Pipeline MIPS Processor with Cache

实验报告

**16307130194 陈中钰**

16级 计算机科学技术学院

**Contents**

[1 总体状况 2](#_Toc517994169)

[2 显示实现 3](#_Toc517994170)

[3 Cache设计 4](#_Toc517994171)

[4 Cache对处理器的影响 7](#_Toc517994172)

[5 仿真测试 7](#_Toc517994173)

[6 性能分析 8](#_Toc517994174)

[7 实验感想 10](#_Toc517994175)

# 总体状况

* 1. Cache

近30年来，由于计算机处理器运行速度的增长迅速，处理器的运行速度比存储器的运行速度快1~2个数量级。为了抵消这种趋势，于是就有了cache：

* 比memory更快，速度与处理器相近；
* 由于速度快，成本也很高，使得cache的大小一般都比较小；
* 储存最常用的指令和数据存储；
* 放在与处理器同一芯片的SRAM中；
* 实现处理器和主存之间的交互；

在有了cache之后，如果处理器需要的数据在cache中可用，那么就可以快速返回（hit），否则才需要从memory中获得数据（miss）。这样可以减少memory的访问，可以大大减少耗时，提高处理器整体运行性能。

一般来说cache的架构：

* 分为L1 cache和L2 cache，其中L1更小但更快，价格更高，L2大些但更慢，价格更低;
* 不仅data memory有cache，instr memory也是有cache的；
* 但在这里，只实现了data memory的一层cache，其他没有实现。
  1. **指令集（共29条指令，红色的是添加的19条指令）**
* 逻辑运算：addi, and, or, add, sub, andi, ori, xor, xori, nor
* 移位运算：sll, srl, sra, lui, nop
* 分支跳转：beq, bne, bgez, bgtz, blez, bltz
* 比较运算：slt, slti
* 内存读写：sw, lw
* 跳转： j, jal, jalr, jr

**注意：**

* **以上指令的实现按照MIPS指令集文档中的格式，故不再附上指令格式要求**
* **jal/jalr指令调用函数，jr函数返回后，紧跟jal/jalr的指令不会被执行**
  1. 规格
* register file: 32bit\*32
* data memory: 32bit\*512
* data memory cache: 4 set \* 4 way \* 4 \* 32bit
* instruction memory: 32bit\*512
  1. Cache设计
* 总体情况：cache的设计是基于原来的pipeline MIPS处理器进行的，设计了一层cache把原来的data memory包裹起来，而处理器的datapath、controller等硬件设计部分基本不变，还有流水线设计、开发板显示设计、冲突处理也基本不变，可以参考之前的流水线实验报告，故在这次报告里面不再叙述。此外，除了新添加的cache测试程序（在下文会有测试结果）以外，其他的测试程序运行结果也同样是正确的，也不再展示了；
* 架构：只设计了data memory的一层cache，主要代码在dcache2.v中，即dcache2模块中。而在top模块中原来是直接调用data memory模块的，改为调用dcache2模块，再在dcache2模块中调用data memory；



* 大小： 4组、4路、每一个block有4个32bit的数据，也就是4x4x4x32bit，而data memory是512x32bit；
* 速度：在hit的情况下可以在1个周期内输出数据，在miss的情况下，如果替换的block不脏，则需要8个周期，如果替换的block是脏的，则需要16个周期，其中8个周期写回脏数据，8个周期读数据；
* 替换策略：LRU；
* 对处理器的影响：当hit的时候，对处理器没有影响；当miss的时候，会使整个处理器暂停，等待data cache的数据的获得。

# 显示实现

（代码实现请看工程文件，在此不展示）

* 由于添加了data cache的模块，于是多了cache的内容、cache的相关控制逻辑、cache的相关信号需要显示出来，显示的方式和以前一致；
* 其他内容基本没有变化，可以参考流水线报告，故不再展示；
* SW[9:7]的模块对应关系：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| SW[9:7] | 0 | 1 | 10 | 11 | 100 | 101 |
| 模块 | 五个流水线阶段的相关数值 | regfile | instr memory | data memory | data cache | data cache模块中其他相关数值和信号值 |

* SW[6:0]选择各个模块中的数值

# Cache设计

（代码请看工程文件dcache2.v，在此不展示）

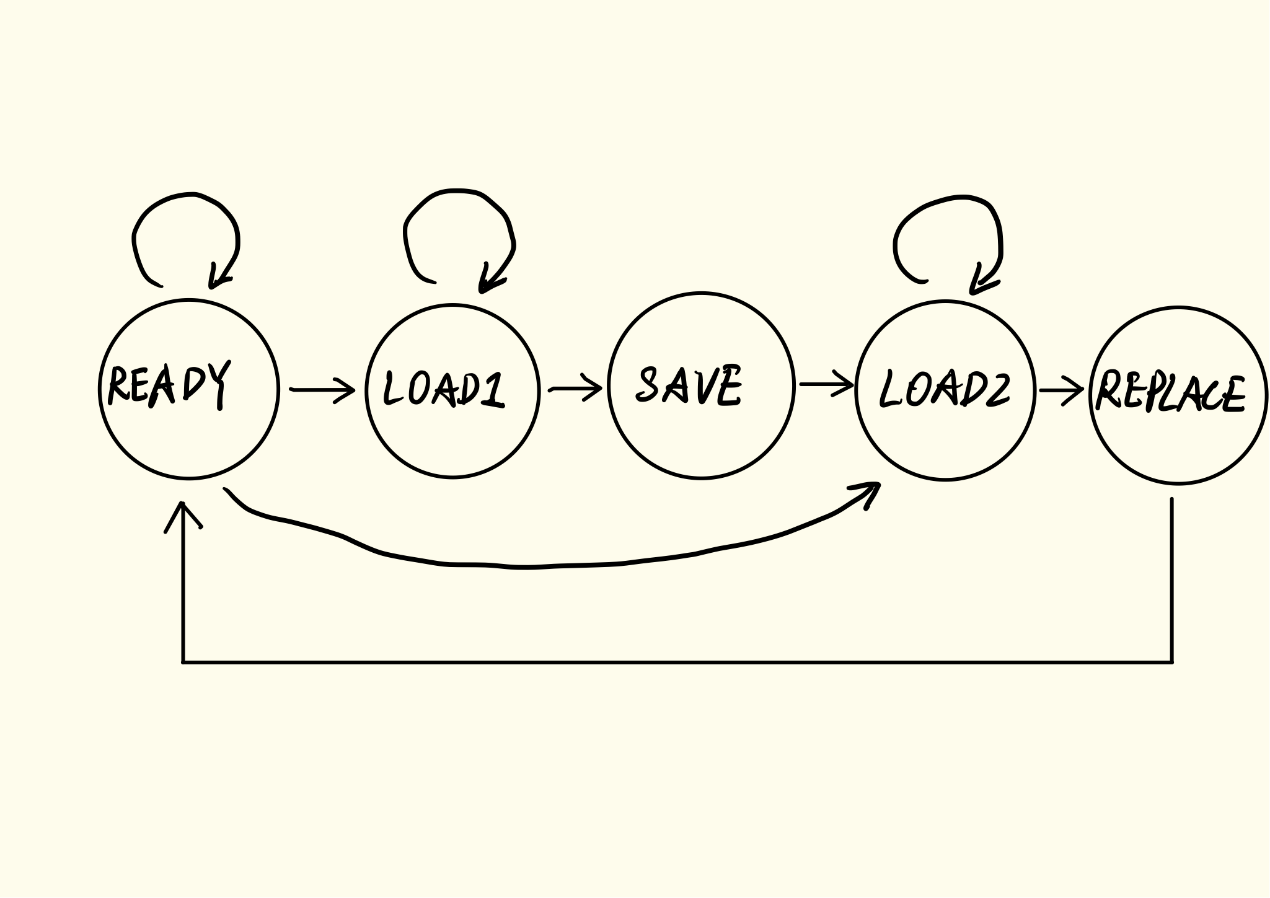
* 1. Cache
* 地址架构：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| address | 8:6 | 5:4 | 3:2 | 1:0 |
| 含义 | tag | set | block offset | offset |

* 实现：reg [132:0] cache2[3:0][3:0]；
* 4行4列的二维数组，每一个数据是一个位长133bit的向量；
* cache2[s][w]代表的是s组、w路的block，而每个block的结构为：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| bit | 132 | 131 | 130:128 | 127:96 | 95:64 | 63:32 | 31:0 |
| 含义 | dirty | valid | tag | data3 | data2 | data1 | data0 |

* 1. FSM



* + 1. cache的运行通过FSM来控制，FSM分为以下5个状态：
* READY：正常状态，不需要读写，或者需要读或写但是hit；
* LOAD1：当需要读或写但是miss的时候，根据替换策略判断要替换的路是dirty的，则需要把dirty的被替换块先写入data memory，再替换，再进行读或写；而在写入dirty块的时候，由于写入data memory较慢，则在写入成功之前，都会处于LOAD1状态；
* SAVE：写回dmem的延时结束，并把dirty块写回dmem，准备进入下一个从dmem读取数据的加载状态LOAD2；
* LOAD2：需要替换cache中的块时，首先要从dmem中读取需要的块，那么在读取出数据之前，都会处于LOAD2状态；
* REPLACE：从dmem中读取出来数据，LOAD2结束，进入了REPLACE阶段，此时把被替换块替换为读取出来的块，并准备进入READY阶段。
  + 1. 而这5个状态有以下几种变化
* 如果不需要进行读写，或者在需要进行读写的情况下是hit的，那么将一直处于READY状态，并根据读或写的需求进行读或写，或者都不进行；
* 在READY状态下，如果需要进行读或写，但是miss，那么将会进入LOAD1或LOAD2状态；
* 如果miss之后，根据替换策略获得的要替换的block不是dirty的，即数据没有更改过，那么会从READY进入LOAD2状态，尝试从dmem中读取需要的数据所在的block，当读取成功后，会进入REPLACE状态，并把对应块进行替换，接着返回到READY阶段，那么这一次就会hit，就能进行读或写的操作了；
* 如果miss之后，根据替换策略获得的要替换的block是dirty的，则要先把dirty的block写回，再进行替换，最后再进行读或写。那么首先要进入LOAD1状态，尝试把数据写入，延时够了，就进入SAVE状态，并最终写回dirty块，然后尝试从dmem中读出目标块，当数据准备好后，进入replace阶段，进行块的替换，再回到ready阶段，这一次就会hit，就能进行读或写的操作了。
  + 1. 访问情况

总的来说，在处理器对cache发出读或写的请求时，之后发生的事情只有这三种情况：

* hit: 直接读/写；
* miss & not dirty: 直接进行替换，再读/写；
* miss & dirty:先把dirty块写回，再进行替换，再读/写。
  + 1. FSM实现
* 通过state logic和nextstate logic进行实现；
* state logic是一个时序模块，通过always语句实现，选择nextstate或者在reset的时候选择READY状态；
* nextstate logic是一个逻辑模块，通过state、读/写的请求信号、是否hit、是否dirty、dmem读/写数据是否ready等信号，来判断nextstate是什么。
  1. decoder及其控制的对应模块

由于cache中需要很多控制信号来控制各个部分的运行，因此仿照controller中的maindec的样式，使用decoder来统一产生控制信号，来分别控制对应模块的运行。decoder通过state状态，以及是否要进行读写、是否hit的信号，来获得以下7个信号：

* read：在read信号为1的时候，会把hit状态下获得的数据加载到cache模块的数据输出上，由if语句进行判断；
* load：由load信号来选择输入到dmem中的地址，可能是把dirty块写回dmem的地址，或者是从dmem读取目标块的地址；
* we：控制是否要向dmem写入数据；
* clean：当clean为1时，已经把dirty块写回了，那么要把cache中对应块标记为不dirty；则选择对cache的路为要替换的路rway，而不是hit的路hway，并把cache中对应块的D位置0；
* replace：当replace为1时，选择对cache的路为要替换的路rway，而不是hit的路hway，并把对应块进行替换；
* write：当write为1时，选择hit的路hway，并把数据写入cache中的对应块；
* ready：当cache的读/写完成后，ready为1，否则为0，并会把整个处理器stall住。
  1. hit
* cache有4路，对每一个路，根据对应tag是否与地址的tag相等，以及是否valid，来产生每个路的hit状况，通过4个hitcheck模块实现；
* 根据每路的hit状态获得最终的hit和hit的路hway；
* 根据地址中的block offset，用MUX4，来从hit中的路对应的block中，读出对应数据，放在trd中；
  1. 从cache读出数据
* 只有hit的情况下，cache中才有目标数据，则在hit的情况下，会在READY状态，那么read信号控制把hit判断中获得的trd加载到readdata上
  1. 向cache写入数据
* clean为1时，写回了脏数据，会把对应的dirty改为0；
* replace为1时，会把对应块替换；
* write为1时，会先根据block offset，用MUX4选出写了数据后的block，再写入cache中。
  1. data memory读写
* 写入dirty块时，会加载dirty块的地址，而读取目标数据块时，会加载对应地址；该地址由load信号来选；
* we信号判断是否要进行写dmem操作；
* 对dmem读写时，会进行读写，并在读写之后用计数器对clk进行计数，在延迟8个周期后，dmem才会输出有效的ready信号，才代表数据准备好了，否则cache会进行等待；
* 计数器：当输入地址不变时，对计数器+1，否则代表要重新加载一个新的数据块，则清0.
  1. LRU替换策略

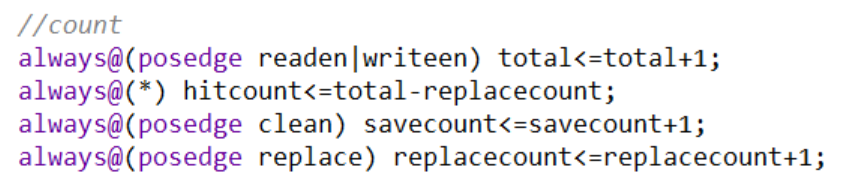
替换策略采用的是LRU，通过记录U[5:0]，并进行维护，来获得会被替换的路rway。

* U有6位，分别记录4个路之间的老幼关系。设4路分别为A、B、C、D，那么U中的6位分别记录AB、AC、AD、BC、BD、CD之间的老幼关系，如U[5]对应AB的老幼关系，若为0，则表示A比B老，否则表示A比B年轻，即A比B在更近的时间内使用过；
* 根据上面的规则，可以判断出那一路是最老的：如AB、AC、AD对应位为000，那么A比另外3路都要老，则要替换的rway路为A路，其他情况依次类推；通过assign以及三目运算的判断符来实现；
* 如果使用了某一路，则只需要含有该路的3个对应信号都修改为该路年轻就可以：如这一次对A进行了读或写，那么要把AB、AC、AD对应位改为111.

有了上面这些之后，就能通过U判断要替换的路rway，并根据情况对U进行更新。

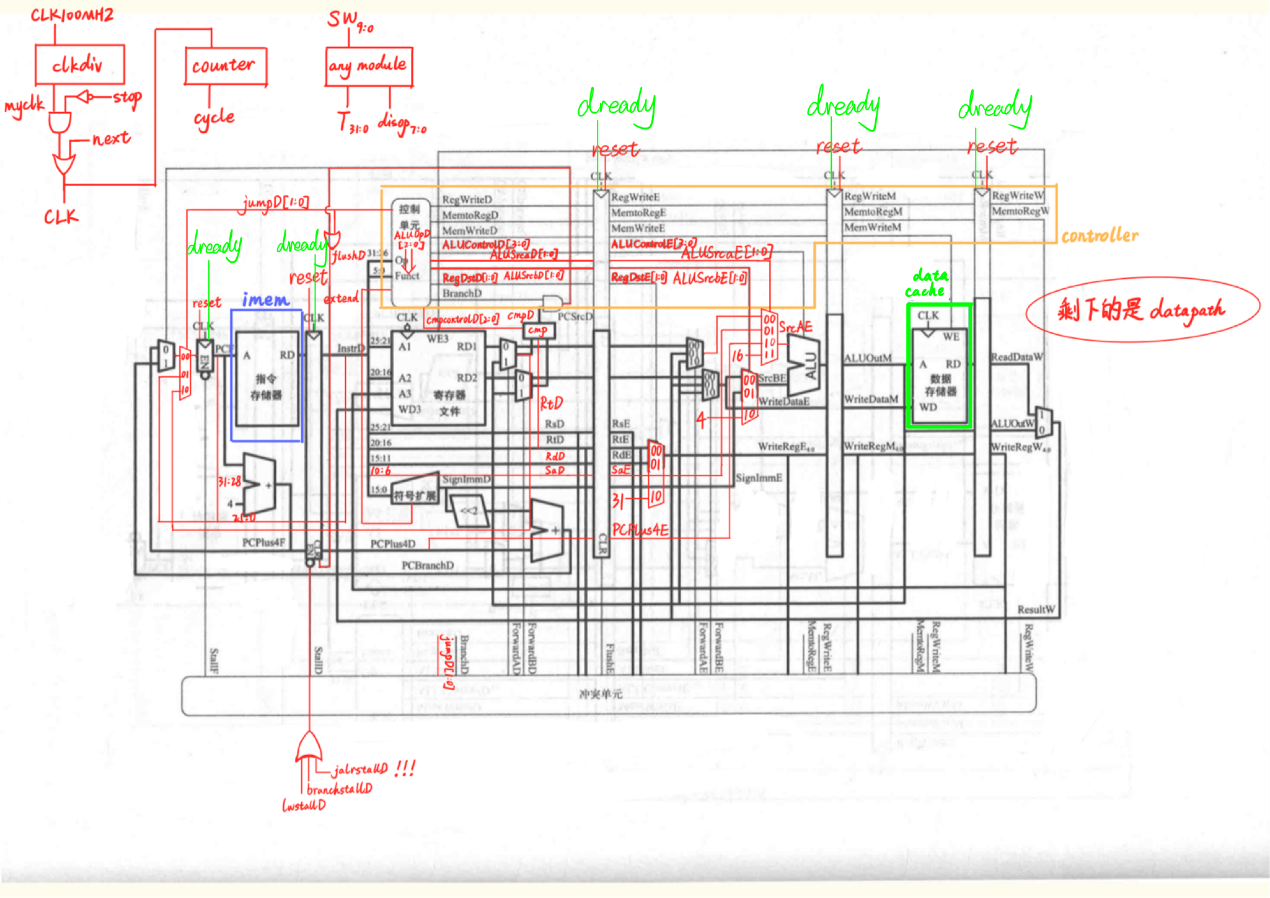
* 1. 计数

对总读写次数total、hit的次数、保存dirty块的次数、replace更换块的次数进行了计数：



# Cache对处理器的影响

（代码请看工程文件，在此不展示）



（原图可以查看**硬件设计.png**）

当cache的数据读写完成后，输出有效dready信号，处理器正常运行；而未读写完时，dready无效，使5个阶段的流水线寄存器的enable信号无效，即都处于stall的状态，也就是整个处理器停下来，等待cache的数据读写。

# 仿真测试

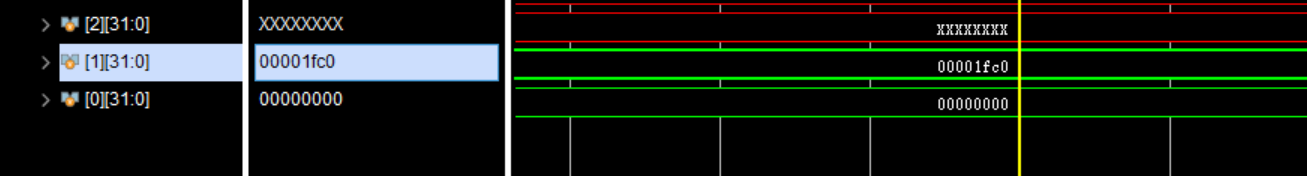
* 1. cache测试情况

cache的测试以下方面要进行测试：

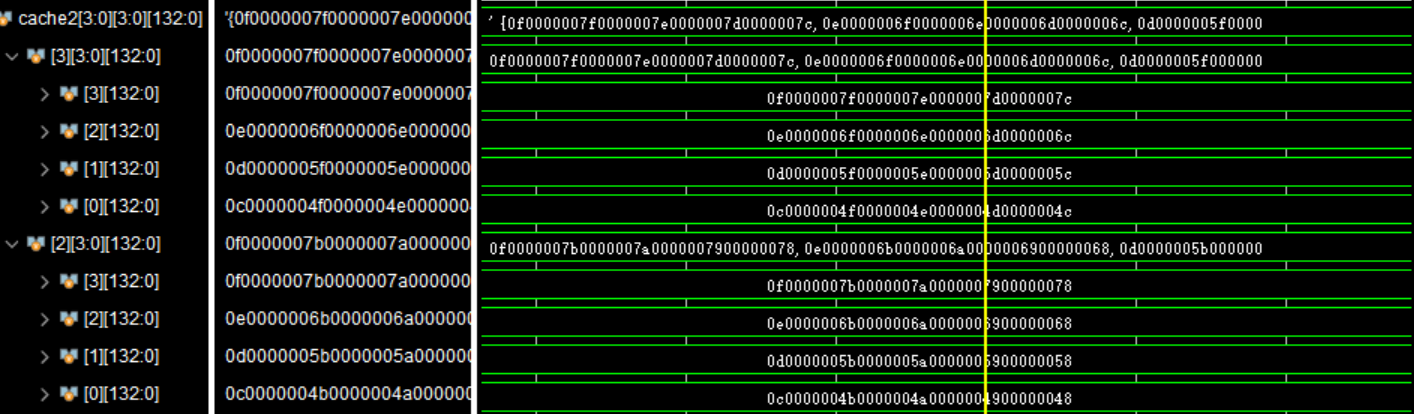
* hit: 直接读/写；
* miss & not dirty: 直接进行替换，再读/写；
* miss & dirty:先把dirty块写回，再进行替换，再读/写。

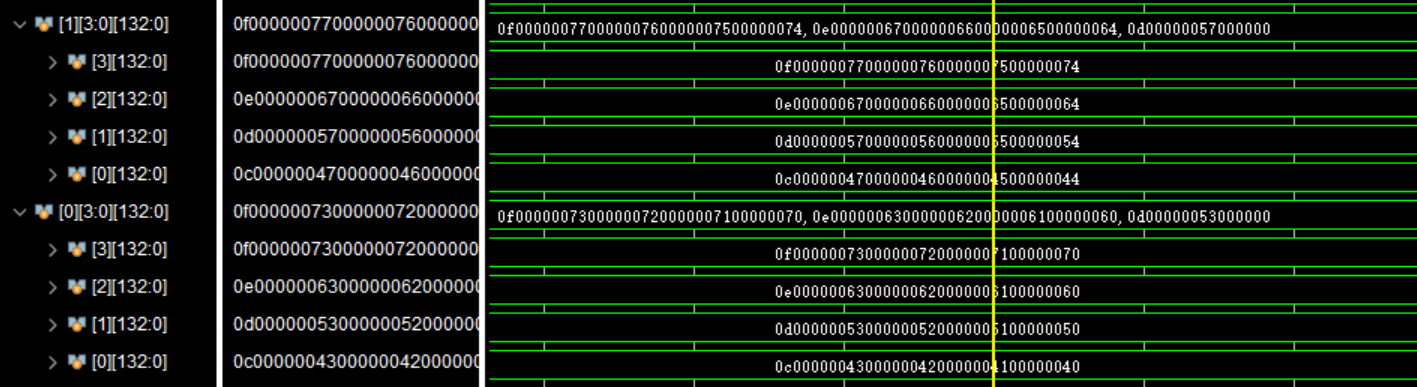
如果把读和写的情况分开，那么就一共是2\*3=6种情况，将在cache.txt代码中测试。此外还需要对cache的替换策略进行单独测试，将在LRU.txt中进行测试。那么cache的全部方面测试完了。

* 1. cache读写测试
* 在cache.txt中测试：依次sw数据0~127进入地址0~127中，再依次把0~127从地址0~127中lw出来，并同时进行累计，存在$1寄存器中。
* 0~127想加的结果为0x1fc0

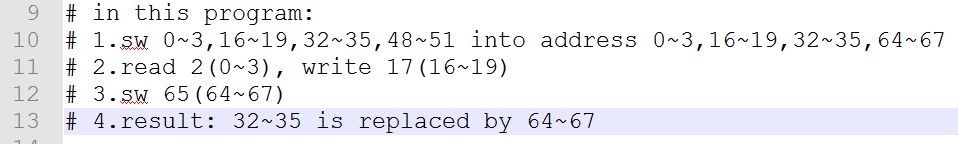


* 最终的cache状态





* 运行过程中的cache状态也是对的，就不一一进行截图了。
  1. 替换策略LRU测试
* 在LRU.txt中进行测试：



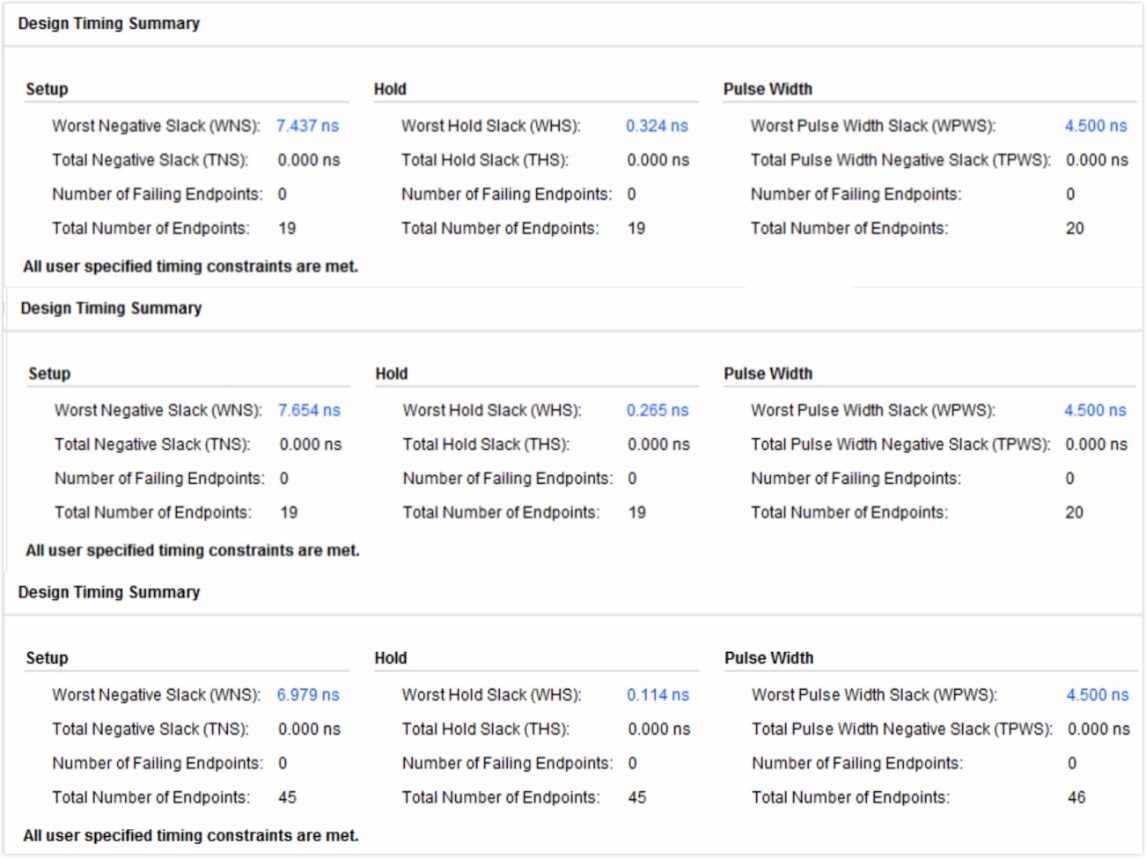
* 在sw操作后第0组中的4路都是满的，LRU年龄大小关系：0路>1路>2路>3路在read操作和sw操作之后，LRU年龄大小关系： 2路>3路>0路>1路，那么再进行sw操作时，还是要放在第0组，则要进行替换，替换的应该是2路
* 最终cache的状态：

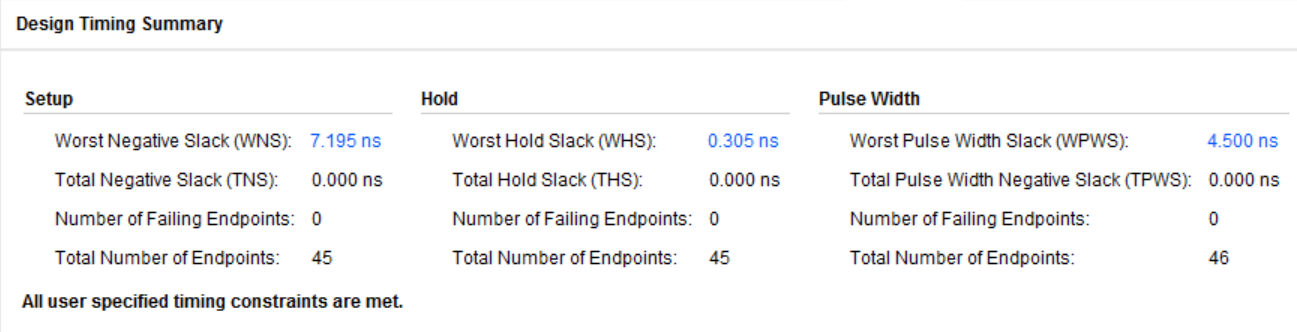


# 性能分析

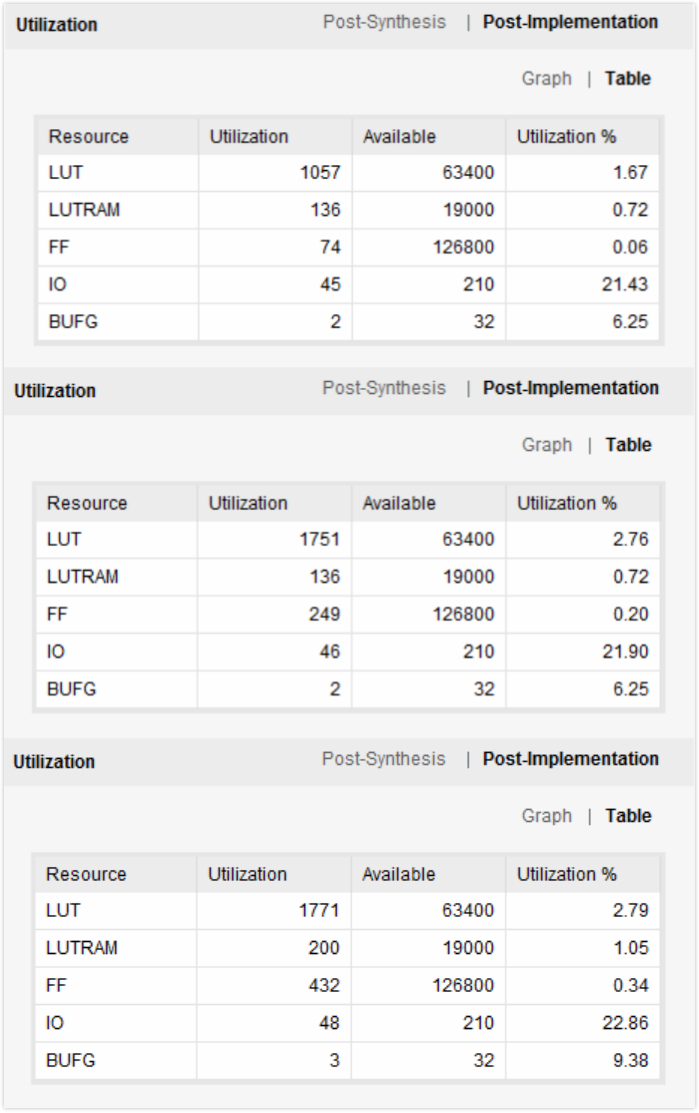
从上到下分别是：单周期、多周期、流水线、cache

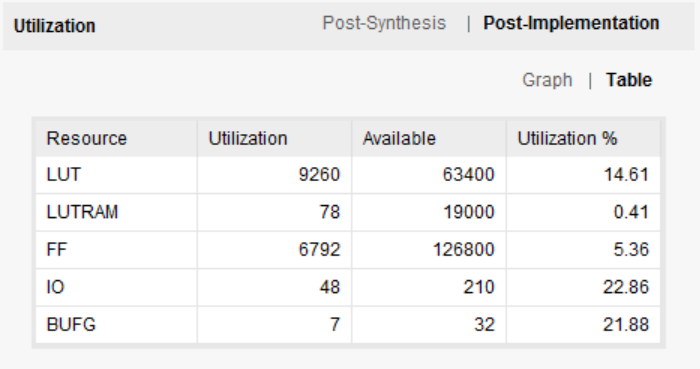
* 1. 时钟占用





* 1. 资源占用





# 实验感想

* 对cache有了更深的了解，对MIPS的学习的理解更加接近现实的MIPS设计；
* 对MIPS指令集有了更深的了解，并学会书写针对性测试cache、LRU替换策略的MIPS指令程序；
* 对verilog语言更加深入；
* 原来打算给instr memory也设计对应的cache，但由于考试周时间不足，最终很遗憾未能实现；
* 能熟练运用SW来实现功能，能使用LED以及8个7段显示数码来直观地显示所有数值、控制信号，能极大地方便debug及正确性验证。