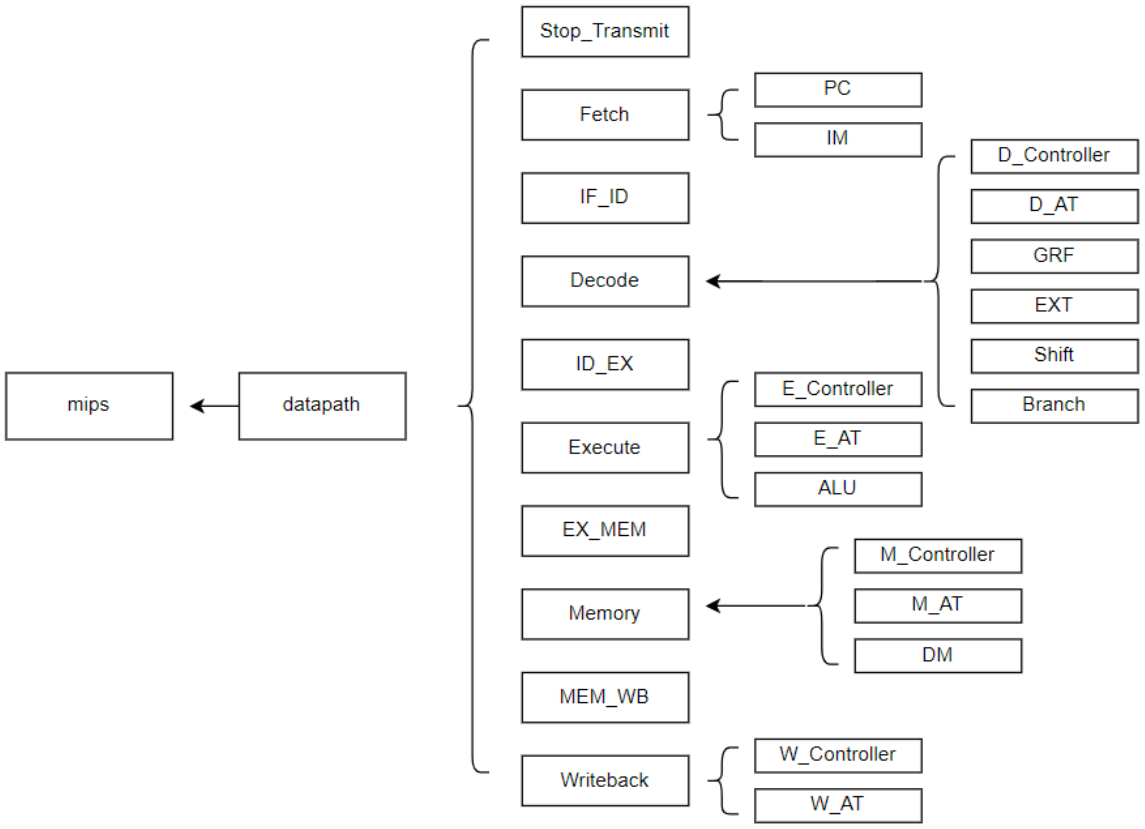
CPU设计文档

构造



结构设计

数据通路

总流水线阶段

阶段	简称	功能概述
取指阶段（Fetch）	F	从指令存储器中读取指令
译码阶段（Decode）	D	从寄存器文件中读取源操作数并对指令译码以便得到控制信号
执行阶段（Execute）	E	使用 ALU 执行计算
存储阶段（Memory）	M	读或写数据存储器
写回阶段（Writeback）	W	将结果写回到寄存器文件

> **取指阶段（Fetch）**
>
> - PC
> - NPC
> - IM
>
> **IF/ID寄存器**
>
> **译码阶段（Decode）**
>
> - GRF
> - EXT
>
> **ID/EX寄存器**
>
> **执行阶段（Execute）**
>
> - ALU
>
> **EX/MEM寄存器**
>
> **存储阶段（Memory）**
>
> - DM
>
> **MEM/WB寄存器**
>
> **写回阶段（Writeback）**
>
> - GRF

指令过程详情

	取指阶段（Fetch）	IF/ID寄存器	译码阶段（Decode）	ID/EX寄存器	执行阶段（Execute）	EX/MEM寄存器	存储阶段（Memory）	MEM/WB寄存器	写回阶段（Writeback）
R型指令	从 PC 寄存器取地址，根据地址从 IM 获得指令编码	存入指令	根据相应寄存器 rs、rt 地址获取其中数据	存入 rs,rt数据，rd地址	ALU 计算	存入ALU计算结果，rd地址	无	存入ALU计算结果，rd地址	根据rd 地址与 WD（ALUout），将相应数据写入 rd 中
I型指令	从 PC 寄存器取地址，根据地址从 IM 获得指令编码	存入指令	获取rs数据,将指令低 16 位取出经过 EXT扩展（Shift移位）	存入rs数据,rt地址，EXT扩展（Shift移位）后结果	ALU 计算（无）	存入ALU计算（Shift移位）结果，rt地址	无	存入ALU计算（Shift移位）结果，rt地址	根据rt 地址与 WD（ALUout/Shift），将相应数据写入 rt中
Lw	从 PC 寄存器取地址，根据地址从 IM 获得指令编码	存入指令	获取rs数据,将指令低 16 位取出经过 EXT扩展	存入rs数据,rt地址，EXT扩展后结果	通过 rs 寄存器的数据与立即数计算地址	存入ALU计算结果，rt地址	根据 ALU 计算出的地址仿存	存入DM取值结果，rt地址	根据rt 地址与 WD（DMout），将相应数据写入 rt中
Sw	从 PC 寄存器取地址，根据地址从 IM 获得指令编码	存入指令	获取rs数据,将指令低 16 位取出经过 EXT扩展	存入 rs,rt数据,EXT扩展后结果	通过 rs 寄存器的数据与立即数计算地址	存入ALU计算结果，rt数据	将 rt 寄存器的数据写入 ALU 计算出的地址	无	无
跳转B型指令	从 PC 寄存器取地址，根据地址从 IM 获得指令编码	存入指令	比较读取出的 rs 和 rt 寄存器的数据，若相等，则通过多选器将拓展后的立即数作为下一周期的 PC 写入 PC 寄存器	无	无	无	无	无	无
跳转J型指令	从 PC 寄存器取地址，根据地址从 IM 获得指令编码	存入指令	通过多选器将拓展后的立即数作为下一周期的 PC 写入	存入PC + 8	无	存入PC + 8	无	存入PC + 8	将跳转前的 PC + 8 写入 31 号 ra 寄存器
跳转R型指令	从 PC 寄存器取地址，根据地址从 IM 获得指令编码	存入指令	通过多选器将 rs 寄存器的数据作为下一周期的 PC 写入	无	无	无	无	无	无

译码器

- **控制器**

	cal_r	cal_i	load	store	mult_div	mf	mt	branch	jal	jr	nop
RegDst	01(Rd)	00(Rt)	00	x	x	01	x	x	10(Reg31)	00	x
MemtoReg	000(ALU)	根据指令	001(DM)	x	x	100(HI/LO)	x	x	011(Adder)	00	x
Regwrite	1	1	1	0	0	1	0	0	1	0	x
ALUop	根据指令	根据指令	010(加)	010(加)	x	x	x	110(减)	x	x	x
ALUsrc	0(Rdata2)	1(EXT)	1	1	0	x	x	0	x	x	x
Memwrite	0	0	0	根据指令	0	0	0	0	x	x	x
Branch	x	x	x	x	x	x	x	1(beq)/4(bne)	2(jal)	3(jr)	x
EXT	x	01(0扩展)	00(符号扩展)	00	x	x	x	00	00	x	x
Shift	x	根据指令	x	x	x	x	x	x	x	x	x

- **乘除控制**

	mult_div	mf	mt
MDop	**根据指令**	**根据指令**	**x**
MFsrc	**x**	**根据指令**	**x**
MTDst	**x**	**x**	**根据指令**
MDwrite	**0**	**0**	**1**

- **具体指令**

cal_r

opcode	000000	000000	000000	000000	000000	000000
funct	100000	100010	100100	100101	101010	101011
	add	sub	and	or	slt	sltu
ALUop	010	110	000	001	011	100

cal_i

opcode	001101	001100	001000	001111
	ori	andi	addi	lui
MemtoReg	000	000	000	010
ALUop	001	000	010	x
Shift	x	x	x	1

load

opcode	100011	100001	100000
	lw	lh	lb

store

opcode	101011	101001	101000
	sw	sh	sb

mult_div

opcode	000000	000000	000000	000000
funct	011000	011001	011010	011011
	mult	multu	div	divu
MDop	001	010	011	100

mf

opcode	000000	000000
funct	010000	010010
	mfhi	mflo
MFsrc	0	1

mt

opcode	000000	000000
funct	010001	010011
	mthi	mtlo
MTDst	0	1

branch

opcode	000100	000101
	beq	bne

jal

opcode	000011
	jal

jr

opcode	000000
funct	001000
	jr

乘除模块

- **信号**

Start

- **只能有效 1 个时钟周期**
- 启动乘除法运算。

Busy

- 通过 Busy 输出标志来反映执行乘除法的延迟

- **寄存器**

HI

LO

- **行为**

- 自 Start 信号有效后的第 1 个 clock 上升沿开始，乘除法部件开始执行运算，同时将 Busy 置位为 1。
- 在运算结果保存到 HI 寄存器和 LO 寄存器后，Busy 位清除为 0。
- 当 Busy 信号或 Start 信号为 1 时，`mult, multu, div, divu, mfhi, mflo, mthi, mtlo` 等乘除法相关的指令均被阻塞在 D 流水级。
- 数据写入 HI 寄存器或 LO 寄存器，均只需 1 个时钟周期。

字节使能

`m_data_byteen[3:0]` 是字节使能信号，其最高位到最低位分别与 `m_data_wdata` 的 `[31:24]`、`[23:16]`、`[15:8]` 及 `[7:0]` 对应（即一位对应一个字节）。

例如，若 `m_data_byteen[3]` 为 1，则 `m_data_wdata[31:24]` 会被写入到 `m_data_addr` 所指向 word 的 `[31:24]`，依此类推。

若 `m_data_byteen` 的任意一位为 1，则代表当前需要写入内存，也就是说可以用 `|m_data_byteen` 代替数据存储器的写使能信号。

`m_data_byteen[3:0]` 主要用于支持 `sb`、`sh` 这两条指令。当处理器执行 `sb`、`sh` 指令时，只要对 EX/MEM 保存的 ALU 计算结果即 32 位地址的低两位进行解读，产生相应的 `m_data_byteen` 信号，就可以“通知” Testbench 中的 DM 该写入哪些字节，具体规则如下：

sw 指令：向 GPR[rt] 写入对应的字

地址 [1:0]	m_data_byteen [3:0]	用途
XX	1111	`m_data_wdata[31:24]` 写入 byte3 `m_data_wdata[23:16]` 写入 byte2 `m_data_wdata[15:8]` 写入 byte1 `m_data_wdata[7:0]` 写入 byte0

sh 指令：向 GPR[rt]15:0 写入对应的半字

地址 [1:0]	m_data_byteen [3:0]	用途
0X	0011	`m_data_wdata[15:8]` 写入 byte1 `m_data_wdata[7:0]` 写入 byte0
1X	1100	`m_data_wdata[31:24]` 写入 byte3 `m_data_wdata[23:16]` 写入 byte2

sb 指令：向 GPR[rt]7:0 写入对应的字节

地址[1:0]	m_data_byteen [3:0]	用途
00	0001	`m_data_wdata[7:0]` 写入 byte0
01	0010	`m_data_wdata[15:8]` 写入 byte1
10	0100	`m_data_wdata[23:16]` 写入 byte2
11	1000	`m_data_wdata[31:24]` 写入 byte3

BE 扩展模块需放在与数据存储器交互数据之前。显然，BE 扩展功能部件还需要有来自控制器的控制信号。

数据扩展模块

对于 `lb`、`lh` 来说，我们需要额外增加一个数据扩展模块。这个模块把从数据存储器读出的数据做符号扩展。

以 `lb` 为例，数据扩展模块输入数据寄存器的 32 位数据，根据 ALU 计算出来的地址最低 2 位从中取出**特定的字节**，并以该字节的最高位为符号位做符号扩展。

参考的接口定义如下：

信号名	方向	描述
A[1:0]	I	最低两位的地址
Din[31:0]	I	输入的 32 位数据
Op[2:0]	I	数据扩展控制码 000：无扩展 001：无符号字节数据扩展 010：符号字节数据扩展 011：无符号半字数据扩展 100：符号半字数据扩展
Dout[31:0]	O	扩展后的 32 位数据

冒险

- **类型**

> ### 结构冒险
>
> - 不同指令同时需要使用同一资源
>
> ### 控制冒险
>
> - 分支指令（如 beq ）的判断结果会影响接下来指令的执行流
>
> ### 数据冒险
>
> - 后面指令需求的数据，正好就是前面指令供给的数据，而后面指令在需要使用数据时，前面供给的数据还没有存入寄存器堆，从而导致后面的指令不能正常地读取到正确的数据

- **解决办法**

> ### 阻塞
>
> - 在 D 级阻塞的时候，像下一流水级传递的指令不应当是 D 级指令，否则 D 级指令还是会向下一流水级传递。所以我们应当插入“指令气泡（bubble）”，也就是 nop 空指令来避免这种情况。实现 CPU 流水级的“空转”
>
> ### 提前分支判断
>
> - 将分支判断提前到 D 级

延迟槽

- 延迟槽是基本上不需要实现的，也就是说，只要不考虑 F 级指令的作废问题，就是实现了延迟槽。唯一需要变更的是，**对于 jal 指令，应当向 31 号寄存器写入 D_PC + 8 或者 F_PC + 4（当 jal 指令在 D 级时）**

转发

- 数据并非一定要等到写入寄存器堆中才可以被使用，直接从后面的流水级的供给者把计算结果发送到前面流水级的需求者来引用
- **AT 法**
  - **A模型**

    A 模型描述的是一个很显然的事情，就是需求者和供给者转发的数据必须是同一个寄存器的值。

    我们的 A 指 Address，也就是**寄存器的地址**（编号）。在转发的时候需要检验需求者和供给者的对应的 A值是否相等，且不为 0.
  - **T模型**

    需求时间-供给时间模型

    对于某一个指令的某一个数据需求，我们定义**需求时间 T*use***为：这条指令**位于 D 级的时候**，再经过多少个时钟周期就必须要使用相应的数据。

```
Tuse的两条性质：

- Tuse是一个定值，每个指令的Tuse 是一定的；
- 一个指令可以有两个Tuse 值。

对于某个指令的数据产出，我们定义供给时间Tnew 为：位于某个流水级的某个指令，它经过多少个时钟周期可以算出结果并且存储到流水级寄存器。

Tnew的两条性质：

- Tnew 是一个动态值，每个指令处于流水线不同阶段有不同的 Tnew值；
- 一个指令在一个时刻至多有一个 Tnew值（一个指令至多写一个寄存器）
```

- **结论**

  当 T*use* ≥T*new*，说明需要的数据可以及时算出，可以通过**转发**来解决。

  当 T*use* <T*new*，说明需要的数据不能及时算出，必须**阻塞**流水线解决。
- **各指令的T*use* ， T*new***

	Rs_T*use*	Rt_Tuse	E_T*new*	M_T*new*	W_T*new*

	Rs_T*use*	Rt_Tuse	E_T*new*	M_T*new*	W_T*new*
cal_r	1	1	1	0	0
cal_i	1	1	1	0	0
Load	1	0	2	1	0
Store	1	2	1	0	0
跳转 B 型指令	0	0	0	0	0
跳转 J 型指令	0	0	0	0	0
跳转 R 型指令	0	0	0	0	0
MF	0	0	1	0	0
MT	1	0	1	0	0

我们知道，当 T*use* ≥T*new* 时，可以通过转发解决；当 T*use* <T*new*必须阻塞流水线。

根据上述的 T*use*和T*new* 的值，我们做出策略矩阵，其中 F 表示转发，S 表示暂停：

		E_Tnew			M_Tnew		W_Tnew
		2	1	0	1	0	0
Rs_Tuse	0	S	S	F	S	F	F
	1	S	F	F	F	F	F
Rt_Tuse	0	S	S	F	S	F	F
	1	S	F	F	F	F	F
	2	F	F	F	F	F	F

根据上表，可以看出只有四种情况需要阻塞：

- E_T*new*=2 ，T*use*=0
- E_T*new*=1 ，T*use*=0
- M_T*new*=1 ，T*use*=0
- E_T*new*=2 ，T*use*=1

除了这四种，剩下的情况就是需要转发的了。在转发中，我们需要特别注意**转发的优先级问题和 rt 域有效性问题。**

转发的优先级

- 我们要选择流水线中**靠前的"新鲜"的数据**进行转发

rt 域有效性问题

- 指有些指令的 rt 域不是用来表示读寄存器编号的，比如 j 指令没有 rt 域、ori 指令的 rt 域表示写入寄存器的编号，那么我们是否需要对这些指令进行特判呢？答案是不需要。对于 rt 域无效的指令，即使我们转发了相应的数据，也不会影响流水线的正确性，因此**无需特判。**

为解决流水线数据冒险，我们可以单独设计一个冒险控制模块，输入用于判断暂停和转发的 A 和 T 信号，输出暂停和转发的控制信号，下面让我们一起来分析该模块内部的逻辑。

- **暂停**

  我们可以根据T*use* 和T*new*值的不同组合构造出 4 种暂停信号。当然除了上述 **T 的条件，暂停时还需要满足 A 的条件**（即读寄存器和写寄存器编号相同且不为 0、写寄存器信号有效）。最后总的暂停信号把两个寄存器分别的暂停信号或起来即可。

  简单起见，我们约定暂停只发生在 D 级，因此当暂停信号有效时，我们需要保持 D 级流水寄存器，清空 E 级流水寄存器。

- **转发**

  我们首先要在需要转发的点位放一个多路选择器，可以让 0 路对应原始数据，剩下的路按照数据优先级从低到高排列，然后利用一个转发控制信号选择正确的值。转发控制信号在冒险控制模块内生成，具体的判断条件是：**读寄存器和写寄存器编号相同且不为 0、写寄存器信号有效（A 条件）以及转发流水级 T*new*=0（T 条件，表示此时数据已经准备好了）**。

- **解决冒险的流水线实现**

译码器的实现

- 在译码的时候，不能只译码出指令信息，还需要译码出指令相关的 AT 信息。只有译码出了 AT 信息，才可以帮助我们进行流水线的决策。
- 采用集中式译码的时候，AT 信息只在 D 级被译码了一次，但是同一个指令的 AT 值在不同的流水线阶段可能会发生变化，所以这就要求我们应当在流水线寄存器中完成流水级间的变化，比如某种实现的 Tnew = max{Next-Tnew-1 , 0} 。也可以不在流水线寄存器中完成，而是开辟一个新的功能单元完成
- 对于分布式译码，因为 AT 信息在每个流水线都被译码，所以就不存在传递变化的问题，但是译码器就必须知道自己所在的流水级，才能给出对应的正确的 AT 值。

阻塞的实现

- 当一个指令到达 D 级后，我们需要将它的T*use*值与后面每一级的 T*new*进行比较（当然还有 A 值的校验），当 T*use* <T*new* 时，我们需要阻塞流水线。
- 阻塞的实现需要改造流水线寄存器和 PC ，我们需要让它们具有以下功能：
  - 冻结 PC 的值；
  - 冻结 D 级流水线寄存器的值；
  - 将 E 级流水线寄存器清零（这等价于插入了一个 nop 指令）。

  此外，还有一个考虑，就是**复位信号和阻塞信号的优先级问题**。请仔细设计信号的优先级来保证流水线的正确性

内部转发的实现

- 对 GPR 采用**内部转发**机制.也就是说，当前 GPR 被写入的值会即时反馈到读取端上。
- 具体的说，当读寄存器时的地址与同周期写寄存器的地址相同时，我们将读取的内容改为写寄存器的内容，而不是该地址可以索引到的寄存器文件中的值。

> **转发的实现**
>
> - 当一个指令到达 D 级后，我们需要将它的T*use*值与后面每一级的 T*new*进行比较（当然还有 *A* 值的校验），当 T*use* ≥T*new* 时，我们需要进行转发。
> - 为了实现转发，我们需要两种多路选择器 MUX，分别对应转发的供给者和需求者
>   - 这是第一种MUX，输入是流水线寄存器的输出，输出是当前指令对应的写数据。
>   - 这是第二种 MUX，输入是各级的第一种 MUX 的输出，输出是当前正确（或者可以容忍的错误）的读数据。
>
> **控制器的实现**
>
> - 我们解决冒险需要进行 AT 值的比较判断，并需要根据判断的结果产生特定的控制信息。这些功能要求我们丰富我们的控制器，使其可以支持这些功能。
> - 我们的控制器需要产生的信号包括但不限于**冻结信号，刷新信号，供给者选择器信号，需求者选择器信号**等。

具体部件

- **Controller（控制器，根据 splitter 得到的 6 位 funct 码和 6 位 instr_index 码确定指令的类型并输出对应的控制信号）**
  - 控制器的设计，从最基本的层面来说，是一个**译码**的过程，将每一条机器指令中包含的信息，转化为给 CPU 各部分的控制信号
  - assign疯狂赋值
  - 指令的opcode和funct

- **Datapath**
  - PC
    - 初始化，起始地址 0x00003000
    - 同步复位
    - 根据PC_sel确定下一条指令地址

指令	PC_sel	Next_PC
无	00	PC+4
branch	01	PC + 4 + Shift_out
jal	10	{{PC[31 : 28]}, instruction[25:0], {2'b00}}
jr	11	grf(Registe[31])

  - IM
    - 根据地址输出指令
    - assign
    - 初始化，起始地址 0x00000000

        - 容量为 4096 × 32bit。

    - GRF
        - 寄存器堆
        - 同步复位
        - assign读，always写
        - 初始化
        - 0寄存器特判

    - ALU
        - assign疯狂输出
        - zero判定

    - DM
        - 注意读写判定
        - assign读，always写
        - 同步复位,复位值为 0x00000000
        - 初始化
        - 容量为 3072 × 32bit

    - EXT
        - assign输出
        - 注意拓展类型

    - Shift
        - assign输出

    - branch
        - assign输出
        - 注意各个跳转指令判定

模块

- PC
- IM
- GRF：寄存器相关
    - Reg1,Reg2
    - Wreg
    - Wdata
- ALU：运算
- DM：内存
- EXT：位扩展
- Shift：移位

- Controller：控制器

结构图

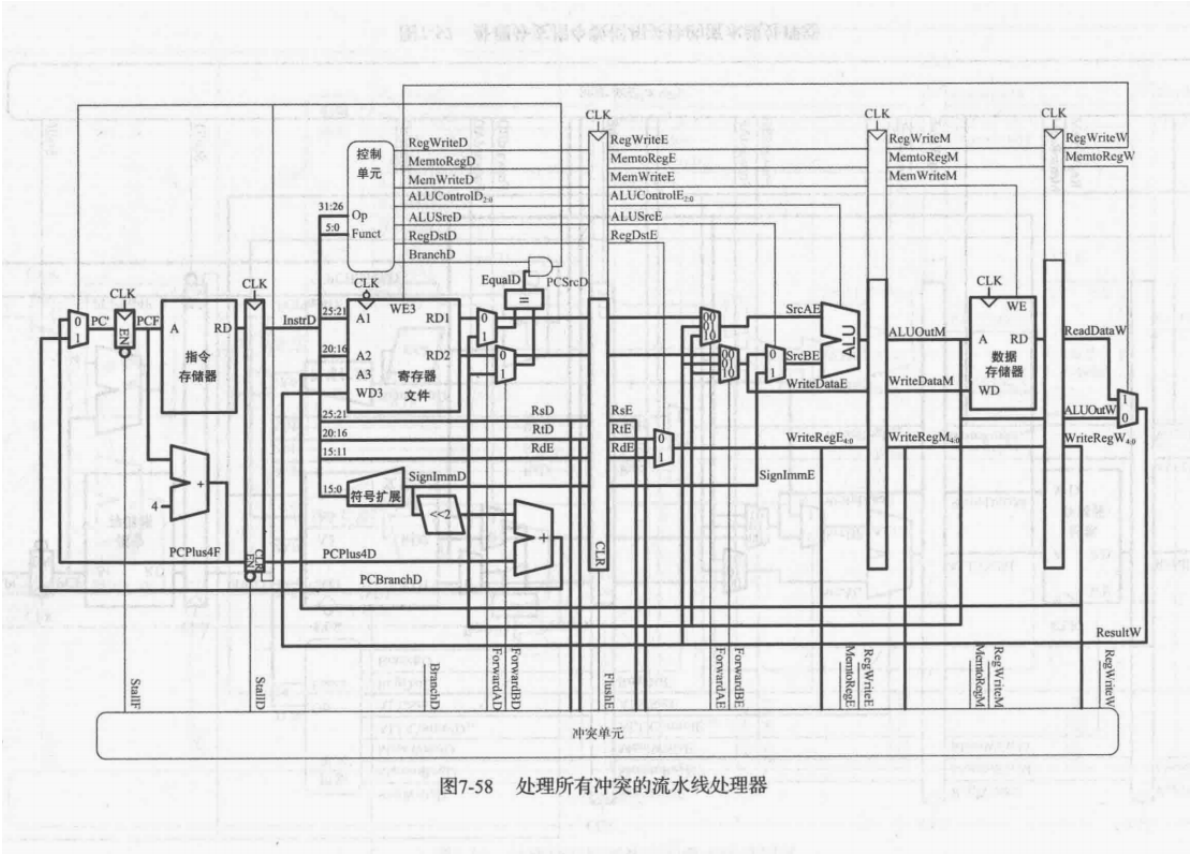


图7-58　处理所有冲突的流水线处理器

测试方案

见附

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？

因为乘除法运算效率远低于 ALU，整合进 ALU 会使效率降低。

独立的 HI、LO 寄存器能增加效率。使用独立寄存器可以使得乘除法运算在进行多个时钟周期时，尽可能的不阻塞其他指令。

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！
> 然后在判断乘数寄存器的低位是低电平还是高电平（0/1）！
> 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，然后在让对齐的被除数与除数比较(双符号位判断)。
> 这里说一下什么是双符号位判断：
> 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在商寄存器上商：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?

当 Busy 信号为 1 时，且后面为`mult，multu，div，divu，mfhi，mflo，mthi，mtlo`等乘除法相关的指令时，阻塞在 D 流水级。

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）

`m_data_byteen[3:0]`是字节使能信号，其最高位到最低位分别与`m_data_wdata`的`[31:24]`、`[23:16]`、`[15:8]`及`[7:0]`对应（即一位对应一个字节）。其一一对应性使得写入的字节位置更为清晰。

同时，所有的S类指令都可以通过字节使能信号决定输入，具有统一性。

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?

实际从 DM 获得的数据和向 DM 写入的数据并不是一字节。当我们处理与以字节为基础单元相关的问题时，按字节读和按字节写的效率会高于按字读和按字写。

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?

- 通过宏定义对指令进行规范，并将指令的相关定义给提出来作为一个单独文件，并通过头文件调用的方式使用。使译码时更加清晰易懂。
- 将指令大致分类，每一类指令的控制信号以及操作等相似，方便译码和数据冲突处理。

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?

在p6中主要是乘除法相关指令之间的冲突。当 Busy 信号或 Start 信号为 1 时，`mult，multu，div，divu，mfhi，mflo，mthi，mtlo`等乘除法相关的指令均被阻塞在 D 流水级。于是需要一个新的stop信号，用于确定关于乘除法相关指令之间的冲突是否需要阻塞。

其他指令之间产生的阻塞与P5大致相同，处理也比较相似，只需要将新增指令加入到其所属类指令的转发阻塞信号之中即可。

测试样例 如

```
#mult/div

ori $1,$0,10
ori $2,$0,5
mult $1,$2
mfhi $3
mflo $4
multu $1,$2
mfhi $3
mflo $4
mthi $4
mtlo $3
```

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。

我是手动构造的样例，通过覆盖率分析工具，我发现自己的测试数据其实还是有很多没有考虑到的情况（万幸CPU对这些情况的处理没有出错）。如果要保证覆盖所有需要测试的情况，还得根据分析工具对测试数据进行进一步的加强，充分考虑转发的来源和接受源。我最后还未测试到的点有乘除法相关指令与其他指令之间的转发问题，以及load相关。前者能通过构造较好的解决，但是与 load 指令相关的构造有一定的难度，这也是我考虑并不周全的原因之一。