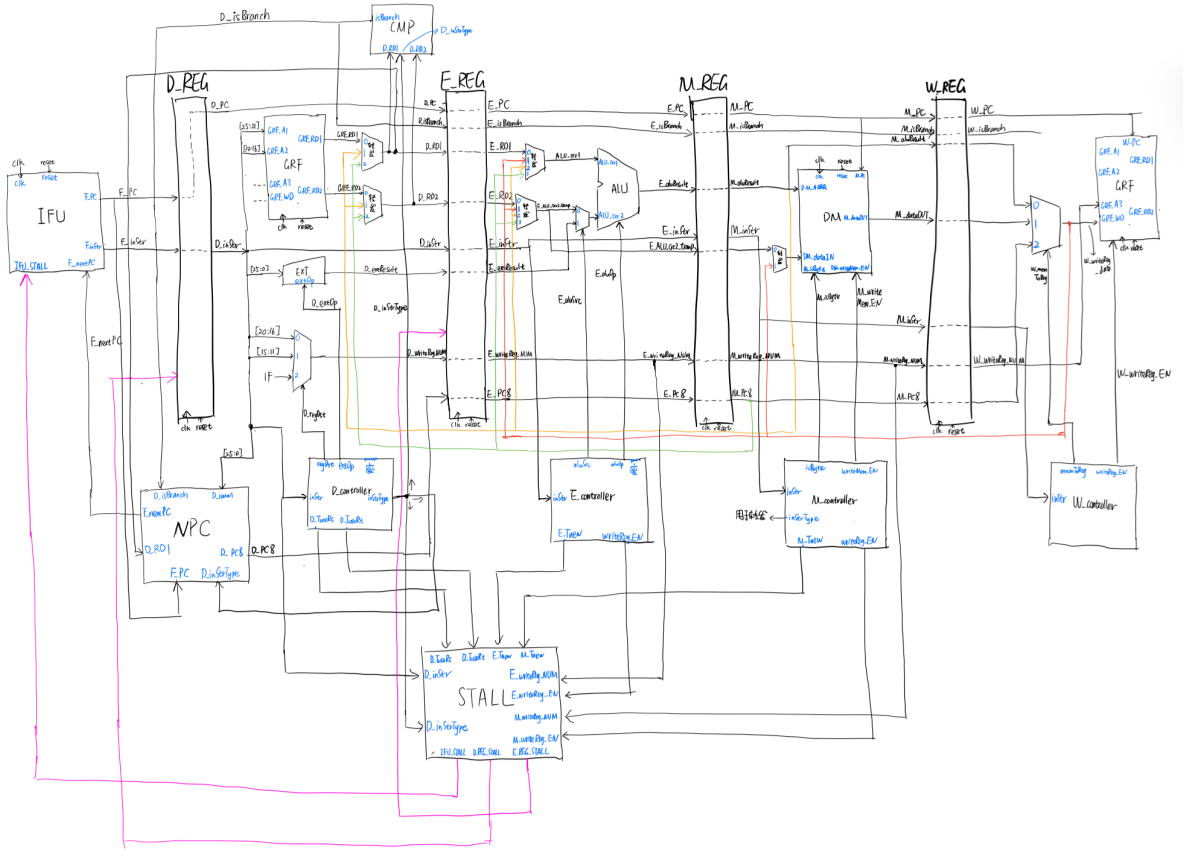
P5设计文档

概述

本次通过Verilog设计单周期CPU架构支持了基本要求的指令add、sub、ori、lw、sw、beq、lui、nop、jal、jr指令，以此为基础又新增了lb、sb功能，即对DM进行了一些修改。首先给出基本框架图：（内部转发实现后W到D级的转发不需要再实现）。



数据通路模块

IFU（取指令单元）

模块内部包含PC（程序计数器）以及IM（指令存储器）。

端口定义

信号名	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
nextPC	I	32	下一条要被执行的指令的地址
IFU_STALL	I	1	使能信号，用于暂停情况
F_PC	O	32	输出当前正在执行的指令的地址
F_inStr	O	32	输出当前正在执行的指令

GRF（通用寄存器组）

该模块内部包含 32 个具有写使能 32 位寄存器，分别对应 MIPS 架构中0~31通用寄存器（其中 0 号寄存器中的值恒为 0，即不具备写使能）。GRF 可以实现同步复位，同时可以根据输入的 5 位地址（0~31）向寄存器堆存取数据，实现定向访存寄存器。

端口定义

信号名	方向	位宽	描述
PC	I	32	用于输出指定信息
clk	I	1	时钟信号
reset	I	1	同步复位信号
GRF_A1	I	5	将对应寄存器的数据读出到 GRF_RD1
GRF_A2	I	5	将对应寄存器的数据读出到 GRF_RD2
GRF_A3	I	5	写入目标寄存器的编号
GRF_WD	I	32	数据输入信号
W_writeReg_EN	I	1	写使能信号
GRF_RD1	O	32	输出 GRF_A1 指定的寄存器中的 32 位数据
GRF_RD2	O	32	输出 GRF_A2 指定的寄存器中的 32 位数据

NPC

负责计算下一条指令的地址并传递给PC。

信号名	方向	位宽	描述
F_PC	I	32	当前指令地址
D_RD1	I	32	用于jr指令，传入地址
D_imm	I	25	立即数来源
D_isBranch	I	1	用于B类指令判断是否跳转
D_inStrType	I	10	传递指令类型
F_nextPC	O	32	输出下一指令地址
D_PC8	O	32	传递PC + 8，用于jal指令的转发

CMP

用于B级跳转指令的判断

信号名	方向	位宽	描述
D_RD1	I	32	参与比较的第一个值（转发后）

信号名	方向	位宽	描述
D_RD2	I	32	参与比较的第二个值（转发后）
D_inStrType	I	10	D级指令类型
isBranch	O	32	输出结果至NPC

EXT

位扩展模块。

信号名	方向	位宽	描述
EXT_imm	I	26	包含需要位扩展的立即数
EXT_extOp	I	2	位扩展方式
D_extResult	O	32	位扩展结果

ALU

该模块主要实现了加法、减法、按位或、立即数加载至高位（LUI）运算。

端口定义

信号名	方向	位宽	描述
aluOp	I	3	ALU 功能选择信号
ALU_src1	I	32	参与 ALU 计算的第一个值
ALU_src2	I	32	参与 ALU 计算的第二个值
E_ALU_result	O	32	输出 ALU 计算结果

DM（数据存储器）

该模块主要通过RAM实现，具有读写功能以及同步复位功能。

端口定义

信号名	方向	位宽	描述
M_PC	I	32	用于输出指定信息
clk	I	1	时钟信号
reset	I	1	同步复位信号
DM_ADDR	I	5	地址输入信号，指向数据储存器中某个存储单元
DM_dataIN	I	32	数据输入信号
DM_writeMem_EN	I	1	写使能信号
M_isByte	I	1	判断是否为字节操作

信号名	方向	位宽	描述
M_dataOUT	O	32	输出 DM_ADDR 指定的存储单元中的 32 位数据

Controller（控制模块）

在控制模块中，我们对指令中 Opcode 域和 Funct 域中的数据进行解码，输出 ALUOp,MemtoReg等10条控制指令，从而对数据通路进行调整，满足不同指令的需求。同时输出指令相关的Tuse和Tnew，便于流水。为实现该模块，我们又在内部设计了两部分 —— 和逻辑（AND Logic）和或逻辑（OR Logic）。前者的功能是识别，从输入的整条指令提取 Opcode 和 Funct ，从而识别为对应的指令，后者的功能是生成，根据输入指令的不同产生不同的控制信号。在每级流水线（D、E、M、W）设置不同的controller便于管理。

控制信号

序号	信号名	位宽	描述	触发指令
1	memToReg	1	GRF 中 WD 接口输入数据选择	lw、lb、jal
2	memWrite_EN	1	DM 写入使能信号	sw、sb
3	aluSrc	1	ALU_src2 接口输入数据选择	ori、lw、sw、lui、lb、sb
4	regWrite_EN	1	GRF 写入使能信号	add、sub、ori、lw、lui、lb、jal
5	extOp	2	立即数符号扩展选择	lw、sw、beq、lb、sb
6	regDst	1	GRF 中 A3 接口输入数据选择	add、sub、jal
7	aluOp	3	ALU 功能选择信号	略
8	inStrType	10	传递指令信息	all-inStr
9	isByte	1	判断是否为字节操作	lb、sb
10	D_TuseRs	2	判断指令对rs的使用时间	all-inStr
11	D_TuseRt	2	判断指令对rt的使用时间	all-inStr
12	E_Tnew	2	判断指令产生新值与E级距离	all-inStr
13	M_Tnew	2	判断指令产生新值与M级距离	all-inStr

STOP（暂停控制器）

暂停模块通过比较Tuse与Tnew的关系，并且在D、E级以及IFU中通过控制使能信号实现在听操作，如下：

```
assign D_rs = D_inStr[25:21];
assign D_rt = D_inStr[20:16];

assign E_Stall_RS = (E_writeReg_NUM == D_rs && D_rs != 0) && (D_TuseRs < E_Tnew)
&& E_writeReg_EN;
assign E_Stall_RT = (E_writeReg_NUM == D_rt && D_rt != 0) && (D_TuseRt < E_Tnew)
&& E_writeReg_EN;

assign M_Stall_RS = (M_writeReg_NUM == D_rs && D_rs != 0) && (D_TuseRs < M_Tnew)
&& M_writeReg_EN;
assign M_Stall_RT = (M_writeReg_NUM == D_rt && D_rt != 0) && (D_TuseRt < M_Tnew)
&& M_writeReg_EN;

assign isStall = E_Stall_RS | E_Stall_RT | M_Stall_RS | M_Stall_RT;

assign IFU_STALL = isStall;
assign D_REG_STALL = isStall;
assign E_REG_STALL = isStall;
```

流水级寄存器

为满足流水线的需要，设置了四个流水级寄存器，分别是D_REG、E_REG、M_REG、W_REG。每级寄存器保存跨越不同流水级的值。

```
module D_REG (
    input clk,
    input reset,
    input D_REG_STALL,  // stop the pipeline
    input [31:0] F_PC,
    input [31:0] F_inStr,
    output [31:0] D_PC,
    output [31:0] D_inStr
);

module E_REG (
    input clk,
    input reset,
    input E_REG_STALL,  // stop the pipeline
    input [31:0] D_PC,
    input [31:0] D_inStr,
    input [31:0] D_PC8,
    input [4:0] D_writeReg_NUM,
    input [31:0] D_RD1,
    input [31:0] D_RD2,
    input [31:0] D_extResult,
    input D_isBranch,
    output [31:0] E_PC,
    output [31:0] E_inStr,
    output [31:0] E_PC8,
    output [4:0] E_writeReg_NUM,
```

```
    output [31:0] E_RD1,
    output [31:0] E_RD2,
    output [31:0] E_extResult,
    output E_isBranch
);

module M_REG (
    input clk,
    input reset,
    input [31:0] E_PC,
    input [31:0] E_inStr,
    input [31:0] E_PC8,
    input [4:0] E_writeReg_NUM,
    input [31:0] E_aluResult,
    input [31:0] E_ALU_src2_temp,
    input E_isBranch,
    output [31:0] M_PC,
    output [31:0] M_inStr,
    output [31:0] M_PC8,
    output [4:0] M_writeReg_NUM,
    output [31:0] M_aluResult,
    output [31:0] M_ALU_src2_temp,
    output M_isBranch
);

module W_REG (
    input clk,
    input reset,
    input [31:0] M_PC,
    input [31:0] M_inStr,
    input [31:0] M_PC8,
    input [4:0] M_writeReg_NUM,
    input [31:0] M_dataOUT,
    input [31:0] M_aluResult,
    input M_isBranch,
    output [31:0] W_PC,
    output [31:0] W_inStr,
    output [31:0] W_PC8,
    output [4:0] W_writeReg_NUM,
    output [31:0] W_dataOUT,
    output [31:0] W_aluResult,
    output W_isBranch
);
```

转发的实现

本次设计主要由MUX实现转发，具体为在mips.v中通过assign语句以及三目运算符进行转发判断：

```
// transition
assign D_RD1 =  (M_inStrType == `jal && D_inStr[25:21] == 5'b11111) ? M_PC8 :
(M_writeReg_NUM == D_inStr[25:21] && D_inStr[25:21] != 0 && M_writeReg_EN) ?
M_aluResult : GRF_RD1;

assign D_RD2 =  (M_inStrType == `jal && D_inStr[20:16] == 5'b11111) ? M_PC8 :
(M_writeReg_NUM == D_inStr[20:16] && D_inStr[20:16] != 0 && M_writeReg_EN) ?
M_aluResult : GRF_RD2;

assign ALU_src1 =   (M_inStrType == `jal && E_inStr[25:21] == 5'b11111) ? M_PC8 :
(M_writeReg_NUM == E_inStr[25:21] && E_inStr[25:21] != 0 && M_writeReg_EN) ?
M_aluResult : (W_writeReg_NUM == E_inStr[25:21] && E_inStr[25:21] != 0 &&
W_writeReg_EN) ? W_writeReg_DATA : E_RD1;

assign E_ALU_src2_temp = (M_inStrType == `jal && E_inStr[20:16] == 5'b11111) ?
M_PC8 : (M_writeReg_NUM == E_inStr[20:16] && E_inStr[20:16] != 0 &&
M_writeReg_EN) ? M_aluResult : (W_writeReg_NUM == E_inStr[20:16] &&
E_inStr[20:16] != 0 && W_writeReg_EN) ? W_writeReg_DATA : E_RD2;

assign DM_dataIN = (W_writeReg_NUM == M_inStr[20:16] && M_inStr[20:16] != 0 &&
W_writeReg_EN) ? W_writeReg_DATA : M_ALU_src2_temp;
```

测试方案

基于Pre教程中的测试样例：

```
ori $a0, $0, 123
ori $a1, $a0, 456
lui $a2, 123            # 符号位为 0
lui $a3, 0xffff         # 符号位为 1
ori $a3, $a3, 0xffff    # $a3 = -1
add $s0, $a0, $a2      # 正正
add $s1, $a0, $a3      # 正负
add $s2, $a3, $a3      # 负负
ori $t0, $0, 0x0000
sw $a0, 0($t0)
sw $a1, 4($t0)
sw $a2, 8($t0)
sw $a3, 12($t0)
sw $s0, 16($t0)
sw $s1, 20($t0)
sw $s2, 24($t0)
lw $a0, 0($t0)
lw $a1, 12($t0)
sw $a0, 28($t0)
sw $a1, 32($t0)
ori $a0, $0, 1
ori $a1, $0, 2
ori $a2, $0, 1
beq $a0, $a1, loop1     # 不相等
beq $a0, $a2, loop2     # 相等
loop1:sw $a0, 36($t0)
loop2:sw $a1, 40($t0)
```

进行了一些补充：

```
ori $a0, $0, 65535
ori $a1, $0, 1234
ori $a1, $a0, 0
lui $t1, 0xffff
ori $t1, $0, 0xffff
ori $t0, 1
add $t2, $t1, $t0
add $t3, $t0, $0
lui $0, 1111
```

并且测试了p5教程中所给出的样例。

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：提前的分支判断会使TuseRt、TuseRs的值提前发生变化，从而影响暂停控制器对于是否需要暂停的判断：比如

```
add $t0, $t1, $t2
beq $t0, $s0, label
```

若判断提前至D级，则需要暂停使得beq中$t0为计算后的新值，而不提前则不需要暂停。

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：若考虑延迟槽，则PC+4指令在当前jal指令进入D级时已经取出并进入流水线，在返回时要从PC+8开始执行，否则PC+4会执行两次，导致程序错误。

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：如果从功能部件进行转发，则会增长关键路径长度，其会使时钟频率变长，降低效率。

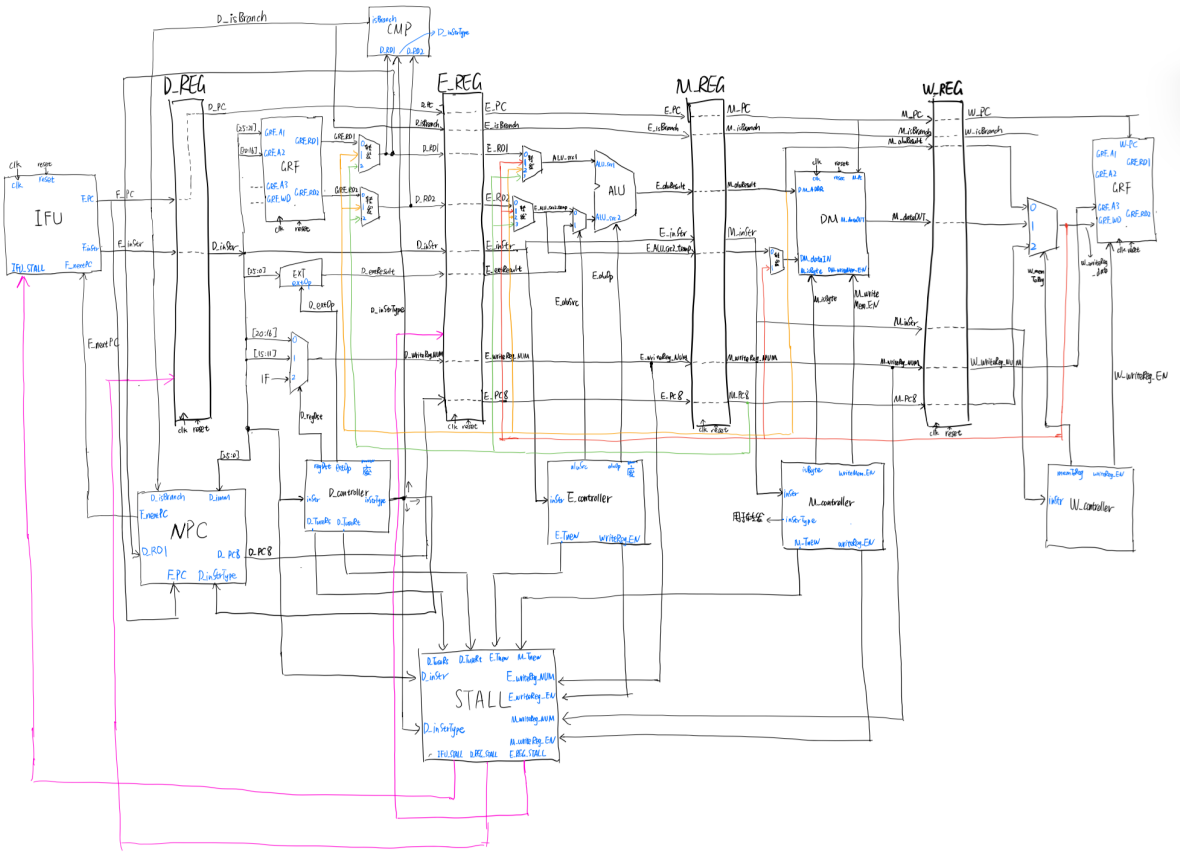
4：我们为什么要使用 GPR 内部转发？该如何实现?

答：GRF的内部转发等价于W级数据转发至D级，若不实现内部转发，则位于D级的指令要使用W级写入的数据时，输出的是旧数据而非新数据。（读写在上升沿同时发生，所以读出的是新写入前的数据）。实现方法如下，只需在GRF对输出进行修改：

```
assign GRF_RD1 = (GRF_A3 && W_writeReg_EN && GRF_A3 == GRF_A1) ? GRF_WD :
register[GRF_A1];
assign GRF_RD2 = (GRF_A3 && W_writeReg_EN && GRF_A3 == GRF_A2) ? GRF_WD :
register[GRF_A2];
```

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：如图所示，黄线、红线、绿线起点为供给者，终点为需求者。



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：可能需要增加控制信号，增加流水级寄存器的传递值以及增加ALU、NPC的计算方法。

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。

答：译码器结构分为两部分，判断指令类型以及输出控制信号：

```
wire rType, nop, add, sub, jr, ori, lw, sw, beq, lui, jal;
assign nop = (opcode == 6'b000000) & (funct == 6'b000000);
assign rType = (opcode == `RTYPE);
assign add = rType & (funct == `ADD);
assign sub = rType & (funct == `SUB);
assign jr = rType & (funct == `JR);
assign ori = opcode == `ORI;
assign lw = opcode == `LW;
assign sw = opcode == `SW;
assign beq = opcode == `BEQ;
assign lui = opcode == `LUI;
assign jal = opcode == `JAL;
assign lb = opcode == `LB;
assign sb = opcode == `SB;
```

```
// inStrType
assign inStrType = nop ? `nop : add ? `add : sub ? `sub :
                   jr ? `jr : ori ? `ori : lw ? `lw : sw ? `sw :
                   beq ? `beq : lui ? `lui : jal ? `jal :
                   lb ? `lb : sb ? `sb : `nop;

// control
assign regDst[0] = add | sub;  // 0: rt, 1: rd, 2: ra;
```

```
assign regDst[1] = jal;
assign aluSrC = lw | sw | lb | sb | ori | lui;  // 0: rt, 1: imm;
assign memWrite_EN = sw | sb;  // 0: no, 1: yes;
assign memToReg[0] = lw | lb;  // 0: alu, 1: mem, 2: PC + 8
assign memToReg[1] = jal;
assign regWrite_EN = add | sub | ori | lui | lw | lb | jal;  // 0: no, 1: yes;
assign aluOp[0] = add | lui | lw | sw | lb | sb;  // 0: -, 1: +, 2: |, 3: lui;
assign aluOp[1] = ori | lui;
assign aluOp[2] = 0;
assign extOp[0] = lw | sw | lb | sb | beq;  // 0: zero ext, 1: sign ext;
assign extOp[1] = 0;
assign isByte = lb | sb;

// STOP
assign D_TuseRs[0] = add | sub | ori | lw | lb | sw | sb | nop | lui | jal;
assign D_TuseRs[1] = nop | lui | jal;
assign D_TuseRt[0] = add | sub | nop | lui | jal | ori | lw | lb | jr;
assign D_TuseRt[1] = sw | sb | nop | lui | jal | ori | lw | lb | jr;
assign E_Tnew[0] = add | sub | ori | lui;
assign E_Tnew[1] = lw | lb;
assign M_Tnew[0] = lw | lb;
assign M_Tnew[1] = 0;
```

优点在于代码整体直观简洁，且便于修改。