实验题目：夏季小学期实验32位MIPS处理器设计

班级：无65班

学号：2016011109

姓名：王春禹

日期：2018年7月20日

其他组员：无65班 孙天宇 2016011108

无65班 潘岩松 2016011110

## 一．实验任务

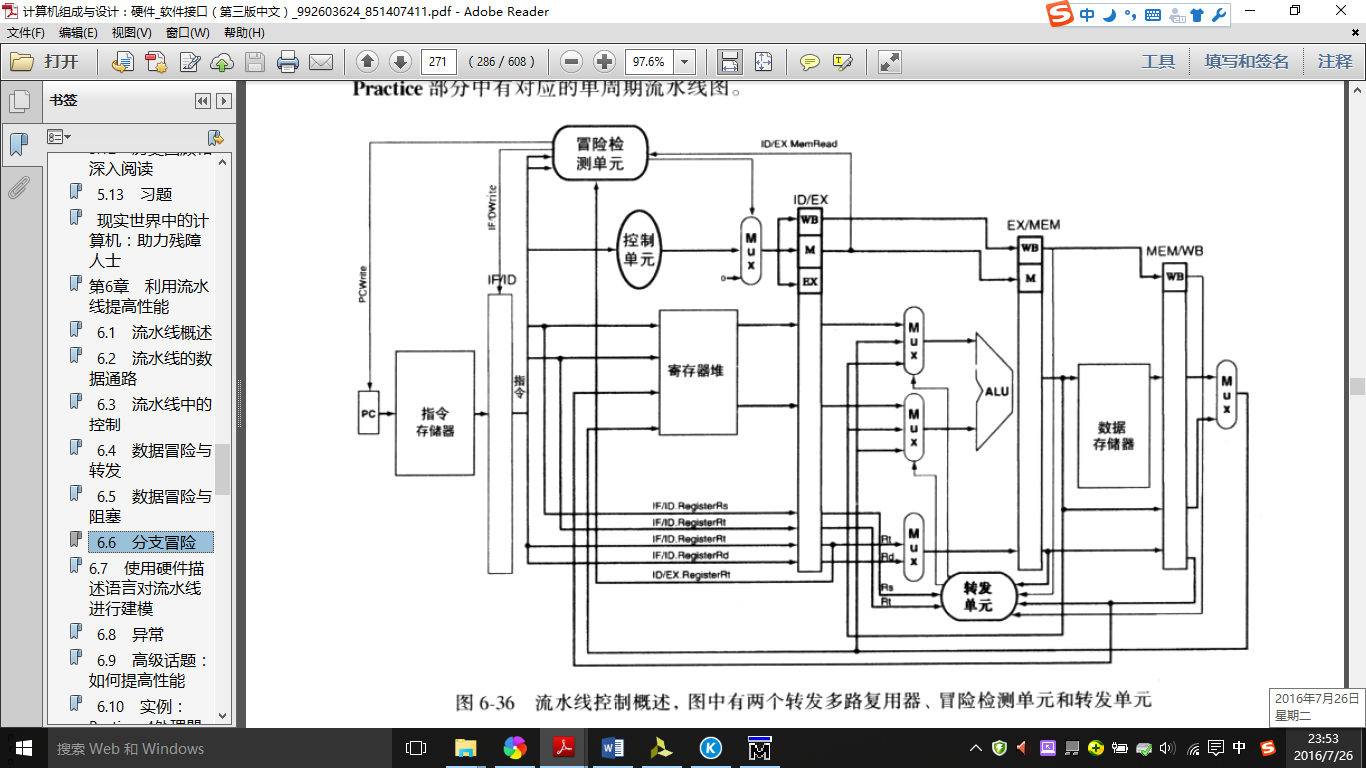
在单周期MIPS处理器的基础上，设计一个5级流水处理器。具体要求如下：

1. 采用完全的forwarding电路解决数据关联问题。 
2. 对于Load-use类竞争采取阻塞一个周期 +Forwarding的方法解决 
3. 对于分支指令在EX阶段判断（提前判断也可以）， 在分支发生时刻取消ID和IF阶段的两条指令。 
4. 对于J类指令在ID阶段判断，并取消IF阶段指令。
5. 将计算最大公约数的程序在流水线MIPS处理器中正常运行。

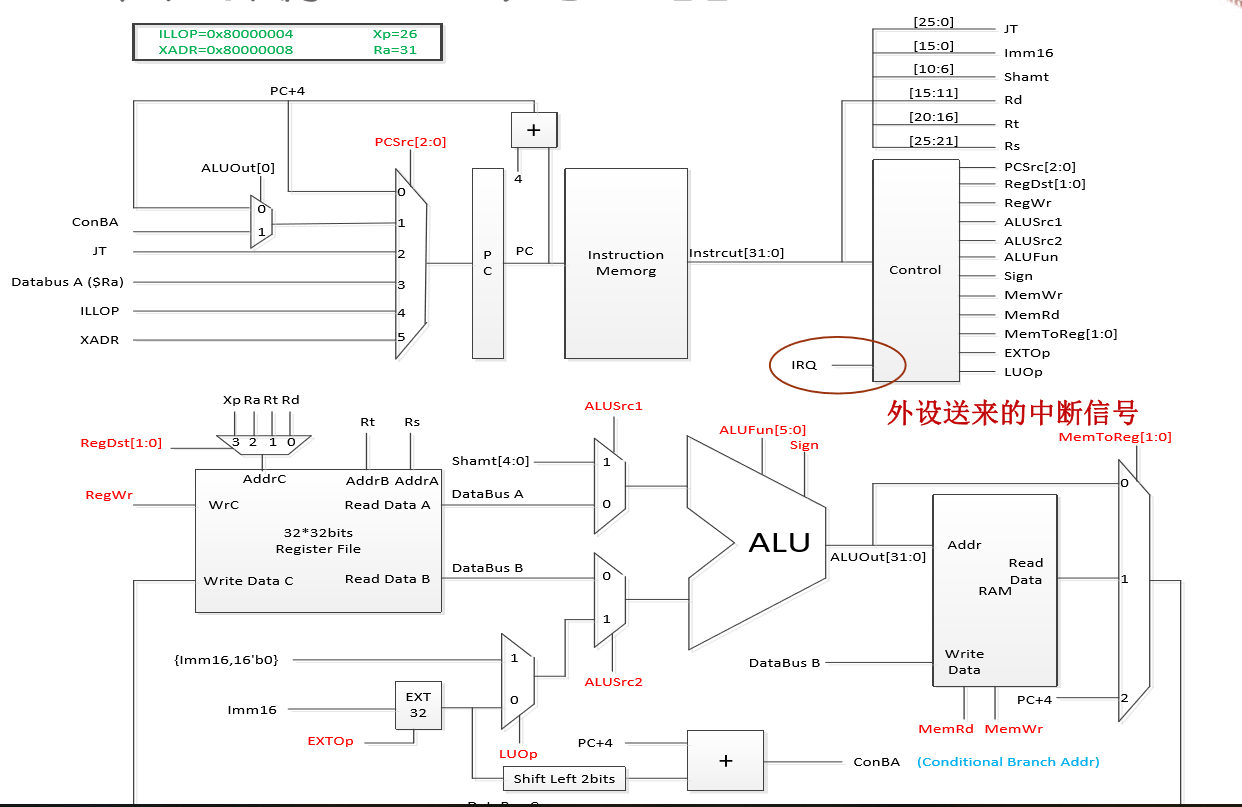
## 二．设计方案

## （一）实验原理

流水线是提高CPU运行效率的关键技术，其核心思想是把多条指令的不同执行阶段重叠起来，使得CPU能同时处理多条指令。这些指令分处于不同的运行周期，使用不同的物理器件。一条MIPS指令一般包括如下五个步骤：一、从存储器中读取指令，一般称其为IF阶段；二、指令解码的同时读取寄存器，一般称其为ID阶段；三、执行操作或计算地址，一般称其为EX阶段；四、在数据存储器中读取操作数，一般称其为MEM阶段；五、将结果写会寄存器，一般称其为WB阶段。对应于这五个阶段，对应着五级流水。相较于单周期MIPS处理器，五级流水与其最主要的区别在于两点：一是需要在五个阶段之间添加**四个流水寄存器**；二是需要解决实验要求中提到的**竞争**问题。



上图为5级流水实现示意图（未显示分支指令冒险检测与单周期模块细节）。图中包括单周期实现的模块、四个流水寄存器、转发单元和冒险检测单元。在我的实现当中，顶层模块中包括6个主要的连接模块：4个流水寄存器为分割的数据通路模块以及转发单元模块与冒险检测单元模块。其中，涉及到主要数据通路的模块有四个，其命名分别为Pipeline\_IF\_ID, Pipeline\_ID\_EX, Pipeline\_EX\_MEM, Pipeline\_MEM\_WB；转发单元和冒险检测单元的模块名称分别为Pipeline\_Forwarding与Pipeline\_Hazard。主要模块中与单周期有关的模块（PC, Ctrl,InsMemory,Register等）沿用单周期代码，略有改动。单周期数据通路如下：

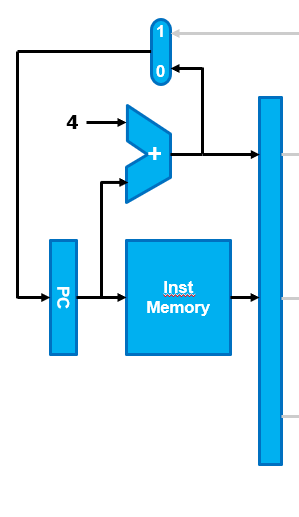


### （二）各模块设计方案

下面，就将对于流水线几个主要模块的设计思路分别进行说明。

1. **Pipeline\_IF\_ID模块**

Pipeline\_IF\_ID模块包括从得到当前指令的PC到将需要的信息传入IF\_ID寄存器的过程。该模块下面包括三个子模块Pipeline\_PC, Pipeline\_InsMem, Pipeline\_IFID\_reg.

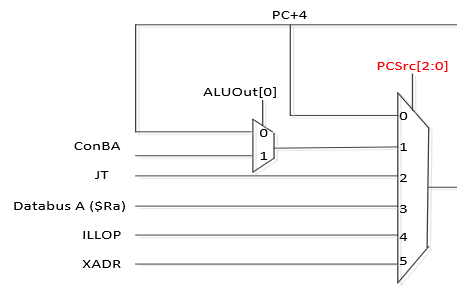


**（1）Pipeline\_PC**

该模块功能为根据当前状态得到PC。

模块输入为clk, reset, nop, ALUOut0, [2:0] PCSrc, JT, [31:0] DatabusA, [31:0] ConBA；输出为[31:0] PC. 其中，nop用于标记存在load-use冒险，ALUOut0为分支中的条件判断结果，JT为J型指令的26位地址，DatabusA的值取自于$Ra，ConBA来自于PC+4与指令中16位立即数左移两位的数值之和。

得到当前PC的方法与单周期中的类似，只需使用多路选择器即可，如下图所示。其中，PCSrc=4时，发生中断，ILLOP=0x8000\_0004；PCSrc=5时，发生异常，XADR=0x8000\_0008。关于load-use冒险的处理，利用nop作为标记，当其为1时，PC保持一个周期不变，实现阻塞



**（2）Pipeline\_InsMem**

该模块功能为从当前PC得到下一条指令。函数的输入为 [6:0] PC\_82, reset, nop, 输出为 [31:0] Ins. 其中，PC\_82为当前PC的第9位到第3位，即PC[8:2]；Ins为当前需要执行的指令。

该过程与单周期实现过程没有太大区别，唯一的区别为当load-use冒险发生时发生阻塞。实现时，当nop为1时，直接将Ins赋值为0；否则正常赋值。

**（3）Pipeline\_IFID\_reg**

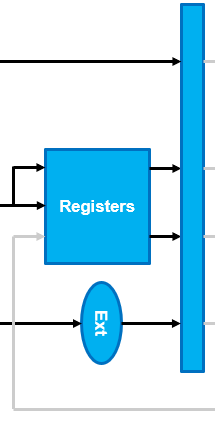
该模块功能为将需要的信息传递到IF\_ID寄存器。函数的输入为clk, reset, nop, Stall, [31:0] PC, [31:0]Ins; 输出为[63:0] IFID. 其中，Stall为顶层函数中的IFID\_Stall, 用于分支发生或者跳转发生的判断；IFID为输出信息，IFID[63:32]对应于传递到寄存器中的指令信息，IFID[31:0]对应于传递到寄存器中的PC信息。

当需要阻塞时（nop=1），需要保持传递到IF\_ID寄存器的PC和指令信息不变，因而IFID不变；而nop为0时，将IFID进行更新。如果需要取消IF阶段指令（stall=1），需要将IFID清空，置为0；否则使用上面得到的IFID.

输出[63:0]IFID是将PC和指令连接起来，这样便于模块的书写，这种处理方法后面也会用到

1. **Pipeline\_ID\_EX 模块**

Pipeline\_ID\_EX模块是从IF\_ID寄存器和MEM\_WB寄存器中得到需要的信息进行处理后传递到ID\_EX寄存器；实现的功能为得到控制信号，对寄存器堆进行读写等操作，将需要信息写入ID\_EX寄存器。对应如上三个功能，该模块含有三个子模块，分别为cpu\_Ctrl, cpu\_Reg, Pipeline\_IDEX\_reg

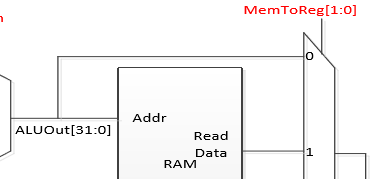


**（1）cpu\_Ctrl**

控制模块与单周期中完全相同，需要实现的功能为从IF\_ID寄存器中的指令信息中得到控制信号。输入为[31:0] Instruct, PC, IRQ ;输出为 [25:0] JT, [15:0] Imm16, [4:0] Shamt, [4:0] Rd, [4:0] Rt, [4:0] Rs, [2:0] PCSrc, [1:0] RegDst, [1:0] MemToReg, [5:0] ALUFun, RegWr, ALUSrc1, ALUSrc2, Sign, MemWr, MemRd, EXTOp, LUOp. 其中，Instruct为IFID[63:32], 即IF\_ID寄存器中的指令信息；PC为IFID[31], 即IF\_ID寄存器中PC信息的最高位。.

**（2）cpu\_Reg**

首先需要将MEM\_WB[63:32]（从数据存储器中读出的数据）与MEM\_WB[31:0]（经过ALU中得到的数据）通过一个二选一的多路选择器；其控制信号为MemToReg[0], 当其为1时，输出DataBusC为前者，否则为后者（这个结论可由下图得出）。

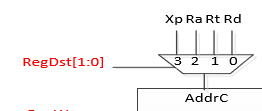


该模块实现的功能为通过需要的控制信号，将需要的数据写入寄存器堆或者从寄存器堆中读出数据。输入为 clk, reset, RegWr, IRQ, MemToReg[1], [4:0] AddrA, [4:0] AddrB, [4:0] AddrC, [31:0] WriteDataC, [31:0] PC；输出为 [31:0] ReadDataA, [31:0] ReadDataB。其中，IRQ为中断控制信号；AddrA,AddrB分别为Rs,Rt，而Rs, Rt 为控制模块的输出；AddrC为从EX\_WB寄存器传回的AddrC数据，对应于MEMWB\_data[68:64]；WriteDataC为从上面提到的DataBusC。

该模块实现的功能有两个：一是向寄存器堆写入数据，二是从寄存器中读出数据。为实现该功能，首先构造了一个31\*32的二维数组，命名为[31:0] Reg[31:1]，对应于1到32号寄存器，分别存储着每个寄存器中的信息。从寄存器中读取信息比较简单，只需要将地址（不为0时）为AddrA和AddrB的寄存器堆中的信息写入DataBusA和DataBusB中；若AddrA或AddrB为0，则对应0号寄存器，其中存储的信息一定为0，因而将DataBusA或DataBusB 赋值为0即可。向寄存器堆中写入数据时，若不发生中断且指令不为JAL/JALR，则将Reg[AddrC]赋值为DataBusC，当然必须要求AddrC不为0且RegWr为1；若发生中断或者指令为JAL/JALR，根据要求，此时写回的寄存器分别为26号寄存器和31号寄存器，写回的数据为PC+4.实现时，用信号enable标记后一种状态。

**（3）Pipeline\_IDEX\_reg**

在进入该模块之前，首先需要进行如下几个步骤。从控制信号单元得到了立即数Imm16，在控制信号EXTOp控制下得到符号扩展数或是非符号扩展数Imm32，在控制信号LUOp控制下得到将Imm16载入高位或是直接为Imm32的LUout。在控制信号RegDst[1:0] 控制下得到了AddrC，具体实现方法如下图；其中，Xp=26，Ra=31. ConBA为PC+4与指令中16位立即数左移两位的数值之和。



该模块实现的功能是在必要信号的控制下，将需要的信息传递到ID\_EX寄存器。该函数的输入为clk, reset, Stall, ALUSrc1, ALUSrc2, Sign, MemWr, MemRd, RegWr, [1:0] MemToReg, [2:0] PCSrc, [5:0] ALUFun, [4:0] Rs, [4:0]Rt, [4:0]Shamt, [4:0]AddrC, [31:0] LUout, [31:0] ConBA, [31:0] DataBusA, [31:0] DataBusB；输出为 [148:0] IDEX\_data, [17:0] IDEX\_control.其中，Stall为ID\_EXStall,当其为1时，表明需要将ID\_EX寄存器中的信息清零，具体情况将在较后部分说明；LUSrc1, ALUSrc2, Sign, MemWr, MemRd, RegWr, MemToReg, PCSrc, ALUFun, Rs, Rt, Shamt均为控制信号单元的输出；AddrC, LUout, ConBA均为上一段说明的数据信息；DataBusA, DataBusB均为从寄存器读出的信息；输出为IDEX\_data, IDEX\_control，对应的信息关系如下：

IDEX\_data={[147:116]ConBA,[115:84]LUout,[83:52]DataBusB,[51:20]DataBusA,[19:15]Shamt,[14:10]Rt,[9:5]Rs,[4:0]AddrC};

IDEX\_control={[16]ALUSrc1,[15]ALuSrc2,[14]Sign,[13]MemWr,[12]MemRd,[11]RegWr,[10:8]PCSrc,[7:6]MemToReg,[5:0]ALUFun};

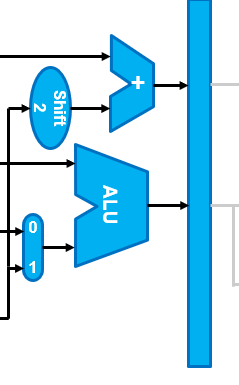
为实现上述功能，在reset有效或Stall信号（即为顶层模块中的ID\_EXStall信号）为1时，将IDEX\_data与IDEX\_control清零；否则将对应信号输入到IDEX\_data与IDEX\_control的对应位置。

**（4）Pipeline\_ID\_EX模块的几点补充**

对于J型指令而言，ID阶段可以得到其跳转的地址JT，因而该模块的输出有[25:0]JT,供Pipeline\_IF\_ID模块使用。而且，当寄存器跳转时，需要知道$Ra中的值，该值存储在DataBusA中；该模块的一个输出为DataBusA，该模块的有一个输出为PCSrc，供Pipeline\_IF\_ID模块使用。该模块的一个输出为PCSrc，对应着顶层模块中的ID\_PCSrc，需要注意。该模块还有一个输出为Stall\_one, 当指令为J型指令或者发生中断、异常时，其值为1，此时应当将IF阶段的值清空。实现起来，利用Pipeline\_IF\_ID模块内部的PCSrc信号判断即可；我的实现方法是利用真值表，当PCSrc[2:1]为2或者1时，令Stall\_one为1。

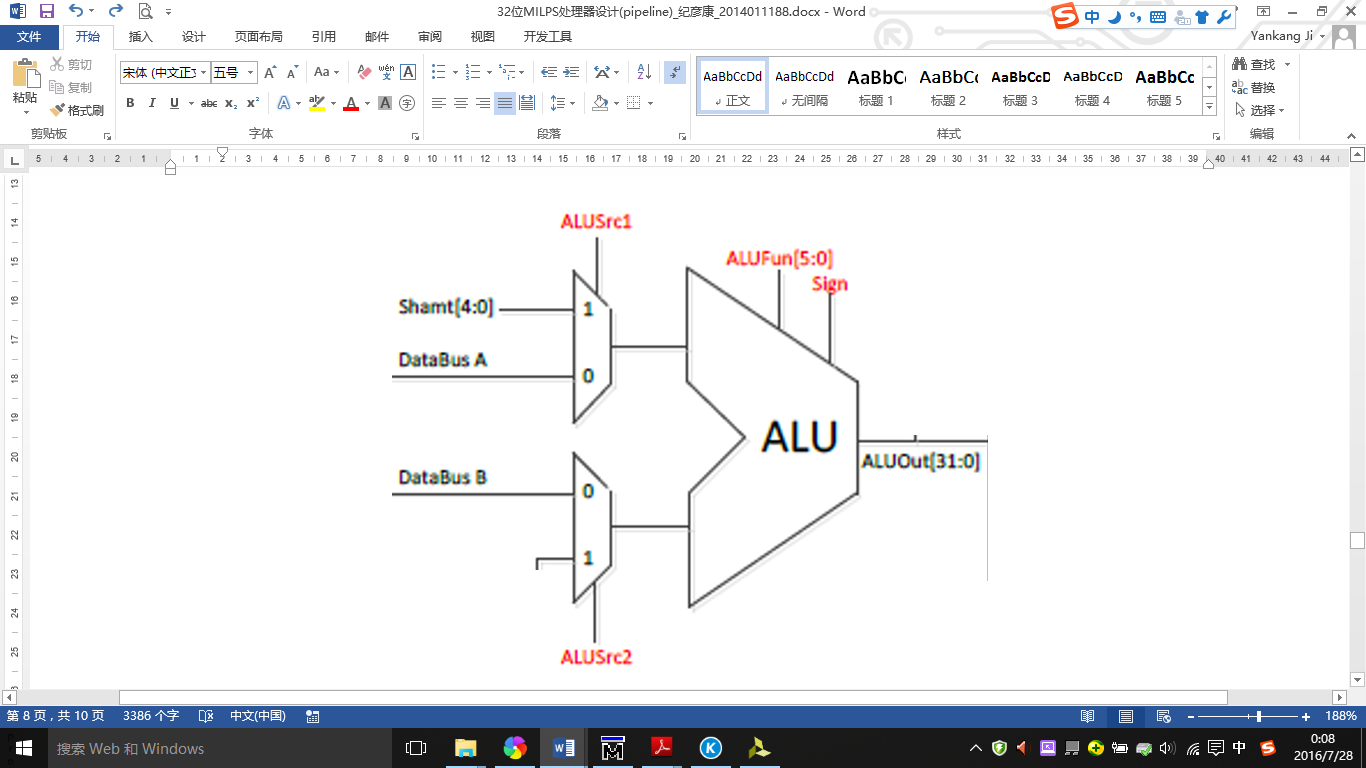
1. **Pipeline\_EX\_MEM模块**

Pipeline\_EX\_MEM模块是从ID\_EX寄存器中得到需要的信息进行处理后传递到EX\_MEM寄存器；实现的功能为对数据进行ALU操作，将由ALU操作中得到的信号及其余必要信号传递到EX\_MEM寄存器。该模块包括2个子模块：ALU, Pipeline\_EXMEM\_reg



**（1）ALU**

在进行ALU操作之前，首先需要得到进行ALU操作的数据。如下图所示，若想得到进行ALU操作的两个操作数，必须均经过一个三选一多路选择器和一个二选一多路选择器；前者与转发单元有关，后者与控制信号有关。对于ALU的第一个输入数，三选一多路选择器的控制信号为由Pipeline\_Forwarding模块得到的forwardA,当其为0到4时，输出分别为IDEX\_data[15:20]（对应IDEX\_databusA）、MEMWB\_Data（对应于主函数中的DataBusC，为Pipeline\_WB\_ID\_EX模块的输出）、EXMEM\_Data（对应于主函数中的EXMEM\_data[31:0]，为Pipeline\_EX\_MEM模块的输出）、和0；接下来通过二选一多路选择器，控制信号为IDEX\_control[15]（对应于IDEX\_ALUSrc1），当其为0和1时，输出分别为三选一多路选择器的输出和{27'b0,IDEX\_data[19:15]}（即为{27'b0,IDEX\_shamt}）。对于ALU的第二个输入，首先通过三选一多路选择器，与第一个的类似；然后通过二选一多路选择器，控制信号IDEX\_control[16]（对应于IDEX\_ALUSrc2），当其为0和1时，输出分别为三选一多路选择器的输出和IDEX\_data[115:84]（对应于IDEX\_LUOut）.



ALU的具体细节不再赘述

**（2）Pipeline\_EXMEM\_reg**

该模块功能是将必要信息传递到EX\_MEM寄存器。该函数的输入为clk, reset, [4:0] IDEX, [4:0]AddrC, [1:0] IDEX\_MemToReg, [2:0] IDEX\_con, [31:0] ALU\_dataB, [31:0] ALUOut；输出为[68:0] EXMEM\_data, [4:0]EXMEM\_control，对应关系如下：

EXMEM\_data={[68:64]AddrC,[63:32]ALU\_dataB,[31:0]ALUOut};

EXMEM\_control={[4]MemWr,[3]MemRd,[2]RegWr,[1:0]MemToReg};

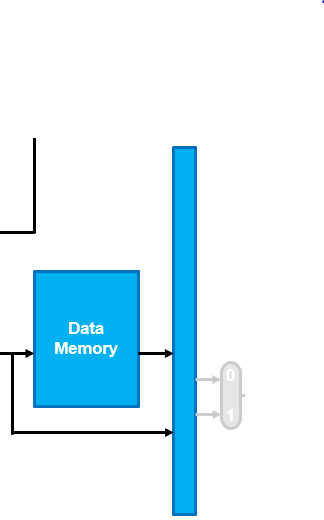
该模块功能的实现只需要在reset=0时，将对应的数据输出到EX\_MEM寄存器中即可。

**（3）Pipeline\_EX\_MEM总体模块的几点补充**

在EX阶段，分支指令可以得到判断，跳转的地址ConBA也将被计算出来；该模块的一个输出就是ConBA，将被用于Pipeline\_IF\_ID的输入。由于该模块可以得到ALU的计算结果，用于判断的ALUOut[0]也将得到；该模块的另一个输出就是Zero=ALUOut[0],将被用于 Pipeline\_IF\_ID的输入。该模块的一个输出为Stall，用于表明指令类型为跳转指令并且跳转发生。对于表明一个指令为跳转指令，只需判断该模块中的PCSrc是否为1；若要判断跳转是否发生，则需要看ALUOut[0]是否为1。此时需要对IF、ID阶段的信息进行清空。

1. **Pipeline\_MEM\_WB模块**

Pipeline\_MEM\_WB模块是从EX\_MEM寄存器中得到需要的数据进行处理后传递到MEM\_WB寄存器，中间包括中断情况下跳转到外设单元；实现的功能为对datamemory进行读操作或者写操作等（包括对于外设的操作），然后将需要的数据传输到MEM\_WB寄存器。对应如上两个功能，该模块含有两个子模块，分别为cpu\_data\_Memory， Pipeline\_MEMWB\_reg



**（1）cpu\_data\_Memory**

该模块功能是对数据存储器进行读操作或者写操作等，并且包括对于外设的书写。该函数与单周期完全一样。由于Pipeline\_MEMWB\_reg模块的输入较为明确，因而在这里不再详细列出cpu\_data\_Memory的输入输出。该模块输入输出包括RX\_DATA, TX\_EN, TX\_STATUS, RX\_STATUS等，起到连接串口数据传输发送器和接收器的作用。

**（2）Pipeline\_MEMWB\_reg**

该模块功能是将必要的信息传输到MEM\_WB寄存器。该函数的输入为clk, reset, [31:0] ReadData [31:0] EXMEM\_ALUOut, [4:0]EXMEM\_AddrC, [2:0]EXMEM\_con；输出为[68:0] MEMWB\_data, [1:0]MEMWB\_MemToReg, MEMWB\_RegWr. 其中，ReadData为CPU\_RAM的输出，是从数据存储器中读出的数据；EXMEM\_ALUOut, EXMEM\_AddrC, EXMEM\_con 分别为EXMEM\_data[31:0], EXMEM\_data[68:64], EXMEM\_control[2:0]；MEMWB\_data所对应的信息如下：

MEMWB\_data={[68:64]EXMEM\_AddrC,[63:32]ReadData,[31:0]EXMEM\_ALUOut};

该模块实现较为简单，只需要在reset=0时，将对应的数据输出到MEM\_WB寄存器中

1. **Pipeline\_Hazard模块**

此模块为冒险检测模块，用于处理load-use冒险。存在load-use冒险有如下两种情况IDEX\_Rt==IFID\_Rs,IDEX\_Rt==IFID\_Rt。前提条件是IDEX\_MemRd为1.

该函数的输入为IDEX\_MemRd, [4:0] IDEX\_Rt, [4:0] IFID\_Rs, [4:0] IFID\_Rt；输出为nop。nop为1时，说明存在load-use冒险；否则不存在。

1. **Pipeline\_Forwarding模块**

该模块用于处理数据关联冒险：当一条指令需要用到上面某一条指令中寄存器中的值，但由于流水线的存在导致寄存器中的值还没有更新。由于寄存器要存储的数值已被计算出来，因而采用Forwarding的方法，提前获得寄存器中的值。Forwarding主要有4种，分别为EXMEM\_Rd==IDEX\_Rs, MEMWB\_Rd==IDEX\_Rs, EXMEM\_Rd==IDEX\_Rt, MEMWB\_Rd==IDEX\_Rt. Fowarding使用的前提必须为EXMEM\_RegWr/MEMWB\_RegWr为1并且EXMEM\_Rd/ MEMWB\_Rd不为0.对于EX/MEM Hazard而言，以上三个条件就已经足够了；但是对于MEM/WB Hazard而言，还要保证不存在EX/MEM Hazard, 这样才可以说存在MEM/WB Hazard。

该函数的输入为EXMEM\_RegWr, MEMWB\_RegWr, [4:0] EXMEM\_Rd, [4:0]MEMWB\_Rd, [4:0]IDEX\_Rs, [4:0]IDEX\_Rt；输出为 [1:0] forwardA, [1:0] forward. forwardA对应于EXMEM\_Rd==IDEX\_Rs, MEMWB\_Rd==IDEX\_Rs;forwardB对应于EXMEM\_Rd==IDEX\_Rt, MEMWB\_Rd==IDEX\_Rt。 对于两者而言，如果其为2，说明存在EX/MEM Hazard；如果为1，说明存在MEM/WB Hazard。

1. **总模块的几点补充**

Pipeline\_total包括上述6个模块以及CPU时钟模块、UART外设。

对于J类指令用Stall\_one标记，分支跳转指令用Stall标记，load-use冒险用nop，当前两者之一发生时，IFID\_Stall置为1，清空IF\_ID寄存器；当后两者之一发生时，IDEX\_Stall置为1，清空ID\_EX寄存器。

PC是由控制信号PCSrc信号决定的，而控制信号是在ID阶段产生的，所以一般情况下，PCSrc就是ID\_PCSrc。.但是当指令为跳转指令时，PC就和平常不一样了，PCSrc也要发生变化。我采用的方法是当ID\_PCSrc=1或EX\_PCSrc =1时，PCSrc均使用EX\_PCSrc的值；后者容易理解，因为此时计算出来跳转的地址，可以选择是否跳转，而前者的话若采用ID\_PcSrc，则会发生跳转，而此时跳转地址还没有计算出来，会出现错误，因而PCSrc均使用EX\_PCSrc的值。

## 三．关键代码与文件清单

**1. 顶层模块关键代码**

1. //外设部分
2. cpu\_baudrate brclk(clk\_sys,reset,clk\_UART);
3. cpu\_clk clock(clk\_sys,reset,clk);
4. cpu\_UART\_R uartr(clk\_sys,clk\_UART,reset,UART\_RX,RX\_STATUS,RX\_DATA);
5. cpu\_UART\_S uartt(clk\_sys,clk\_UART,reset,TX\_EN,TX\_DATA,TX\_STATUS,UART\_TX );
7. //IFID部分
9. assign PCSrc=(ID\_PCSrc==3'b001||EX\_PCSrc == 3'b001)? EX\_PCSrc: ID\_PCSrc;
10. assign IFID\_Stall=Stall\_one|Stall;
11. Pipeline\_IF\_ID total\_IF\_ID(
12. .clk(clk),
13. .reset(reset),
14. .nop(nop),
15. .Stall(IFID\_Stall),
16. .PCSrc(PCSrc),
17. .ALUOut0(Zero),
18. .JT(JT),
19. .ConBA(ConBA),
20. .DatabusA(DataBusA),
21. .IFID(IFID),
22. .PC31(PC31));
24. //IDEX部分
25. assign IDEX\_Stall=nop|Stall;
26. Pipeline\_ID\_EX total\_WB\_ID\_EX(
27. .PC31(PC31),
28. .clk(clk),
29. .reset(reset),
30. .IRQ(IRQ),
31. .IDEX\_Stall(IDEX\_Stall),
32. .Stall\_one(Stall\_one),
33. .IFID(IFID),
34. .MEMWB\_data(MEMWB\_data),
35. .MEMWB\_RegWr(MEMWB\_RegWr),
36. .MEMWB\_MemToReg(MEMWB\_MemToReg),
37. .Rt(Rt),
38. .Rs(Rs),
39. .PCSrc(ID\_PCSrc),
40. .JT(JT),
41. .DataBusA(DataBusA),
42. .DataBusC(DataBusC),
43. .IDEX\_data(IDEX\_data),
44. .IDEX\_control(IDEX\_control));
45. //EXMEM部分
47. Pipeline\_EX\_MEM total\_EX\_MEM(
48. .clk(clk),
49. .reset(reset),
50. .forwardA(forwardA),
51. .forwardB(forwardB),
52. .EXMEM\_Data(EXMEM\_data[31:0]),
53. .MEMWB\_Data(DataBusC),
54. .IDEX\_data(IDEX\_data),
55. .IDEX\_control(IDEX\_control),
56. .EXMEM\_data(EXMEM\_data),
57. .EXMEM\_control(EXMEM\_control),
58. .Zero(Zero),
59. .PCSrc(EX\_PCSrc),
60. .ConBA(ConBA),
61. .Stall(Stall));

64. //MEMWB部分
65. Pipeline\_MEM\_WB total\_MEM\_WB(
66. .clk(clk),
67. .reset(reset),
68. .led(led),
69. .AN(AN),
70. .digi(digi),
71. .IRQ(IRQ),
72. .EXMEM\_data(EXMEM\_data),
73. .EXMEM\_control(EXMEM\_control),
74. .MEMWB\_data(MEMWB\_data),
75. .MEMWB\_RegWr(MEMWB\_RegWr),
76. .MEMWB\_MemToReg(MEMWB\_MemToReg),
77. .UART\_TXD(TX\_DATA),
78. .RX\_DATA(RX\_DATA),
79. .TX\_EN(TX\_EN),
80. .TX\_STATUS(TX\_STATUS),
81. .RX\_STATUS(RX\_STATUS),
82. .An(An),
83. .Digital(Digital));
84. //Forwarding部分
85. Pipeline\_Forwarding total\_Forwarding(
86. .EXMEM\_RegWr(EXMEM\_control[2]),
87. .MEMWB\_RegWr(MEMWB\_RegWr),
88. .EXMEM\_Rd(EXMEM\_data[68:64]),
89. .MEMWB\_Rd(MEMWB\_data[68:64]),
90. .IDEX\_Rs(IDEX\_data[9:5]),
91. .IDEX\_Rt(IDEX\_data[14:10]),
92. .forwardA(forwardA),
93. .forwardB(forwardB));
94. //Hazard部分
95. Pipeline\_Hazard total\_Hazard(
96. .IDEX\_MemRd(IDEX\_control[12]),
97. .IDEX\_Rt(IDEX\_data[14:10]),
98. .IFID\_Rs(Rs),
99. .IFID\_Rt(Rt),
100. .nop(nop));

**2. IFID模块：**

1. Pipeline\_PC IF\_ID\_PC(
2. .nop(nop),
3. .clk(clk),
4. .reset(reset),
5. .PCSrc(PCSrc),
6. .ALUOut0(ALUOut0),
7. .ConBA(ConBA),
8. .JT(JT),
9. .Databus\_A(DatabusA),
10. .PC(PC));
12. Pipeline\_InsMem IF\_ID\_InsMem(
13. .reset(reset),
14. .nop(nop),
15. .PC\_82(PC[8:2]),
16. .Ins(Ins));
18. Pipeline\_IFID\_reg IF\_ID\_reg(
19. .clk(clk),
20. .reset(reset),
21. .nop(nop),
22. .Stall(Stall),
23. .PC(PC),
24. .Ins(Ins),
25. .IFID(IFID));

**IFID寄存器：**

1. wire[63:0] IFID\_stop;
2. assign IFID\_stop=nop? IFID:{Ins,PC};
4. always @(posedge clk or negedge reset)
5. begin
6. **if**(!reset) IFID<=64'b0;
7. **else** IFID<=Stall? 64'b0:IFID\_stop;
8. end

**3. IDEX模块：**

1. cpu\_Ctrl ID\_EX\_Control(.PC31(PC31),
2. .Instruct(IFID[63:32]),
3. .PC(IFID[31:0]),
4. .IRQ(IRQ),
5. .JT(JT),
6. .Imm16(Imm16),
7. .shamt(Shamt),
8. .Rd(Rd),
9. .Rt(Rt),
10. .Rs(Rs),
11. .ALUFun(ALUFun),
12. .PCSrc(PCSrc),
13. .RegDst(RegDst),
14. .MemToReg(MemToReg),
15. .RegWr(RegWr),
16. .ALUSrc1(ALUSrc1),
17. .ALUSrc2(ALUSrc2),
18. .Sign(Sign),
19. .MemWr(MemWr),
20. .MemRd(MemRd),
21. .EXTOp(EXTOp),
22. .LUOp(LUOp));
24. assign DataBusC=MEMWB\_MemToReg[0]?MEMWB\_data[63:32]:MEMWB\_data[31:0];
25. cpu\_Reg ID\_EX\_Register(
26. .clk(clk),
27. .reset(reset),
28. .IRQ(IRQ),
29. .RegWr(MEMWB\_RegWr),
30. .MemToReg1(MemToReg[1]),
31. .AddrA(Rs),
32. .AddrB(Rt),
33. .AddrC(MEMWB\_data[68:64]),
34. .WriteDataC(DataBusC),
35. .PC(IFID[31:0]),
36. .ReadDataA(DataBusA),
37. .ReadDataB(DataBusB));
39. assign Imm32=EXTOp?{{16{Imm16[15]}},Imm16}:{16'h0000,Imm16};//·ûºÅÎ»À©Õ¹
40. assign LUout=LUOp?{Imm16,16'b0}:Imm32;//ÊÇ·ñÔØÈë¸ßÎ»
41. assign AddrC=(RegDst==0)?Rd:(RegDst==1)?Rt:(RegDst==2)?5'b1\_1111:5'b1\_1010;//Ð´ÈëÎ»ÖÃ
42. assign ConBA={IFID[31],{IFID[30:0]+31'd4+{Imm32[28:0],2'b00}}};//·ÖÖ§Î»ÖÃ
44. Pipeline\_IDEX\_reg ID\_EX\_reg(
45. .clk(clk),
46. .reset(reset),
47. .Stall(IDEX\_Stall),
48. .Rs(Rs),
49. .Rt(Rt),
50. .AddrC(AddrC),
51. .Shamt(Shamt),
52. .LUout(LUout),
53. .ConBA(ConBA),
54. .DataBusA(DataBusA),
55. .DataBusB(DataBusB),
56. .ALUSrc1(ALUSrc1),
57. .ALUSrc2(ALUSrc2),
58. .PCSrc(PCSrc),
59. .ALUFun(ALUFun),
60. .Sign(Sign),
61. .MemWr(MemWr),
62. .MemRd(MemRd),
63. .RegWr(RegWr),
64. .MemToReg(MemToReg),
65. .IDEX\_data(IDEX\_data),
66. .IDEX\_control(IDEX\_control));
68. assign Stall\_one=((PCSrc[2:1]==2'b10)||(PCSrc[2:1]==2'b01));

**IDEX寄存器：**

1. always @(posedge clk or negedge reset)
2. begin
3. **if**(!reset)
4. begin
5. IDEX\_data<=148'b0;
6. IDEX\_control<=17'b0;
7. end
8. **else** **if**(Stall)begin
9. IDEX\_data<=148'b0;
10. IDEX\_control<=17'b0;
11. end
12. **else**
13. begin
14. IDEX\_data<={ConBA,LUout,DataBusB,DataBusA,Shamt,Rt,Rs,AddrC};
15. IDEX\_control<={ALUSrc1,ALUSrc2,Sign,MemWr,MemRd,RegWr,PCSrc,MemToReg,ALUFun};
16. end
17. end

**4. EXMEM模块**

1. always @(\*)
2. begin
3. **if**(forwardA==2'b00) ALU\_dataA<=IDEX\_data[51:20];//databusA
4. **else** **if**(forwardA==2'b01) ALU\_dataA<=MEMWB\_Data;
5. **else** **if**(forwardA==2'b10) ALU\_dataA<=EXMEM\_Data;
6. end
8. always @(\*)
9. begin
10. **if**(IDEX\_control[16])//ALUSrc1
11. ALU\_A<={27'b0,IDEX\_data[19:15]};//shamt
12. **else** ALU\_A<=ALU\_dataA;
13. end
15. always @(\*)
16. begin
17. **if**(forwardB==2'b00) ALU\_dataB<=IDEX\_data[83:52];//databusB
18. **else** **if**(forwardB==2'b01) ALU\_dataB<=MEMWB\_Data;
19. **else** **if**(forwardB==2'b10) ALU\_dataB<=EXMEM\_Data;
20. end
22. always @(\*)
23. begin
24. **if**(IDEX\_control[15])//ALUSrc2
25. ALU\_B<=IDEX\_data[115:84];//luout
26. **else** ALU\_B<=ALU\_dataB;
27. end
29. ALU EX\_MEM\_ALU(
30. .A(ALU\_A),
31. .B(ALU\_B),
32. .ALUFun(IDEX\_control[5:0]),
33. .Sign(IDEX\_control[14]),
34. .OUT(ALUOut));
36. Pipeline\_EXMEM\_reg EX\_MEM\_reg(
37. .clk(clk),
38. .reset(reset),
39. .IDEX\_AddrC(IDEX\_data[4:0]),
40. .IDEX\_MemToReg(IDEX\_control[7:6]),
41. .IDEX\_con(IDEX\_control[13:11]),
42. .ALU\_dataB(ALU\_dataB),
43. .ALUOut(ALUOut),
44. .EXMEM\_data(EXMEM\_data),
45. .EXMEM\_control(EXMEM\_control));
47. assign Zero=ALUOut[0];
48. assign PCSrc=(IDEX\_control[10:8]==3'b001)?3'b001:3'b0;
49. assign ConBA=IDEX\_data[147:116];
50. assign Stall=Zero&(PCSrc==3'b001);

**EXMEM寄存器：**

1. always @(posedge clk or negedge reset)
2. begin
3. **if**(!reset) begin
4. EXMEM\_data<=69'b0;
5. EXMEM\_control<=5'b0;
6. end
7. **else** begin
8. EXMEM\_data<={IDEX\_AddrC,ALU\_dataB,ALUOut};
9. EXMEM\_control<={IDEX\_con,IDEX\_MemToReg};
10. end
11. end

**5. MEMWB模块：**

1. cpu\_data\_Memory MEM\_WB\_RAM(
2. .clk(clk),
3. .reset(reset),
4. .Addr(EXMEM\_data[31:0]),
5. .WriteData(EXMEM\_data[63:32]),
6. .MemRd(EXMEM\_control[3]),
7. .MemWr(EXMEM\_control[4]),
8. .ReadData(ReadData),
9. .led(led),
10. .AN(AN),
11. .digital(digi),
12. .IRQ(IRQ),
13. .UART\_TXD(UART\_TXD),
14. .RX\_DATA(RX\_DATA),
15. .TX\_EN(TX\_EN),
16. .TX\_STATUS(TX\_STATUS),
17. .RX\_STATUS(RX\_STATUS),
18. .An(An),
19. .Digital(Digital));
21. Pipeline\_MEMWB\_reg EXE\_WB\_reg(
22. .clk(clk),
23. .reset(reset),
24. .ReadData(ReadData),
25. .EXMEM\_ALUOut(EXMEM\_data[31:0]),
26. .EXMEM\_AddrC(EXMEM\_data[68:64]),
27. .EXMEM\_con(EXMEM\_control[2:0]),
28. .MEMWB\_data(MEMWB\_data),
29. .MEMWB\_RegWr(MEMWB\_RegWr),
30. .MEMWB\_MemToReg(MEMWB\_MemToReg));

**MEMWB寄存器：**

1. always @(posedge clk or negedge reset)
2. begin
3. **if**(!reset)
4. begin
5. MEMWB\_data<=68'b0;
6. MEMWB\_MemToReg<=2'b0;
7. MEMWB\_RegWr<=1'b0;
8. end
9. **else**
10. begin
11. MEMWB\_data<={EXMEM\_AddrC,ReadData,EXMEM\_ALUOut};
12. MEMWB\_MemToReg<=EXMEM\_con[1:0];
13. MEMWB\_RegWr<=EXMEM\_con[2];
14. end
15. end

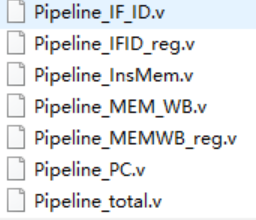
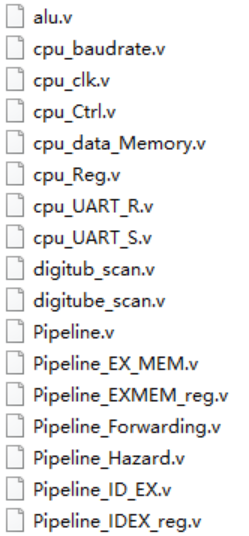
**6. Forwarding模块：**

1. always @(\*)
2. begin
3. **if**((EXMEM\_RegWr)&&(|EXMEM\_Rd)&&(EXMEM\_Rd==IDEX\_Rs)) //EX/MEM Hazard
4. forwardA<=2'b10;
5. **else** **if**((MEMWB\_RegWr)&&(|MEMWB\_Rd)&&((|EXMEM\_RegWr)||(EXMEM\_Rd!=IDEX\_Rs))&&(MEMWB\_Rd==IDEX\_Rs)) //MEM/WB Hazard.
6. forwardA<=2'b01;
7. **else** forwardA<=2'b0;
8. end
10. always @(\*)
11. begin
12. **if**((EXMEM\_RegWr)&&(|EXMEM\_Rd)&&(EXMEM\_Rd==IDEX\_Rt))  //EX/MEM Hazard
13. forwardB<=2'b10;
14. **else** **if**((MEMWB\_RegWr)&&(|MEMWB\_Rd)&&((|EXMEM\_RegWr)||(EXMEM\_Rd!=IDEX\_Rt))&&(MEMWB\_Rd==IDEX\_Rt))  //MEM/WB Hazard.
15. forwardB<=2'b01;
16. **else** forwardB<=2'b0;
17. end

**7. Hazard模块**

1. always @(\*)
2. begin
3. **if**(IDEX\_MemRd&((IDEX\_Rt==IFID\_Rs)|(IDEX\_Rt==IFID\_Rt)))   nop<=1;
4. **else** nop<=0;
5. end

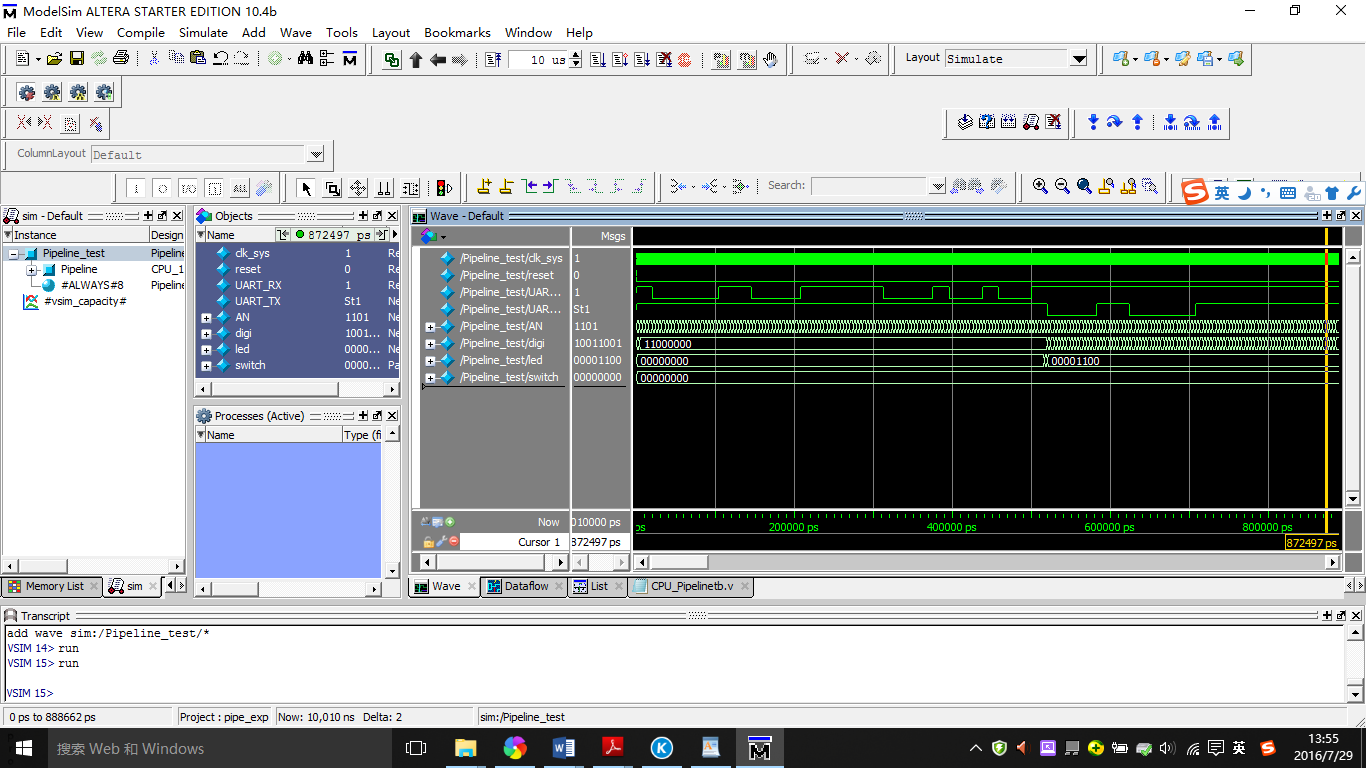
**8. 文件清单**



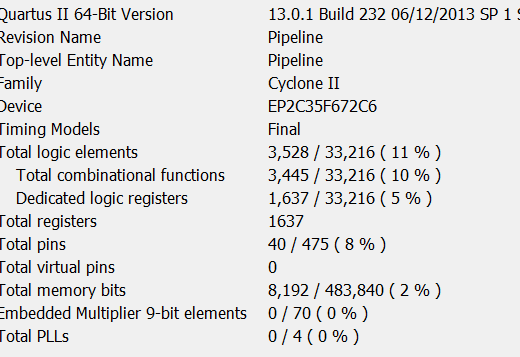
**注：Pipeline.v为最顶层文件**

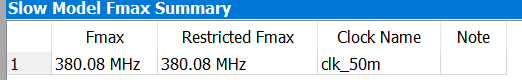
## 四．仿真结果与分析

下图为Pipeline\_total仿真结果的波形图。从上到下的信号分别为clk\_sys, reset, UART\_RX, UART\_TX, AN, digi；由于以前的书写代码习惯，将实验指导书中的[11:0]digi 拆分成了7段数码管（带小数点）的信号[7:0]digi与7段数码管扫描信号[3:0]AN. 仿真时，UART\_RX串行输入的两个数字分别为24与36，UART\_TX并行输出的结果为12，led并行输出结果为12，四个7段数码管的输出分别为24与36，符合实验要求。



## 五．综合情况





（这个频率我也没法解释做了什么优化，队友改了下alu就从100多飙到了380）

## 六．硬件调试情况

该处理器并未设置成只输出一次，故串口小助手上会看到多个重复输出。硬件状况这部分忘记拍照，实际结果在验收当天已检验

## 七．实验总结

通过本次cpu大作业，我进一步掌握了流水线处理器的原理和实现方法，通过将近一周的调试也增强了verilog debug的能力和与队友合作、沟通的能力，收获颇丰

我的设计思路源自与同学的讨论，不分五个阶段来写，而是以四个寄存器为结点，把每两个阶段的数据传输弄明白，这样确实比较清晰。由于代码风格的关系，不太喜欢把好几个模块放在一个文件里，清单比较乱，但是整体思路比较清晰

调试过程可以说很艰辛，主要方法就是把PC\_82和指令拎出来，对着MIPS代码，一个周期一个周期比较，如果是执行过程不对就检查MIPS代码，也可能是控制信号的问题，如果执行过程对，应该得到的结果不对，就要追踪这个信号，从PC开始沿数据通路挨个阶段检查。实验中遇到问题很多，时间间隔关系记得不完全了，现列举部分：

（1）第一条指令跳转地址就不对，先是MIPS写的地址部分不对，然后发现cpu\_Ctrl模块00101前没加5‘b

（2）MIPS跳转到output部分后不断循环，将output第三条改为nop指令得以解决

（3）输出结果为全0，检查发现输入数据没读入。这就比较棘手。从IFID阶段开始，发现读入数据有两条lw指令存在冒险，检查冒险处理单元和nop信号，发现PC中没加nop，需要nop阻塞一个周期。然后检查寄存器，输入数据没存到寄存器，追踪数据来源发现DatabusC没读上，MemToReg控制的不对，检查真值表发现应该取决于MemToReg[0]。然后到数据存储器，这里问题就比较大，因为队友写单周期的时候我就不清楚他的逻辑，改的时候先是把两个非阻塞赋值变成阻塞赋值以使其立即赋值，之后控制写入UART\_RXD的read也有改动。在同学提醒下修改了一个MIPS寄存器。一圈下来，数据终于读入了

（4）寄存器部分打算采用for循环给寄存器赋值，上网查知道序号i应声明为integer或数组

（5）烧到板子上还是数据存储器模块有问题（好像是不能有多个触发）新增一个always用RX\_STATUS触发得以解决

有的问题调试起来真的是费时，有一次连续调了12小时，正好也是下雨天点的外卖，连坐一天下来做梦都是代码和波形。验收前一晚我们还在debug，晚上才烧板子，慌乱之中还是稳住了阵脚，在半夜调试成功

总之现在还是比较有成就感的，也很感激给予我帮助的同学们