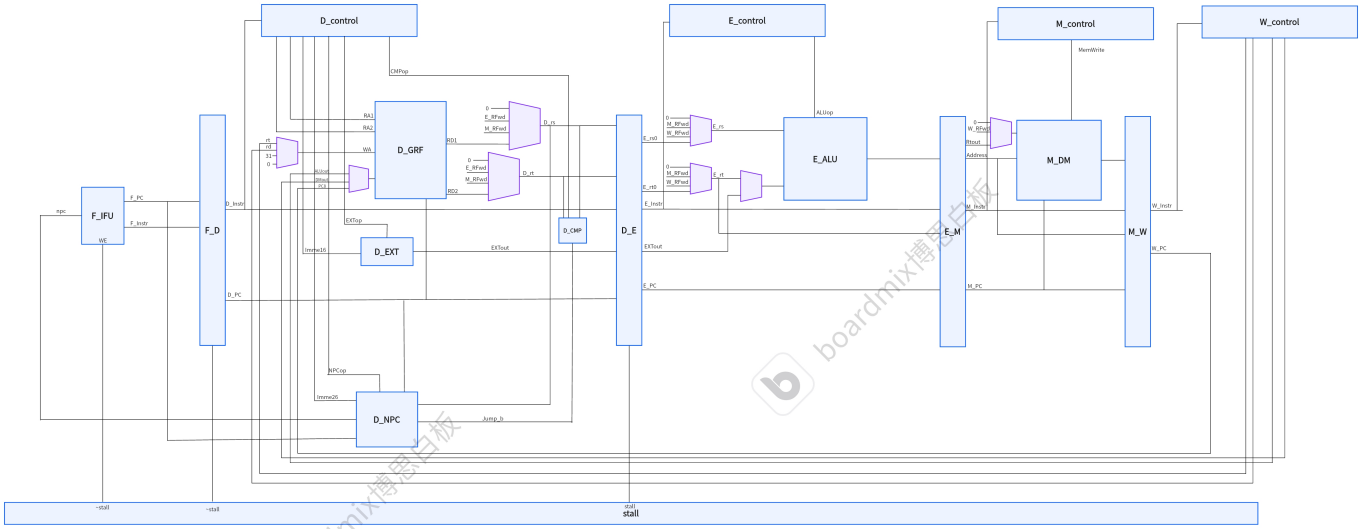
顶层设计



R型指令：

31:26	25:21	20:16	15:11	10:6	5:0
opcode	rs	rt	rd	shamt	func

I型指令：

31:26	25:21	20:16	15:0
opcode	rs(base)	rt	imme16(offset)

J型指令：

31:26	25:0
opcode	imme26(instr_index)

要实现的指令集： **R型指令：{add,sub,jr} I型指令：{ori,lui,beq,lw,sw} J型指令：{jal} 其他指令：{nop}**

命名规范：

1. 每个模块均使用大写字母命名，命名方式为模块所在层级+英文简称，比如E_ALU表示处在E级的ALU模块
2. 每个非流水线寄存器模块的端口信号均以大写字母开头，辅以小写字母，命名方式为英文简称（clk和reset除外）,对于流水线寄存器中需要进行流水的信号，输入信号用前一个阶段+信号简称命名，输入信号用后一个阶段+信号简称命名，比如F_PC、D_PC
3. 在mips.v中所有实例化模块均以模块命名的小写字母形式命名，比如e_alu,所有连线信号均以小写字母命名，命名方式为英文简称，英文简称有冲突时辅以英文说明
4. 四个流水线寄存器使用前后层级+REG命名，比如D_E_REG

主要部件

1. GRF

参与读写操作，比P4新增了内部转发功能，当读地址和写地址相同时，读出数据改写为写入数据，而非寄存器中的值，具体实现如下：

```
assign RD1 = (RA1 == 5'b00_000)?32'b00:
             ((RA1 == WA) && RegWrite)?WD:
             grf[RA1];
assign RD2 = (RA2 == 5'b00_000)?32'b00:
             ((RA2 == WA) && RegWrite)?WD:
             grf[RA2];
```

除此之外，还将RegWrite设置为1，而不是由控制器产生写寄存器的控制信号（事实上可以直接将这个端口删去），这是因为控制器能够产生针对不同指令的写寄存器的地址，对于不写寄存器的指令，直接将写入地址设置为0，并在GRF中规定无法对$0进行改写。

信号	方向	描述
clk	输入	时钟信号
reset	输入	异步复位信号
RegWrite	输入	控制信号，为真时GRF方可写入数据
RA1[4:0]	输入	读地址
RA2[4:0]	输入	读地址
WA[4:0]	输入	写地址
WD[31:0]	输入	要写入的数据
PC[31:0]	输入	当前指令的地址
RD1[31:0]	输出	读出的值
RD2[31:0]	输出	读出的值

2. IFU

信号	方向	描述
clk	输入	时钟信号
reset	输入	异步复位信号
enable	输入	使能端
NPC[31:0]	输入	下一条指令的地址
PC[31:0]	输入	指令地址

信号	方向	描述
Instr[31:0]	输出	指令

3. ALU

> 与P4不同的是，P5将两个操作数的比较功能单独拿出来形成一个比较模块，这是为了提前进行beq指令的比较，减少指令的浪费。

信号	方向	描述
A[31:0]	输入	第一个运算数
B[31:0]	输入	第二个运算数
ALUop[2:0]	输入	ALU的控制信号
ALUout[31:0]	输出	ALU的计算结果

4. DM

> 设计和P4相同
> 存储数据，用RAM实现，容量为3072×32bit，RAM在verilog中可以用reg型变量来实现
> 地址范围：0x00000000~0x00002FFF

信号	方向	描述
clk	输入	时钟信号
reset	输入	同步复位信号
MemWrite	输入	控制信号，为真时DM方可写入数据
WA[31:0]	输入	写入数据的地址
WD[31:0]	输入	要写入的数据
PC[31:0]	输入	当前指令的地址
RD[31:0]	输出	读出的数据

5. NPC

信号	方向	描述
Offset[15:0]	输入	16位运算数(beq)
Instr_index[25:0]	输入	26位运算数(jal)
NPCop[2:0]	输入	NPC的控制信号

信号	方向	描述
JrReg[31:0]	输入	jr指令存入PC的数据
F_PC[31:0]	输入	F级的PC值
D_PC[31:0]	输入	D级的PC值
Zero	输入	来自CMP模块，用于beq指令的判断
Npc[31:0]	输入	下一条指令的地址

6. CMP

P5新增模块，将ALU中的数值比较功能单拎出来，用于beq指令的判断

信号	方向	描述
RD1	输入	参与比较的第一个寄存器的值
RD2	输入	参与比较的第二个寄存器的值
CMPop[1:0]	输入	比较方式的选择信号
Jump_b	输出	两值相同置1，不同置0

7. EXT

设计和P4相同
将32位指令中的16位immediate符号扩展或者零扩展为32位
将16位immediate加载至高位，低位补0

信号	方向	描述
Imme16[15:0]	输入	需要被扩展的16位立即数
ExtOp	输入	控制信号，选择符号扩展还是零扩展
Imme32[31:0]	输出	扩展后的32位立即数

控制器设计

采用分布式控制器，将P4中的译码模块DIS并入控制器模块，由于各指令的译码也在控制器中实现，
因此各指令的Tuse和Tnew也在控制器中确定。

信号	方向	描述

信号	方向	描述
Instr[31:0]	输入	输入的32位指令信号
Rs[4:0]	输出	
Rt[4:0]	输出	
Rd[4:0]	输出	add,sub指令的目的寄存器
Shamt[4:0]	输出	
Offset[15:0]	输出	
Instr_index[25:0]	输出	
ALUout[2:0]	输出	ALU的控制信号
EXTop	输出	EXT的控制信号，1时符号扩展，反之进行0扩展
RegWrite	输出	GRF写使能信号，为1时方可写寄存器，具体实现时恒为1
NPCop[2:0]	输出	NPC控制信号
CMPop[1:0]	输出	CMP控制信号
MemWrite	输出	DM写使能信号，为1时方可写RAM
ALUSrc[2:0]	输出	选择ALU第二个数据的来源
FwSel[2:0]	输出	选择转发数据
RegWriteDst[4:0]	输出	每个指令写寄存器的地址
Tuse_rs[1:0]	输出	每个指令rs寄存器的Tuse
Tuse_rt[1:0]	输出	每个指令rt寄存器的Tuse
Tnew_E[1:0]	输出	每个指令在E级的Tuse
Tnew_M[1:0]	输出	每个指令在M级的Tuse

冲突解决

> 利用AT法解决
> Tuse是指某一指令在D级再经过多少个时钟周期就必须要使用相应的数据，Tnew是指位于某个流水级的某个指令，它经过多少个时钟周期可以算出结果并且存储到流水线寄存器里。

指令	A信号	$Tnew-E$	$Tnew-M$	$Tuse-rs$	$Tuse-rt$
add	rd	1	0	1	1
sub	rd	1	0	1	1
ori	rt	1	0	1	null

指令	A信号	$Tnew-E$	$Tnew-M$	$Tuse-rs$	$Tuse-rt$
lw	rt	2	1	1	null
sw	0	0	0	1	2
beq	0	0	0	0	0
lui	rt	1	0	1	null
jal	31	0	0	null	null
jr	0	0	0	0	null

A信号是该指令的目的寄存器，如没有则取0;null表示该指令不使用该寄存器，在具体实现时信号用可取的
最大值表示。
特别地，我的lui指令的Tnew-E为1，因为我将lui的计算并入了E级的ALU模块，而非在D级的EXT模块实现。

除了可以精确到指令进行设计以外，还可以对指令进行分类，同一类的指令具有相同的$Tnew$和$Tuse$,参考学长博客：

指令分类

类型名	信号名	数量	指令
算术逻辑寄存器	ALREG	16	_arith(算数)：add,addu,sub,subu _logic(逻辑):and,or,xor,nor _shift(移位):sll,srl,sra,sllv,srlv,srav _comp(比较):slt,sltu
算术逻辑立即数	ALIMM	7	_sign(有符号扩展):addi,addiu,slti,sltiu _zero(无符号扩展):andi,ori,xori
高位加载	EXTLUI	1	lui
两数比较分支	BRANCHE	2	beq,bne
与零比较分支	BRANCHZ	4	bltz,bgez,vlez,bgtz
仅读取跳转	JUMPR	1	jr
仅写入跳转	JUMPW	1	jal
读取写入跳转	JUMPRW	1	jalr
读取内存	LOAD	5	lw,lh,lb,lbu,lhu
写入内存	STORE	3	sw,sh,sb

指令的AT

信号名	$Tuse-rs$	$Tuse-rt$	$Tnew-E$	$Tnew-M$
ALREG	1	1	1	0

信号名	$Tuse-rs$	$Tuse-rt$	$Tnew-E$	$Tnew-M$
ALIMM	1	null	1	0
EXTLUI	1	null	1	0
BRANCHE	0	0	0	0
BRANCHZ	0	null	0	0
JUMPR	0	null	0	0
JUMPW	null	null	0	0
JUMPRW	0	null	0	0
LOAD	1	null	2	1
STORE	1	2	0	0

阻塞设计

> 设计专门的stall模块来处理合适要进行阻塞，stall模块会输出stall信号

首先要明确阻塞发生的条件：当冲突发生时，较旧的指令无法及时提供给较新的指令其所计算出来的值，便需要插入一个nop空泡，也就是使用阻塞，具体条件是**旧指令寄存器的写入地址和新指令寄存器的读出地址相同且读出地址不为0，而且旧指令的$Tnew$ $>$ 新指令的$Tuse$。** 课程要求**阻塞发生在D级**，因此当进行阻塞时，要将PC冻结，将F_D_REG的值冻结，将D_E_REG的值清零，也就是说我们要为IFU模块增加使能信号，为F_D_REG增加使能信号，为D_E_REG增加复位信号（异步复位），但是为了模块的统一性，我们为所有流水线寄存器都设置使能端`enable`和复位端`reset`,对于非上述两个流水线寄存器，使能信号恒为1，复位信号恒为0

如何实现阻塞?

将Tuse、Tnew和每个指令写寄存器的地址的获取集成到控制器中，在stall模块中示例化控制器获取每个指令的Tuse和Tnew以及写寄存器的地址，然后按照上述进行阻塞的条件生车工stall信号，具体代码实现如下：

```
    wire stall_E_rs = ((D_RegReadDst_rs != 5'b00_000) && (D_RegReadDst_rs ==
E_RegWriteDst) && (Tnew_E > Tuse_rs));
    wire stall_E_rt = ((D_RegReadDst_rt != 5'b00_000) && (D_RegReadDst_rt ==
E_RegWriteDst) && (Tnew_E > Tuse_rt));
    wire stall_M_rs = ((D_RegReadDst_rs != 5'b00_000) && (D_RegReadDst_rs ==
M_RegWriteDst) && (Tnew_M > Tuse_rs));
    wire stall_M_rt = ((D_RegReadDst_rt != 5'b00_000) && (D_RegReadDst_rt ==
M_RegWriteDst) && (Tnew_M > Tuse_rt));
    wire stall_E = stall_E_rs | stall_E_rt;
    wire stall_M = stall_M_rs | stall_M_rt;
    assign stall = stall_E | stall_M;
```

转发设计

和阻塞一样，首先要明确转发发生的条件。当一个较新的指令要使用的寄存器里的值还没有被较旧的指令写入到寄存器里面，但是已经被较旧的指令给算出来，这时候较新的指令便可以直接使用较旧的指令计算出来的值而不是使用寄存器里的值，这就是转发。

如何实现转发?

正如上文所说，所谓转发，就是新指令直接使用旧指令的结果，因此只需要比较新旧指令的目的寄存器的地址是否相同便可，相同而且$Tuse>=Tnew$便可以转发，否则便阻塞。 对于P5的指令集，写入寄存器的来源有三个：jal的PC+8，ALU的输出结果，DM的输出结果。而对于E级的转发只能是PC+8，因为这时候ALUout和DMout还没有产生，M级的转发有PC+8和ALUout，W级的转发有PC+8、ALUout、DMout,我们可以在控制器中增加 转发结果的选择信号，依照指令来选择转发结果，比如add就选择ALUout，lw就选择DMout，具体数据通路见思考题5。

延迟槽

清空延迟槽 在默认情况下，延迟槽中的指令都是默认执行的，但是有些指令（基本上都是分支跳转指令）在某些条件下不想让延迟槽中的指令执行，这就是清空延迟槽，比如bltzall，bgezall。

如何实现清空延迟槽?

法1：在CMP模块中新增一个输出flush，当判断为不需要进行跳转时，将flush置为1，并将F_D_REG清空，但是需要特别注意的是，当且仅当当前周期的阻塞信号为0时，我们才能执行清空延迟槽的操作。

省流：当且仅当当前周期的阻塞信号为0且CMP判断为需要清空延迟槽时，将F_D_REG清空。

具体如下：

```
F_D_REG_reset = (~stall && D_instr == new && flush);
```

法2：当监测到该指令不跳转时，F级的指令插入nop,具体如下：

```
F_instr = (!D_b_jump && D_bgezall) ? 32'd0 : im[pc[13:2] - 12'hc00];
```

思考题

1.我们使用提前分支判断的方法尽早产生结果来减少因不确定而带来的开销，但实际上这种方法并非总能提高效率，请从流水线冒险的角度思考其原因并给出一个指令序列的例子。

如果beq要比较的两个寄存器均和前面的指令产生数据相关，则不会提升效率，比如：

```
lw $1,0($3)
lw $2,0($4)
beq $1,$2,lable
```

此时必须等到第二条lw向$2中存入数据后才能进行判断，只能阻塞。

2.因为延迟槽的存在，对于 jal 等需要将指令地址写入寄存器的指令，要写回 PC + 8，请思考为什么这样设计?

> 延迟槽的作用就是无论是否跳转，都执行跳转指令的下一个指令，而这一条指令的地址就是PC + 4，跳转应该返回这条指令的下一条指令的地址，也就是PC + 8

3.我们要求大家所有转发数据都来源于流水寄存器而不能是功能部件（如 DM 、 ALU ），请思考为什么?

> 由于存在毛刺现象，由功能部件生成的数据并不稳定，直接使用功能部件产生的数据可能会出现错误，而流水线寄存器提供的数据则比较稳定，能够保证数据的正确性。

4.我们为什么要使用 GPR 内部转发？该如何实现?

> 使用GRF内部转发可以解决从W级向D级转发的问题，W级的所有指令的Tuse都是0，因此都可以进行转发。
> 实现方式如下：

```
assign RD1 = (RA1 == 5'b00_000)?32'b00:
             ((RA1 == WA) && RegWrite)?WD:
             grf[RA1];
assign RD2 = (RA2 == 5'b00_000)?32'b00:
             ((RA2 == WA) && RegWrite)?WD:
             grf[RA2];
//与P4相比新增写入地址和读出地址的相等判断，相等则进行转发，不等则输出寄存器的值
```

5.我们转发时数据的需求者和供给者可能来源于哪些位置？共有哪些转发数据通路?

> 需求者：D级GRF的两个输出端口，E级ALU的两个输入端口，M级DM的输入端口。
> 供给者：E级的PC+8，M级的PC+8和ALUout,W级的PC+8、ALUout和DMout
> 转发数据流：E_RFwd -> D_RD1
> E_RFwd -> D_RD2
> M_RFwd -> D_RD1
> M_RFwd -> D_RD2
> W_RFwd -> D_RD1 //GRF内部转发
> W_RFwd -> D_RD2 //GRF内部转发
> M_RFwd -> E_ALUinA
> M_RFwd -> E_ALUinB
> W_RFwd -> E_ALUinA

```
W_RFwd -> E_ALUinB
W_RFwd -> M_DMin
```

6.在课上测试时，我们需要你现场实现新的指令，对于这些新的指令，你可能需要在原有的数据通路上做哪些扩展或修改？提示：你可以对指令进行分类，思考每一类指令可能修改或扩展哪些位置。

指令类型	修改内容
算术逻辑寄存器	ALU模块增加相应运算 控制器增加相应的ALUop
算术逻辑立即数	ALU模块增加相应运算 控制器增加相应的ALUop和EXTop EXT增加相应扩展方式
立即数加载	ALU模块增加相应运算 控制器增加相应的ALUop
比较和跳转	CMP模块增加相应的比较方式 控制器增加相应的NPC和CMP控制信号
读内存	DM模块增加相应的读取方式 控制器增加相应的DM控制信号
写内存	DM模块增加相应的读取方式 控制器增加相应的DM控制信号

7.确定你的译码方式，简要描述你的译码器架构，并思考该架构的优势以及不足。

我采用分布式译码
只写一个主控制器，在D、E、M、W级各实例化一个分支控制器，每个分支控制器仅生成该级的控制信号。
优势：由于仅需要流水指令，减少了流水信号的数量；提高了代码的可读性
不足：资源使用率不如集中式控制器