- **P5_CPU**
  - 流水线架构
    - 取指阶段（Fetch）
      - PC
      - im.v
        - 端口定义
      - npc.v
        - 端口定义
      - d_reg.v
        - 端口定义
    - 译码阶段（Decord）
      - Stall
      - ctrl.v
        - 端口定义
        - 控制信号
      - grf.v
        - 端口定义
      - ext.v
        - 端口定义
      - cmp.v
        - 端口定义
      - e_reg.v
        - 端口定义
    - 执行阶段（Excute）
      - alu.v
        - 端口定义
      - m_meg.v
        - 端口定义
      - dm.v
        - 端口定义
      - w_reg.v
        - 端口定义
  - 流水线冒险
    - 冒险的种类
      - 结构冒险
      - 控制冒险
      - 数据冒险
  - 思考题
  - 课上指令
    - BONALL
    - 由补码求原码
  - 测试数据
    - M <- W
    - E <- W
    - D <- W
    - E <- M
    - D <- M
    - D <- E
    - 代码

P5_CPU

流水线架构

阶段	简称	功能概述
取指阶段（Fetch）	F	从指令存储器中读取指令
译码阶段（Decode）	D	从寄存器文件中读取源操作数并对指令译码以便得到控制信号
执行阶段（Execute）	E	使用 ALU 执行计算
存储阶段（Memory）	M	读或写数据存储器
写回阶段（Writeback）	W	将结果写回到寄存器文件

- 流水线寄存器以其供给数据的流水级的简称命名

取指阶段（Fetch）

PC

这里的 PC 采用下面这种方式直接**在顶层模块** $mips$ **中**实现：

```
    reg [31:0] PC;

    initial begin
        PC = 32'h00003000;
    end

    always @(posedge clk) begin
        if (reset) begin
            PC <= 32'h00003000;
        end
        else begin
            PC <= NPC_NPC;
        end
    end
```

im.v

端口定义

名称	描述	位宽	方向
PC	当前指令地址	32	I
instr	取出的指令	32	O

npc.v

端口定义

名称	描述	位宽	方向
Stall	暂停信号	1	I
PC	当前指令地址	32	I
imm26	26位立即数	26	I
EXT	位扩展器结果	32	I
ra	寄存器的值	32	I
nPC_sel	跳转控制信号	2	I
isSame	相等比较信号	1	I
isNega	相反比较信号	1	I
NPC	下一条指令地址	32	O
flush	清空延迟槽信号	1	O

- 若暂停信号 $Stall$ 有效，则 NPC 保持不变
- 若对应指令满足清空延迟槽条件，则输出清空延迟槽信号 $flush$ 为 1

d_reg.v

端口定义

名称	描述	位宽	方向
clk	时钟信号	1	I
reset	同步复位信号	1	I
Stall	暂停信号	1	I
flush	清空延迟槽	1	I
instr_in	F级指令	32	I
PC_in	F级PC	32	I
instr_out	D级指令	32	O
PC_out	D级PC	32	O

- 若暂停信号 $Stall$ 有效，则 $instr\_out$ 与 PC_out 保持不变
- 若清空延迟槽信号 $flush$ 有效**且暂停信号** $Stall$ **无效**，则清空 D 级寄存器

译码阶段（Decord）

Stall

当一个指令到达 D 级后，我们需要将它的 T_{use} 值与后面每一级的 T_{new} 进行比较，并进行 A 值的校验。

- 当 $T_{use} \geq T_{new}$ 时，我们可以通过**转发**来解决冒险
- 当 $T_{use} < T_{new}$ 时，我们就需要**阻塞**流水线

```
    assign Stall_RS = (((T_use_rs < T_new_E) & (rs == A3_E) & RegWrite_E) |
                       ((T_use_rs < T_new_M) & (rs == A3_M) & RegWrite_M)) &
                        (rs != 5'd0);
    assign Stall_RT = (((T_use_rt < T_new_E) & (rt == A3_E) & RegWrite_E) |
                       ((T_use_rt < T_new_M) & (rt == A3_M) & RegWrite_M)) &
                        (rt != 5'd0);
    assign Stall = Stall_RS | Stall_RT;
```

ctrl.v

端口定义

名称	描述	位宽	方向
op	指令操作码	6	I
funct	指令功能码	6	I
RegDst	GRF写地址选择信号	2	O
ExtOP	位扩展控制码	2	O
ALUSrc	ALU操作数B选择信号	1	O
ALUOP	ALU控制码	4	O
RegWrite	寄存器写使能	1	O
MemtoReg	寄存器写数据选择信号	2	O
storeOP	数据存储器写数据控制信号	2	O
DextOP	数据存储器数据扩展信号	3	O
nPC_sel	跳转控制信号	3	O
T_use_rs	GPR[rs]的需求时间	2	O
T_use_rt	GPR[rt]的需求时间	2	O

- 这里采用了**控制信号驱动型**的**集中式译码**

控制信号

	add(addu)	sub(subu)	and(or)	slt(sltu)	addi(addiu)	andi(ori)	lui	lw	sw
RegDst	01	01	01	01	00	00	00	00	X
ExtOp	X	X	X	X	SIGN	ZERO	HIGN	SIGN	SIGN
ALUSrc	0	0	0	0	1	1	1	1	1
ALUOP	ADD	SUB	AND(OR)	SLT(SLTU)	ADD	AND(OR)	ADD	ADD	ADD
RegWrite	1	1	1	1	1	1	1	1	0
MemWrite	0	0	0	0	0	0	0	0	1
MemtoReg	00	00	00	00	00	00	00	01	X
nPC_sel	PC4	PC4	PC4	PC4	PC4	PC4	PC4	PC4	PC4
T_use_rs	1	1	1	1	1	1	3	1	1
T_use_rt	1	1	1	1	3	3	3	3	2

- 信号 $storeOP$ 只对指令 $sw(2'd1), sh(2'd2), sb(2'd3)$ 生效，未列于表中
- 信号 $DextOP$ 只对指令 $lw(dext\_lw), lh(dext\_lh), lhu(dext\_lhu), lb(dext\_lb), lbu(dext\_lbu)$ 生效，未列于表中
- 指令的需求时间 T_{use} 是指这条指令位于 D 级的时候，**再经过多少个时钟周期就必须要使用相应的数据**
- 实际上，T_{use} 的最大值为 $2'd2$，当指令用不到 $GPR[rs]$ 或 $GPR[rt]$ 时，我们将对应的置 T_{use} 为 $2'd3$，这并不会影响我们对转发和暂停的判断

grf.v

端口定义

名称	描述	位宽	方向
clk	时钟信号	1	I
reset	同步复位信号	1	I
RegWrite	寄存器写使能	1	I
PC	当前指令的地址	32	I
A1	读地址1	5	I
A2	读地址2	5	I
A3	读地址3	5	I
WD	写数据	32	I
RD1	输出A1指定的寄存器数据	32	O
RD2	输出A2指定的寄存器数据	32	O

ext.v

端口定义

名称	描述	位宽	方向
imm16	16位立即数	16	I
ExtOP	位扩展控制码	2	I
EXT_Result	位扩展结果	32	O

cmp.v

端口定义

名称	描述	位宽	方向
GRF_RD1	GRF读数据1	32	I
GRF_RD2	GRF读数据2	32	I
isSame	相等比较信号	1	O
isNega	相反比较信号	1	O

- 这里添加一个 CMP 模块是为了将 B **类跳转指令的**判定**提前至** D **级**来进行，不再使用 E 级的 ALU 来进行判定

e_reg.v

端口定义

名称	描述	位宽	方向
clk	时钟信号	1	I
reset	同步复位信号	1	I
Stall	暂停信号	1	I
PC_in	D级PC	32	I
op	D级指令操作码	6	I
funct	D级指令功能码	6	I
shamt_in	D级shamt	5	I
ALUSrc_in	D级ALUSrc	1	I
ALUOP_in	D级ALUOP	4	I
RegWrite_in	D级RegWrite	1	I
MemtoReg_in	D级MemtoReg	2	I
storeOP_in	D级storeOP	2	I

名称	描述	位宽	方向
DextOP_in	D级DextOP	3	I
A1_in	D级寄存器读地址1	5	I
A2_in	D级寄存器读地址2	5	I
GRF_RD1_in	D级寄存器读数据1	32	I
GRF_RD2_in	D级寄存器读数据2	32	I
A3_in	D级寄存器写地址	5	I
EXT_Result_in	D级位扩展结果	32	I
PC_out	E级PC	32	O
shamt_out	E级shamt	5	O
T_new	E级指令供给时间	2	O
ALUSrc_out	E级ALUSrc	1	O
ALUOP_out	E级ALUOP	4	O
RegWrite_out	E级RegWrite	1	O
MemtoReg_out	E级MemtoReg	2	O
storeOP	E级storeOP	2	O
DextOP	E级DextOP	3	O
A1_out	E级寄存器读地址1	5	O
A2_out	E级寄存器读地址2	5	O
GRF_RD1_out	E级寄存器读数据1	32	O
GRF_RD2_out	E级寄存器读数据2	32	O
A3_out	E级寄存器写地址	5	O
EXT_Result_out	E级位扩展结果	32	O

- 指令的供给时间 T_{new} 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**存储到流水级寄存器里**
- 若暂停信号 $Stall$ 有效，则清空 E_REG (效果同 $reset$)

执行阶段（Excute）

alu.v

端口定义

名称	描述	位宽	方向
A	操作数A	32	I
B	操作数B	32	I
shamt	移位数	5	I
ALUOp	ALU控制码	4	I
C	运算结果	32	O
isZero	零判断	1	O

m_meg.v

端口定义

名称	描述	位宽	方向
clk	时钟信号	1	I
reset	同步复位信号	1	I
PC_in	E级PC	32	I
T_new_in	E级指令供给时间	2	I
RegWrite_in	E级RegWrite	1	I
MemtoReg_in	E级MemtoReg	2	I
storeOP_in	E级storeOP	2	I
DextOP_in	E级DextOP	3	I
A1_in	E级寄存器读地址1	5	I
A2_in	E级寄存器读地址2	5	I
A3_in	E级寄存器写地址	5	I
ALU_C_in	E级ALU计算结果	32	I
GRF_RD2_in	E级寄存器读数据2	32	I
PC_out	M级PC	32	O
T_new_out	M级指令供给时间	2	O
RegWrite_out	M级RegWrite	1	O
MemtoReg_out	M级MemtoReg	2	O
storeOP_out	M级storeOP	2	O
DextOP_out	M级DextOP	3	O

名称	描述	位宽	方向
A1_out	M级寄存器读地址1	5	O
A2_out	M级寄存器读地址2	5	O
A3_out	M级寄存器写地址	5	O
ALU_C_out	M级ALU计算结果	32	O
GRF_RD2_out	M级寄存器读数据2	32	O

存储阶段（Memory)

dm.v

端口定义

名称	描述	位宽	方向
clk	时钟信号	1	I
reset	同步复位信号	1	I
PC	M级PC	32	I
A	数据存储器读写地址	5	I
WD	DM写数据	2	I
byteen	数据存储器字节写使能	4	I
RD	DM读数据	32	O

- 这里可以用 $|byteen$ 来代替原先的数据存储器写使能信号 $MemWrite$

w_reg.v

端口定义

名称	描述	位宽	方向
clk	时钟信号	1	I
reset	同步复位信号	1	I
PC_in	M级PC	32	I
T_new_in	M级指令供给时间	2	I
RegWrite_in	M级RegWrite	1	I
MemtoReg_in	M级MemtoReg	2	I
A3_in	M级寄存器写地址	5	I
ALU_C_in	M级ALU计算结果	32	I

名称	描述	位宽	方向
DM_RD_in	M级DM读数据	32	I
PC_out	W级PC	32	O
T_new_out	W级指令供给时间	2	O
RegWrite_out	W级RegWrite	1	O
MemtoReg_out	W级MemtoReg	2	O
A3_out	W级寄存器写地址	5	O
ALU_C_out	W级ALU计算结果	32	O
DM_RD_out	W级DM读数据	32	O

流水线冒险

冒险的种类

结构冒险

结构冒险是指**不同指令同时需要使用同一资源**的情况。例如在普林斯顿结构中，指令存储器和数据存储器是同一存储器，在取指阶段和存储阶段都需要使用这个存储器，这时便产生了结构冒险。我们的实验采用哈佛体系结构，将指令存储器和数据存储器分开，因此不存在这种结构冒险。

另一种结构冒险在于寄存器文件需要在 D 级和 W 级同时被使用（读写）时并且读和写的寄存器为同一个寄存器时。

控制冒险

控制冒险，是指**分支指令（如 beq ）的判断结果会影响接下来指令的执行流**的情况。在判断结果产生之前，我们无法预测分支是否会发生。然而，此时流水线还会继续取指，让后续指令进入流水线。这时就有可能导致错误的产生，即不该被执行的指令进入到了指令的执行流中。

数据冒险

流水线之所以会产生数据冒险，就是因为后面指令需求的数据，正好就是前面指令供给的数据，而后面指令在需要使用数据时，前面供给的数据还没有存入寄存器堆，从而导致后面的指令不能正常地读取到正确的数据。

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。

答：将分支判断提前至 D 级进行时，我们常常需要利用 E 级 ALU 计算的结果，而这个结果在下一个周期才能存入 E 级寄存器，这时会使得流水线被**阻塞**一个周期，流水线的效率受到了一定的影响。

```
ori  $t0, $0, 0x1
beq  $t0, $0, loop
nop
loop:
...
```

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?

答：因为地址为 $PC+4$ 的指令位于延迟槽中，一定会被执行，跳转时要跳转到的指令地址为其后一条指令，即地址为 $PC+8$ 的指令。

*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?*

答：若不这样做，会导致我们的流水线不能正常地**并行**运行，而是会将某两个流水级**串行**运行。这样降低了流水线工作的效率。

4、我们为什么要使用 GPR 内部转发？该如何实现?

答：因为 GPR 既属于 D 级，也属于 W 级。当一条指令在 D 级需要读出寄存器内的数据时，若此时位于 W 级的指令正在写寄存器，就可以利用 GPR 的内部转发将这个写数据直接读出。

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?

答：需求者可能来源于 D 、 E 、 M 级，供给者可能来源于 E 、 M 、 W 级。

转发数据通路有： $D<-E$, $D<-M$, $D<-W$, $E<-M$, $E<-W$, $M<-W$ 。

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。

答：课上指令大致可以分为计算类指令、条件跳转类指令、条件访存类指令。

- 对于计算类指令，一般情况下只需要在译码时添加一个 $ALUOp$ ，在 ALU 中来实现其运算操作即可。
- 对于条件跳转类指令，可以在译码时添加一个 nPC_sel ，在 NPC 模块中实现其对应的 NPC 值，必要时应在 NPC 以及 CMP 模块中加入一些控制信号来辅助判断。比如可以添加 $flush$ 信号来判断是否需要清空延迟槽。
- 对于条件访存类指令，可能会增加 $GPR[rs]$ 值的流水，可能会因为写入的不确定性导致转发数据需要额外的判断。

7、简要描述你的译码器架构，并思考该架构的优势以及不足。

答：我的译码器采用的是**控制信号驱动型**的**集中式译码**。

优势为只需要在 D 级进行一次译码即可，比较简单粗暴，且指令数较多时代码量不见得很多。

不足为后续每一级流水级寄存器都要传递大量的控制信号，写起来比较繁琐复杂~~（已经深刻体会到子）~~。

课上指令

BONALL

先**链接**。如果 $GPR[rs]$ 和 $GPR[rt]$ 互为相反数，则跳转，否则**清空延迟槽**。

```
ori  $7, $0, 0x1
ori  $8, $0, 0x2
sub  $9, $7, $8
beq  $9, $7, loop1
ori  $10, $0, 0x9999
ori  $10, $0, 0x8888
loop1:
ori  $10, $0, 0x7777
nop
nop
nop
beq  $9, $8, loop2
ori  $11, $0, 0x9999
ori  $11, $0, 0x8888
loop2:
ori  $11, $0, 0x7777
nop
```

由补码求原码

```
    if ($signed(A) > 0) begin
        Result = A;
    end
    else if (A == 32'b0) begin
        Result = 32'h8000_0000;
    end
    else begin
        Result = {A[31], ~(A[30:0] - 31'h1)};
    end
```

测试数据

接收新数据的流水级<-发射新数据的流水级

M <- W

M级要接收数据：

- FW_DM_WD_out

W级可发射数据：

- PC_W + 32'h8
- ALU_C_W

E <- W

E级要接收数据：

- FW_ALUA_out
- FW_ALUB_out

W级可发射数据：

- PC_W + 32'h8
- ALU_C_W

D <- W

D级要接收数据：

- FW_RD1_out
- FW_RD2_out

W级可发射数据：

- PC_W + 32'h8
- ALU_C_W

E <- M

E级要接收数据：

- FW_ALUA_out
- FW_ALUB_out

M级可发射数据：

- PC_M + 32'h8
- ALU_C_M

D <- M

D级要接收数据：

- FW_RD1_out
- FW_RD2_out

M级可发射数据：

- PC_M + 32'h8
- ALU_C_M

D <- E

D级要接收数据：

- FW_RD1_out
- FW_RD2_out

E级可发射数据：

- PC_E + 32'h8
- EXT_Result_E

代码

```
######## M <- W #########
# FW_DM_WD_out <- ALU_C_W
ori $t1, $0, 0x1
nop
nop
nop
add $t1, $t1, $t1
sw $t1, 0($0)
nop
nop
nop
# FW_DM_WD_out <- PC_W + 32'h8
ori $t1, $0, 0x1
nop
nop
nop
jal test_m_w
sw $ra, 0($0)
ori $t1, $0, 0x2
j test_m_w_end
nop
test_m_w:
ori $t1, $0, 0x3
jr $ra
nop
nop
nop
test_m_w_end:
nop
nop
nop
######## E <- W #########
# FW_ALU_out <- ALU_C_W
ori $t1, $0, 0x1
nop
nop
nop
add $t2, $t1, $t1
nop
add $t3, $t2, $t2
nop
nop
nop
# FW_ALU_out <- PC_W + 32'h8
jal test_e_w
nop
j test_e_w_end
nop
test_e_w:
add $t4, $ra, $ra
nop
nop
nop
```

```
jr $ra
nop
test_e_w_end:
nop
nop
nop
########## D <- W #########
# FW_RD12_out <- ALU_C_W
ori $t1, $0, 0x1111
nop
nop
nop
ori $t2, $0, 0x2222
nop
nop
nop
add $t3, $t1, $t1 # W
nop # M
nop # E
beq $t3, $t2, test_d_w_1 # D
nop
ori $t1, $0, 0x3333
test_d_w_1:
ori $t1, $0, 0x1111
nop
nop
nop
######
add $t4, $t1, $t1 # W
nop # M
nop # E
beq $t2, $t4, test_d_w_2 # D
nop
ori $t1, $0, 0x5555
test_d_w_2:
ori $t1, $0, 0x6666
nop
nop
nop
# FW_RD12_out <- PC_W + 32'h8
ori $t5, $0, 0x314c
nop
nop
nop
jal test_d_w # W
nop # M
j test_d_w_end
nop
test_d_w:
nop # E
beq $ra, $t5, test_d_w_3 # D
nop
ori $t1, $0, 0x3333
test_d_w_3:
nop
```

```
nop
nop
jr $ra
nop
test_d_w_end:
nop
nop
nop
############ E <- M ############
# FW_ALU_out <- ALU_C_M
ori $t1, $0, 0x1
nop
nop
nop
add $t2, $t1, $t1 # M
add $t3, $t2, $t2 # E
nop
nop
nop
# FW_ALU_out <- PC_M + 32'h8
jal test_e_m # M
add $t1, $ra, $ra # E
j test_e_m_end
nop
test_e_m:
nop
nop
nop
jr $ra
nop
test_e_m_end:
nop
nop
nop
########### D <- M ############
# FW_RD12_out <- ALU_C_M
ori $t1, $0, 0x1111
nop
nop
nop
ori $t2, $0, 0x2222
nop
nop
nop
ori $t3, $0, 0
nop
nop
nop
add $t3, $t1, $t1 # M
nop # E
beq $t3, $t2, test_d_m_1 # D
nop
ori $t1, $0, 0x3333
test_d_m_1:
ori $t1, $0, 0x1111
```

```
nop
nop
nop
######
add $t4, $t1, $t1 # M
nop # E
beq $t2, $t4, test_d_m_2 # D
nop
ori $t1, $0, 0x5555
test_d_m_2:
ori $t1, $0, 0x6666
nop
nop
nop
# FW_RD12_out <- PC_M + 32'h8
ori $t5, $0, 0x3268
nop
nop
nop
jal test_d_m # M
nop # E
j test_d_m_end
nop
test_d_m:
beq $ra, $t5, test_d_m_3 # D
nop
ori $t1, $0, 0x3333
test_d_m_3:
nop
nop
nop
jr $ra
nop
test_d_m_end:
nop
nop
nop
########### D <- E ############
# FW_RD12_out <- EXT_Result_E
lui $t1, 0x9
nop
nop
nop
lui $t2, 0x9 # E
beq $t1, $t2, test_d_e_1
nop
ori $t1, $0, 0x3333
test_d_e_1:
nop
nop
nop
# FW_RD12_out <- PC_M + 32'h8
```