五级流水线CPU设计文档

(一)CPU设计要求

32位五级流水线CPU

支持指令

```
add, sub, and, or, slt, sltu, lui
addi, andi, ori
lb, lh, lw, sb, sh, sw
mult, multu, div, divu, mfhi, mflo, mthi, mtlo
beq, bne, jal, jr
```

(二)关键模块设计

IFU

模块定义

信号名	方向	描述
clk	I	时钟信号
Reset	I	异步复位信号
NPCop[1:0]	I	两位控制信号，控制NPC的的值 00：PC←PC+4 01：PC←$PC_{31\cdots28}$||instr_index||0_2 10：PC←PC + 4 + sign_extend(offset||0_2) 11：PC←GPR[rs]
instr_index[26:0]	I	26位输入信号，用于PC的计算
offset[16:0]	I	16位输入信号，PC的偏移量
Reg[32:0]	I	32位输入信号，用于寄存器地址的跳转
Judge	I	一位输入，作为跳转的判断依据
PC_in[31:0]	I	32位输入，D级的PC地址
PCwrite	I	控制是否进行PC修改

信号名	方向	描述
Instr[31:0]	O	32为输出信号，输出当前要执行的指令
PC[31:0]	O	32位输出信号，当前PC的地址

NPC

模块定义

信号名	方向	描述
PC[31:0]	I	32位输入信号，当前指令的地址
NPCop[1:0]	I	两位控制信号，控制NPC的的值 00：PC←PC+4 01：PC←$PC_{31…28}$||instr_index||0_2 10：PC←PC + 4 + sign_extend(offset||0_2) 11：PC←GPR[rs]
instr_index[26:0]	I	26位输入信号，用于PC的计算
offset[16:0]	I	16位输入信号，PC的偏移量
Reg[32:0]	I	32位输入信号，用于寄存器地址的跳转
Judge	I	一位输入，作为跳转的判断依据
NPC[31:0]	O	32位输出，输出下一条指令的地址

功能定义

序号	功能名称	描述
1	计算PC的下一个值	两位控制信号，控制NPC的的值 00：PC←PC+4 01：PC←$PC_{31…28}$||instr_index||0_2 10： 结合equal,n_equal,less,big,big or equal,less or equal判断是否需要跳转 PC←PC + 4 + sign_extend(offset||0_2) 11：PC←GPR[rs]
2	是否跳转判断	Judge=1，跳转 Judge=0，不跳转

IF_ID

模块定义

信号名	方向	描述
clk	I	时钟信号
reset	I	同步复位信号
IF_ID_en	I	寄存器写入控制信号
PC_F[31:0]	I	32位输入信号，F级的PC

信号名	方向	描述
Instr_F[31:0]	I	32位输入信号，F级的指令
PC_D	O	32位输出信号，D级的PC
Instr_D	O	32位输出，D级的指令

CTRL

模块定义

信号名	方向	描述
OP[5:0]	I	6位输入信号，指令操作码
Func[6:0]	I	6位输入信号，指令的func段
RegDst[1:0]	O	GRFA3输入端控制信号 00 : A3←$Instr_{20...16}$ 01 : A3←$Instr_{15...11}$ 11 : A3←0x1f
RegWrite	O	寄存器写入控制信号 0 : 不能向GRF写入 1 : 可以向GRF写入
EXTop	O	功能选择信号 0：Imm无符号拓展到32位 1：Imm符号位拓展到32位
ALUsrc[1:0]	O	ALUSrcB输入控制信号 00 : SrcB←RD2 01 : SrcB←EXTImm 10 : SrcB←sll指令的s SrcA←RD2
ALUctrl[2:0]	O	3位输出信号，选择ALU的功能 000 : SrcA + SrcB 001 : SrcA - SrcB 010 : A |B 011 : A & B 100 : A>>B 101 : $signed(A)>>>B 110 : A<B 置1 111 : A<<B
Menwrite	O	内存写入控制信号 0 : 不能向DM写入 1 : 可以向DM写入

信号名	方向	描述
MemtoReg[1:0]	O	控制向寄存器的写入数据 00 : WD←ALUResult 01 : WD←RD 10 : WD←$NPC_{31...0}$ 11 : WD←[$ALUResult_{15...0}$||0_{16}]

信号名	方向	描述
NPCop[1:0]	O	两位控制信号，控制NPC的的值 00：PC←PC+4 01：PC←$PC_{31...28}$||instr_index||0_2 10：PC←PC + 4 + sign_extend(offset||0_2) 11：PC←GPR[rs]
CMPop[2:0]	O	用于指示进行何种跳转判断 000 : 判断equal 001 : 判断n_equal 010 : 判断less 011 : 判断big 100 : 判断less or equal 101 : 判断big or equal
DMop[1:0]	O	用于lb,sb,lh,sh等操作的拓展 00 : 正常读取，以字为单位 01 : 用于lh或sh Menwrite = 1--->sh Menwrite = 0--->lh 11 : 用于lb或sb Menwrite = 1--->sb Menwrite = 0--->lb

CMP

信号名	方向	描述
RD1[31:0]	I	32位输入，作为要比较的值
RD2[31:0]	I	32位输入，作为要比较的值
CMPop[2:0]	I	用于指示进行何种跳转判断 000 : 判断equal 001 : 判断n_equal 010 : 判断less 011 : 判断big 100 : 判断less or equal 101 : 判断big or equal
Judge	O	一位输出，作为跳转的判断依据

GRF

模块定义

信号名	方向	描述
clk	I	时钟信号
reset	I	复位信号，将32个寄存器中的值全部清零 1：复位 0：无效
WE	I	写使能信号 1：可向GRF中写入数据 0：不能向GRF中写入数据
A1[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的值读出到RD1
A2[4:0]	I	5位地址输入信号，指定32个寄存器中的一个，将其中存储的值读出到RD2
A3[4:0]	I	5位地址输入信号，指定32个寄存器中的一个作为写入的目标寄存器
WD[31:0]	I	32位数据输入信号
RD1[31:0]	O	输出A1指定的寄存器的32位数据
RD2[31:0]	O	输出A2指定寄存器中的32位数据

功能定义

序号	功能名称	描述
1	复位	reset信号有效是，所有寄存器存储的数值清零
2	读数据	读出A1，A2地址对应寄存器中所存储的数据到RD1，RD2
3	写数据	当WE有效且时钟上升沿来临时，将WD写入A3对应的寄存器中

EXT

信号名	方向	描述
Imm[15:0]	I	15位输入立即数
EXTop	I	功能选择信号 0：Imm无符号拓展到32位 1：Imm符号位拓展到32位
EXTImm[31:0]	O	32位输出信号，输出Imm拓展之后的数

ID_EX

模块定义

信号名	方向	描述
clk	I	时钟信号
reset	I	同步复位信号
ID_EX_clr	I	阻塞清零信号
PC_D[31:0]	I	32位输入，D级PC
A3_D[4:0]	I	5位输入，待写入寄存器编号
RD1_D[31:0]	I	32位输入，从GRF[A1]读出
RD2_D[31:0]	I	32位输入，从GRF[A2]读出
RD1_Sel_D[1:0]	I	RD1的转发控制信号
RD2_Sel_D[1:0]	I	RD2的转发控制信号
EXTImm_D[31:0]	I	拓展后的三十二位立即数
Instr_D[31:0]	I	32为输入，D级的指令
A2_D[4:0]	I	D级使用的寄存器编号
A1_D[4:0]	I	E级使用的寄存器编号
A1_E[4:0]	O	E级使用的寄存器编号
A2_E[4:0]	O	D级使用的寄存器编号
Instr_E[31:0]	O	32为输出，E级的指令
PC_E[31:0]	O	32位输出，E级PC
A3_E[4:0]	O	5位输出，待写入寄存器编号
RD1_E[31:0]	O	32位输出，从GRF[A2]读出
RD2_E[31:0]	O	32位输出，从GRF[A2]读出
EXTImm_E[31:0]	O	拓展后的三十二位立即数
RD1_Sel_D_reg[1:0]	O	RD1的转发控制信号
RD2_Sel_D_reg[1:0]	O	RD2的转发控制信号

ALU

模块定义

信号名	方向	描述

信号名	方向	描述
SrcA[31:0]	I	32位输入信号，第一个操作数A
SrcB[31:0]	I	32位输入信号，第二个操作数B
s[4:0]	I	sll可能会用到的信号
ALUControl[2:0]	I	3位输入信号，选择ALU的功能 000 : SrcA + SrcB 001 : SrcA - SrcB 010 : A | B 011 : A & B 100 : 101 : 110 :
ALUResult[31:0]	O	32位输出信号，输出运算结果

MDU

模块定义

信号名	方向	描述
clk	I	时钟信号
reset	I	复位信号
SrcA[31:0]	I	32位输入信号，待操作数
SrcB[31:0]	I	32位输入信号，待操作数
Start	I	乘除指令开始信号
MDUclr	I	MDU清空信号
MDUop[2:0]	I	控制MDU的指令执行
HI[31:0]	O	32位输出，高三十二位寄存器
LO[31:0]	O	32位输出，低三十二位寄存器
MDUout[31:0]	O	32位输出，MDU的输出
Busy	O	Busy信号，MDU正在工作

功能定义

序号	功能名称	描述
1	复位	reset信号有效时，将HI，LO，HI_tmp，LO_tmp全部清空
2	清空数据	MUDclr信号有效时，将HI，LO，HI_tmp，LO_tmp全部清空

序号	功能名称	描述
3	乘除法相关指令	`define mult 3'b000 `define multu 3'b001 `define div 3'b010 `define divu 3'b011 `define mfhi 3'b100 `define mflo 3'b101 `define mthi 3'b110 `define mtlo 3'b111

EX_DM

信号名	方向	描述
clk	I	时钟信号
reset	I	同步复位信号
PC_E[31:0]	I	32位输入，E级PC
A3_E[4:0]	I	5位输入，待写入寄存器编号
Instr_E[31:0]	I	32位输入，E级的指令
RD2_E[31:0]	I	32位输入，从GRF[A2]读出
ALUresult_E[31:0]	I	32位输入，从ALU读出
A2_E[4:0]	I	5位输入，待写入WD的寄存器编号
A2_M[4:0]	O	5位输出，待写入WD的寄存器编号
PC_M[31:0]	O	32位输出，M级PC
A3_M[4:0]	O	5位输出，待写入寄存器编号
Instr_M[31:0]	O	32位输出，M级指令
ALUresult_M[31:0]	O	32位输出
RD2_M[31:0]	O	32位输出，待写入DM的WD端

BE

模块定义

信号名	方向	描述
A[31:0]	I	32位输入，待写入的数据的地址
WD[31:0]	I	32位输入，待写入的数据
BEop[1:0]	I	2位控制信号

信号名	方向	描述
m_data_rdata[31:0]	I	32位输出信号，从地址中读出的数据
m_data_byteen[3:0]	O	4位输出信号，对外置存储器的使能信号
m_data_wdata[31:0]	O	32位输出信号，最终写入内存的数据

功能定义

序号	功能名称	描述
1	控制写入数据种类	m_data_wdata = (BEop==2'b00) ? WD: (BEop==2'b01&&A[1]==1'b0) ? {RD24_31,RD16_23,WD8_15,WD0_7}: (BEop==2'b01&&A[1]==1'b1) ? {WD8_15,WD0_7,RD8_15,RD0_7}: (BEop==2'b10&&A[1:0]==2'b00)? {RD24_31,RD16_23,RD8_15,WD0_7}: (BEop==2'b10&&A[1:0]==2'b01)? {RD24_31,RD16_23,WD0_7,RD0_7}: (BEop==2'b10&&A[1:0]==2'b10)? {RD24_31,WD0_7,RD8_15,RD0_7}: (BEop==2'b10&&A[1:0]==2'b11)? {WD0_7,RD16_23,RD8_15,RD0_7} : 32'b0;
2	控制使能信号	m_data_byteen = (BEop==2'b00) ? 4'b1111: (BEop==2'b01&&A[1]==1'b0) ? 4'b0011: (BEop==2'b01&&A[1]==1'b1) ? 4'b1100: (BEop==2'b10&&A[1:0]==2'b00)? 4'b0001: (BEop==2'b10&&A[1:0]==2'b01)? 4'b0010: (BEop==2'b10&&A[1:0]==2'b10)? 4'b0100: (BEop==2'b10&&A[1:0]==2'b11)? 4'b1000: 4'b0000;

DE

模块定义

信号名	方向	描述
A[31:0]	I	32位输入，待读出的数据的地址
Din[31:0]	I	32位输入，从地址中读出的数据
DEop[2:0]	I	3位控制信号
Dout[31:0]	I	32位输出信号，最终读取的数据

功能定义

序号	功能名称	描述

序号	功能名称	描述
1	控制读取数据种类	Dout = (DEop == 3'b000)? Din : (DEop == 3'b001)? unsigned_byte: (DEop == 3'b010)? signed_byte : (DEop == 3'b011)? unsigned_half : (DEop == 3'b100)? signed_half : 32'b0;

DM_WB

模块定义

信号名	方向	描述
clk	I	时钟信号
reset	I	同步复位信号
PC_M[31:0]	I	32位输入，M级PC
A3_M[4:0]	I	5位输入，待写入寄存器编号
Instr_M[31:0]	I	32位输入，M级的指令
RD_M[31:0]	I	32位输入，DM的输出端
PC_WB[31:0]	O	32位输出，WB级PC
ALUresult_M[31:0]	I	32位输入，从ALU读出
A3_WB[4:0]	O	5位输出，待写入寄存器编号
Instr_WB[31:0]	O	32位输出，WB级指令
RD_WB[31:0]	O	32位输出，DM的输出端的值
ALUresult_WB[31:0]	O	32位输出

HAZARD

模块定义

信号名	方向	描述
clk	I	时钟信号
Instr_D[31:0]	I	D级指令
Instr_E[31:0]	I	E级指令
Regwrite_E	I	E级寄存器写使能
Regwrite_M	I	M级寄存器写使能
PCwrite	O	PC写使能

信号名	方向	描述
IF_ID_en	O	IF_ID寄存器写使能
ID_EX_clr	O	ID_EX寄存器清零
Num_use_rs_D[4:0]	O	D指令待使用的rs
Num_use_rt_D[4:0]	O	D指令待使用的rt
Tnew_E_[1:0]	O	D指令待使用rs时间
Tnew_M_[1:0]	O	D指令待使用rt时间
MDUclr	O	MDU清空信号

HAZARD_E

模块定义

信号名	方向	描述
Instr_E[31:0]	I	E级指令
Tnew_E[1:0]	O	E级指令的Tnew
Num_new_E[4:0]	O	E级指令待修改寄存器编号

HAZARD_M

模块定义

信号名	方向	描述
Instr_E[31:0]	I	E级指令
Tnew_E[1:0]	I	E级指令的Tnew
Num_new_E[4:0]	I	E级指令待修改寄存器编号
Tnew_M	O	M级指令的Tnew
Num_new_M	O	M级指令待修改寄存器编号

(三)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?

因为乘法除法需要在多个周期内执行，用单独的HI,LO寄存器可以减少阻塞周期，提高效率，因为在乘除法部件工作时ALU模块还可以进行其他指令的工作，做到高效率执行。

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。

乘法

首先CPU会初始化三个通用寄存器用来存放被乘数，乘数，部分积的二进制数，部分积寄存器初始化为0，再判断乘数寄存器的低位是低电平还是高电平，如果为0则将乘数寄存器右移一位，同时将部分积寄存器也右移一位，在位移时遵循计算机位移规则，乘数寄存器低位溢出的一位丢弃，部分积寄存器低位溢出的一位填充到乘数寄存器的高位，同时部分积寄存器高位补0。如果为1则将部分积寄存器加上被乘数寄存器，在进行移位操作。当所有乘数位处理完成后部分积寄存器做高位，乘数寄存器做低位就是最终乘法结果。

除法

首先CPU会初始化三个寄存器,用来存放被除数，除数，部分商。余数(被除数与除数比较的结果)放到被除数的有效高位上。CPU做除法时和做除法时是相反的，乘法是右移，除法是左移，乘法做的是加法，除法做的是减法。首先CPU会把被除数bit位与除数bit位对齐，然后再让对齐的被除数与除数比较(双符号位判断)。比如01-10=11(前面的1是符号位) 1-2=-1 计算机通过符号位和后一位的bit位来判断大于和小于，那么01-10=11 就说明01小于10，如果得数为01就代表大于，如果得数为00代表等于。如果得数大于或等于则将比较的结果放到被除数的有效高位上然后再商寄存器上商：1 并向后多看一位(上商就是将商的最低位左移1位腾出商寄存器最低位上新的商)如果得数小于则上商：0 并向后多看一位然后循环做以上操作当所有的被除数都处理完后，商做结果被除数里面的值就是余数。

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?

对于stall信号，我增加了如果Start或者Busy信号有效，且D级时**mfhi,mflo,mthi,mtlo**时进行阻塞。

如果D级也是乘除法，使用MDUclr将MDU的内容清空，相当于被下一条指令覆盖。

```
assign stall =            ……
               ((Busy==1'b1
||Start==1'b1)&&Instr_D[31:26]==`R&&Instr_D[5:0]==`mfhi_I) |
               ((Busy==1'b1
||Start==1'b1)&&Instr_D[31:26]==`R&&Instr_D[5:0]==`mflo_I) |
               ((Busy==1'b1
||Start==1'b1)&&Instr_D[31:26]==`R&&Instr_D[5:0]==`mthi_I) |
               ((Busy==1'b1
||Start==1'b1)&&Instr_D[31:26]==`R&&Instr_D[5:0]==`mtlo_I) |1'b0;
```

```
assign MDUclr = ((Busy==1'b1
||Start==1'b1)&&Instr_D[31:26]==`R&&Instr_D[5:0]==`mult_I) |
                ((Busy==1'b1
||Start==1'b1)&&Instr_D[31:26]==`R&&Instr_D[5:0]==`multu_I)|
                ((Busy==1'b1
||Start==1'b1)&&Instr_D[31:26]==`R&&Instr_D[5:0]==`div_I)  |
                ((Busy==1'b1
||Start==1'b1)&&Instr_D[31:26]==`R&&Instr_D[5:0]==`divu_I) |1'b0;
```

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）

清晰性方面，按字节使能相当于onehot编码，1为写入，0为不写入，直观清楚。

统一性方面，对于各种处理内存指令只需设置好字节使能信号就可以控制数据的写入，不用通过取出该字的数据后再次拼接，对各种指令的操作表现的一样。

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？

在按字节读和按字节写时，实际从 DM 获得的数据和向 DM 写入的数据是同一节，因为读写所使用的地址都来自上一级的ALUresult，但是对应的字节不能保证一样，因为写入的和读出的有特定的模块控制。

在有sb，sh这种不是对于完整字节操作的指令来说，按字节的读写效率会更高，如果按字处理，需要先从内存中读出该字节的所存储的数据，接下来根据需要进行拼接，最后组成完整的字节存入地址中，从内存中读出所消耗的组合逻辑延迟会降低效率，以字节读写就会消除这方面的影响。

6、为了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？

我使用了**控制信号驱动型**去设置各种指令的Tuse和Tnew，以及相应的要使用的或者写入的寄存器，对于控制单元只传入指令，这些信号的处理都在内部实现，在外部只输出转发或者暂停控制信号。通过控制信号驱动型，可以直观的添加新的指令，设置新指令的Tnew，Tuse等各项数据，让这些数据都集中在一起，便于检查和不容易遗忘信号。设置好各种Tnew和Tuse，转发和暂停在P5的基础上就自动支持了。

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？

R类型Rd（写），后续指令Rs或Rt（读）

使用旁路转发，以add为例

```
=========================
add $t0,$t1,$t2
add $a0,$t0,$t1           M到E级转发
=========================
add $t0,$t1,$t2
nop
add $a0,$t0,$t1           M到D级转发
=========================
add $t0,$t1,$t2
nop
nop
add $a0,$t0,$t1          WB向D级转发
```

load指令Rt（写），隔条指令Rs或Rt（读）

```
=========================
lw $t0,0($0)
add $t0,$t0,$t0          stall一个周期，WB向E级转发
=========================
lw $t0,0($0)
nop
add $t0,$t0,$t0          WB向E级转发
=========================
lw $t0,0($0)
nop
nop
add $t0,$t0,$t0          WB向D级转发
```

mult、multu、div、divu和mfhi、mflo、mthi、mtlo

如果E级的Busy或Start信号有效，如果是其他非乘除法相关指令，不会阻塞，使用ALU即可，如果是乘除法相关指令，如果是**mfhi、mflo、mthi、mtlo**，将它们阻塞在D级，直到乘除法结束；如果是**mult、multu、div、divu**，清空乘除法模块中的操作，将D级流水到E级。

```
=========================
mult $t1,$t2
mflo $a0                 会阻塞直到乘除法结束
=========================
mult $t1,$t2
div  $a0,$a1             mult指令被div指令覆盖
```

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。

完全随机的测试样例会导致冲突样例难以达到测试的效果，由于寄存器编号都是随机生成，难以保证相邻的指令可以构成冒险，除此以外，由于数据的随机生成对于加载内存的指令偏移量可能不规范，如对sw指令，可能不是4的整数倍。除此以外，随机生成不能保证延迟槽跳转的科学性，因为可能在一条跳转指令的延迟槽中还有一条跳转指令，这个显然是不合理的。