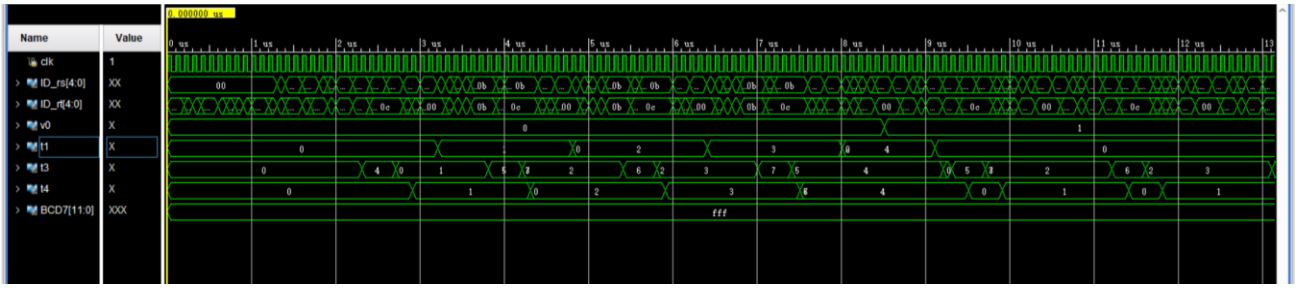
```
RAM_data[0] <= 32'h04030201;
RAM_data[1] <= 32'h04030201;
RAM_data[2] <= 32'h05030201;
```

为了实现后仿真分析字符串识别部分和显示部分的简便，暂时将模式串长度改为 4，字符串长度改为 8，具体数据如上图，其中 RAM_data[0]是模式串，同时更改 IM 中指令对应的字符串长度，理论上仿真结束后$v0应该存放 **0x00000001**，BCD7 在 **0xef9，0xdc0，

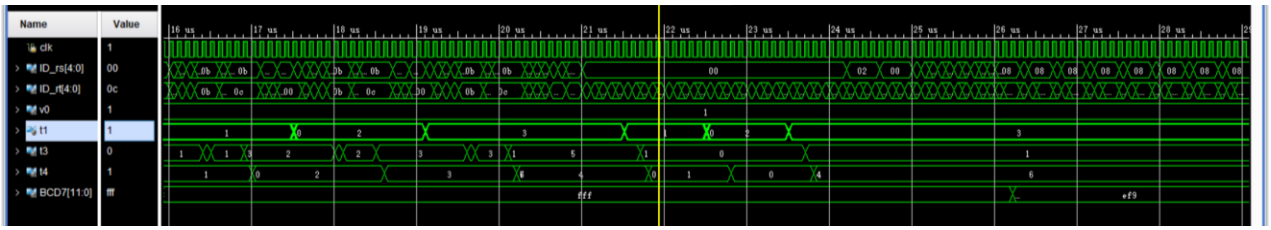
0xbc0，0x7c0 之间循环变化，其中高 4bit 分别对应 AN3，AN2，AN1，AN0，7 段数码管从右向左显示对应使能信号是$(\mathbf{1110})_2$，$(\mathbf{1101})_2$，$(\mathbf{1011})_2$，$(\mathbf{0111})_2$，对应十六进制的 e，d，b 和 7；f9 对应数码管显示 1，c0 对应显示 0。



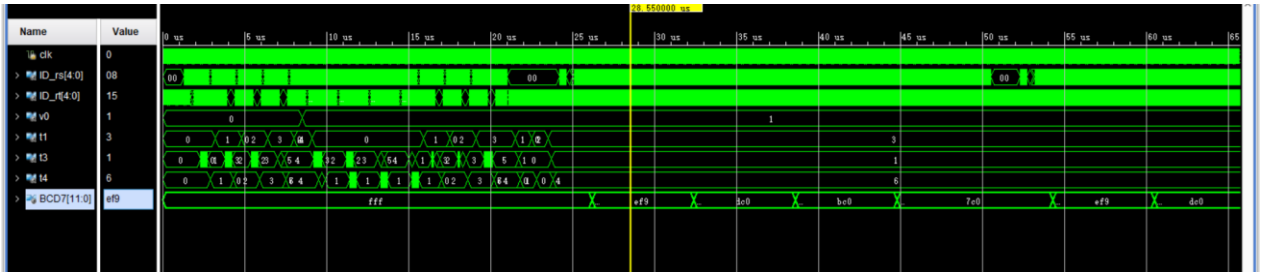
整体波形图，输出观察$t1, t3, t4, v0$四个寄存器以及七段数码管 BCD7，主要分析内层j的循环和最后 BCD7 的输出结果



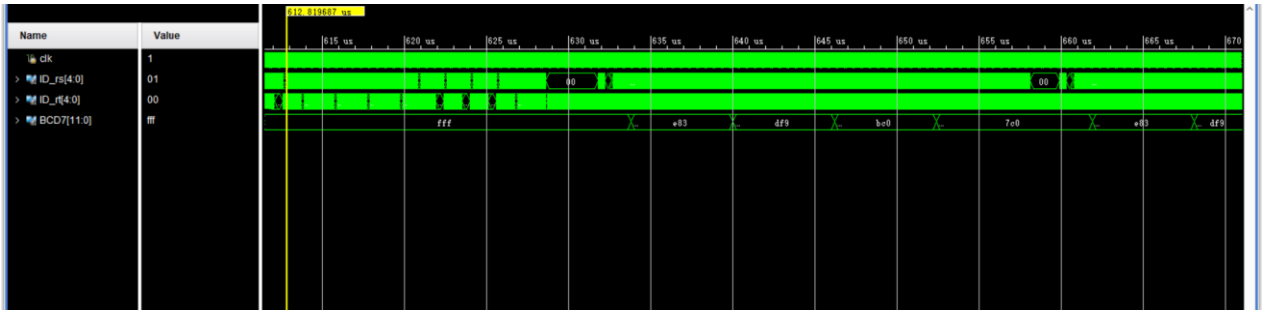
$i = 0$的第一次循环，$t1$从 0 变化到 1，2，3，4，$t3$读取字符串，$t4$读取模式串,由于字符串和模式串前四位是相同的，j（$t1$）从 0 一直增加，至$j < len_pattern$不成立（$j = 4$）时跳出此次i循环，识别到一次模式串，储存计数结果的$v0$从 0 增加到 1,之后进行$i = 1$，2，3，4的循环，$i = 1$，2，3 读取的字符和模式串首位就不相同，因此j在复位到 0 后没有增加到 1，$i = 4$时字符串后四位是 **0x01，0x02，0x03，0x05**，j从 0 增加到 3，由于模式串第四位 **0x04≠0x05**，j循环跳出，之后i自加到 5，不满足$i <= len_str - len_pattern$条件跳出关于$i$的外层循环，字符串识别部分结束。



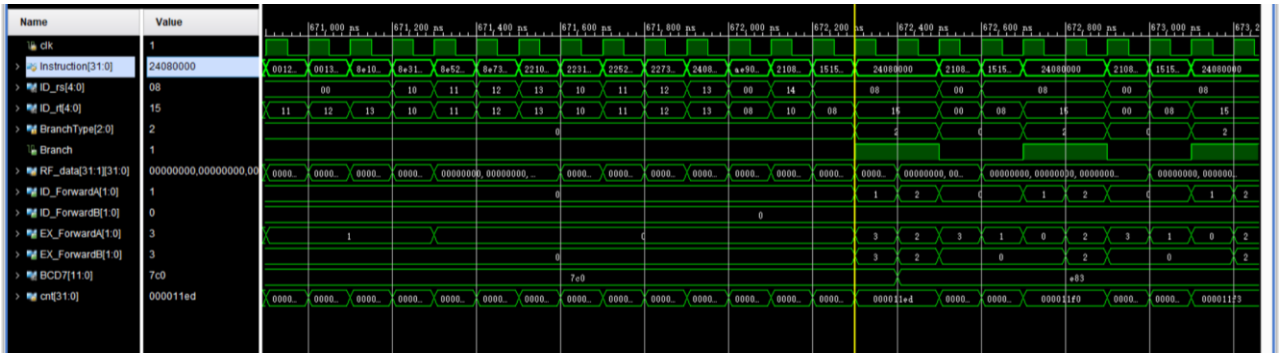
下一部分指令用于将$v0$的值转换成 16 进制数字显示在七段数码管上，此时$t1$，$t2$，$t3$，$t4$用于将七段数码管对应的 8 位控制信号存入 DataMemory 对应位置，监视的$t1$负责存入 **0x1，0x6，0xb** 的 8 位控制信号，即 **0x92，0x82，0x83** 由于监视信号只看后三比特，对应在仿真结果上是图中的 1，2，3，与理论一致。



最后看七段数码管对应数据，在 **0xef9，0xdc0，0xbc0，0x7c0** 之间循环变化，与理论预测一致。



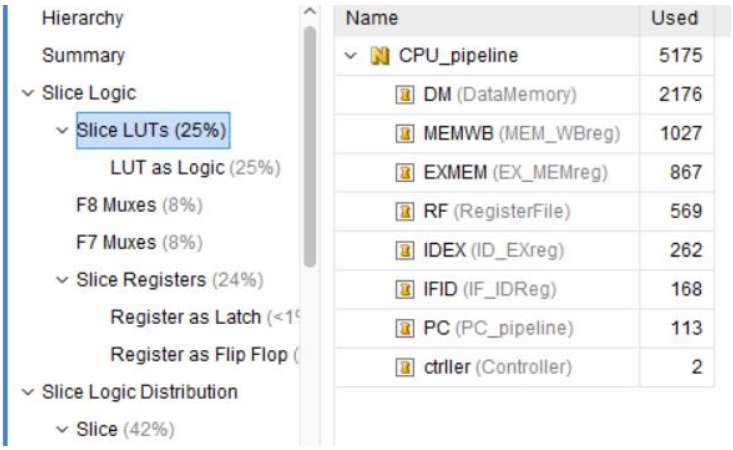
将 DataMemory 和 IM 数据改为正常值后，再次进行后仿，指令执行完成后 BCD7 的结果和前仿以及理论分析一致，可以验证 CPU 设计和指令设计的正确性。



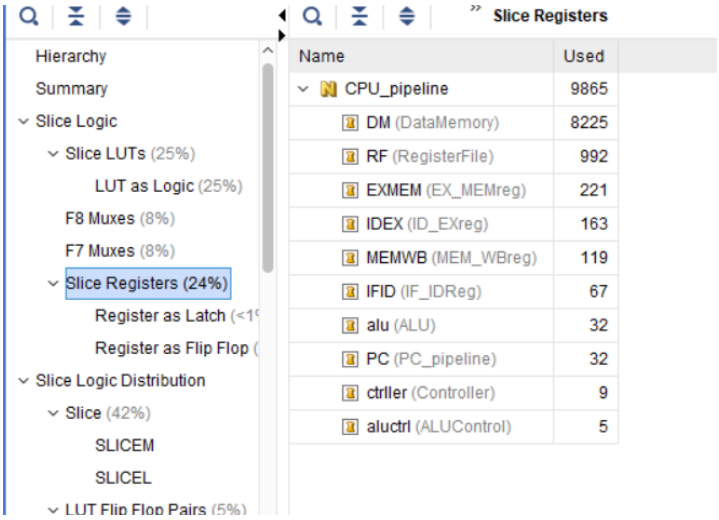
由前仿结果，从程序开始运行至底部 j 指令执行用时$672.3us$，仿真设置时钟周期$T = 100ns$，时钟周期数$C = \frac{672.3u}{100n} = 6723$，在 PC 中加入计数器，由于不存在完全相同的相邻两条指令，也就是 PC 每一次更新(非 Stall 情况)都是一次指令执行，共执行指令 N=**0x000011ed=(4589)**$_{10}$，因此$CPI = \frac{C}{N} = \frac{6723}{4589} = 1.465$，由于指令中分支指令占比较大，需要 stall 的情况较多，因此 CPI 偏离 1 也较大。

五、综合情况（面积和时序性能）

1）面积



CPU 占用 5175 个查找表



CPU 占用 9865 个寄存器

2）时序性能：


```
31
32  create_clock -period 10.000 -name CLK -waveform {0.000 5.000} [get_ports clk]
33
```

为了让 vivado 尽可能优化数据通路，约束文件时钟周期设置为 10ns，因此最小时钟周期$= 10n - (-3.466)n = 13.466ns$,最高时钟频率$f_{max} = \frac{1}{13.466n} = 74.261MHz$

六、硬件调试情况

由于主频偏离 100M 较远，上板子时使用了分频器，调整输入 CPU 的时钟周

期为 10kHz，按下复位键后经过一段时间板子仍能显示正确结果。

七、思想体会

完成整个实验的过程中 debug 是最困难的一部分，我遇到的 bug 是 bf 算法输出结果一直是 0，分析循环过程，发现内层循环永远只执行一次。

```
25    lb $t3,0($t3) # str[i+j]
26    lb $t4,0($t4) # pattern[j]
27
28    bne $t3,$t4,exit_j
29    addi $t1,$t1,1 # j+=1
30    blt $t1,$a2,loop_j # j<len_pattern
31    exit_j:
```

把$t3$，$t4$加入仿真波形窗，发现问题出在分支指令在存在数据冒险时的转发有问题，整个汇编指令只有这处分支的数据冒险明显影响程序结果。

```
timer1:
addi $t0,$t0,1
bne $t0,$s5,timer1
```

末尾显示部分的 bne 语句也存在数据冒险，但是由于这只是个计数循环，数据冒险也会发生但唯一的后果是这一位显示的时钟周期数+1，很难察觉。为了解决这个 bug，我跑了很多次后仿，我的体会是：跑后仿之前代码一定要先仔细检查一遍，确保没有问题再进行 synthesis 和 implementation，前仿可以不跑综合直接进行，跑的也快，随手改代码很方便；但是后仿要依赖综合的网格，有时发现有个小问题改了代码就要从综合重新跑起，综合跑完还要进行缓慢的后仿，时间成本巨大；同时后仿虽然很慢，但是有些上板子会发生的 bug 只能在后仿体现出来，前仿并不会出问题，比如上面说的分支指令数据冒险导致结果错误在前仿并不会发生，只有在后仿和上板子时会突然出现，因此我认为后仿对 debug 用处非常大，可以很方便的监视信号变化，观察这些信号往往

就会发现问题所在。