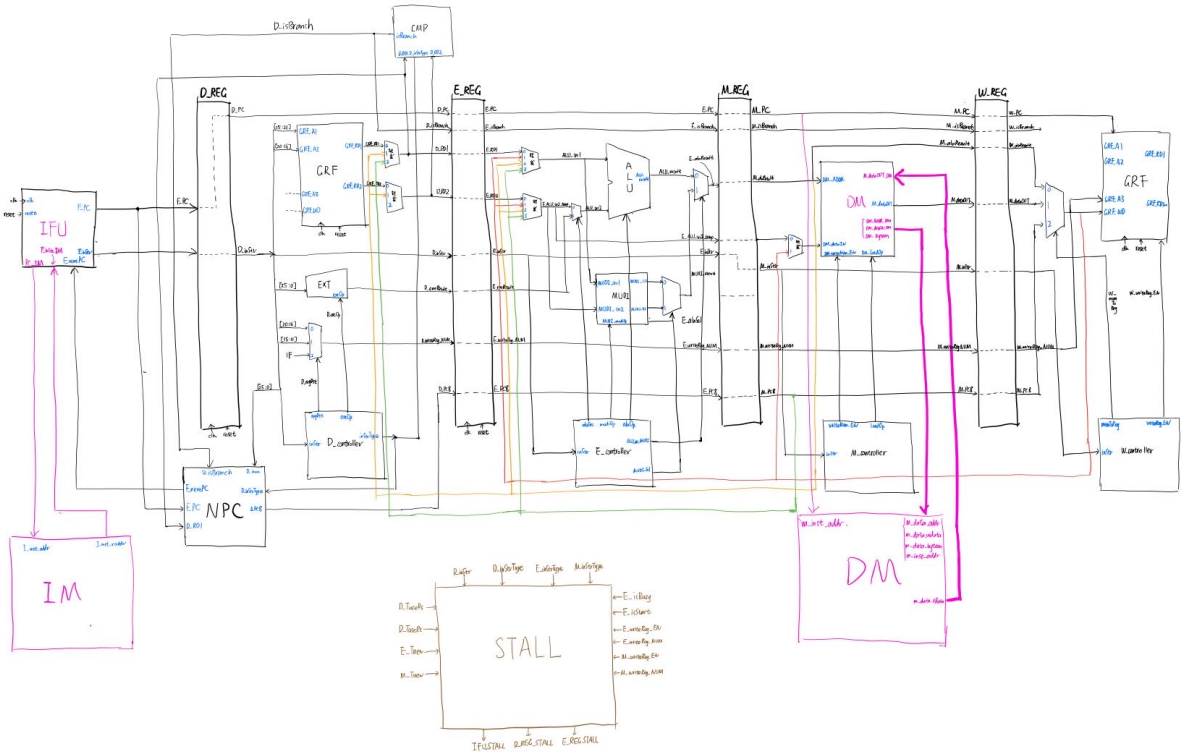
P6设计文档

概述

本次通过Verilog设计流水线CPU架构支持了基本要求的指令指令。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
IFU_STALL	I	1	使能信号，用于暂停情况
nextPC	I	32	下一条要被执行的指令的地址
F_inStr_IM	I	32	外部 IM 传入的指令
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

信号名	方向	位宽	描述
PC_IM	O	32	传至外部 IM 的PC值

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）
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
ALU_result	O	32	输出 ALU 计算结果

MUDI（乘除模块）

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
E_isStart	I	1	乘除计算指令开始信号
MUDI_mudiOp	I	3	MUDI 功能选择信号
MUDI_src1	I	32	参与 MUDI 计算的第一个值
MUDI_src2	I	32	参与 MUDI 计算的第二个值

信号名	方向	位宽	描述
isBusy	O	1	MUDI 计算判断信号
MUDI_HI	O	32	HI 寄存器值
MUDI_LO	O	32	LO 寄存器值

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
DM_writeMem_EN	I	1	写使能信号
M_inStrType	I	10	M 级指令类型
DM_ADDR	I	32	传入存取数据的地址
DM_dataIN	I	32	传入待存储的数据
M_dataOUT_DM	I	32	从外部 DM 传入的数据
M_dataOUT	O	32	输出 M_dataOUT_DM 的数据
DM_ADDR_DM	O	32	向外部 DM 传入地址
DM_dataIN_DM	O	32	向外部 DM 传入数据
DM_byteen_DM	O	32	向外部 DM 传入写使能信号

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
1	inStrType	10	传递指令信息
2	regDst	2	GRF 中 A3 接口输入数据选择
3	aluSrc	1	ALU_src2 接口输入数据选择
4	writeMem_EN	1	DM 写入使能信号
5	memToReg	1	GRF 中 WD 接口输入数据选择

序号	信号名	位宽	描述
6	writeReg_EN	1	GRF 写入使能信号
7	aluOp	3	ALU 功能选择信号
8	mudiOp	3	MUDI 功能选择信号
9	extOp	2	立即数符号扩展选择
10	MUDI_sel	1	HI, LO 寄存器选择
11	ALU_or_MUDI	1	ALU, MUDI 结果选择
12	isStart	1	乘除有关指令开始信号
13	D_TuseRs	2	判断指令对rs的使用时间
14	D_TuseRt	2	判断指令对rt的使用时间
15	E_Tnew	2	判断指令产生新值与E级距离
16	M_Tnew	2	判断指令产生新值与M级距离

STALL（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 Stall_MUDI = E_start || E_busy;

assign isStall = E_Stall_RS | E_Stall_RT | M_Stall_RS | M_Stall_RT | Stall_MUDI;

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

```
lui $1,0x0111
ori $1,$1,0x0111
lui $2,0x0222
ori $2,$2,0x0222
lui $3,0x0333
ori $3,$3,0x0333
lui $4,0x0444
ori $4,$4,0x0444
lui $5,0x0555
ori $5,$5,0x0555
add $6,$1,$2
sub $7,$6,$2
```

```
or $8,$7,$0
slt $8,$7,$6
sw $1,8($0)
sh $2,12($0)
sb $3,32($0)
lw $1,32($0)
lb $2,13($0)
lh $3,8($0)
mult $1,$2
mfhi $4
mflo $5
div $3,$1
mfhi $4
mflo $5
mthi $6
mtlo $7
addi $1,$0,100
andi $2,$1,0xffff
beq $1,$2,label1
sb $2,16($0)
sw $1,20($0)
label1:
bne $1,$2,label2
nop
jal label2
nop
addi $1,$0,200
addi $2,$0,300
label2:
mult $1,$2
mfhi $4
mflo $5
bne $1,$2,label3
nop
jr $ra
label3:
nop
nop
lui $1,0xffff
ori $1,$1,0xffff
slt $3,$2,$1
sltu $4,$2,$1
sltu $5,$1,$2
sw $1,0($0)
sb $4,0($0)
sb $4,1($0)
sb $4,2($0)
sb $4,3($0)
sw $1,0($0)
sh $4,0($0)
sh $4,2($0)
lui $1,0x1234
ori $1,$1,0x5678
sw $1,4($0)
lb $2,4($0)
lb $2,5($0)
```

```
lb $2,6($0)
lb $2,7($0)
lh $2,4($0)
lh $2,6($0)
addi $1,$0,0x1234
sb $1,0($0)
mthi $1
mfhi $3
lui $1,0x8000
ori $1,$1,0x0004
ori $2,$0,0x0002
mult $1,$2
mfhi $3
mflo $4
multu $1,$2
mfhi $3
mflo $4
sb $4,1($0)
lb $4,21($0)
sltu $1,$4,$2
addi $ra,$ra,20
jal label4
addi $ra,$ra,4
label4:
nop
nop
addi $4,$4,1
ori $1,$0,0x3178
ori $ra,$0,0x0000
jr $1
addi $ra,$ra,20
beq $3,$4,label4
addi $4,$4,3
bne $2,$3,label5
slt $1,$2,$3
nop
label5:
ori $ra,$0,0x318c
addi $ra,$ra,8
label6:
jr $ra
addi $ra,$ra,8
nop
jal label6
mflo $2
addi $ra,$ra,8
addi $ra,$ra,4
mthi $ra
mfhi $1
jr $1
lui $1,100
```

并且测试了p6教程中所给出的样例。

思考题

1：为什么需要有单独的乘除法部件而不是整合进 ALU？为何需要有独立的 HI、LO 寄存器？

答：乘除相关的指令涉及到时序逻辑，而ALU的功能目前仅涉及到组合逻辑，所以乘除相关指令更适合放到单独的模块中。如果不使用独立的HI、LO寄存器，即在GRF中添加HI、LO寄存器，那么mfhi、mflo、mthi、mtlo等指令关于HI、LO寄存器的操作会变得混乱，会与通用寄存器的写入、读取操作混淆。

2：真实的流水线 CPU 是如何使用实现乘除法的？请查阅相关资料进行简单说明。

答：CPU执行乘法指令是以加法为基础进行的，乘法的本质是若干个相同的数相加。CPU执行除法指令是以减法为基础的，减法也是以加法为基础的，除法的本质是看被除数能够减去除数几次，这个“几次”便是得到的商，减去若干次后剩下的不够再减一次的部分便是余数。

3：请结合自己的实现分析，你是如何处理 Busy 信号带来的周期阻塞的？

答：在暂停模块中加入对MUDI模块有关的Start信号和Busy信号的考虑，当二者有一个有效时，则触发暂停。

4：请问采用字节使能信号的方式处理写指令有什么好处？（提示：从清晰性、统一性等角度考虑）

答：字节使能信号一方面保证了数据传输之间的一致性，即所有存取指令向DM传输的、从DM传来的地址和数据都是32位，另一方面也使存取的具体操作变得更加清晰。

5：请思考，我们在按字节读和按字节写时，实际从 DM 获得的数据和向 DM 写入的数据是否是一字节？在什么情况下我们按字节读和按字节写的效率会高于按字读和按字写呢？

答：不是，写入和读出的数据为32位。当按字节对数据进行处理的指令较多时按字节读写会有更高的效率。

6：为对抗复杂性你采取了哪些抽象和规范手段？这些手段在译码和处理数据冲突的时候有什么样的特点与帮助？

答：将乘除模块的计算结果与ALU的计算结果在E级通过多路选择器并到一起，这样p6的转发操作与p5完全相同不用修改。

7：在本实验中你遇到了哪些不同指令类型组合产生的冲突？你又是如何解决的？相应的测试样例是什么样的？

答：新的冒险冲突主要是针对乘除运算器和寄存器的使用争夺问题。1.乘除寄存器在运算时未能产生相应的HI和LO时需要执行mfio/mfhi/mtlo/mthi指令，这时直接阻塞即可。2.mtlo/mthi指令需要利用的Rs寄存器的最新的值可能还未产生，此时利用P5就构建好的转发路径即可。3.mflo/mfhi指令改变了寄存器的值.若有之后的指令需要使用相同寄存器时，同理进行转发即可。

8：如果你是手动构造的样例，请说明构造策略，说明你的测试程序如何保证**覆盖**了所有需要测试的情况；如果你是**完全随机**生成的测试样例，请思考完全随机的测试程序有何不足之处；如果你在生成测试样例时采用了**特殊的策略**，比如构造连续数据冒险序列，请你描述一下你使用的策略如何**结合了随机性**达到强测的效果。

答：。首先给每个寄存器赋上不同的随机初值，再对其进行生成不同的操作。完全随机代码的缺点就是可能难以覆盖所有的矛盾冲突，因此我特意手动构造了一些关于冲突的指令，对于这些冲突的构造也是主要针对第7个思考题中提及的矛盾。