CPU设计方案综述

总体设计概述

本 CPU 为 Verilog 实现的流水线 MIPS-CPU，支持的指令集包括 `lb，lbu，lh，lhu，lw，sb，sh，sw，add，addu，sub，subu，mult，multu，div，divu，sll，srl，sra，sllv，srlv，srav，and，or，xor，nor，addi，addiu，andi，ori，xori，lui，slt，slti，sltiu，sltu，beq，bne，blez，bgtz，bltz，bgez，j，jal，jalr，jr，mfhi，mflo，mthi，mtlo`。为了实现这些功能，CPU 主要包含了 `IFU，GRF，DM，ALU，MDU，IF_ID，ID_EX，EX_MEM，MEM_WB，SFU，DEC，CTRL` 这些模块。

关键模块定义

1.IFU

介绍

取指令单元，内部包括 PC(程序计数器)、IM(指令存储器)及相关逻辑，其中 PC 具有同步复位功能，IM 的容量为 容量为 32bit * 4096，起始地址为 0x00003000。

端口定义

端口	输入/输出	位宽	描述
nPC	I	32	设置下一个 PC 值。
WE	I	1	使能端。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
I	O	32	当前指令。
PC	O	32	当前 PC 值。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将 PC 设置为 0x00003000。
2	取指令	当时钟上升沿到来时，I 输出当前 PC 对应的指令。
3	取PC值	当时钟上升沿到来时，PC 输出当前 PC 值。
4	设置PC值	当时钟上升沿到来且使能端有效时，将 nPC 设置为当前 PC 值。

2.GRF

介绍

通用寄存器组，也称为寄存器文件、寄存器堆，可以存取 32 位数据，具有同步复位功能。寄存器标号为 0 到 31，其中 0 号寄存器读取的结果恒为 0。

端口定义

端口	输入/输出	位宽	描述
PC	I	32	当前指令的 PC 值。
A1	I	5	指定 32 个寄存器中的一个，将其存储的数据读出到 RD1。
A2	I	5	指定 32 个寄存器中的一个，将其存储的数据读出到 RD2。
A3	I	5	指定 32 个寄存器中的一个，作为写入的目标寄存器。
WD	I	32	写入寄存器的数据信号。
WE	I	1	写使能信号，高电平有效。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
RD1	O	32	输出 A1 指定的寄存器中的数据。
RD2	O	32	输出 A2 指定的寄存器中的数据。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将所有寄存器的值设置为 0x00000000。
2	读数据	读出 A1 和 A2 地址对应寄存器中存储的数据到 RD1 和 RD2；当 WE 有效时会将 WD 的值会实时反馈到对应的 RD1 或 RD2，即内部转发。
3	写数据	当 WE 有效且时钟上升沿到来时，将 WD 的数据写入 A3 对应的寄存器中。

3.ALU

介绍

算术逻辑单元，提供 32 位的多种运算功能。

端口定义

端口	输入/输出	位宽	描述
A	I	32	参与 ALU 计算的第一个值。
B	I	32	参与 ALU 计算的第二个值。
Op	I	4	ALU 功能的选择信号，具体见功能定义。
AO	O	32	ALU 的计算结果。

功能定义

序号	功能名称	功能描述
1	按位与	当 Op = 0 时，AO = A & B。
2	按位或	当 Op = 1 时，AO = A | B。
3	加法	当 Op = 2 时，AO = A + B。
4	减法	当 Op = 3 时，AO = A - B。
5	左移 16 位	当 Op = 4 时，AO = A << 16。
6	有符号比大小	当 Op = 5 时，AO = A < B。
7	无符号比大小	当 Op = 6 时，AO = A < B。（有符号）
8	逻辑左移	当 Op = 7 时，AO = A << B[4:0]。
9	逻辑右移	当 Op = 8 时，AO = A >> B[4:0]。
10	算术右移	当 Op = 9 时，AO = A >>> B[4:0]。
11	异或	当 Op = 10 时，AO = A ^ B。
12	或非	当 Op = 11 时，AO = ~(A | B)。

4.MDU

介绍

乘除块，内有 hi 和 lo 两个寄存器，可以进行乘除运算，完成相关乘除指令，并模拟乘除运算的延时。

端口定义

端口	输入/输出	位宽	描述
A	I	32	参与乘除运算的第一个值。
B	I	32	参与乘除运算的第二个值。
Op	I	4	MDU 功能的选择信号，具体见功能定义。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
Busy	O	1	当前是否正在进行乘除运算。
hi	O	32	hi 寄存器的值。
lo	O	32	lo 寄存器的值。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将 MDU 中所有值设为 0x00000000。
2	有符号乘	当 Op = 1 时，将 A * B 的高 32 位和低 32 位分别写入 hi 和 lo 寄存器，延迟为 5 个时钟周期。（有符号）
3	无符号乘	当 Op = 2 时，将 A * B 的高 32 位和低 32 位分别写入 hi 和 lo 寄存器，延迟为 5 个时钟周期。（无符号）
4	有符号除	当 Op = 3 时，将 A / B 的余数和商分别写入 hi 和 lo 寄存器，延迟为 10 个时钟周期。（有符号）
5	无符号除	当 Op = 4 时，将 A / B 的余数和商分别写入 hi 和 lo 寄存器，延迟为 10 个时钟周期。（有符号）
6	写 hi 寄存器	当 Op = 7 时，将 A 写入 hi 寄存器。
7	写 lo 寄存器	当 Op = 8 时，将 A 写入 lo 寄存器。

5.DM

介绍

数据存储器，可以存取 32 位数据，容量为 32bit * 4096，具有同步复位功能，起始地址为 0x00000000。

端口定义

端口	输入/输出	位宽	描述
PC	I	32	当前指令的 PC 值。
A	I	5	读取或写入数据的地址。
WD	I	32	写入 DM 中的数据。
Op	I	2	DM 功能的选择信号，具体见功能定义。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
RD	O	32	根据 A 和 Op 输出对应的数据。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将 DM 中所有值设为 0x00000000。
2	有符号读字节	当 Op = 0 时，输出对应字节的 32 位有符号扩展。
3	无符号读字节	当 Op = 1 时，输出对应字节的 32 位无符号扩展。
4	有符号读半字	当 Op = 2 时，输出对应半字的 32 位有符号扩展。
5	无符号读半字	当 Op = 3 时，输出对应半字的 32 位无符号扩展。
6	读字	当 Op = 4 时，输出对应字。
7	写字节	当 Op = 5 且时钟上升沿来临时，将 WD[7:0] 写入 A 对应的地址。
8	写半字	当 Op = 6 且时钟上升沿来临时，将 WD[15:0] 写入 A 对应的地址。
9	写字	当 Op = 7 且时钟上升沿来临时，将 WD 写入 A 对应的地址。

6. IF_ID

介绍

I 级和 D 级间的流水线寄存器，同时产生 D 级的控制信号。

端口定义

端口	输入/输出	位宽	描述
nI	I	32	下一个指令。
nPC	I	32	下一个 PC 值。
WE	I	1	使能端
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
I	O	32	当前指令。
PC	O	32	当前 PC 值。
PCSrc	O	4	0：PC = PC + 4。 1：跳转到 beq 指令对应的跳转地址。 2：跳转到 jal 指令和 j 指令对应的跳转地址。 3：跳转到 jalr 指令和 jr 指令对应的跳转地址。 4：跳转到 bne 指令对应的跳转地址。 5：跳转到 blez 指令对应的跳转地址。 6：跳转到 bgtz 指令对应的跳转地址。 7：跳转到 bltz 指令对应的跳转地址。 8：跳转到 bgez 指令对应的跳转地址。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将所有值设为 0x00000000。
2	读数据	读出各个当前寄存器对应的值。
3	写数据	当 WE 有效且时钟上升沿到来时，将各个对应的值写入寄存器中。

7. ID_EX

介绍

D 级和 E 级间的流水线寄存器，同时产生 E 级的控制信号。

端口定义

端口	输入/输出	位宽	描述
nI	I	32	下一个指令。
nRD1	I	32	下一个 RD1 值。
nRD2	I	32	下一个 RD2 值。
nPC	I	32	下一个 PC 值。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
I	O	32	当前指令。
RD1	O	32	当前 RD1 值。
RD2	O	32	当前 RD2 值。
PC	O	32	当前 PC 值。
ALUOp	O	4	ALU 功能的选择信号，具体见 ALU 模块的功能定义。
ALUSrc	O	3	0：ALU 的 A 输入端选择 RD1 输出端，ALU 的 B 输入端选择 RD2 输出端。 1：ALU 的 A 输入端选择 RD1 输出端，ALU 的 B 输入端选择 I[15:0] 的 32 位无符号扩展。 2：ALU 的 A 输入端选择 RD1 输出端，ALU 的 B 输入端选择 I[15:0] 的 32 位有符号扩展。 3：ALU 的 A 输入端选择 RD2 输出端，ALU 的 B 输入端选择 I[10:6]。 4：ALU 的 A 输入端选择 RD2 输出端，ALU 的 B 输入端选择 RD1 输出端。
MDUOp	O	3	MDU 功能的选择信号，具体见 MDU 模块的功能定义。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将所有值设为 0x00000000。
2	读数据	读出各个当前寄存器对应的值。
3	写数据	当时钟上升沿到来时，将各个对应的值写入寄存器中。

8. EX_MEM

介绍

E 级和 M 级间的流水线寄存器，同时产生 M 级的控制信号。

端口定义

端口	输入/输出	位宽	描述
nI	I	32	下一个指令。
nAO	I	32	下一个 ALU 计算结果。
nWD	I	32	下一个写入 DM 的值。
nPC	I	32	下一个 PC 值。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
I	O	32	当前指令。
AO	O	32	当前 ALU 计算结果。
WD	O	32	当前写入 DM 的值。
PC	O	32	当前 PC 值。
MemOp	O	3	DM 功能的选择信号，具体见 DM 模块的功能定义。
MemtoReg	O	2	0：准备写回 GRF 的 WD 输入端选择 AO 输出端。 1：准备写回 GRF 的 WD 输入端选择 DM 的 RD 输出端。 2：准备写回 GRF 的 WD 输入端选择 PC + 8。

功能定义

序号	功能名称	功能描述
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将所有值设为 0x00000000。
2	读数据	读出各个当前寄存器对应的值。
3	写数据	当时钟上升沿到来时，将各个对应的值写入寄存器中。

9. MEM_WB

介绍

M 级和 W 级间的流水线寄存器，同时产生 W 级的控制信号。

端口定义

nI	**I**	**32**	**下一个指令。**
nPC	I	32	下一个 PC 值。
nWD	I	32	下一个写入 GRF 的值。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
I	O	32	当前指令。
PC	O	32	当前 PC 值。
WD	O	32	当前写入 GRF 的值。
RegWrite	O	1	0：GRF 的写使能端 WE 无效。 1：GRF 的写使能端 WE 有效。
RegDst	O	2	0：准备写回 GRF 的 A3 输入端选择 I[20:16]。 1：准备写回 GRF 的 A3 输入端选择 I[15:11]。 2：准备写回 GRF 的 A3 输入端选择 31。
RegWrite	O	1	0：GRF 的写使能端 WE 无效。 1：GRF 的写使能端 WE 有效。

功能定义

序号	**功能名称**	**功能描述**
1	同步复位	当时钟上升沿到来且同步复位信号有效时，将所有值设为 0x00000000。
2	读数据	读出各个当前寄存器对应的值。
3	写数据	当时钟上升沿到来时，将各个对应的值写入寄存器中。

10. SFU

介绍

暂停 (Stall) 和 转发 (Forward) 的控制单元，产生两者的控制信号。

端口与功能定义

端口	输入/输出	位宽	描述
if_id_I	I	32	IF/ID 流水线寄存器当前的指令。
id_ex_I	I	32	ID/EX 流水线寄存器当前的指令。
ex_mem_I	I	32	EX/MEM 流水线寄存器当前的指令。
mem_wb_I	I	32	MEM/WB 流水线寄存器当前的指令。
Stall	O	1	暂停信号。
Busy	I	1	乘除块的 Busy 信号。
Start	I	1	乘除块的 Start 信号。
F_if_id_rs	O	3	0：D 级 Rs 选择 GRF 的 RD1 输出端。 1：D 级 Rs 选择 ID/EX 流水线寄存器的 PC + 8 输出端。 2：D 级 Rs 选择 EX/MEM 流水线寄存器的 PC + 8 输出端。 3：D 级 Rs 选择 EX/MEM 流水线寄存器的 AO 输出端。
F_if_id_rt	O	3	0：D 级 Rt 选择 GRF 的 RD2 输出端。 1：D 级 Rt 选择 ID/EX 流水线寄存器的 PC + 8 输出端。 2：D 级 Rt 选择 EX/MEM 流水线寄存器的 PC + 8 输出端。 3：D 级 Rt 选择 EX/MEM 流水线寄存器的 AO 输出端。
F_id_ex_rs	O	3	0：E 级 Rs 选择 ID/EX 流水线寄存器 的 RD1 输出端。 2：E 级 Rs 选择 EX/MEM 流水线寄存器的 PC + 8 输出端。 3：E 级 Rs 选择 EX/MEM 流水线寄存器的 AO 输出端。 4：E 级 Rs 选择 MEM/WB 流水线寄存器的 WD 输出端。
F_id_ex_rt	O	3	0：E 级 Rt 选择 ID/EX 流水线寄存器 的 RD2 输出端。 2：E 级 Rt 选择 EX/MEM 流水线寄存器的 PC + 8 输出端。 3：E 级 Rt 选择 EX/MEM 流水线寄存器的 AO 输出端。 4：E 级 Rt 选择 MEM/WB 流水线寄存器的 WD 输出端。
F_ex_mem_rt	O	3	0：M 级 Rt 选择 EX/MEM 流水线寄存器的 WD 输出端。 4：M 级 Rt 选择 MEM/WB 流水线寄存器的 WD 输出端。

数据冒险分析表

IF/ID当前指令				ID/EX (Tnew)				EX/MEM (Tnew)				MEM/WB (Tnew)			
指令类型	源寄存器	Tuse		jal 0/rd	cal_i 1/rt	cal_r 1/rd	load 2/rt	jal 0/rd	cal_r 0/rd	cal_i 0/rt	load 1/rt	jal 0/rd	cal_r 0/rd	cal_i 0/rt	load 0/rt
j	rs/rt	0			暂停	暂停	暂停				暂停				
cal_r	rs/rt	1					暂停								
cal_i	rs	1					暂停								
ld	rs	1					暂停								
st	rs	1					暂停								
st	rt	2													
				PC + 8				PC + 8	AO	AO		WD	WD	WD	WD
流水级	源寄存器	涉及指令	控制信号												
IF/ID	rs	j	F_if_id_rs												
	rt	j	F_if_id_rt												
ID/EX	rs	cal_r, cal_i, ld, st	F_id_ex_rs												
	rt	cal_r, st	F_id_ex_rt												
EX/MEM	rt	st	F_ex_mem_rt												

11.DEC

介绍

指令分类译码器，输入一个指令，返回该指令在数据冒险中对应的类型。

端口与功能定义

端口	输入/输出	位宽	描述
I	I	32	需要分类的指令。
j	O	1	需要读寄存器的跳转指令。
r	O	1	除 `jalr, jr` 外的 R 型指令。
i	O	1	I 型指令。
ld	O	1	load 型指令。
st	O	1	store 型指令。
jal	O	1	`jal, jalr` 指令。
rs	O	5	I[25:21]。
rt	O	5	I[20:16]。
rd	O	5	如果指令为 `jal` 输出 31，否则输出 I[15:11]。

12.CTRL

介绍

指令译码器，输入 Op 和 Func，输出该指令是哪个指令。本 CPU 采用分布式译码，对应控制信号的功能已在上述四个流水线寄存器中详细描述。

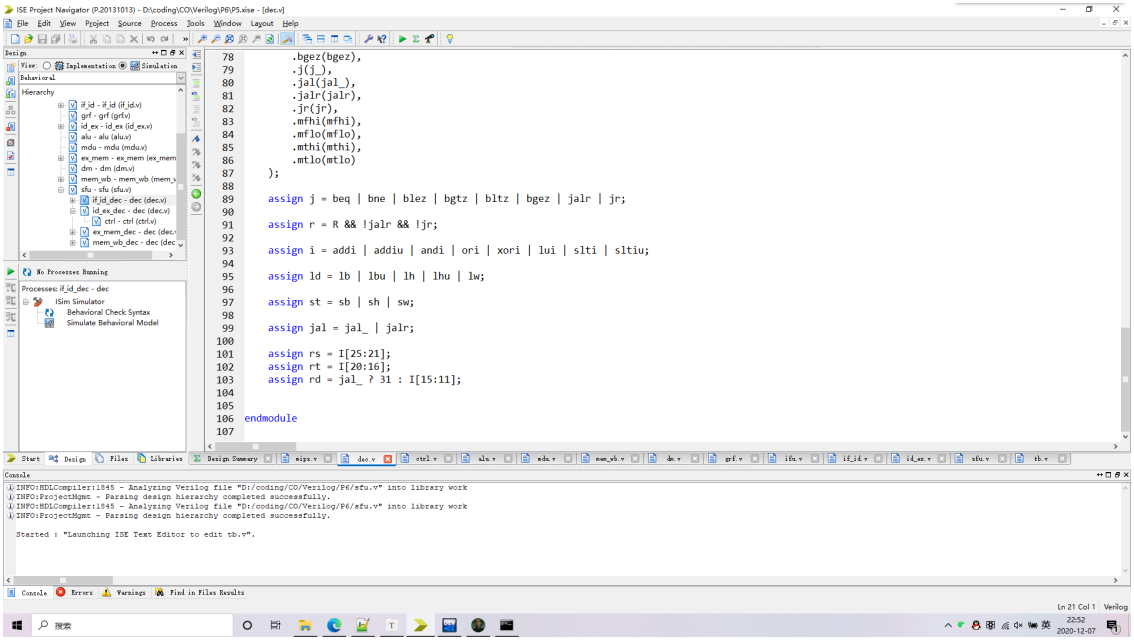
端口与功能定义

端口	输入/输出	位宽	描述
Op	I	6	所有指令的操作码，对应 I[31:26]。
Func	I	6	R 指令中辅助识别的操作码，对应 I[5:0]。
每种指令	O	1	判断是否是每种指令。

重要机制实现方法

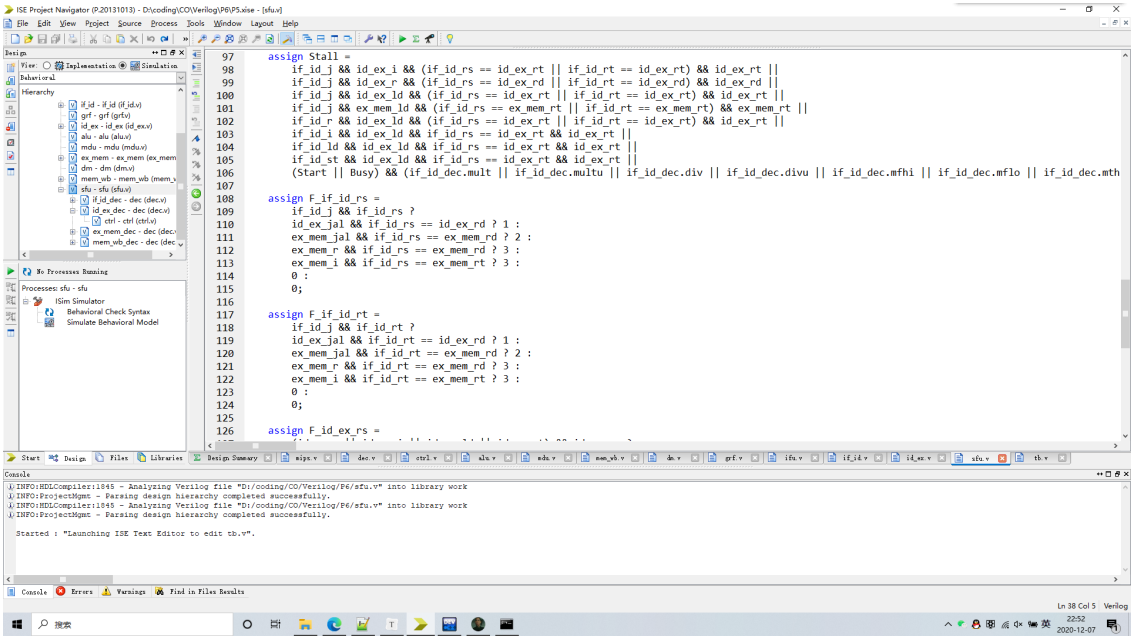
通过指令分类和 Tuse-Tnew 法解决数据冒险

- 首先通过 Tuse-Tnew 的定义对指令进行分类。



```
        .bgez(bgez),
        .j(j_),
        .jal(jal_),
        .jalr(jalr),
        .jr(jr),
        .mfhi(mfhi),
        .mflo(mflo),
        .mthi(mthi),
        .mtlo(mtlo)
    );

    assign j = beq | bne | blez | bgtz | bltz | bgez | jalr | jr;

    assign r = R && !jalr && !jr;

    assign i = addi | addiu | andi | ori | xori | lui | slti | sltiu;

    assign ld = lb | lbu | lh | lhu | lw;

    assign st = sb | sh | sw;

    assign jal = jal_ | jalr;

    assign rs = I[25:21];
    assign rt = I[20:16];
    assign rd = jal_ ? 31 : I[15:11];


endmodule
```

- 再列出对应的 Tuse-Tnew 表，已在 SFU 部分展示。
- 再通过 Tuse-Tnew 表实现暂停信号与转发信号的生成。



```
    assign Stall =
        if_id_j && id_ex_i && (if_id_rs == id_ex_rt || if_id_rt == id_ex_rt) && id_ex_rt ||
        if_id_j && id_ex_r && (if_id_rs == id_ex_rd || if_id_rt == id_ex_rd) && id_ex_rd ||
        if_id_j && id_ex_ld && (if_id_rs == id_ex_rt || if_id_rt == id_ex_rt) && id_ex_rt ||
        if_id_j && ex_mem_ld && (if_id_rs == ex_mem_rt || if_id_rt == ex_mem_rt) && ex_mem_rt ||
        if_id_r && id_ex_ld && (if_id_rs == id_ex_rt || if_id_rt == id_ex_rt) && id_ex_rt ||
        if_id_i && id_ex_ld && if_id_rs == id_ex_rt && id_ex_rt ||
        if_id_ld && id_ex_ld && if_id_rs == id_ex_rt && id_ex_rt ||
        if_id_st && id_ex_ld && if_id_rs == id_ex_rt && id_ex_rt ||
        (Start || Busy) && (if_id_dec.mult || if_id_dec.multu || if_id_dec.div || if_id_dec.divu || if_id_dec.mfhi || if_id_dec.mflo || if_id_dec.mth

    assign F_if_id_rs =
        if_id_j && if_id_rs ?
        id_ex_jal && if_id_rs == id_ex_rd ? 1 :
        ex_mem_jal && if_id_rs == ex_mem_rd ? 2 :
        ex_mem_r && if_id_rs == ex_mem_rd ? 3 :
        ex_mem_i && if_id_rs == ex_mem_rt ? 3 :
        0 :
        0;

    assign F_if_id_rt =
        if_id_j && if_id_rt ?
        id_ex_jal && if_id_rt == id_ex_rd ? 1 :
        ex_mem_jal && if_id_rt == ex_mem_rd ? 2 :
        ex_mem_r && if_id_rt == ex_mem_rd ? 3 :
        ex_mem_i && if_id_rt == ex_mem_rt ? 3 :
        0 :
        0;

    assign F_id_ex_rs =
```

- 最后，将暂停信号和转发信号连接到对应的部件上。

测试方案

自动测试工具

全自动化对拍器

- 重复下述过程无数次：
  - 使用 C++ 随机化生成一段 MIPS 汇编代码。
  - 使用魔改版的 Mars，执行汇编指令时会按照格式输出评测机需要的信息（即需要 $display 的信息）。
  - 使用 Mars 的命令行代码编译并执行，将输出导入到 `m.out`。
  - 使用 Mars 的命令行代码编译并将机器码导出到 `code.txt`。
  - 使用 IVerilog 命令行代码将 testbench 文件转换为可进行仿真的 `tb.out` 文件。
  - 使用 IVerilog 命令行代码 执行仿真，并将输出导入到 `v.out`。
  - 使用 C++ 删掉 `v.out` 中多余的信息（例如一些警告和时间），将 `m.out` 和 `v.out` 分别排序。
  - 使用 `fc` 将 `v.out` 和 `m.out` 进行比对，在第一组 `WA` 的数据点停下。
- 代码如下：
  - 数据生成器：

```
#include <bits/stdc++.h>

using namespace std;

vector<int> r;
mt19937 mt(time(0));
uniform_int_distribution<int>
    u16(0, (1 << 16) - 1),
    s16(-(1 << 15), (1 << 15) - 1),
    siz(0, 15),
    reg(0, 2),
    grf(1, 30),
    shift(0, 31),
    I(1, 40),
    J(41, 49),
    IJ(1, 49),
    one(11, 40);

int cnt, tot;

int getR(){
    return r[reg(mt)];
}

void solve(int i){
    int x, X;
    switch(i){
        case 1:
            x = getR();
            printf("ori $%d, $0, 0\n", x);
            printf("lb $%d, %d($%d)\n", getR(), siz(mt), x);
            tot += 2;
            break;
        case 2:
            x = getR();
            printf("ori $%d, $0, 0\n", x);
            printf("lbu $%d, %d($%d)\n", getR(), siz(mt), x);
            tot += 2;
            break;
```

```
        case 3:
            x = getR();
            printf("ori $%d, $0, 0\n", x);
            printf("lh $%d, %d($%d)\n", getR(), siz(mt) >> 1 << 1, x);
            tot += 2;
            break;
        case 4:
            x = getR();
            printf("ori $%d, $0, 0\n", x);
            printf("lhu $%d, %d($%d)\n", getR(), siz(mt) >> 1 << 1, x);
            tot += 2;
            break;
        case 5:
            x = getR();
            printf("ori $%d, $0, 0\n", x);
            printf("lw $%d, %d($%d)\n", getR(), siz(mt) >> 2 << 2, x);
            tot += 2;
            break;
        case 6:
            x = getR();
            printf("ori $%d, $0, 0\n", x);
            printf("sb $%d, %d($%d)\n", getR(), siz(mt), x);
            tot += 2;
            break;
        case 7:
            x = getR();
            printf("ori $%d, $0, 0\n", x);
            printf("sh $%d, %d($%d)\n", getR(), siz(mt) >> 1 << 1, x);
            tot += 2;
            break;
        case 8:
            x = getR();
            printf("ori $%d, $0, 0\n", x);
            printf("sw $%d, %d($%d)\n", getR(), siz(mt) >> 2 << 2, x);
            tot += 2;
            break;
        case 9:
            x = getR();
            printf("ori $%d, $%d, 1\n", x, x);
            printf("div $%d, $%d\n", getR(), x);
            tot += 2;
            break;
        case 10:
            x = getR();
            printf("ori $%d, $%d, 1\n", x, x);
            printf("divu $%d, $%d\n", getR(), x);
            tot += 2;
            break;
        case 11:
            printf("add $%d, $0, $%d\n", getR(), getR());
            tot++;
            break;
        case 12:
            printf("addu $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 13:
            x = getR();
```

```
            printf("sub $%d, $%d, $%d\n", getR(), x, x);
            tot++;
            break;
        case 14:
            printf("subu $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 15:
            printf("mult $%d, $%d\n", getR(), getR());
            tot++;
            break;
        case 16:
            printf("multu $%d, $%d\n", getR(), getR());
            tot++;
            break;
        case 17:
            printf("slt $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 18:
            printf("sltu $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 19:
            printf("sll $%d, $%d, %d\n", getR(), getR(), shift(mt));
            tot++;
            break;
        case 20:
            printf("srl $%d, $%d, %d\n", getR(), getR(), shift(mt));
            tot++;
            break;
        case 21:
            printf("sra $%d, $%d, %d\n", getR(), getR(), shift(mt));
            tot++;
            break;
        case 22:
            printf("sllv $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 23:
            printf("srlv $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 24:
            printf("srav $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 25:
            printf("and $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 26:
            printf("or $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 27:
            printf("xor $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
```

```
            break;
        case 28:
            printf("nor $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 29:
            printf("addi $%d, $%d, %d\n", getR(), getR(), 0);
            tot++;
            break;
        case 30:
            printf("addiu $%d, $%d, %d\n", getR(), getR(), s16(mt));
            tot++;
            break;
        case 31:
            printf("andi $%d, $%d, %d\n", getR(), getR(), u16(mt));
            tot++;
            break;
        case 32:
            printf("ori $%d, $%d, %d\n", getR(), getR(), u16(mt));
            tot++;
            break;
        case 33:
            printf("xori $%d, $%d, %d\n", getR(), getR(), u16(mt));
            tot++;
            break;
        case 34:
            printf("lui $%d, %d\n", getR(), u16(mt));
            tot++;
            break;
        case 35:
            printf("slti $%d, $%d, %d\n", getR(), getR(), s16(mt));
            tot++;
            break;
        case 36:
            printf("sltiu $%d, $%d, %d\n", getR(), getR(), s16(mt));
            tot++;
            break;
        case 37:
            printf("mfhi $%d\n", getR());
            tot++;
            break;
        case 38:
            printf("mflo $%d\n", getR());
            tot++;
            break;
        case 39:
            printf("mthi $%d\n", getR());
            tot++;
            break;
        case 40:
            printf("mtlo $%d\n", getR());
            tot++;
            break;
        case 41:
            printf("beq $%d, $%d, label%d\n", getR(), getR(), ++cnt);
            solve(I(mt));
            solve(I(mt));
            printf("label%d: ", cnt);
```

```
            solve(I(mt));
            tot++;
            break;
        case 42:
            printf("bne $%d, $%d, label%d\n", getR(), getR(), ++cnt);
            solve(I(mt));
            solve(I(mt));
            printf("label%d: ", cnt);
            solve(I(mt));
            tot++;
            break;
        case 43:
            printf("blez $%d, label%d\n", getR(), ++cnt);
            solve(I(mt));
            solve(I(mt));
            printf("label%d: ", cnt);
            solve(I(mt));
            tot++;
            break;
        case 44:
            printf("bgtz $%d, label%d\n", getR(), ++cnt);
            solve(I(mt));
            solve(I(mt));
            printf("label%d: ", cnt);
            solve(I(mt));
            tot++;
            break;
        case 45:
            printf("bltz $%d, label%d\n", getR(), ++cnt);
            solve(I(mt));
            solve(I(mt));
            printf("label%d: ", cnt);
            solve(I(mt));
            tot++;
            break;
        case 46:
            printf("bgez $%d, label%d\n", getR(), ++cnt);
            solve(I(mt));
            solve(I(mt));
            printf("label%d: ", cnt);
            solve(I(mt));
            tot++;
            break;
        case 47:
            printf("j label%d\n", ++cnt);
            solve(I(mt));
            solve(I(mt));
            printf("label%d: ", cnt);
            solve(I(mt));
            tot++;
            break;
        case 48:
            printf("jal label%d\n", ++cnt);
            X = getR();
            printf("ori $%d, $0, 16\n", X);
            solve(one(mt));
            printf("label%d: addu $%d, $%d, $31\n", cnt, X, X);
            printf("jr $%d\n", X);
```

```
            puts("nop");//solve(I(mt));
            tot += 4;
            break;
        case 49:
            printf("jal label%d\n", ++cnt);
            X = getR();
            printf("ori $%d, $0, 16\n", X);
            solve(one(mt));
            printf("label%d: addu $%d, $%d, $31\n", cnt, X, X);
            printf("jalr $%d, $%d\n", getR(), X);
            puts("nop");//solve(I(mt));
            tot += 4;
            break;
    }
}

int main(){
    r.push_back(grf(mt)), r.push_back(grf(mt)), r.push_back(grf(mt));
    freopen("test.asm", "w", stdout);
    puts("ori $28, $0, 0");
    puts("ori $29, $0, 0");
    while(tot < 900) solve(IJ(mt));
}
```

- 格式处理器：

```
#include <bits/stdc++.h>
#define maxn 100086

using namespace std;

vector<string> v, w;
char s[maxn];

int main(){
    freopen("v.out", "r", stdin);
    while(gets(s) != NULL){
        string S = s;
        v.push_back(s);
    }
    freopen("v.out", "w", stdout);

    for(int i = 0;i < v.size();i++){
        if(v[i].length() <= 20 || v[i][20] != '@') continue;
        w.push_back(v[i].substr(20));
    }
    sort(w.begin(), w.end());
    for(int i = 0;i < w.size();i++) printf("%s\n", w[i].c_str());

    v.clear();
    freopen("m.out", "r", stdin);
    while(gets(s) != NULL){
        string S = s;
        v.push_back(s);
    }
    freopen("m.out", "w", stdout);
    sort(v.begin(), v.end());
```

```
    for(int i = 0;i < v.size();i++){
        if(v[i][0] == '@') printf("%s\n", v[i].c_str());
    }
}
```

- 评测代码：

```
#include <bits/stdc++.h>

using namespace std;

char s[10086];
int cnt = 0;

int main(){
    while(1){
        system("gen.exe");
        system("java -jar Mars_Changed.jar db mc CompactDataAtZero nc
test.asm > m.out");
        system("java -jar Mars_Changed.jar mc CompactDataAtZero a dump
.text HexText code.txt test.asm > log.txt");
        system("iverilog -o tb.out -y D:\\coding\\CO\\Verilog\\P6
D:\\coding\\CO\\Verilog\\P6\\tb.v");
        system("vvp tb.out > v.out");
        system("del.exe");
        system("fc v.out m.out > log.txt");
        freopen("log.txt", "r", stdin);
        gets(s), gets(s);
        printf("test%d:", ++cnt);
        if(s[0] != 'F'){
            puts("Wrong Answer!");
            break;
        }
        puts("Accepted!");
    }
  }
```

- 效果图如下：

思考题

- 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？
    - 乘除法延迟远大于 ALU，整合进 ALU 那么根据木桶原理 CPU 整体周期将大幅增加。增加 HI 和 LO 寄存器可以让乘除法指令和其它指令并行执行，需要结果时再取出即可。
- 参照你对延迟槽的理解，试解释“乘除槽”。
    - 当乘除法进行或即将开始时，乘除有关指令会被阻塞在 IF/ID 流水线寄存器，相当于处于“乘除槽”。
- 举例说明并分析何时按字节访问内存相对于按字访问内存性能上更有优势。（Hint： 考虑C语言中字符串的情况）
    - 当访问类型只占一个字节时，比如 `char`。
- 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？

  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。

  此思考题请同学们结合自己测试CPU使用的具体手段，按照自己的实际情况进行回答。
    - 主要是数据冒险和控制冒险，分别通过暂停转发以及比较前移+延迟槽解决。
    - 数据生成器采用了特殊策略：单组数据中除了 0 和 31 号寄存器外，至多涉及 3 个寄存器。一方面，这样产生的代码中，邻近的指令几乎全部都存在数据冒险，可以充分测试转发和暂停；另一方面，当测试数据的组数一定多，几乎涉及了每个寄存器，避免了只测试部分寄存器。此外，所有跳转指令都是特殊构造的，不会进入死循环的同时如果跳转出错可以输出中体现。
    - 对于一些会产生异常的指令，为防止 MARS 报错，进行了一定的规避。
- 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？
    - 主要采用了指令分类的方法，P6 完全沿用了 P5 的分类方法，新增的指令对应的特点都没有脱离这些分类，因此对于每条指令而言，只需译码后将其加入对应的分类，数据通路部分和 P5 完全类似，转发部分完全不用改，暂停部分只需添加一个因乘除块而导致的暂停。