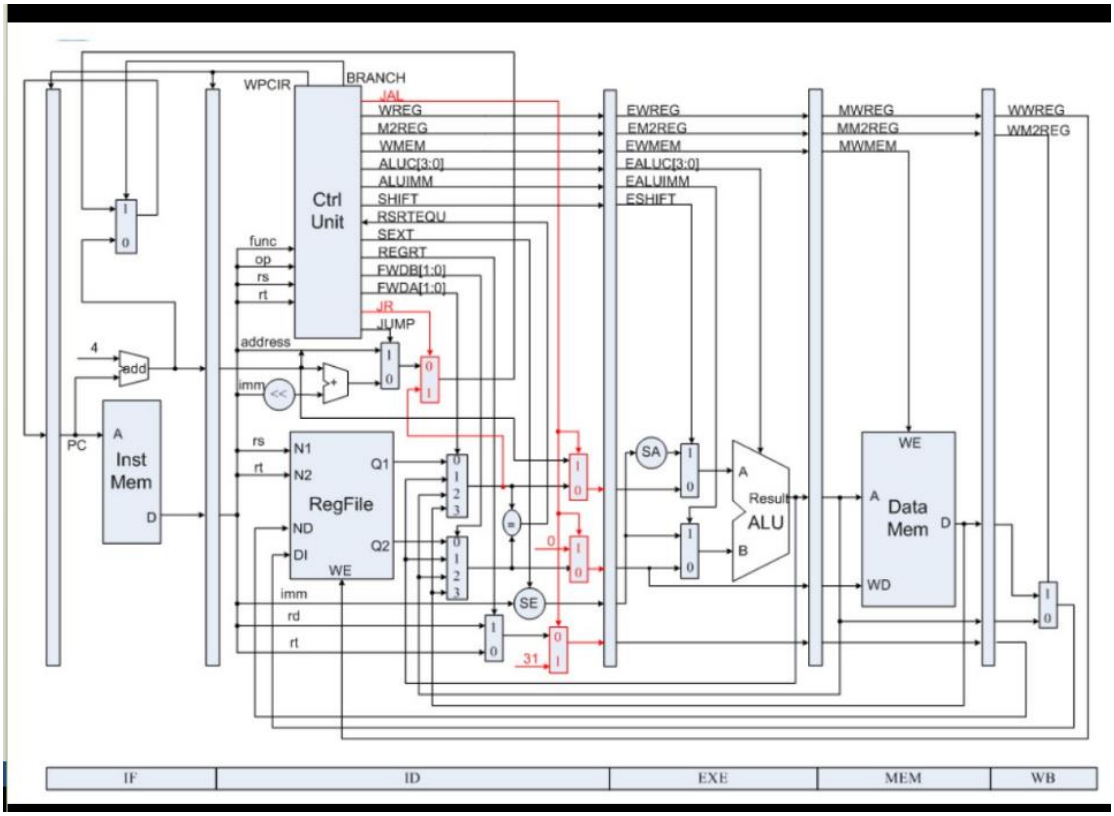
Document

The Appendix B&C in the textbook has discussed the principle of pipeline well but lack enough details about how to implement a pipeline CPU or how to deal with different hazards. Thus, this document plans to discuss some important details about how to implement the pipeline with Verilog.

下面是流水线 CPU 的基本图（此图在实验课件上有），我的设计是在此基础上进行修改的(因为这些信号不足以应对一些 hazard，所以我的多了几个 Ctrl Unit 信号)。



Component Function:

Latch: There are five Latches corresponding to each step. These Latches should update their data at the positive edge of CPU clock.
Inst_Mem: instruction memory which stores instructions

Ctrl Unit: control the data flow at the ID step and control whether the latches of previous two steps should stall.

RegFile: control the data flow of 32 registers.

SE: sign or zero extend unit which decides whether zero-extend immediate or sign-extend immediate should be output.

=: checks whether the value of rt is equal to the value of rd.

ALU: arithmetic unit which should do **16 kinds of operation**, including add, sub, xor ,or shift, nor and so on.

1. **Memory**

建议大家在 MEMORY 模块可以首先像上图一样分成 Instruction Memory 和 Data Memory 两块，最后再合成一个 Memory.这样一开始可以不考虑 Memory 部分的 structure hazard。当然实际上 Memory structure hazard 也不是很难处理，你也可以一开始就设计整个的。

Memory clock

Memory 的时钟不适合用 CPU 的时钟：如果选取 CPU 时钟作为 Memory 的时钟的话，那么时钟上升沿更新 PC 值得时候，由于 Memory 的时钟也是上升沿，所以 Memory 是无法马上在此周期内根据更新的 PC 值取出指令的。**因此解决方法是你可以用比 CPU 时钟更快的时钟作为 Memory 的时钟，或者将 Memory 的时钟设置为反 CPU 时钟。**前一种方法是指让 Memory 的时钟周期比 CPU 的短，从而能够在 CPU 的一个周期内将更新的 PC 值传入 Memory 中及时更新指令。后一种方法是指上升沿的时候锁存器更新 PC 值，随后在高电平的时候锁存器把更新的 PC 值传送给 Memory，然后下降沿的时候从 Memory 读取指令。**这两种方法同样适用于数据 Memory。**

2. **Register file clock**

结构冒险：读、写寄存器可能会同时存在。

解决方法：由于写是前面执行的指令的操作，所以应该首先更新，即应该先写后读。因此，**应该设置为 CPU 时钟下降沿的时候进行写寄存器**，即在下降沿的时候更新寄存器的值，随后的低电平传给 RegFile 的输出寄存器，上升沿到来的时候已经更新的输出寄存器的值可以传给下一级的锁存器。下面是我的参考代码（端口名与群上的示例代码有一些不一样）：

```
module Regs(
  input clk,
      input rst,
      input L_S,
      input[4:0] R_addr_A,
      input[4:0] R_addr_B,
      input[4:0] Wt_addr,
      input[31:0] Wt_data,
      output[31:0] rdata_A,
      output[31:0] rdata_B
       );
        reg [31:0] register [1:31];
      integer i;

      assign rdata_A = (R_addr_A == 0)? 0: register[R_addr_A];
      assign rdata_B = (R_addr_B == 0)? 0: register[R_addr_B];

      always @ (negedge clk or posedge rst) begin
            if(rst == 1)
                  for(i = 1; i < 32; i = i+1)
```

```
                    register[i] <= 0;
            else if((Wt_addr != 0) && (L_S == 1))
                register[Wt_addr] <= Wt_data;
        end

endmodule
```

Ctrl Unit

Control unit is the most important and hard part in the design of pipeline CPU, because it has to carefully deal with several kinds of hazards.

我的 Ctrl Unit 的设计大致如下：

```
module Ctrl(…..);
always@(posedge clk) begin
//对于目的寄存器的队列以及 opcode 队列的操作
end

always@(negedge clk)begin
//判断,状态转移以及控制数据冒险和控制冒险的操作
end
end module
```

2.1 Data Hazard

There are three kinds of data hazards, one-step-forward data hazard, two-step-forward data hazard and memory-register data hazard, respectively.

2.1.1 one-step-forward and two-step-forward data hazards

控制单元需要维持一个目的寄存器（也就是指令中需要写的寄存器 rd 或者 rt）编号的队列。我的实现当中用 REG_HEAP[14:0]做作为队列，如果指令中有写寄存器操作，那么 REG_HEAP[14:0] <= {REG_HEAP[9:0], rd/rt}，否则如果指令没有写寄存器或者当前是 stall 状态那么 REG_HEAP[14:0] <= {REG_HEAP[9:0], 5'b0}。

注意这里是非阻塞赋值，因此当新的一条指令进来时，REG_HEAP[4:0]是上一条指令的写寄存器，REG_HEAP[9:5]是上上一条指令的写寄存器，以此类推。

One-step-forward example:

add r1, r2, r3

add r4, r1, r2

The second instruction needs the value of the destination register of the first instruction, but the value of the register has not been updated.

Solution: control unit should compare the REG_HEAP[4:0] and rt, rs, then if they are equal and are not 5'b0， it means one-step-forward and you should set FWDA/FWDB.

Two-step-forward example:

add r1, r2, r3

add r4, r5, r6

add r7, r1, r5

The third instruction needs the value of the destination register of the first instruction, but the value of the register has not been updated.

Solution: control unit should compare the REG_HEAP[9:5] and rt, rs, then if they

are equal and are not 5'b0， it means two-step-forward and you should set FWDA/FWDB

NOTE:

One-step-forward is prior to two-step-forward, for example:

add $1, $2, $3

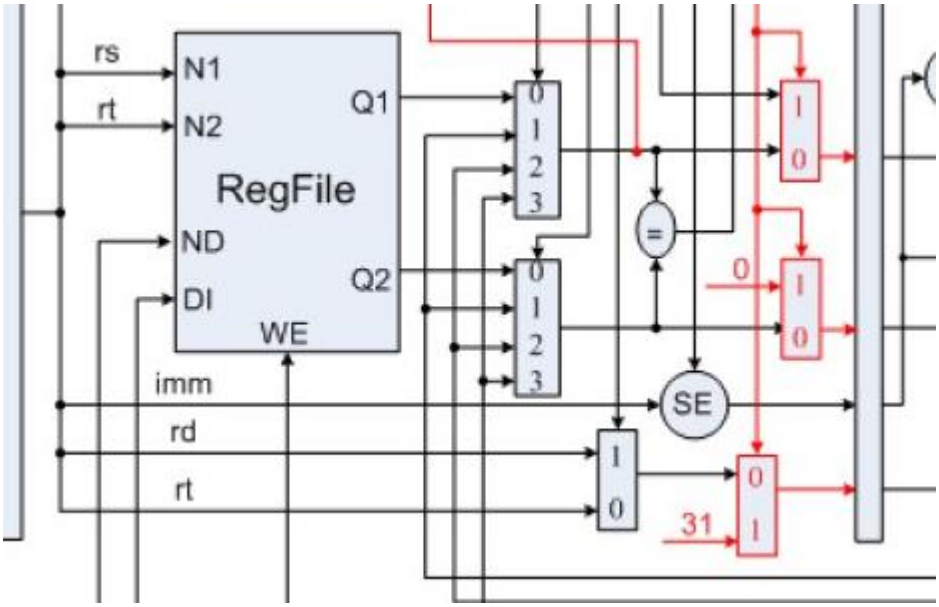
add $1, $3, $4

add $5, $1, $3

In this example, both one-step-forward and two-step-forward exists but we only do the one-step-forward instead of the two-step-forward for the third instruction.

2.1.2 memory-register hazard

There are two four-to-one multiplexers in ID step that help to choose which data the instruction should use.



Here the main problem is caused by the lw instruction. There are two kinds of hazards.

Example one:

lw $1, 200($0)

add $2, $1, $3

The second instruction has to wait for the result of the first instruction, and thus when the second instruction should wait in the ID step and then allow the first instruction to execute the EXE step. Then when the first instruction is in the MEM step, there will be a forward action in the ID step in order to choose the data from the data memory.

In conclusion, the solution is that when control unit finds that rs or rt in the ID step is the same as the destination register of previous lw instruction, then it should stall in the ID step for one clock and then forward to choose the data from memory.

Example two:

lw $1, 200($0)

add $2, $3, $4

add $5. $1, $6

The third instruction needs the value of the first instruction. Since when the third instruction is in ID step, the first instruction is in the MEM step, all we need to do is only to forward the data in the MEM step.

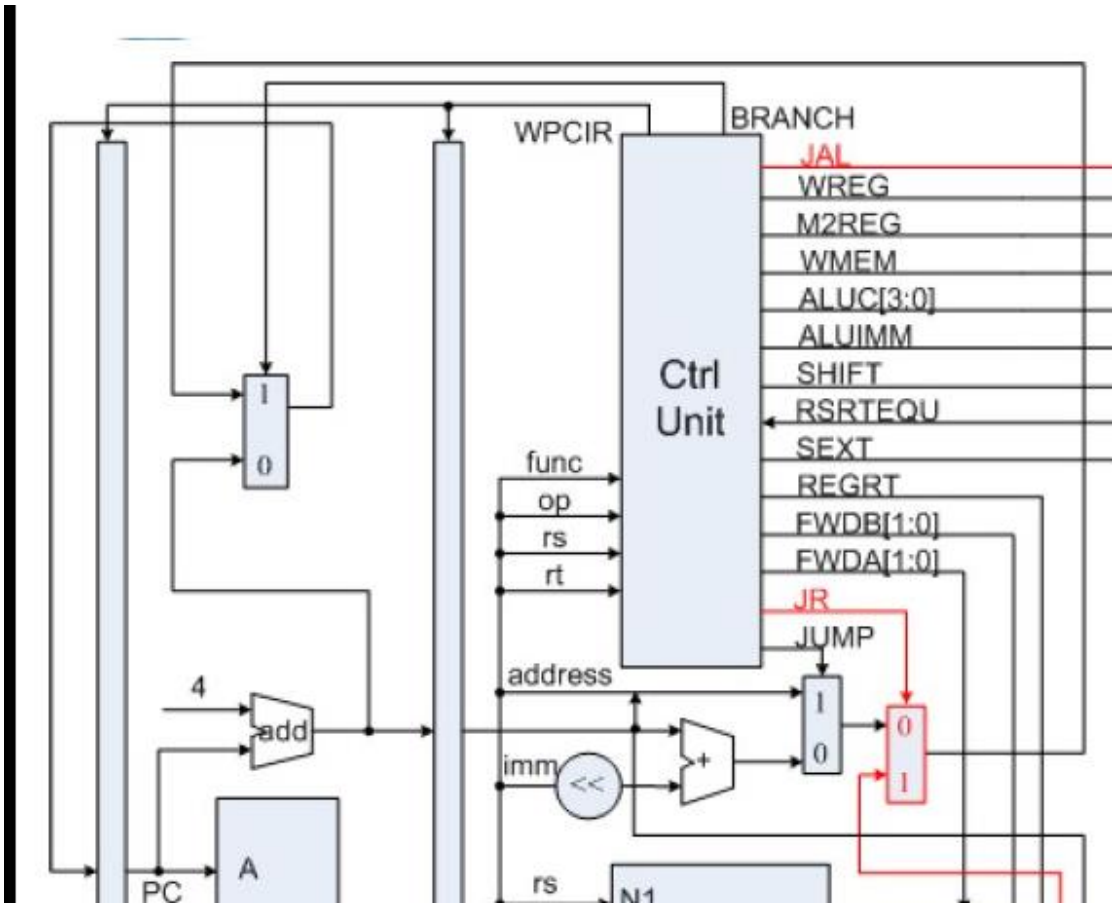
In conclusion, the solution is that when control unit finds that rs or rt in the ID step is the same as the destination register of the instruction before the previous instruction, then it should choose the data from the MEM step.

NOTE:

为了能够知道上一条指令以及上上条指令是什么需要在控制单元维持一个关于 opcode 的队列。操作与控制单元的寄存器号队列基本一样。

2.2.1 control hazard

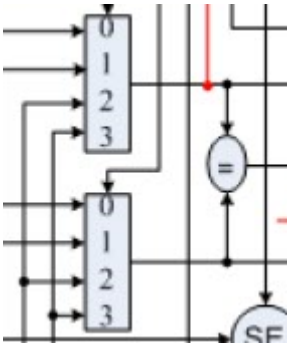
控制冒险主要由跳转指令引起，j, bne ,beq 的操作都类似。下面以 j 指令为示例进行讨论：



当在 ID step 发现 j 指令的时候，需要置位的信号有 JUMP=1, JR=0, BRANCH=1。另外需要让 IF step 进行 stall，所以需要将 WPCIR 置位 1. 当 j 指令过后的一个时钟控制单元需要重新将 WPCIR 置位 0.在此我是使用状态转移实现的。控制单元识别是 J 指令以后从正常执行状态跳转到 STALL 状态，然后在

STALL 状态中重新返回正常执行状态，这样就实现了一个时钟以后可以正常执行后面的指令。

bne 与 beq 指令与 j 指令类似。只是多了判断等于(=)单元的输出。如果条件符合则操作与 J 一致，否则正常执行后面的指令。



APPENDIX

All instructions you should implement:

ZheJiang University

MIPS Instructions

Bit #	31..26	25..21	20..16	15..11	10..6	5..0	Operations	
R-type	op	rs	rt	rd	sa	func		
add		rs	rt	rd	00000	100000	rd = rs + rt; with overflow	PC+=4
addu		rs	rt	rd	00000	100001	rd = rs + rt; without overflow	PC+=4
sub		rs	rt	rd	00000	100010	rd = rs - rt; with overflow	PC+=4
subu		rs	rt	rd	00000	100011	rd = rs - rt; without overflow	PC+=4
and		rs	rt	rd	00000	100100	rd = rs & rt;	PC+=4
or		rs	rt	rd	00000	100101	rd = rs | rt;	PC+=4
xor		rs	rt	rd	00000	100110	rd = rs ^ rt;	PC+=4
nor		rs	rt	rd	00000	100111	rd = ~(rs | rt);	PC+=4
slt	000000	rs	rt	rd	00000	101010	if(rs < rt)rd = 1; else rd = 0; <(signed)	PC+=4
sltu		rs	rt	rd	00000	101011	if(rs < rt)rd = 1; else rd = 0; <(unsigned)	PC+=4
sll		00000	rt	rd	sa	000000	rd = rt << sa;	PC+=4
srl		00000	rt	rd	sa	000010	rd = rt >> sa (logical);	PC+=4
sra		00000	rt	rd	sa	000011	rd = rt >> sa (arithmetic);	PC+=4
sllv		rs	rt	rd	00000	000100	rd = rt << rs;	PC+=4
srlv		rs	rt	rd	00000	000110	rd = rt >> rs (logical);	PC+=4
srav		rs	rt	rd	00000	000111	rd = rt >> rs(arithmetic);	PC+=4
jr		rs	00000	00000	00000	001000		PC=rs

ZheJiang University

MIPS Instructions

Bit #	31..26	25..21	20..16	15..11	10..6	5..0	Operations	
I-type	op	rs	rt	immediate				
addi	001000	rs	rt	imm			rt = rs + (sign_extend)imm; with overflow	PC+=4
addiu	001001	rs	rt	imm			rt = rs + (sign_extend)imm;without overflow	PC+=4
andi	001100	rs	rt	imm			rt = rs & (zero_extend)imm;	PC+=4
ori	001101	rs	rt	imm			rt = rs | (zero_extend)imm;	PC+=4
xori	001110	rs	rt	imm			rt = rs ^ (zero_extend)imm;	PC+=4
lui	001111	00000	rt	imm			rt = imm << 16;	PC+=4
lw	100011	rs	rt	imm			rt = memory[rs + (sign_extend)imm];	PC+=4
sw	101011	rs	rt	imm			memory[rs + (sign_extend)imm] <-- rt;	PC+=4
beq	000100	rs	rt	imm			if (rs == rt) PC+=4 + (sign_extend)imm <<2;	PC+=4
bne	000101	rs	rt	imm			if (rs != rt) PC+=4 + (sign_extend)imm <<2;	PC+=4
slti	001010	rs	rt	imm			if (rs < (sign_extend)imm) rt =1 else rt = 0; less than signed	PC+=4
sltiu	001011	rs	rt	imm			if (rs < (zero_extend)imm) rt =1 else rt = 0; less than unsigned	PC+=4
J-type	op	address						
j	000010	address					PC = (PC+4)[31..28],address<<2	
jal	000011	address					PC = (PC+4)[31..28],address<<2 ; $31 = PC+4	