计算机组成原理实验报告

一、CPU设计方案综述

（一）总体设计概述

本CPU为Verilog实现的流水线MIPS - CPU，支持的指令集包含{`addu，subu，ori，lw，sw，beq，lui，j，jal，jr，nop`}。为了实现这些功能，CPU主要包含了IM、GRF、……，这些模块按照....的顶层设计逐级展开。

本设计中，不再设计`NPC`模块，而使用在`mips`中构造`MUX`的方式选择传递给PC的信号。

在本设计中，对模块之间解耦合是设计重点和精要，因此，单个模块的设计将会比较简单，和单周期非常相似，但是在`mips.v`中，要在这里写全部的`mux`，采取这样的方案是避免单列`mux.v`接口过于冗杂。

在本设计中，各级流水线寄存器尽可能多地保留数据，而非仅针对本设计支持的十条指令。

（二）关键模块定义

1. IM

介绍

Instruction Memory，指令存储器，支持初始化时一次性读入全部指令，并根据给定PC读出指令，支持0x3000-0x4000。

端口	输入/输出	位宽	描述
F_PC	I	32	输入指令地址
F_instr	O	32	输出对应指令

2. PC

介绍

获取指令，支持同步复位至0x3000。

端口	输入/输出	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
F_WE	I	1	是否继续读取下一条指令
NPC	I	32	下一条指令地址
PC	O	32	输出当前指令

hint:本设计中，将MUX模块全部集成在`mips.v`中，以便于接线。值得注意的是，本设计交由`mips.v`完成转发与选择信号，降低了模块间耦合度，即，`PC`仅需专注于在每个时钟上升沿给出下一个指令即可。

3. GRF

介绍

通用寄存器文件，可以存取32位数据，支持同步复位，寄存器编号为0~31，其中0号寄存器的读出值恒为0。

端口	输入/输出	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
GRF_WE(命名待更改)	I	1	写使能信号
PC(命名待更改)	I	32	当前指令（用于评测姬输出）
D_A1	I	5	A1
D_A2	I	5	A2
D_A3	I	5	A3
D_WD	I	32	WD
D_RD1	O	32	RD1
D_RD2	O	32	RD2

本设计的输入端口中的GRF_WE和PC同样由`mips.v`中实现。

值得注意的是，由于是在流水线中，因此存在一种情况，即，当前`GRF_WE == 1'b1`（要写入）且`A3 == A1/A2 != 1'b0`，即，当前要写入的寄存器恰为`A1`或`A2`，这时，如果继续读取原来GRF中的寄存器值将产生错误，需要读取的是WD。在本设计中将这部分实现于GRF。

4. 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	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
F_WE	I	1	写使能信号（用于处理暂停）
F_instr	I	32	下一个指令
F_PC	I	32	下一个PC
D_instr	O	32	当前指令
D_PC	O	32	当前PC
D_PCsrc	O	8	8'd0：PC = PC + 4。 8'd1：如果 Zero = 1，PC 跳转到 beq 指令对应的跳转地址；否则依旧执行 PC = PC + 4。(判断zero是否为0在mips.v中实现) 8'd2：跳转到 jal 指令和 j 指令对应的跳转地址。 8'd3：跳转到 jr 指令对应的跳转地址。

5. GSU（General Splitter Unit）

介绍

通用解码单元，对给定指令分析其各种数据。

端口	方向	位宽	描述
instr	I	32	给定指令
opcode	O	6	当前给定指令opcode
funct	O	6	当前给定指令funct
I25_21	O	5	instr[25:21]
I20_16	O	5	instr[20:16]
I15_11	O	5	instr[15:11]
I10_6	O	5	instr[10:6]
IMM16	O	16	instr[15:0]
IMM26	O	26	instr[25:0]
addu	O	1	判断addu
subu	O	1	判断subu
ori	O	1	判断ori
lw	O	1	判断lw
sw	O	1	判断sw
beq	O	1	判断beq
lui	O	1	判断lui
j	O	1	判断j
jal	O	1	判断jal
jr	O	1	判断jr
nop	O	1	判断nop

6. ID_EX

介绍

D级、E级间流水线寄存器，同时产生E级的控制信号。

端口	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
D_instr	I	32	下一指令
D_PC	I	32	下一PC
D_RD1	I	32	下一RD1
D_RD2	I	32	下一RD2
E_instr	O	32	当前指令
E_PC	O	32	当前PC
E_RD1	O	32	当前RD1
E_RD2	O	32	当前RD2
E_ALUOp	O	8	当前ALU选择信号（详见ALU模块说明）
E_ALU_Bsrc	O	8	选择ALUB的输入端（详见ALU模块说明）

7. ALU

端口	方向	位宽	描述
E_ALU_A	I	32	操作数A
E_ALU_B	I	32	操作数B
E_ALUOp	I	8	操作信号
E_ALUResult	O	32	输出

ALUOp信号表

ALUOp	功能	描述
8'd0	按位与	E_ALU_A & E_ALU_A
8'd1	按位或	E_ALU_A | E_ALU_B
8'd2	加法	E_ALU_A + E_ALU_B
8'd3	减法	E_ALU_A - E_ALU_B
8'd4	B左移16位	E_ALU_B<<16

8. EX_MEM

介绍

E级和M级之间流水线寄存器，同时产生M级的控制信号。

端口	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
E_instr	I	32	E指令
E_PC	I	32	EPC
E_RD1	I	32	ERD1
E_RD2	I	32	ERD2
E_ALUResult	I	32	EALUResult
E_DMWD	I	32	EDMWD
M_instr	O	32	M指令
M_PC	O	32	MPC
M_RD1	O	32	MRD1
M_RD2	O	32	MRD2
M_ALUResult	O	32	MALUResult
M_DMWD	O	32	MDMWD
M_DMWE	O	1	DM写使能信号
M_MemtoReg	O	8	选择回写到GRF的数据（详见）

9. DM

介绍

数据存储器，数据位宽32位，容量为**12KB**(32bit/word×**3072word**)，支持同步复位，起始地址为0x0000_0000。

端口	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
M_PC	I	32	当前指令PC
M_DMA	I	32	当前读取或写入数据地址
M_DMWD	I	32	写入DM的数据
M_DMWE	I	1	DM写使能信号
M_DMRD	O	32	读出数据

10. MEM_WB

介绍

MEM_WB级寄存器，由M级向W级流水，同时产生W级的控制信号。

端口	方向	位宽	描述
clk	I	1	时钟信号
reset	I	1	同步复位信号
M_instr	I	32	M级instr
M_PC	I	32	M级PC
M_RD1	I	32	M级RD1
M_RD2	I	32	M级RD2
M_ALUResult	I	32	M级ALUResult
M_GRFWD	I	32	M级GRFWD
W_instr	O	32	W级instr
W_PC	O	32	W级PC
W_RD1	O	32	W级RD1
W_RD2	O	32	W级RD2
W_ALUResult	O	32	W级ALUResult
W_GRFWD	O	32	W级GRFWD
W_RegDst	O	8	W级RegDst
W_RegWE	O	1	W级RegWE

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	方向	位宽	描述
instr	I	32	需要分类的指令
j	O	1	需要读寄存器的跳转指令，如 `beq,jalr,jr`（在本CPU中暂不支持 `jalr`）
r	O	1	除 `jalr,jr` 外的 `R` 型指令（这里需要特别注意课上指令）；简单的 R 型算术运算指令：`addu、subu`。
i	O	1	`I` 型指令；带有立即数的 I 型算数、逻辑运算指令：`ori、lui`。
load	O	1	`load` 型指令，包括 `lw`
store	O	1	`store` 型指令，包括 `sw`；内存写入指令：`sw`。
jal	O	1	`jal,jalr` 型指令
rs	O	4	I[25:21]
rt	O	4	I[20:16]
rd	O	4	如果指令为 `jal` 输出31，如果是课上新添的需要写入特定寄存器的指令则为对应寄存器，否则为I[15:11]

值得注意的是，在 `DEC` 内部，实例化了 `GSU`,值得注意的是，

（三）重要机制实现方法

1. 跳转

NPC放在D级，beq比较器放在D级，jal指令将PC+8的值流水后在W级写入31号寄存器。j,jr指令直接跳转，但jr需要rs的值可能需要阻塞。

2. 流水线延迟槽

跳转指令后都有延迟槽，值都进行计算，因此jal需要写入PC+8的值。

3. 转发

转发发生在D,E,M级，D级转发的值来自M级和W级（因为GRF没采用内部转发机制），E级转发的值来自M级和W级，M级转发的值来自W级，主要针对lw,sw指令，如果lw指令后紧跟sw指令且使用的寄存器值相同，则需要转发。

二、测试方案

（一）典型测试样例

10组测试数据

testpoint 0

```
test_code.asm
ori $t0, $0, 123
addu $a0, $t0, $0
lw $a0, -123($a0)

lui $t3, 123
lui $t3, 0
lw $t4, 0($t3)

ori $t0, $0, 0x3028
addu $a0, $t0, $0
jr $a0

ori $t0, $0, 0x3038
sw $t0, 0($0)
lw $a0, 0($0)
jr $a0
nop

jal f
nop
j end
nop
f: jr $ra
nop
end:

jal tag
nop
tag: lw $t0, -0x3058($ra)

lw $s0, -0x3038($t0)
lw $s1, -0x3038($s0)
lw $s2, -0x3038($s1)
sw $t0, -0x3038($s2)
```

```
code.txt
3408007b
01002021
8c84ff85
3c0b007b
3c0b0000
8d6c0000
34083028
01002021
00800008
34083038
ac080000
8c040000
00800008
```

```
00000000
0c000c12
00000000
08000c14
00000000
03e00008
00000000
0c000c16
00000000
8fe8cfa8
8d10cfc8
8e11cfc8
8e32cfc8
ae48cfc8
```

```
test_ans
@00003000: $ 8 <= 0000007b
@00003004: $ 4 <= 0000007b
@00003008: $ 4 <= 00000000
@0000300c: $11 <= 007b0000
@00003010: $11 <= 00000000
@00003014: $12 <= 00000000
@00003018: $ 8 <= 00003028
@0000301c: $ 4 <= 00003028
@00003024: $ 8 <= 00003038
@00003028: *00000000 <= 00003038
@0000302c: $ 4 <= 00003038
@00003038: $31 <= 00003040
@00003050: $31 <= 00003058
@00003058: $ 8 <= 00003038
@0000305c: $16 <= 00003038
@00003060: $17 <= 00003038
@00003064: $18 <= 00003038
@00003068: *00000000 <= 00003038
```

（二）自动测试工具

1. 测试样例生成器

2. 自动执行脚本

3. 正确性判定脚本

三、思考题

流水线冒险

1. 在采用本节所述的控制冒险处理方式下，PC 的值应当如何被更新？请从数据通路和控制信号两方面进行说明。
    - 将 IFU 的 nPC 输入端和 D 级的一个 MUX 输出相连，该 MUX 的输入为正常执行命令的下一个 PC 地址以及各种跳转指令对应的下一个 PC 地址。
    - ID/EX 流水线寄存器直接译码产生 PCSrc 控制信号，配合前移至 D 级的比较器产生的相等信号，用来控制上述的 MUX。
2. 对于 jal 等需要将指令地址写入寄存器的指令，为什么需要回写 PC+8 ？

    PC + 4 的位置是延迟槽，跳回时应该跳到延迟槽下一条指令，即 PC + 8。

数据冒险的分析

为什么所有的供给者都是存储了上一级传来的各种数据的**流水级寄存器**，而不是由 ALU 或者 DM 等部件来提供数据？

如果从非流水线寄存器部件转发，那么某一级的总延迟就会增加，从而根据木桶效应，时钟周期就会增加，总效率反而降低，得不偿失。

AT 法处理流水线数据冒险

1. “转发（旁路）机制的构造”中的Thinking 1-4；
    1. 如果不采用已经转发过的数据，而采用上一级中的原始数据，会出现怎样的问题？试列举指令序列说明这个问题。
        - 计算过程或存储过程中会用到还未更改过的寄存器值，从而出错。例如：

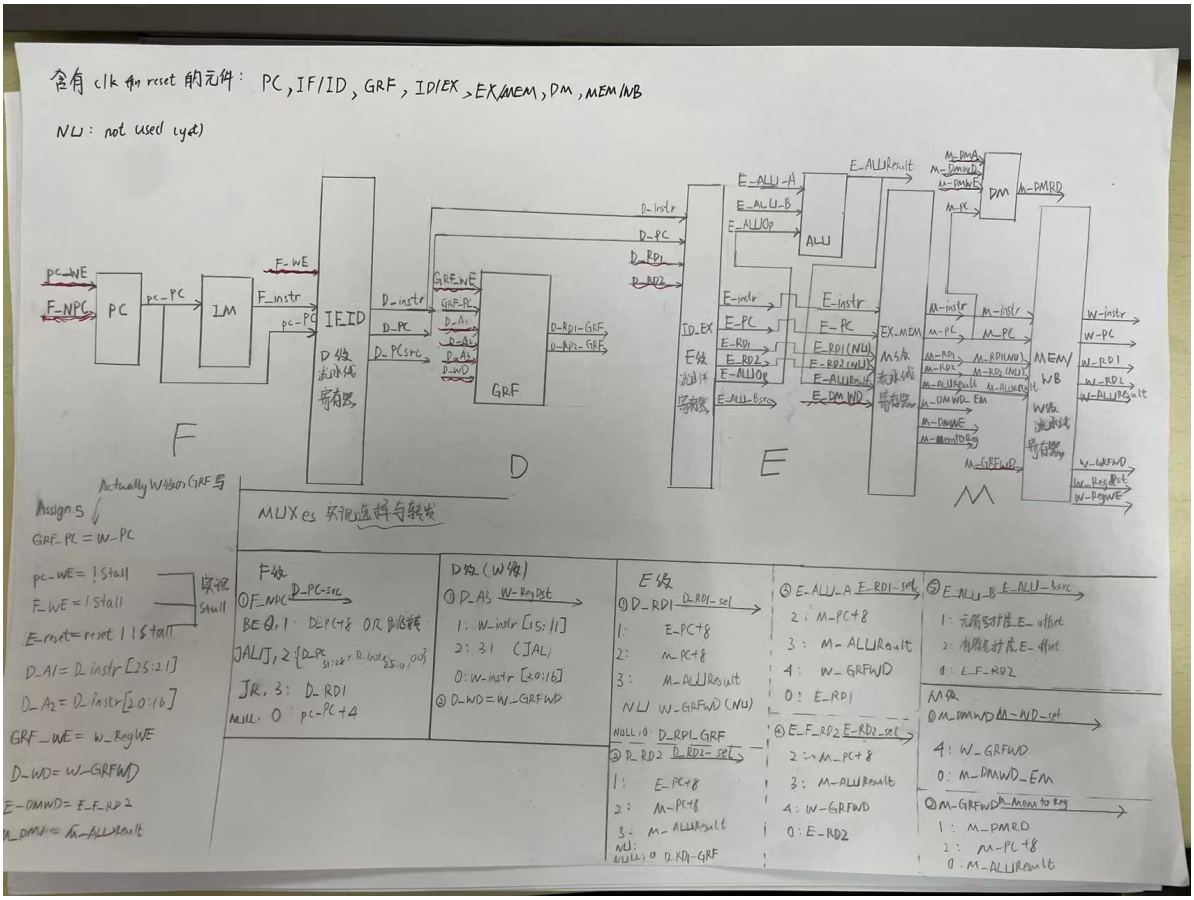
        ```
        ori $1, $0, 1
        nop
        nop
        nop
        nop
        lw $1, 0($0)
        sw $1, 4($0)
        ```

        这样 `sw` 指令就会把 1 存到 DM 中。
    2. 我们为什么要对GPR采用内部转发机制？如果不采用内部转发机制，我们要怎样才能解决这种情况下的转发需求呢？
        - GPR采用内部转发机制相当于 MEM/WB 流水线寄存器的值直接实时反馈到 GPR 的输出端，从而当前处于 D 级的指令可以直接用到对应寄存器的值。
        - 如果不采用内部转发机制，需要额外建立从 MEM/WB 流水线寄存器转发到 D 级的数据通路。
    3. 为什么0号寄存器需要特殊处理？
        - 因为指令可以对 0 号寄存器赋值，只是不会造成实际作用，但是转发过程中如果不特判就默认 0 号寄存器的值被更改了，从而造成错误。
    4. 什么是“最新产生的数据”？
        - 根据指令的执行顺序，越后执行的指令更改的寄存器的值越新，按照 ID/EX、EX/MEM、MEM/WB 的顺序，越靠前所转发出的信息越新，因此优先级更高。
2. 在 AT 方法讨论转发条件的时候，只提到了“供给者需求者的A相同，且不为 0”，但在 CPU 写入 GRF 的时候，是有一个 we 信号来控制是否要写入的。为何在 AT 方法中不需要特判 we 呢？为了**用且仅用** A 和 T 完成转发，在翻译出 A 的时候，要结合 we 做什么操作呢？
    - AT 方法如是说：

        > 只要**当前位点的读取寄存器地址和某转发输入来源的写入寄存器地址相等且不为0**

        那么既然是要写入的，WE 必然为 1，因此不用特判。
    - 对于确实要写入的寄存器，不做改变；对于条件判断后不要写，将寄存器地址赋为0.

后记



Sorce

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 `addu，subu，lw，sw，beq，ori，lui，addi，j，jal，jr，jalr，nop`。为了实现这些功能，CPU 主要包含了 `IFU，GRF，DM，ALU，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 * 1024，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 位按位与、按位或、加法、减法、左移 16 位、判断相等的功能。

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。

4.DM

介绍

数据存储器，可以存取 32 位数据，容量为 32bit * 1024，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。
WE	I	1	写入数据信号，高电平有效。
clk	I	1	时钟信号。
reset	I	1	同步复位信号。
RD	O	32	输出 A 指定地址的数据。

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	读数据	读出 A 地址对应的数据到 RD。
3	写数据	当 WE 有效且时钟上升沿到来时，将 WD 的数据写入 A 对应的地址。

5. 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	2	0：PC = PC + 4。 1：如果 Zero = 1，PC 跳转到 beq 指令对应的跳转地址；否则依旧执行 PC = PC + 4。 2：跳转到 jal 指令和 j 指令对应的跳转地址。 3：跳转到 jr 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。

6. 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	2	0：ALU 的 B 输入端选择 RD2 输出端。 1：ALU 的 B 输入端选择 I[15:0] 的 32 位无符号扩展。 2：ALU 的 B 输入端选择 I[15:0] 的 32 位有符号扩展。

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。

7. 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。
MemWrite	O	1	0：DM 的写使能端 WE 无效。 1：DM 的写使能端 WE 有效。
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。

8. 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。

9. 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。
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

10.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，包括 `beq, jalr, jr`。
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，包括 `lw`。
st	O	1	store 型指令，包括 `sw`。
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]。

11.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，具体见下方真值表。

控制信号真值表

		R	addu	subu	lw	sw	ori	lui	addi	beq	j	jal	jr	jalr
	Op	000000			100011	101011	001101	001111	001000	000100	000010	000011		
	Func		100001	100011									001000	001001
PCSrc		0			0	0	0	0	0	1	2	2	3	3
MemtoReg		0			1		0	0	0			2		2
MemWrite		0				1								
ALUOp			2	3	2	2	1	4	2					
ALUSrc		0			2	2	1	1	2	0				
RegWrite		1			1	0	1	1	1	0		1		
RegDst		1			0		0	0	0			2		

重要机制实现方法

使用 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，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。

- 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，将比对信息输入到 `log.txt`。
- 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),
    I(1, 7),
    J(8, 11),
    IJ(1, 11),
    one(3, 7);

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("lw $%d, %d($%d)\n", getR(), siz(mt) >> 2 << 2, x);
            tot += 2;
            break;
        case 2:
            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 3:
            printf("addu $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 4:
            printf("subu $%d, $%d, $%d\n", getR(), getR(), getR());
            tot++;
            break;
        case 5:
            printf("addi $%d, $%d, %d\n", getR(), getR(), 0);
            tot++;
            break;
        case 6:
            printf("ori $%d, $%d, %d\n", getR(), getR(), u16(mt));
            tot++;
            break;
        case 7:
            printf("lui $%d, %d\n", getR(), u16(mt));
            tot++;
            break;
        case 8:
            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 9:
            printf("j label%d\n", ++cnt);
            solve(I(mt));
            solve(I(mt));
            printf("label%d: ", cnt);
            solve(I(mt));
            tot++;
            break;
        case 10:
            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 11:
            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());
```

```
    }
}
```

- bat 代码：

```
:start
gen.exe
java -jar Mars_Changed.jar db mc CompactDataAtZero nc test.asm > m.out
java -jar Mars_Changed.jar mc CompactDataAtZero a dump .text HexText
code.txt test.asm
iverilog -o tb.out -y D:\coding\CO\Verilog\P5
D:\coding\CO\Verilog\P5\tb.v
vvp tb.out > v.out
del.exe
fc v.out m.out >> log.txt
goto start
```

- 效果图如下：
  -
  -

思考题

流水线冒险

1. 在采用本节所述的控制冒险处理方式下，PC的值应当如何被更新？请从数据通路和控制信号两方面进行说明。
   - 将 IFU 的 nPC 输入端和 D 级的一个 MUX 输出相连，该 MUX 的输入为正常执行命令的下一个 PC 地址以及各种跳转指令对应的下一个 PC 地址。
   - ID/EX 流水线寄存器直接译码产生 PCSrc 控制信号，配合前移至 D 级的比较器产生的相等信号，用来控制上述的 MUX。
2. 对于jal等需要将指令地址写入寄存器的指令，为什么需要回写PC+8？
   - PC + 4 的位置是延迟槽，跳回时应该跳到延迟槽下一条指令，即 PC + 8。

数据冒险的分析

1. 为什么所有的供给者都是存储了上一级传来的各种数据的**流水级寄存器**，而不是由ALU或者DM等部件来提供数据？
   - 如果从非流水线寄存器部件转发，那么某一级的总延迟就会增加，从而根据木桶效应，时钟周期就会增加，总效率反而降低，得不偿失。

AT法处理流水线数据冒险

1. “转发（旁路）机制的构造”中的Thinking 1-4；
   1. 如果不采用已经转发过的数据，而采用上一级中的原始数据，会出现怎样的问题？试列举指令序列说明这个问题。
      - 计算过程或存储过程中会用到还未更改过的寄存器值，从而出错。例如：

```
ori $1, $0, 1
nop
nop
nop
nop
lw $1, 0($0)
sw $1, 4($0)
```

这样 `sw` 指令就会把 1 存到 DM 中。

2. 我们为什么要对GPR采用内部转发机制？如果不采用内部转发机制，我们要怎样才能解决这种情况下的转发需求呢？
    - GPR采用内部转发机制相当于 MEM/WB 流水线寄存器的值直接实时反馈到 GPR 的输出端，从而当前处于 D 级的指令可以直接用到对应寄存器的值。
    - 如果不采用内部转发机制，需要额外建立从 MEM/WB 流水线寄存器转发到 D 级的数据通路。
3. 为什么0号寄存器需要特殊处理？
    - 因为指令可以对 0 号寄存器赋值，只是不会造成实际作用，但是转发过程中如果不特判就默认 0 号寄存器的值被更改了，从而造成错误。
4. 什么是“最新产生的数据”？
    - 根据指令的执行顺序，越后执行的指令更改的寄存器的值越新，按照 ID/EX、EX/MEM、MEM/WB 的顺序，越靠前所转发出的信息越新，因此优先级更高。

2. 在AT方法讨论转发条件的时候，只提到了“供给者需求者的A相同，且不为0”，但在CPU写入GRF的时候，是有一个we信号来控制是否要写入的。为何在AT方法中不需要特判we呢？为了**用且仅用**A和T完成转发，在翻译出A的时候，要结合we做什么操作呢？
    - AT 方法如是说：

      > 只要**当前位点的读取寄存器地址和某转发输入来源的写入寄存器地址相等且不为0**

      那么既然是要写入的，WE 必然为 1，因此不用特判。
    - 不需要做任何操作。

在线测试相关说明

1. 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。