Lab3实验报告

## 整体设计思路分析

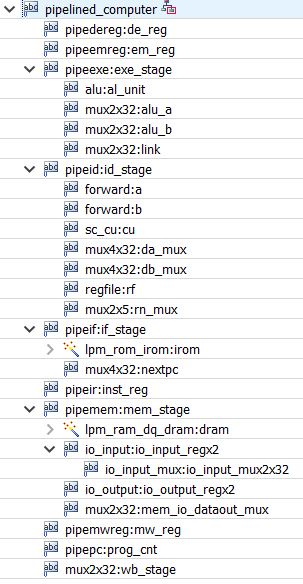
在这次的流水线CPU实验中，通过参考提供的顶层模块代码，整体分为十个模块。分别是五个流水线段，即pipeif，pipeid，pipeexe，pipemem和wb。还包括四个流水线寄存器用于分割这五个流水线段，以及一个程序计数器。

流水线CPU设计与单周期的不同是冒险的引入。这个实验中指令和数据存储器是分开的，因而不存在结构冒险。为处理数据冒险，在id阶段引入两个直通选择，fwda和fwdb。此外，针对特殊的lw指令，还需要额外插入一个停顿周期，这是通过wpcir信号控制流水线寄存器实现的。为处理结构冒险，即可能的跳转，在取指if阶段，对pcsource进行判断，若对应跳转的信号，则将指令全部置零，相当于插入一条nop指令。

整体的设计参考是课程PPT中的流水线CPU结构图。如下所示，在代码设计时，除了exe阶段的ea应改为偏移量sa以外，其余设计均是按照该图进行。



顶层模块代码与示例相同不再展示，工程文件实例如下图所示



## 具体代码说明

在这一部分将依次按照流水线五个流水段的顺序进行说明。

首先是程序计数器pipepc，可以将其看作最前面一级流水段的输入流水线寄存器。该模块的功能是将pc值更新为下一个新的pc。此外，考虑到reset信号的置零控制以及lw指令下特殊数据冒险的停顿控制。

|  |
| --- |
| module pipepc( npc,wpcir,clock,resetn,pc );  input [31:0] npc;  input clock,resetn,wpcir;  output [31:0] pc;  reg [31:0] pc;  always @ (negedge resetn or posedge clock)  if (resetn == 0) // 清零  pc <= -4;  else  if (wpcir != 0)  begin  pc <= npc; // 若不是lw指令带来的特殊数据冒险则更新pc  end  endmodule |

接下来是取指pipeif流水段。该模块的功能是取指令并确定pc的下一个值。与单周期相同的地方是都有通过irom取指令，不同的地方是多了一个四路选择器。根据pcsource判断有无跳转来确定下一个pc。在单周期中是放在mem阶段进行的，在流水线中为了降低冒险带来的损失将其前移到pipeif阶段。然而，在面临跳转相关的控制冒险时，依然需要再插入一条空指令来避免出错。通过pcsource来判断，当其为00时是PC+4，其余都是跳转相关指令，因此均通过硬件实现空指令的插入。

|  |
| --- |
| module pipeif( pcsource,pc,bpc,da,jpc,npc,pc4,ins,mem\_clock );  input [1:0] pcsource;  input mem\_clock;  input [31:0] pc, bpc, jpc, da;  output [31:0] npc, pc4, ins;  wire [31:0] npc, pc4, ins;  assign pc4 = pc + 4;  wire [31:0] fetched\_ins;  mux4x32 nextpc( pc4, bpc, da, jpc, pcsource, npc ); // 下一个pc值  lpm\_rom\_irom irom( pc[7:2], mem\_clock, fetched\_ins );  assign ins = (pcsource[0]|pcsource[1])? 32'h0:fetched\_ins;  endmodule |

接下来是if与id段之间的流水线寄存器pipeir。该寄存器将pc4与ins传递至id段。与pipepc类似，同样有reset和wpcir参与控制。由于代码与程序计数器类似，这里不再给出，具体可见代码文件。

接下来是pipeid流水段。该模块的功能包括cu控制单元，寄存器堆，以及两个解决数据冒险的直通处理选择器。这部分与单周期CPU的不同是多了对数据冒险的直通处理，而其他信号的处理和单周期相同。在数据冒险的处理中，需要考虑三种直通情况，分别是ex段末尾的alu输出端；mem末尾的alu输出端和mem末尾的存储器输出端。前两种情况可以完全通过直通选择的方式解决，而最后一种情况与lw相关，不仅需要直通，还需要插入停顿周期。Wpcir信号即用来处理该情况，当wpcir信号为0时，所有其他信号均置零，同时pc也暂停更新一次。

|  |
| --- |
| assign wpcir = ~(em2reg & ((ern==drs)|(ern==drt)) & ~dwmem);  //wpcir是针对lw取出来的值在下一条指令中使用的情况  regfile rf( drs, drt, wdi, wrn, wwreg, mem\_clock, resetn, q1, q2 ); //寄存器堆  mux4x32 da\_mux( q1, ealu, malu, mmo, fwda, da ); // 四选一 可能的直通  mux4x32 db\_mux( q2, ealu, malu, mmo, fwdb, db );  mux2x5 rn\_mux( inst[15:11], inst[20:16], regrt, drn );  wire dwmem\_tmp, dwreg\_tmp, dm2reg\_tmp, dshift\_tmp, daluimm\_tmp, djal\_tmp;  wire [3:0] daluc\_tmp;  sc\_cu cu(inst[31:26], inst[5:0], rsrtequ, dwmem\_tmp, dwreg\_tmp, regrt, dm2reg\_tmp, daluc\_tmp, dshift\_tmp, daluimm\_tmp,  pcsource, djal\_tmp, sext); //与单周期相同  assign dwreg = wpcir?dwreg\_tmp:1'b0;  assign dm2reg = wpcir?dm2reg\_tmp:1'b0;  assign dwmem = wpcir?dwmem\_tmp:1'b0;  assign daluimm = wpcir?daluimm\_tmp:1'b0;  assign dshift = wpcir?dshift\_tmp:1'b0;  assign djal = wpcir?djal\_tmp:1'b0;  assign daluc = wpcir?daluc\_tmp:4'b0;  //两个直通信号的处理  forward a(ewreg, mwreg, ern, mrn, mm2reg, em2reg, drs, fwda);  forward b(ewreg, mwreg, ern, mrn, mm2reg, em2reg, drt, fwdb); |

而直通信号处理模块forward则是根据结构图设计。没有数据冒险时fwda信号为00，ex段末尾的alu输出端fwda信号为01，mem末尾的alu输出端fwda信号为10，mem末尾的存储器输出端fwda信号为11，forward模块具体代码如下，与课程PPT中提供的fwda相关代码一致。

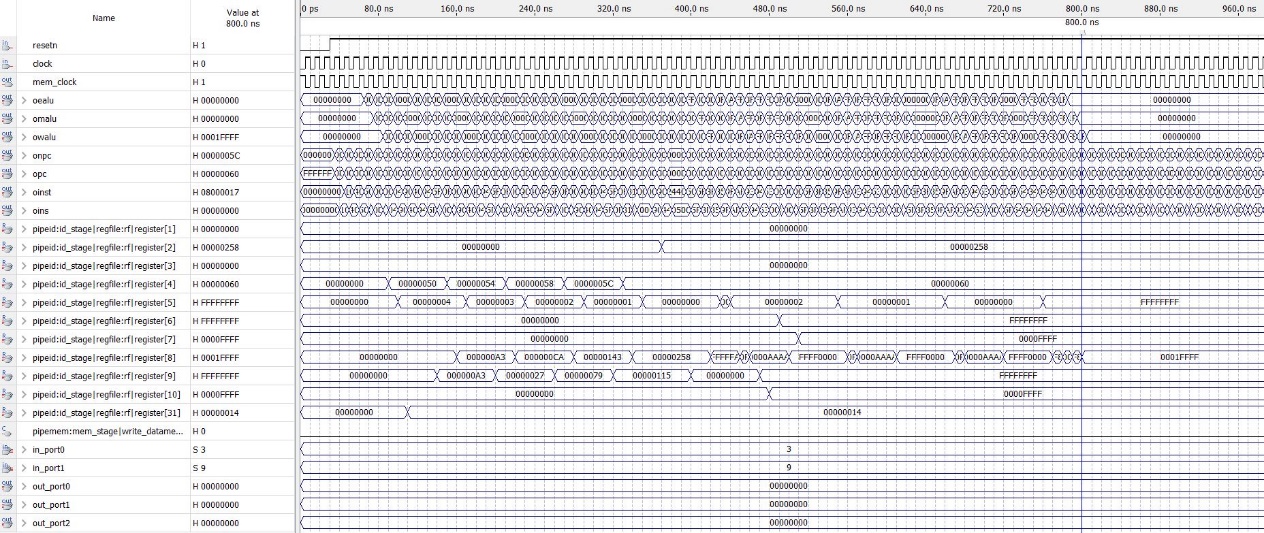
|  |
| --- |
| module forward(ewreg, mwreg, ern, mrn, mm2reg, em2reg, drs, fwda);  input ewreg, mwreg, mm2reg, em2reg;  input [4:0] ern, mrn, drs;  output [1:0] fwda;  reg [1:0] fwda;    always @(\*)  begin  fwda = 2'b00; //default forward a: no hazards  if(ewreg & (ern != 0) & (ern == drs) & ~ em2reg) begin  fwda<=2'b01; //select exe\_alu  end else begin  if (mwreg & (mrn != 0) & (mrn == drs) & ~ mm2reg) begin  fwda<=2'b10; //select mem\_alu  end else begin  if (mwreg & (mrn != 0) & (mrn == drs) & mm2reg) begin  fwda<=2'b11; //select mem\_lw  end  end  end  end  endmodule |

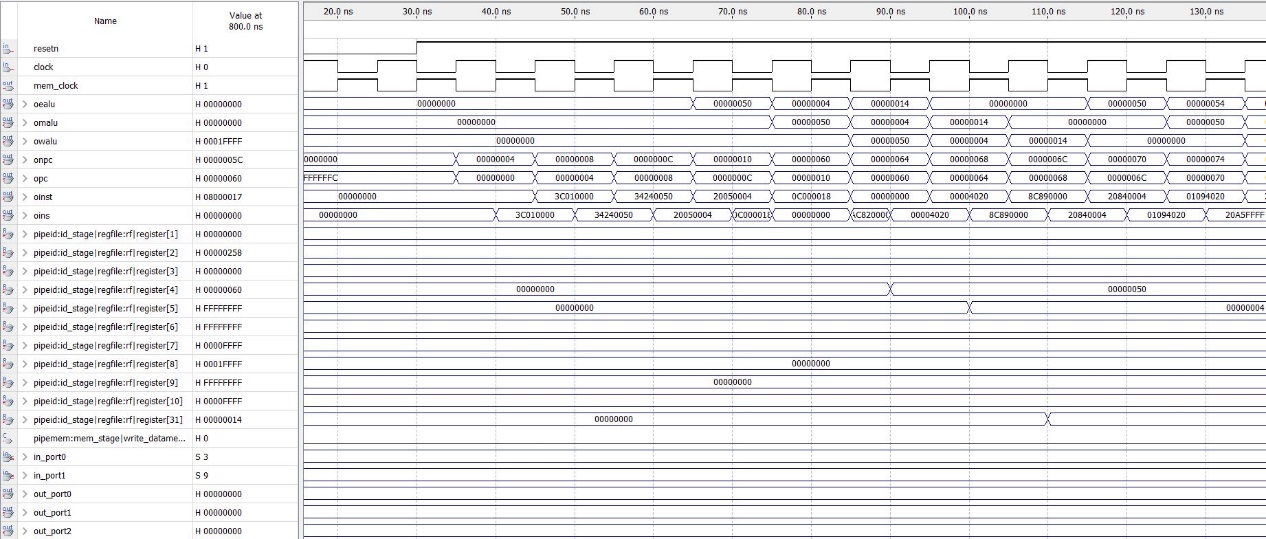
剩下的三个流水线寄存器pipedereg，pipeemreg和pipemwreg功能是相同的，都是将前一段的信号锁存并在时钟沿传递至下一段，而要传递的信号种类与结构图一致。只需要额外注意在resetn信号为0时保持为零即可。具体可见代码文件。

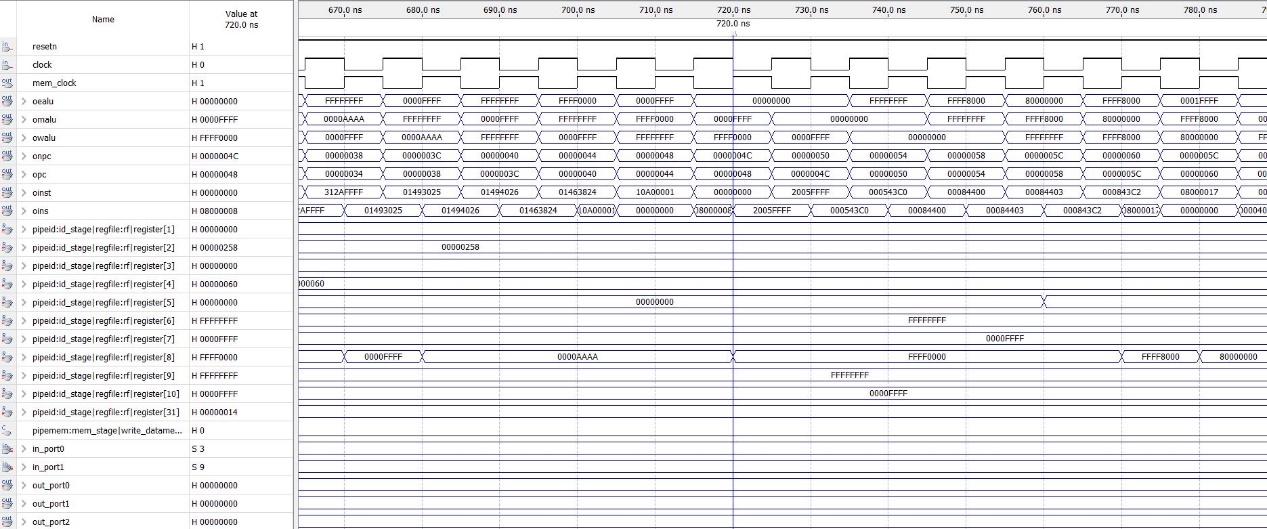
最后是pipeexe流水段pipemem流水段(writeback流水段仅包括一个二路多选器在此忽略)。基本上和单周期设计一致。变动的地方是exe段有一个pc+4的操作，mem段因为存储器时钟和主时钟反向，信号的确定以存储器时钟为准即可。

## 仿真结果展示

首先是整体图如下，在处理了控制冒险后，最后的指令到达死循环，R8最后的值停在0001FFFFH，R2最后的值为258，与提供的结果图一致。



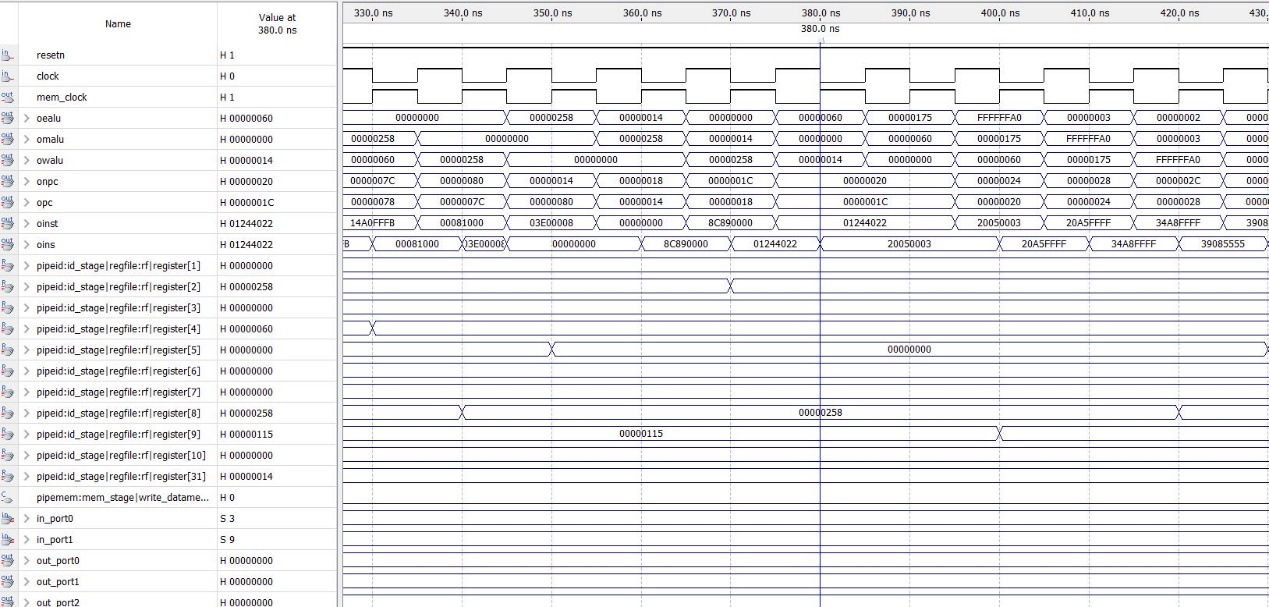
接下来是跳转指令相关的处理细节图。在这次实验中我是通过硬件实现跳转相关的判断和停顿，所以在图中反映出来就是跳转成功的指令后都会插入一条nop指令。以前四条指令为例，如下图。在70ns时，读到一条指令0c000018，接着在ID阶段译码后，其对应pcsource不是00，于是在IF阶段紧接着插入一条空指令00000000，从图中oinst可以看出，插入空指令后，下一条指令就是跳转地址对应的指令了。

对于设计指导中提到的例子，beq指令与j指令紧接着的情况，在插入空指令后同样可以得到解决，其过程与上图类似。在beq指令10a00001不满足跳转条件时，继续读取下一条跳转指令08000008并执行。在beq指令满足跳转条件后，插入一条空指令避免读取原本的下一条跳转指令，从而实现正确地跳转到2005ffff指令。相关细节图如下

还有一种停顿类型，即lw指令的下一条指令需要用到刚取出来的数据，此时则必须停顿一个周期。在测试的inst.mif中，相关指令为

|  |
| --- |
| 5 : 8c890000; % (14) lw $9, 0($4) #check sw %  6 : 01244022; % (18) sub $8, $9, $4 #sub: $8 <- $9 - $4 % |

而在图中的表示如下，可以看到，sub指令保持了两个周期不变，即停顿了一个周期。

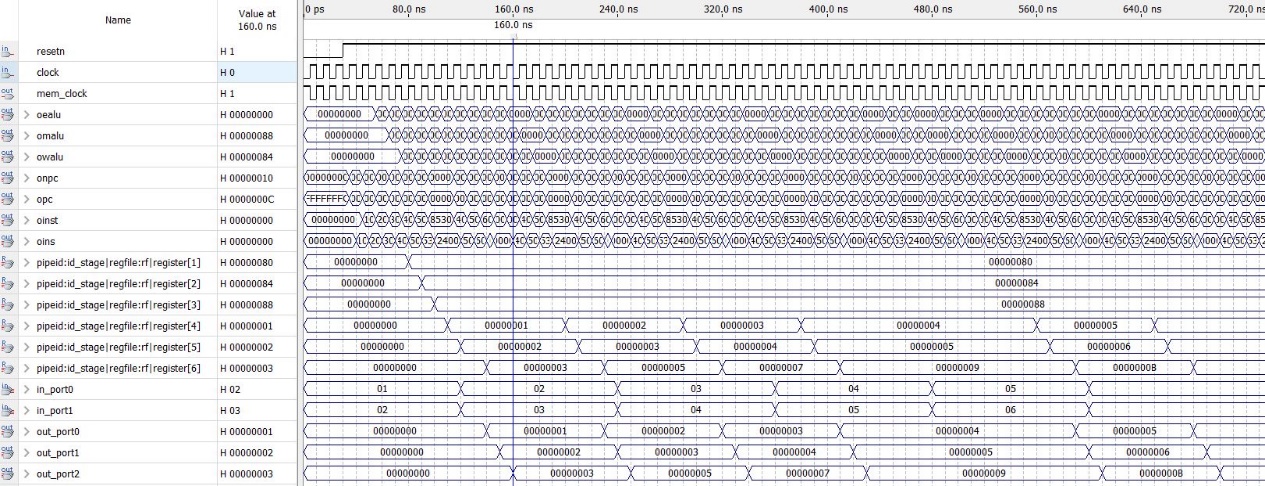


## IO结果展示

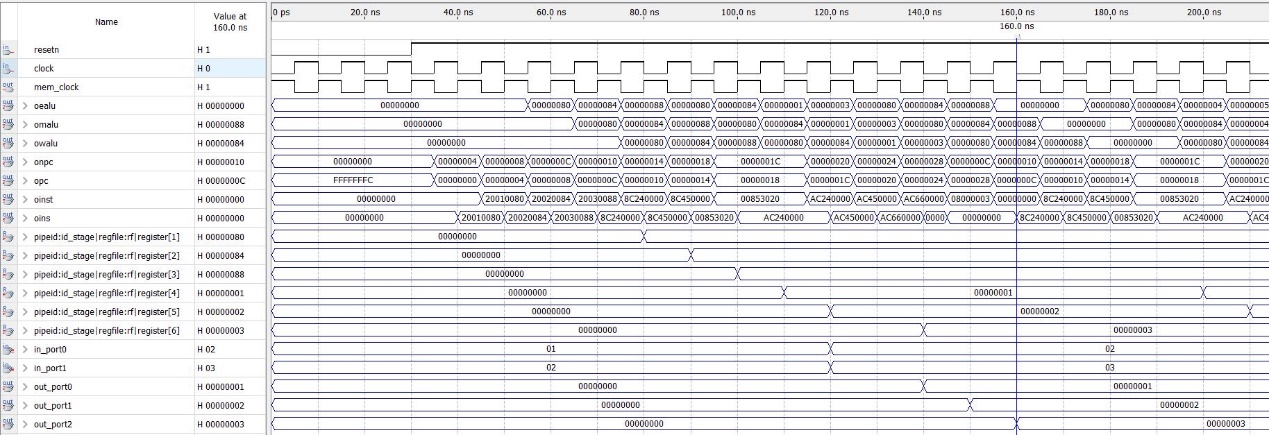
和设计说明一样，通过新建的vwf波形仿真文件，进行IO口的测试。测试指令如下

|  |
| --- |
| 0 : 20010080; % (00) main: addi $1, $0, 128 # outport0,inport0 %  1 : 20020084; % (04) addi $2, $0, 132 # outport1, inport1 %  2 : 20030088; % (08) addi $3, $0, 136 # outport2 %  3 : 8c240000; % (0c) loop: lw $4, 0($1) # input inport0 to $4 %  4 : 8c450000; % (10) lw $5, 0($2) # input inport1 to $5 %  5 : 00853020; % (14) add $6, $4, $5 # add inport0 with inport1 to $6 %  6 : ac240000; % (18) sw $4, 0($1) # output inport0 to outport0 %  7 : ac450000; % (1c) sw $5, 0($2) # output inport1 to outport1 %  8 : ac660000; % (20) sw $6, 0($3) # output result to outport2 %  9 : 08000003; % (24) j loop # % |

两个输入端口的输入数据分别为12345和23456，其结果总体图如下。



细节图如下，和设计指导一致。第一条写80H到R1的指令从40 ns开始执行，经过流水第五个clock后R1改变，110ns时候inport0的值1被读入 R4 120ns时候inport1的值2被读入R5，130ns没有动作，140 ns 时才将求和结果输出到 R6。这正是100ns时候流水线CPU为lw指令处理硬件上插入了一个 bubble引起的。(如果硬件上不做处理，要得到正确结果就需要通过软件的方式在第4行指令后添加 nop 产生一个bubble)。160ns时R6里的求和结果3被 输出到outport2端口。在140ns时执行跳转指令，循环执行第3-9行的loop代码段。



## 参考资料

GitHub链接：<https://github.com/shuiqinggang/EI332>

<https://github.com/Schemeer/EI332>