实验六 综合设计

目录

- 实验六 综合设计
  - 目录
  - 附录文件一览
  - 补全指令
  - 动态预测
    - 饱和计数器
    - 基于全局历史的分支预测
    - 跳转指令cache
    - 具体逻辑设计
    - 测试数据
  - 数据Cache
    - 模块接口
    - 变量的声明
    - hit 与 寻道
      - hit
      - 寻道
    - FIFO
    - 读取
    - cache操作
    - 主存

附录文件一览

```
.
├── ALU.v      #ALU
├── ImmGen.v   #生成Imm
├── InsCache.v  #跳转指令cache，BTB
├── MEM_CACHE.v #内存cache
├── Nexys4DDR.xdc
├── cpu.v
├── cpuDownload.v
├── decoder.v   #16位译码器
├── encoder.v   #4位编码器
├── pdu-v1.1.v
├── register_32_32.v
└── saturatingCounter.v
```

补全指令

最后实现的所有可执行指令如图所示

实现均由原本的实现稍作修改得到，比如 `lui` 由 `auipc` 得到，将 `auipc` 中的[+PC]部分改为[+0]即得

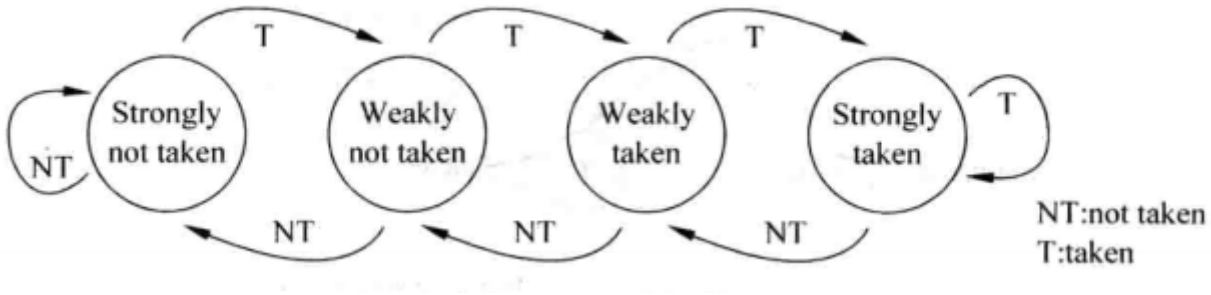
其他指令扩展类似不表

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	101	rd	0010011	I srli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

动态预测

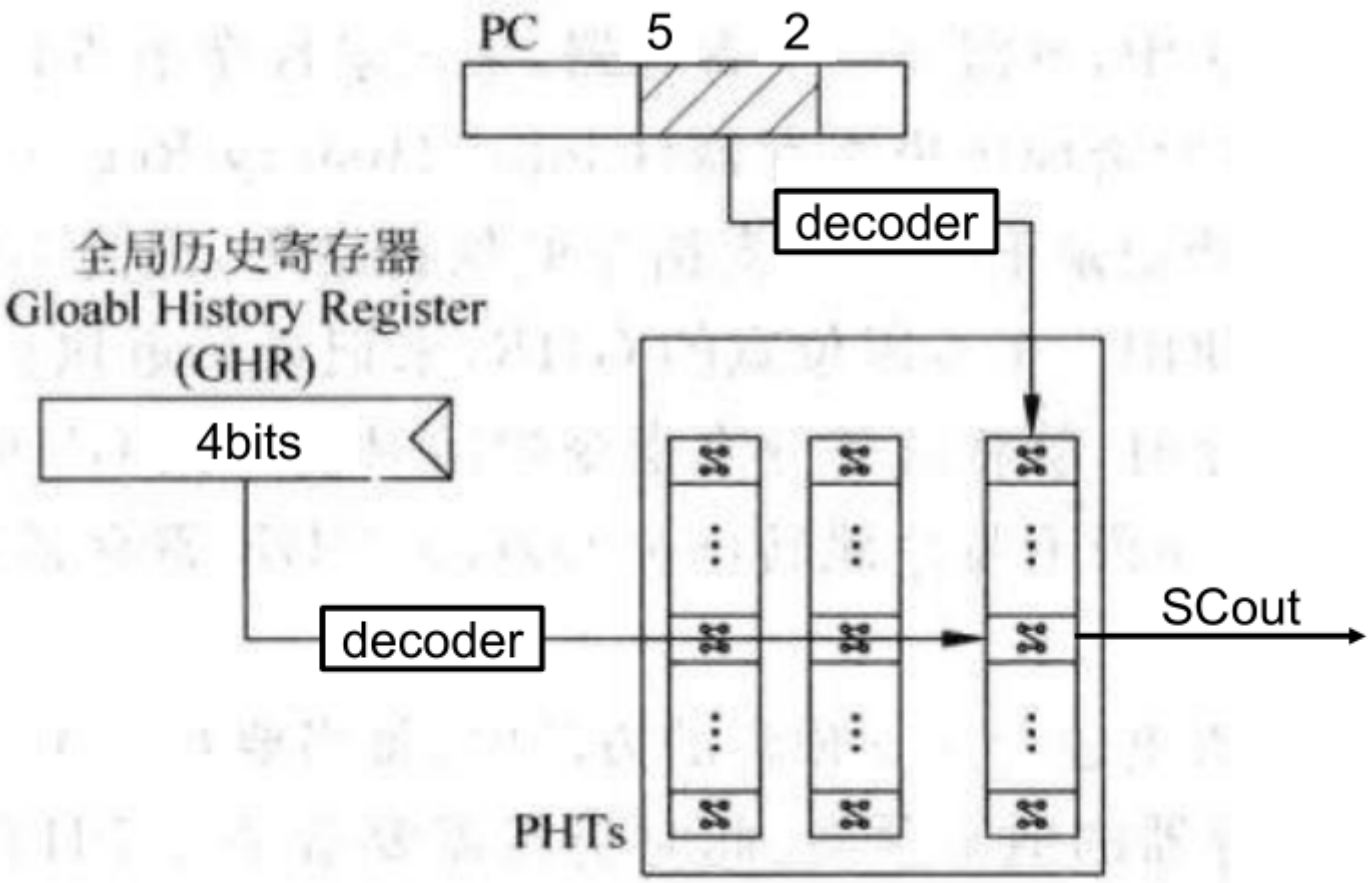
饱和计数器

有四个状态，编码为 `00-11` ，则是否跳转取决于 `bit[1]` ，代码如下




```
reg [1:0] counter;
always @(posedge clk or negedge rstn) begin
    if(~rstn) counter<=1;//弱不跳转
    else
    if(we)
        if(inIfBranch)
        begin
            if(counter==2'b11) counter<=counter;
            else counter<=counter+1;
        end
        else
        begin
            if(counter==2'b00) counter<=counter;
            else counter<=counter-1;
        end
end
assign outIfBranch = counter [1];
```

基于全局历史的分支预测



不同地址的 Branch 指令需要不同的饱和计数器，而不可能将所有的饱和计数器按照地址例化出来，因此我们按照直接映射的方式，将 PC[5:2] 作为key映射一个饱和寄存器，得到一个具有16个饱和寄存器的PHT表

并且通过增加一个全局历史寄存器GHR，记录曾经跳转的历史，增加了对更为复杂情况的预判，而GHR为4bits，对应16个PHT表

综上共花费64B的寄存器资源

每次 branch 发生时，都会读取该表，并且在结果判明后更新该表

```
//4bits 全局历史寄存器，对应16个PHT
reg[3:0] GHR;

always @(posedge clk or negedge rstn) begin
  if(~rstn) GHR=4'b0101;
  else if(Branch_Ex)
        GHR={GHR[2:0],zero};
end
```

```
//编码器，用于定位饱和计数器 SC
wire [15:0]x;
decoder_4t16 decoder_x(GHR,x);

//一个 PHT 由 PCD_r[5:2] 确定，即一个 PHT 有16个 saturatingCounter 即饱和计数器

wire [15:0]y_ex;
decoder_4t16 decoder_y_ex(PCE_r[5:2],y_ex);

// saturatingCounter 的输出
wire [15:0]SCout[15:0];

genvar i;
genvar j;
generate
    for(i=0; i<16; i=i+1)
      for(j=0;j<16;j=j+1)
      begin
        wire out;
        saturatingCounter SC(clk,rstn,SCwe & (x[i]) & (y_ex[i]),
                             zero,SCout[i][j]);
      end
endgenerate

wire branchTrue = SCout[GHR][PCD_r[5:2]];
```

跳转指令cache

因为在流水线中预测跳转后指令也不能立即取出，故需要一个cache（或BranchTargetBuffer、BTB）来保存预测跳转之后的指令

因此设计一个全相连、FIFO、不具有自我取指的半cache，其更新cache需要外界设计支持，即当输入 `memWE` 有效时，判断cache内是否已存在该tag，若无，将 `memAdd` 和 `memData` 写入cache； `memWE` ， `memAdd` 和 `memData` 需要外界提供

其中单元设置为： `valid` 一位， `tag` 三十位， `Data` 三十二位

其中 `tag` 存跳转之后的地址，因为地址模四，故只需要取三十位作为 `tag` 即可

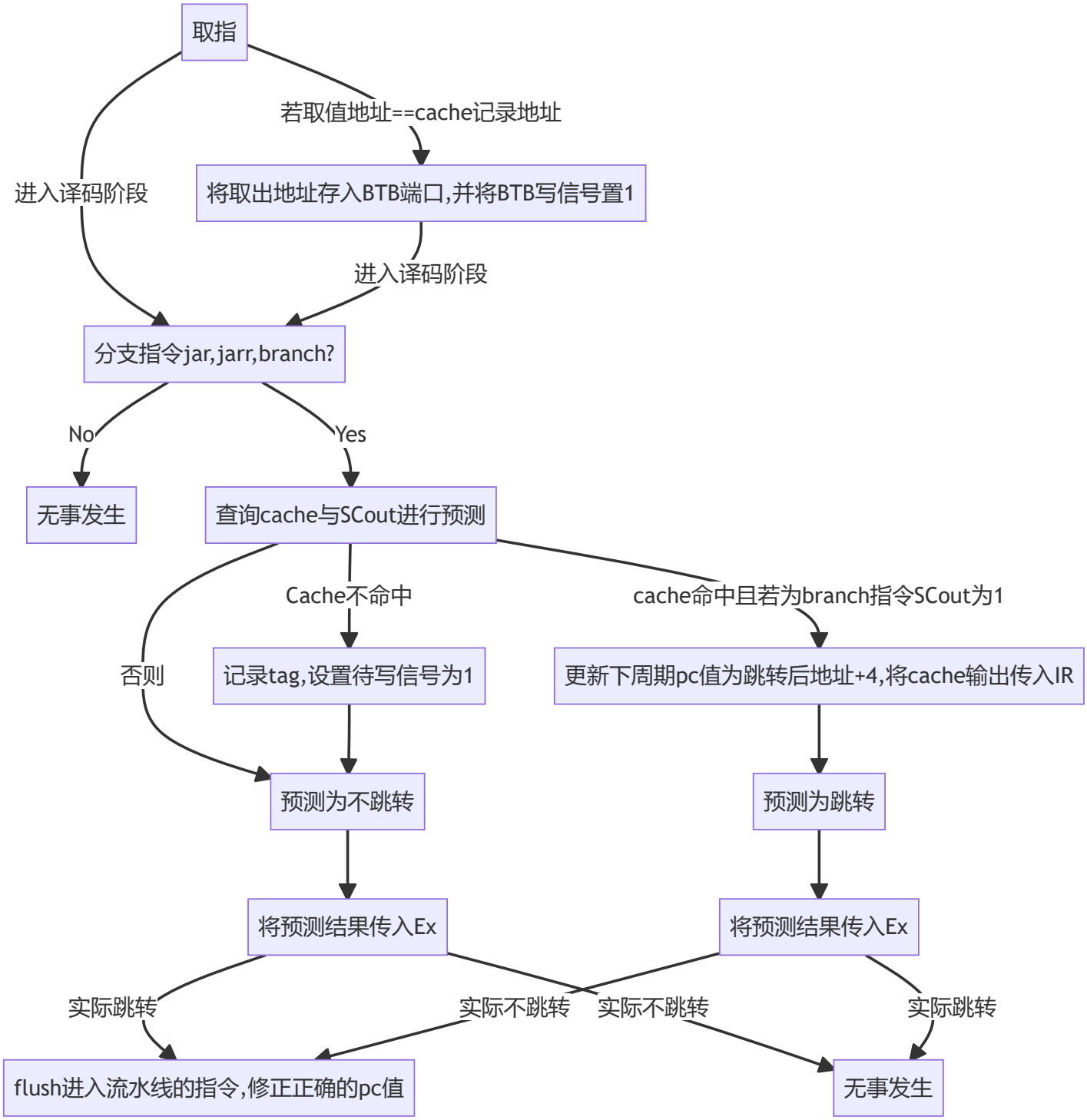
```
module InsCache (
  input clk,
  input [31:0] address,     //cpu给的地址
  output [31:0] data,       //从cache中取出的数据
  output hit,                //是否hit

  input memWE,               //数据准备好的信号
  input [31:0] memAdd,       //写入数据对应的地址
  input [31:0] memData       //从主存输入用来更新Cache的Data
);
```

```
//全相连，FIFO，共有8个缓存地址

parameter WIDTH=8;
parameter WIDTH_CNT = 3;
reg [WIDTH-1:0] valid=0;
reg [29:0] tag[WIDTH-1:0];//用[31:2]来确定tag
reg [31:0] cacheData[WIDTH-1:0];//对应的地址

reg[WIDTH_CNT-1:0] cnt=0;//用于FIFO的计数器


//FIFO计数器与数据更新
wire w;
assign w=memWE &~ (|(eqW & valid));//是否更新=是否写入&&写入数据不在cache内
always @(posedge clk) begin
    if(w)
    if(cnt==WIDTH-1) cnt=0;
    else cnt=cnt+1;
end
always @(posedge clk) begin
    if(w)
    begin
        valid[cnt]<=1;
        tag[cnt]<=memAdd[31:2];
        cacheData[cnt]<=memData;
    end
end



reg[WIDTH-1:0] eq,eqW;
integer i;
always @(*) begin
    for(i=0; i<WIDTH; i=i+1)
    begin
    eq[i]=(address[31:2]==tag[i]);//是否相等
    eqW[i]=(memAdd[31:2]==tag[i]);
    end
end

wire [WIDTH-1:0] sig;
assign sig = eq & valid;
assign hit = |sig;
wire [WIDTH_CNT-1:0] sel;
encoder_16bits encoder_16bits({0,sig},sel);
assign data=cacheData[sel];
```

当指令为 branch 和 jar 指令时，cpu会尝试获取cache，而对于 jarr 因为其跳转地址依赖于寄存器故不支持跳转cache

具体逻辑设计

由下图所示


取指
若取值地址==cache记录地址
进入译码阶段
将取出地址存入BTB端口,并将BTB写信号置1
进入译码阶段
分支指令jar,jarr,branch?
No
Yes
无事发生
查询cache与SCout进行预测
Cache不命中
cache命中且若为branch指令SCout为1
否则
记录tag,设置待写信号为1
更新下周期pc值为跳转后地址+4,将cache输出传入IR
预测为不跳转
预测为跳转
将预测结果传入Ex
将预测结果传入Ex
实际跳转
实际不跳转
实际不跳转
实际跳转
flush进入流水线的指令,修正正确的pc值
无事发生

测试数据

以256个数据排序为例，随机生成多组数据，测试命中率

预测次数	预测成功次数	命中率
0x60be	0x52dd	$\frac{21,213}{24,766}=0.8565$
0x60be	0x528f	$\frac{21,135}{24,766}=0.8534$
0x60be	0x5284	$\frac{21,124}{24,766}=0.8529$
0x60be	0x5086	$\frac{20,614}{24,766}=0.8324$
0x60be	0x5485	$\frac{21,647}{24,766}=0.8741$
平均命中率		0.8543

而考察最内层的循环，有

```
LOOP2:
blt t2,t1,LOOP2FIN  #跳转概率基本等于1
lw t4,4(t1)
bge t3,t4,skip      #跳转概率约0.5
mv t5,t3
mv t3,t4
mv t4,t5
skip:
sw t4,0(t1)
addi t1,t1,4
j LOOP2             #跳转概率等于1
LOOP2FIN:
```

其理论命中率为$66.6\%+33.3\%*0.5=83.25\%$

设计的cpu在该程序中较好地实现了预测，基本达到及超过理论上的命中率

数据Cache

cache类型：基于全写法和不按写分配法以及FIFO换出策略的组相联cache

模块接口

将ip核例化得到的主存与cache一起封装成一个模块

在ip核端口基础上加入写请求 读请求 复位信号

```
module MEM_CACHE (
  input [7:0] Address,    //cpu给的地址
  input [31:0] WriteData,       //写的数据
  input [7:0] DebugAddr,     //chk用 仅可读
  input clk,
  input wr_req,//写请求信号
  input rd_req,
  input rstn,
  output wire [31:0] ReadData,      //从cache中取出的数据
  output wire [31:0] DebugData,
  //debug
  output wire hit_m
);
```

变量的声明

```
localparam way_cnt = 2;//组相连度
//cache与主存之间使用全写法和不按写分配法
//使用块大小为一字（一字四字节）两组 共计8块的直接映射cache  总容量8*2*1=16字
//FIFO策略
//ADDR: 2:0为index 7:3为tag

reg [31:0] cache[7:0][way_cnt - 1:0];
reg valid[7:0][way_cnt - 1:0];//标记有效位
reg [4:0] tag[7:0][way_cnt - 1:0];//标签位


reg FIFO[7:0][way_cnt:0];// 实际上就是一个计数器 升满了表示要润了
reg [way_cnt-1:0] outway;//记录哪个通道的数据要被换出 似乎用不到这么多位宽233

wire [31:0] ip_readdata;
wire [2:0] index;
wire [4:0] tag_in;
// reg [way_cnt-1 : 0] way_addr;//记录数据来自哪个通道
wire [way_cnt-1 : 0] way_addr;//记录数据来自哪个通道
assign index = Address[2:0];
assign tag_in = Address[7:3];
```

hit 与 寻道

hit

```
wire hit;
assign hit = (valid[index][0] && tag[index][0] == tag_in) | (valid[index][1] && tag[index][1] =
```

由于cpu需要在上升沿到来之前就准备好从主存中读取的数据，也就是说读取必须在上升沿之前完成

由于always@*出现的时序问题 hit使用wire类型 使用assign赋值 这样可以保证hit永远是最新的值

拓展时只需要复制多几次即可

寻道

由于采用组相联 数据在一个index内是任意存储的 所以读取前需要确定数据在哪个通道

```
assign way_addr = (valid[index][0] && tag[index][0] == tag_in) ? 0 : 1;
```

拓展时同理 多复制几次即可

FIFO

思路如下：

1. 首先查找有无未使用的块 若有 跳转3 否则跳转2
2. FIFO使用数值来标记进入顺序 从小到大以此为新进入和旧进入 查找FIFO=way_cnt 即可找到队列头需换出的元素
3. 初始化该位置FIFO 修改其他元素FIFO的值

```
integer i,j;

integer free = 0;//标识是否还有空闲块
always @(posedge clk or negedge rstn)begin
  if(~rstn)begin
    for(i = 0;i < 8;i = i + 1)begin
      for(j = 0;j < way_cnt; j = j + 1)begin
        FIFO[i][j] = 0;
      end
    end
  end
  else begin
    if(~hit && (wr_req | rd_req))begin
      //选出冲突时换出的块
      for(i = 0;i < way_cnt;i = i + 1)begin
        if(FIFO[index][i] == 0)begin//FIFO从1开始升序到way_cnt也就是组相联度
        //0表示还没开始 其实相当于valid位
          outway = i;
          free = 1;
        end
      end
      if(free == 0)begin
        for(i = 0;i < way_cnt;i = i +1)begin
          if(FIFO[index][i] == way_cnt)begin//最大值说明是最早进去的
          //最好是能break 但是应该也不会有多个通道都是最大值
            outway = i;
            FIFO[index][i] =0;
          end
        end
      end
      if(FIFO[index][outway] == 0)begin
        for(i = 0;i < way_cnt;i = i + 1)begin
          if(FIFO[index][i]!=0)begin
            FIFO[index][i] = FIFO[index][i] + 1;
          end
        end
      end
      FIFO[index][outway] = 1;//新数据会被换入到这里 所以可以先赋值为1
    end
  end
end
```

读取

使用assign读取以达到在上升沿到来之前读取成功的目的

```
assign ReadData = (hit == 1) ? cache[index][way_addr] : ip_readdata;
```

cache操作

分未hit 非hit 乘 读取 写入四种情况

非hit写入直接写主存IP核 hit读取由上述assign处理 只需要处理另外两种情况

```
always@(posedge clk or negedge rstn)begin
  if(!rstn)begin
    for(i = 0;i < 8;i = i + 1)begin
      for(j = 0;j < way_cnt; j = j + 1)begin
        valid[i][j] = 1'b0;
        cache[i][j] = 0;
        tag[i][j] = 0;
      end
    end
  end
  else begin
    if(hit)begin
      if(wr_req)begin//写请求
        cache[index][way_addr] <= WriteData;
        //ip核的话直接写就是了 不需要在这里操作
      end
      //读取完全用assign
    end
    else begin
      if(rd_req)begin
        //读信号有效时才换入
        //不是写的话就是默认的读取模式 读的话需要从内存中载入到缓存 当然为了速度 先直接返回内存读的值
        cache[index][outway] <= ip_readdata;
        valid[index][outway] <= 1'b1;
        tag[index][outway] <= tag_in;
      end
      //写不命中直接写主存就是了 似乎不需要处理
    end
  end
end
```

主存

直接使用ip核 由于是全写法 debug信号直接访问主存也是最新的值

```
DataMem DataMem(
  .a(Address),
  .d(WriteData),
  .dpra(DebugAddr),
  .clk(clk),
  .we(wr_req),
  .spo(ip_readdata),
  .dpo(DebugData)//由于使用的全写法和不按写分配法 所以内存中同样也是最新的值 直接读就可以了
);
```