Cache 设计说明

一、设计考虑点

1、数据与指令访问并行处理：

此 Cache 使用在流水线 CPU 设计中进行数据以及指令的存取以及修改。

流水线大体分取指、译码、执行、访存、写回五个阶段，对于指令的访问在第一个阶段，即取指阶段，对于数据的访问在第四个阶段，即访存阶段。流水线的一大特征是不同的阶段是并行进行的，取指的同时也会进行数据的访问。

因此需要对于一个周期中同时到达的数据访问请求以及指令访问请求进行处理，并在一定时间内将数据以及指令请求处理完毕，并返回 response 信号，以指示 CPU 数据或者指令访问请求处理完毕，CPU 可以继续执行。

2、I$miss 且 D$在处理请求的情况

此时 I$下一步，应该读 D$以及 RAM，以查找所需的指令。但是 D$是在占用状态，D$正在处理来自 Core 的请求，因此应当先处理完 D$的的任务，Response 之后，再处理 I$miss 的情况，之前 I$需要等待 D$直到 D$处于空闲状态。

3、访问处理速率

由于我们设计的是一级 Cache，因此其命中率较高，大概有 80%左右的命中率，剩余 20%情况均摊到 miss 或者 miss 且 dirty 的情况。而我们使用的是 core 等待 cache 响应的机制，因此要提高 cpu 的速率，应考虑尽量缩短 core 对于 cache 进行访问的时间。

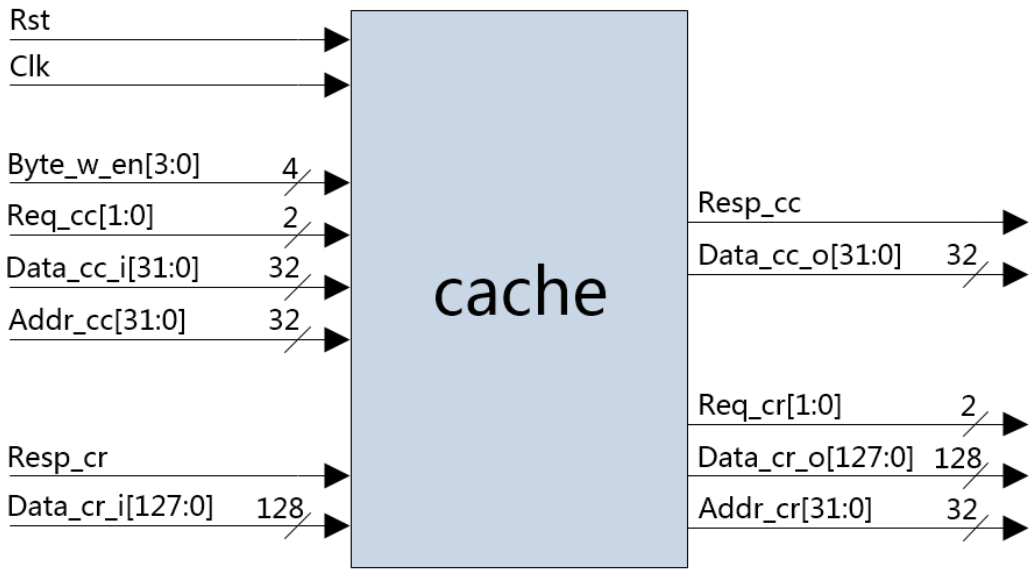
首要先提高命中时的时间花销，此处我们设计使用了状态机转换，因此命中固定的花销为 3 个时钟，要想进一步提高，则需要用一些方法，比如 clk 高电平做一些事，低电平做另一些事，应当可以提高一定的速率。而目前进使用状态机机制应当无法进一步提高。

其他情况由于只占到 20%左右，由于只是对 cache 的初步实现，此处可以几乎忽略掉。

二、 顶层 Cache 设计

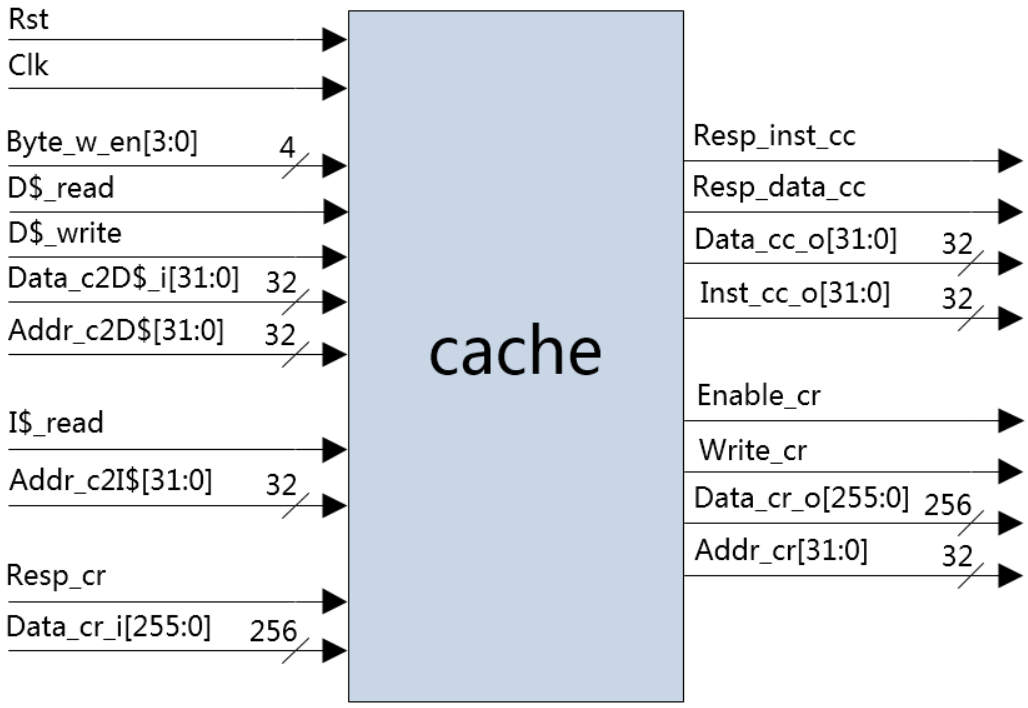
1、设计问题点

第一次设计没有考虑到指令和数据的并行访问请求，同时由于 Cache 大小会决定功耗，对于电路进行了一次精简，产生了一个问题，即一次只能处理一个请求，数据以及指令请求到达只能每次处理一个。需要再在外侧封装一层控制器，两个请求命中时间，若是指令请求为 n1，数据情况为 n2，那么原本需要 max(n1, n2)的变成了 n1+n2，导致 CPU 的等待时间会较长，且无法发挥将 Cache 分为 I$和 D$的特点。产生线路如下：



考虑指令数据请求并行处理，添加一部分使 Data 与 Inst 请求并行的线路，同时 D$中相对 Write 添加 Read 信号，而不是精简到只有 Write 信号是为了使其本身有使能的作用（1 时有效，0 无效），若单独添加使能信号也可以，但是为了与 Inst 请求对称，且不多添加线路便如此设计，最终得到的框图如下。

2、实际电路框图：



3、框图端口说明：

序号	接口名	宽度(bit)	输入/输出	另一端接口部件	作　用
1	Rst	1	输入		复位信号
2	Clk	1	输入		时钟
3	Byte_w_en	4	输入	Core	由于 MIPS 指令中有对单字节读写功能，添加对数据字节写使能信号
4	D$_read	1	输入	Core	为 1 则 Data Cache 读，为 0 无效
5	D$_write	1	输入	Core	为 1 则 Data Cache 写，为 0 无效
6	Data_c2D$_i	32	输入	Core	由 Core 写到 Data Cache 的数据(Data_core_to_data_cache)
7	Addr_c2D$	32	输入	Core	Core 读或写 Data Cache 数据的地址
8	I$_read	1	输入	Core	为 1 则 Instruction Cache 读，为 0 无效
9	Addr_c2I$	32	输入	Core	Core 读 Instruction Cache 的指令的地址
10	Resp_cr	1	输入	RAM	RAM 读写请求 Response，表示请求处理完毕
11	Data_cr_i	256	输入	RAM	从 RAM 读到 Cache 的数据
12	Resp_inst_cc	1	输出	Core	Core 对 Cache 的指令读请求处理完毕信号，为 1 则完毕
13	Resp_data_cc	1	输出	Core	Core 对 Cache 的数据读写请求处理完毕信号，为 1 则完毕
14	Data_cc_o	32	输出	Core	Core 读 Cache 数据请求该地址读出的数据
15	Inst_cc_o	32	输出	Core	Core 读 Cache 指令请求该地址读出的指令
16	Enable_cr	1	输出	RAM	RAM 的使能信号
17	Write_cr	1	输出	RAM	1 则 Cache 向 RAM 写，0 则 Cache 从 RAM 读
18	Data_cr_o	256	输出	RAM	Cache 向 RAM 写回的数据
19	Addr_cr	32	输出	RAM	Cache 向 RAM 写回数据的地址

三、 Cache 内部设计

1、设计问题点

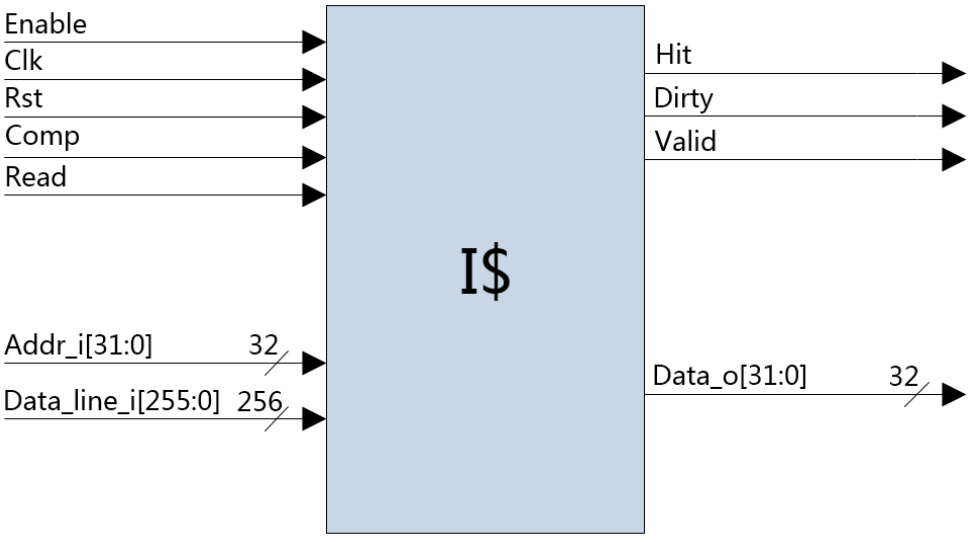
若出现 I$ miss 情况需要向 D$发送读请求，此时应当考虑边界情况，即 D$在读或写时，同时出现了 I$ miss 的状况。则此时交由控制器处理，先对于 D$的读写状况（无论是 miss 抑或存在 dirty）进行处理，处理完之后（此处的处理完是指 D$的 response 信号传到了 Core，并使用完的下一个周期，否则 D$的 hit 信号与 address 信号会一直被 D$的读写请求占用，无法处理 I$读 D$的情况）。控制器转为对于 i$ miss 的情况进行处理即可。

2、I$的设计

⑴、考虑因素：

指令 cache 来自 core 的访问只有读请求，而当 miss 的时候会从数据 cache 或者 RAM 读取一行数据，并进行写入。因此输入数据需要一个选择器用以选择是接收来自数据 cache 的数据还是来自 RAM 的数据。

⑵、电路框图



⑶、端口说明

序号	接口名	宽度(bit)	输入/输出	作　用
1	Enable	1	输入	使能信号，用于指示其余输入信号是否有效
2	Clk	1	输入	时钟
3	Rst	1	输入	复位信号
4	Comp	1	输入	为 1 则比较每行中的 tag 与输入地址相应字段，否则不比较
5	Read	1	输入	为 1 则读 I$，I$输出指令，为 0 则向 I$写入指令
6	Addr_i	32	输入	Core 要读 I$的指令的地址
7	Data_line_i	256	输入	D$或者 RAM 读来的指令串，需要写入 I$
8	Hit	1	输出	为 1 则 cache 命中，输出正确的指令
9	Dirty	1	输出	为 1 则该行是修改过，为 0 未修改，I$该输出无意义
10	Valid	1	输出	为 1 则该行有效，为 0 无效，目前没有用，用于之后拓展
11	Data_o	32	输出	从 I$读出的指令

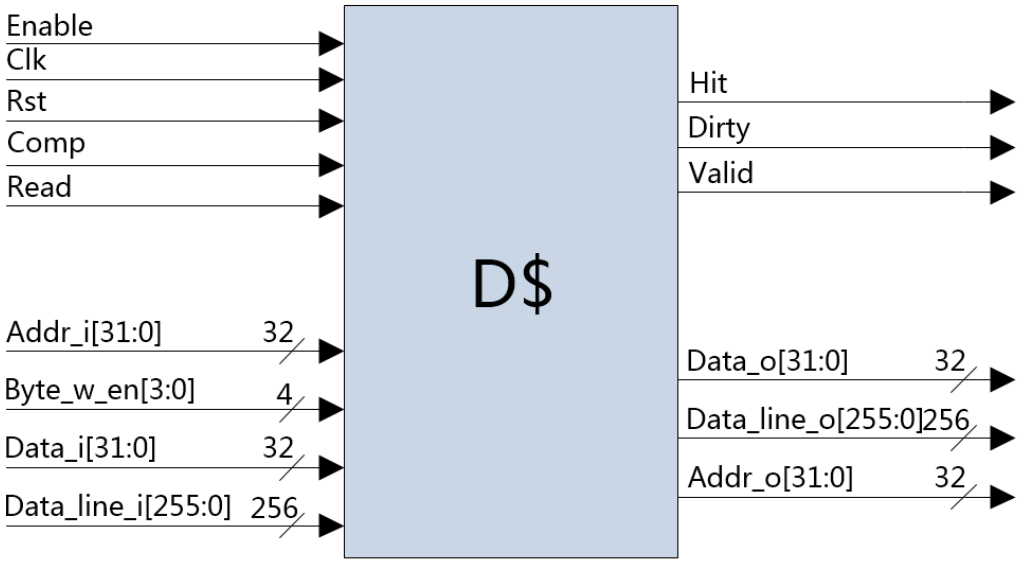
3、 D$的设计

⑴、考虑因素

对于 D$的访问可能来自 I$也可能来自 Core，不过此问题无需 D$进行管理，交给控制器管理即可。

由于 MIPS 指令集中存在对于一个字节的修改以及加载的指令，Cache 需要对于此功能进行支持，因此需要一个 byte_w_en，字节的写使能信号进行对字节的操作，而此信号由 cpu 发出。

(2)、电路框图



(2)、端口说明

序号	接口名	宽度(bit)	输入/输出	作　用
1	Enable	1	输入	使能信号，用于指示其余输入信号是否有效
2	Clk	1	输入	时钟
3	Rst	1	输入	复位信号
4	Comp	1	输入	为 1 则比较每行中的 tag 与输入地址相应字段，否则不比较
5	Read	1	输入	为 1 则读 D$，D$输出数据，为 0 则向 D$写入数据
6	Addr_i	32	输入	Core 要访问 D$的数据的地址
7	Byte_w_en	4	输入	字节写使能信号，用以支持字节的访问需求
8	Data_in	32	输入	需要写入的数据
9	Data_line_i	256	输入	RAM 读来的数据块，需要写入 D$
10	Hit	1	输出	为 1 则 cache 命中，输出正确的数据
11	Dirty	1	输出	为 1 则该行是修改过，为 0 未修改
12	Valid	1	输出	为 1 则该行有效，为 0 无效，目前没有用，用于之后拓展
13	Data_o	32	输出	从 D$读出的数据，传到 Core
14	Data_line_o	256	输出	从 D$读出的数据行，传到 I$或者 RAM，用以写入，32 位数据与行数据同时输出
15	Addr_o	32	输出	要写回 RAM 的数据行地址

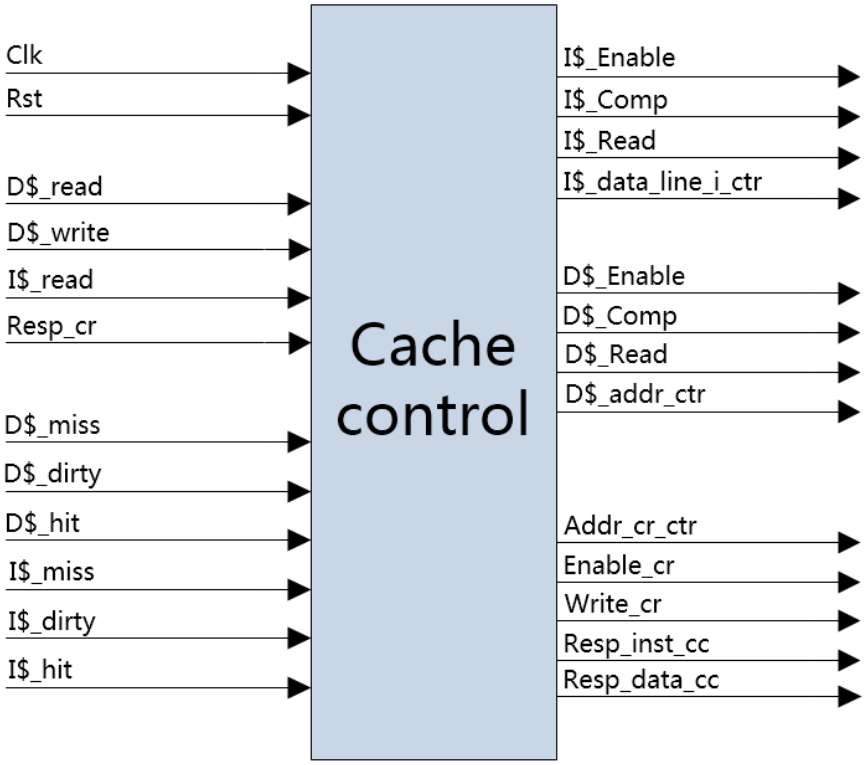
4、 Cache 控制器的设计

⑴、考虑因素

用以控制 I$与 D$的同步，且与 Core 和 RAM 的交互，实现正确的数据流动。

因此除了普通的对于两个 cache 的控制信号外，需要管理 I$的输入由 RAM 来的指令或者 D$来的指令，D$需要处理的是 I$的请求地址还是 Core 的请求地址，向 RAM 发送读写请求的地址是 I$还是 D$的地址，以及一些响应信号。

⑵、电路框图



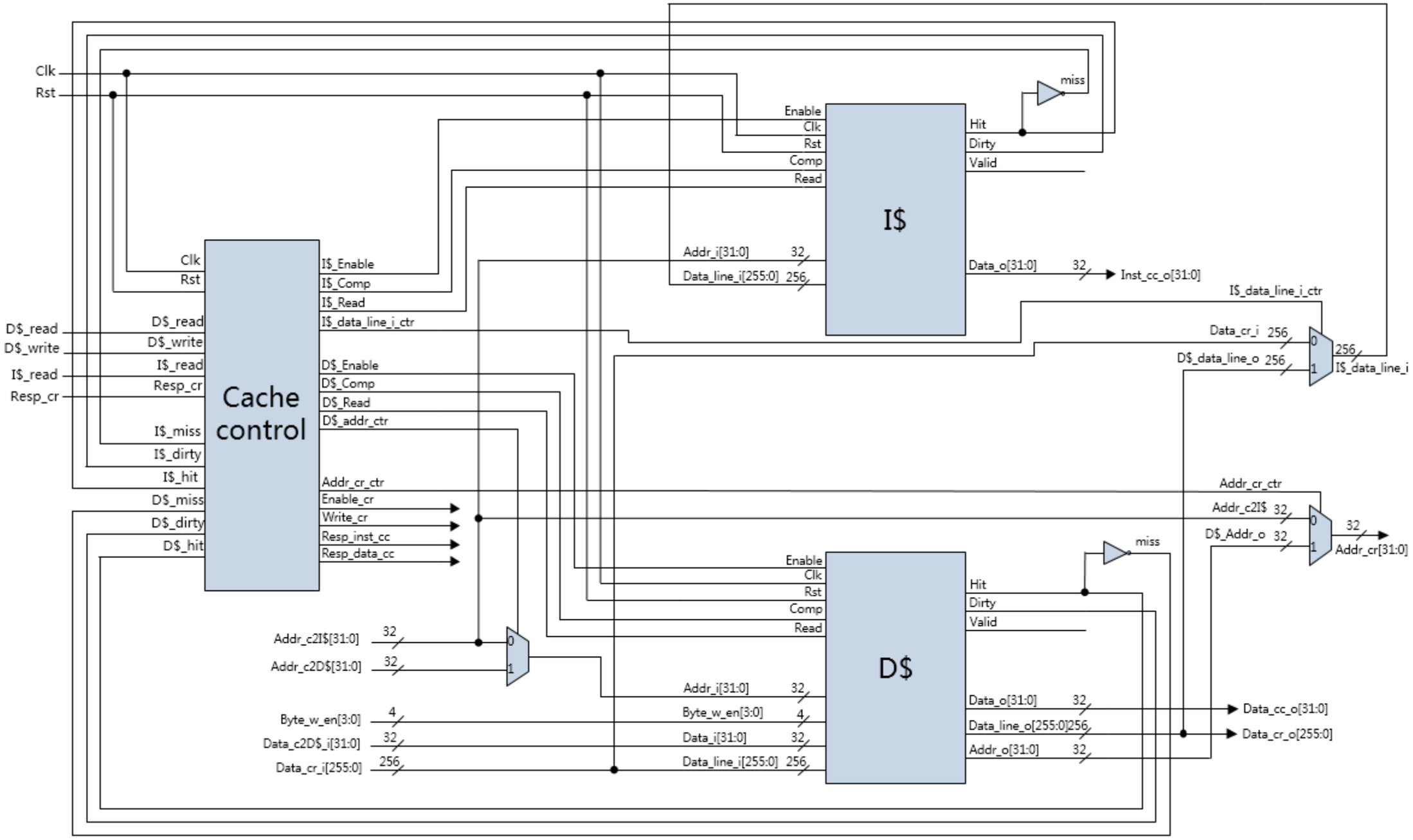
(3)、部分端口说明

由于信号在电路图功能中较为明显，此处仅说明一些较为特殊的信号。

序号	接口名	宽度(bit)	输入/输出	作　用
1	Resp_cr	1	输入	来自 RAM 的响应信号，Response_cache_ram
2	I$_data_line_i_ctr	1	输出	选择写入 I$的指令来自 D$还是 RAM
3	D$_addr_ctr	1	输出	选择 D$的访问数据地址是 I$的还是 D$的
4	Addr_cr_ctr	1	输出	选择输出到 RAM 的地址是来自 I$还是 D$
5	Resp_inst_cc	1	输出	响应 Core 的指令请求信号，访问 I$
6	Resp_data_cc	1	输出	响应 Core 的数据请求信号，访问 D$

5、Cache 内部连线

根据以上的分析以及两个 Cache 以及控制器的设计进行电路图的连接，结果如下：



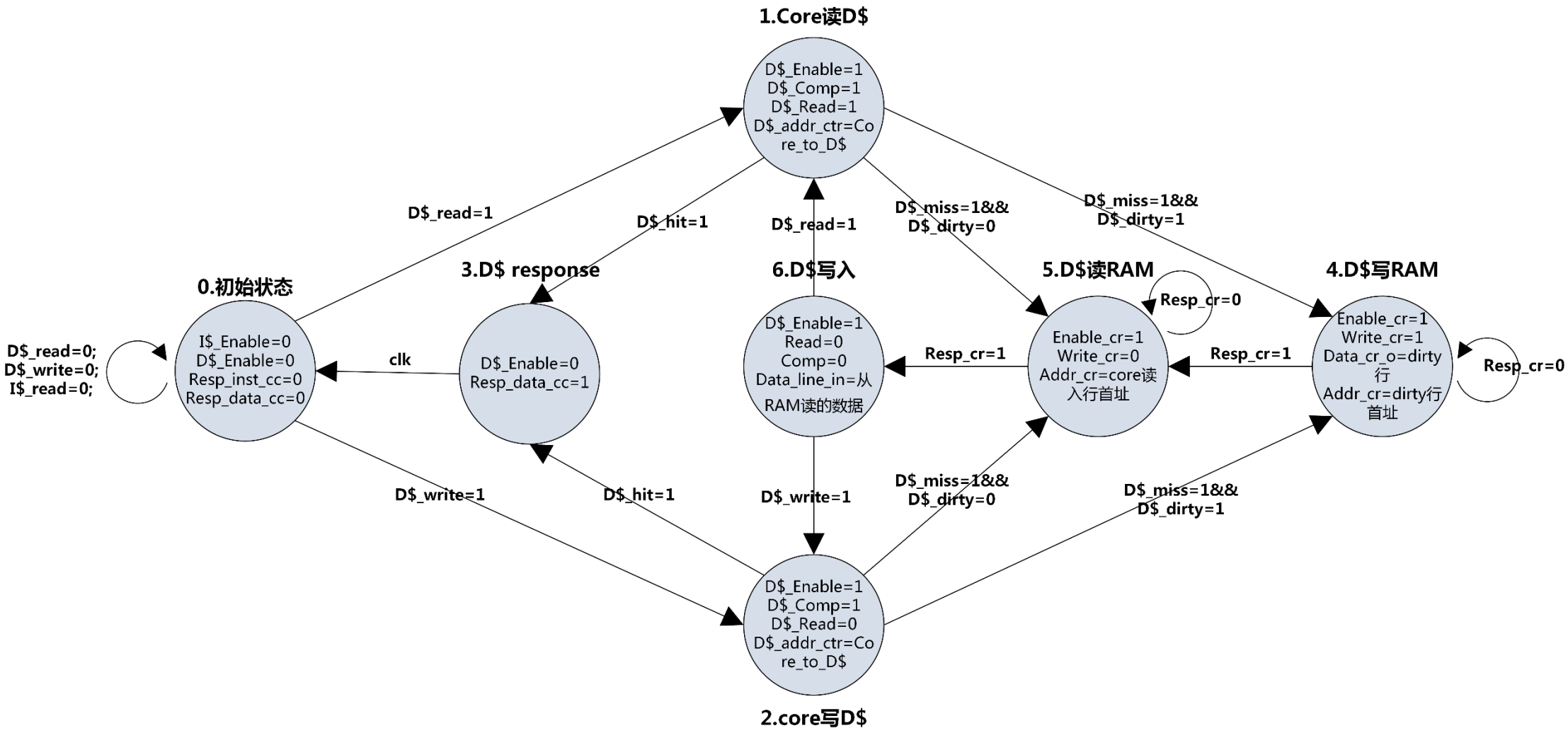
四、状态图转换设计

I$与 D$需要进行并行处理，若仅适用一个状态转换图，那么则为串行的结构，无法达到预期的并行处理的设计。因此，此处将原本设计出的串行处理的状态图一分为二，一部分为 D$的处理，另一部分为 I$的处理，同时由于需要进行 I$对于 D$的访问实现，需要在 I$的处理中对一些特定的条件进行判断（D$处理完毕才能进行来自 I$访问的处理）。

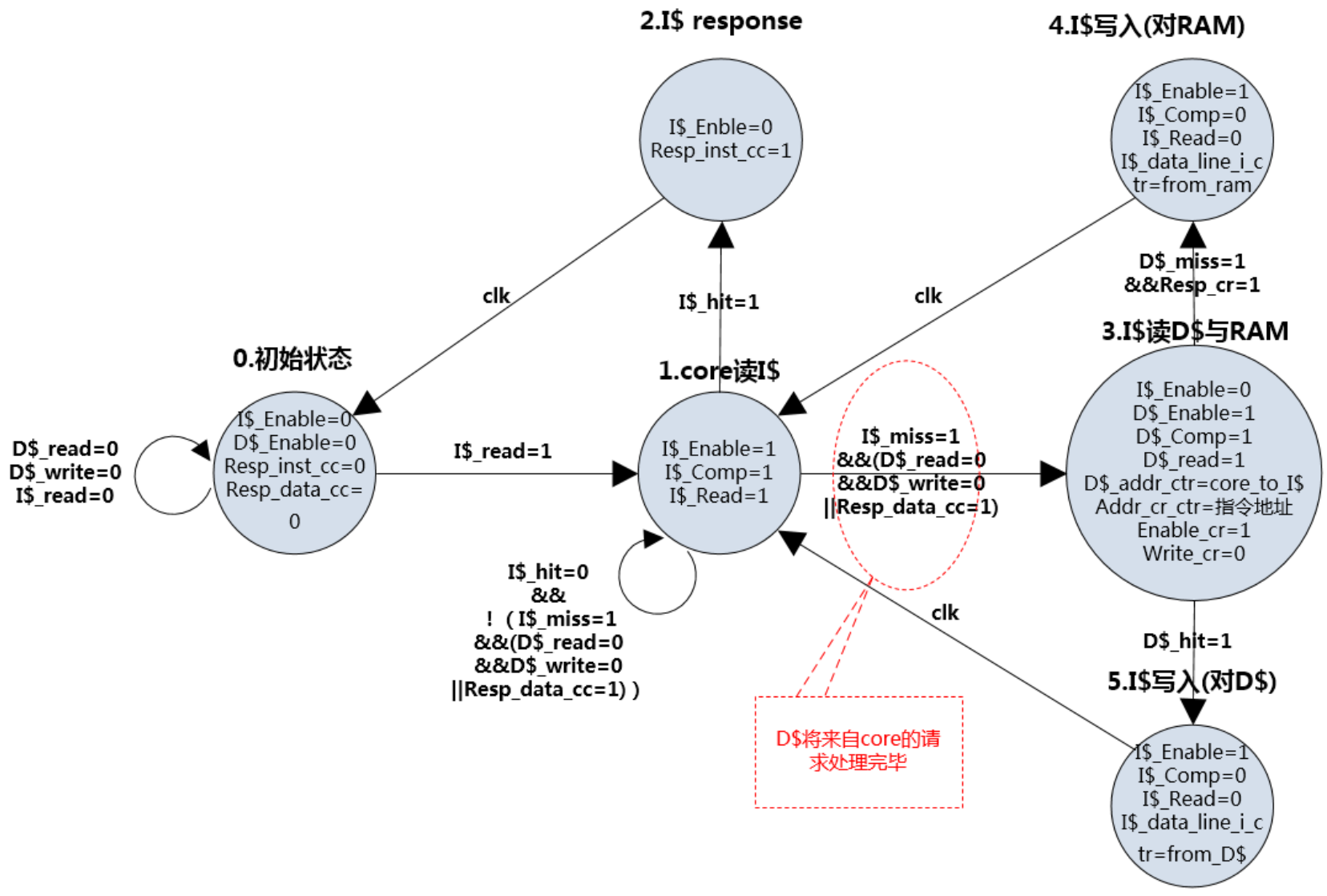
而实际只有一个控制器，因此需要在该控制器中实现两个状态图的并行转换。

实现的转换图如下：

1、D$的状态转换图



2、I$的状态转换图



五、信号值输出表

由于使用了两个状态转换图，因此，输出的控制信号会有重叠，但是不会产生冲突，因此，控制器需要对两个转换图输出的信号值进行一个“与”操作，此处我们约定将无所谓的信号值全赋值为 0，有效的信号为 1，无效的信号为 0。则这样可以得到最终的信号输出。

我们将两个状态转换图的每个状态所产生的信号值在此处一一列举，表格如下：

D$信号表：

状态\信号值	I$_Enable	I$_Comp	I$_Read	I$_data_line_i_ctr	D$_Enable	D$_Comp	D$_Read	D$_addr_ctr	Addr_cr_ctr	Enable_cr	Write_cr	Resp_inst_cc	Resp_data_cc
0.初始状态	0	0	0	0	0	0	0	0	0	0	0	0	0
1.core 读 D$	0	0	0	0	1	1	1	1	0	0	0	0	0
2.core 写 D$	0	0	0	0	1	1	0	1	0	0	0	0	0
3.D$response	0	0	0	0	0	0	0	0	0	0	0	0	1
4.D$写 RAM	0	0	0	0	0	0	0	0	1	1	1	0	0
5.D$读 RAM	0	0	0	0	1	0	0	1	1	1	0	0	0
6.D$写入	0	0	0	0	1	0	0	1	0	0	0	0	0

I$信号表：

状态\信号值	I$_Enable	I$_Comp	I$_Read	I$_data_line_i_ctr	D$_Enable	D$_Comp	D$_Read	D$_addr_ctr	Addr_cr_ctr	Enable_cr	Write_cr	Resp_inst_cc	Resp_data_cc
0.初始状态	0	0	0	0	0	0	0	0	0	0	0	0	0
1.core 读 I$	1	1	1	0	0	0	0	0	0	0	0	0	0
2.I$ response	0	0	0	0	0	0	0	0	0	0	0	1	0
3.I$读 D$与 RAM	0	0	0	0	1	1	1	0	0	1	0	0	0
4.I$写入(对 RAM)	1	0	0	0	0	0	0	0	0	0	0	0	0
5.I$写入(对 D$)	1	0	0	1	0	0	0	0	0	0	0	0	0

注：D$信号表中从 D$写 RAM 到 D$读 RAM，需要将 D$_Addr_o 由 dirty 行首址转换为读入行首址，转换根据 D$的信号不同之处来进行相应的操作。