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

1.1 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。

1.2 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**

1.3 规格

- register file: 32bit*32
- data memory: 32bit*512
- data memory cache: 4 set * 4 way * 4 * 32bit
- instruction memory: 32bit*512

1.4 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。

2 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

3 Cache 设计

（代码请看工程文件 dcache2.v，在此不展示）

3.1Cache

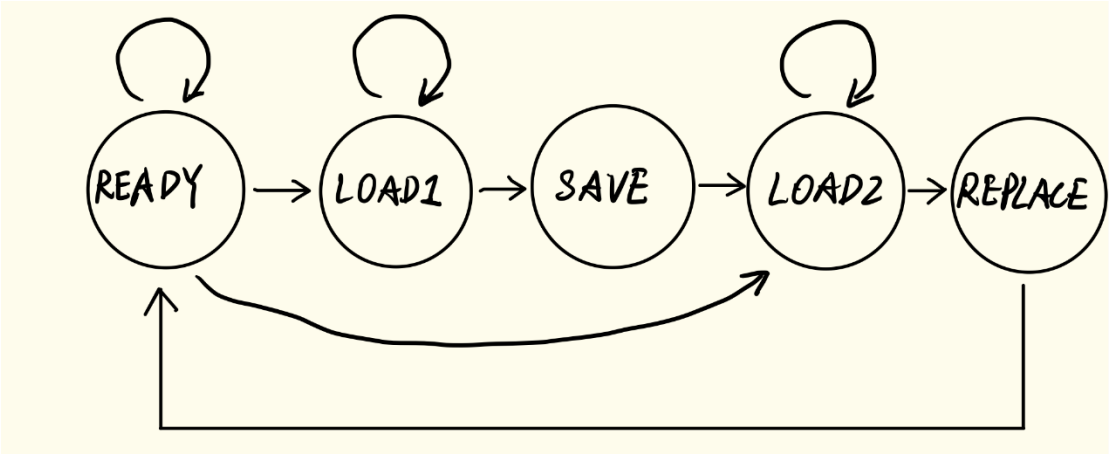
- 地址架构：

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

3.2FSM



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。
2. 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。

3. 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。

4. 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。

3.3decoder 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。

3.4hit

- 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；

3.5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

3.6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。

3.7data 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.

3.8LRU 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。

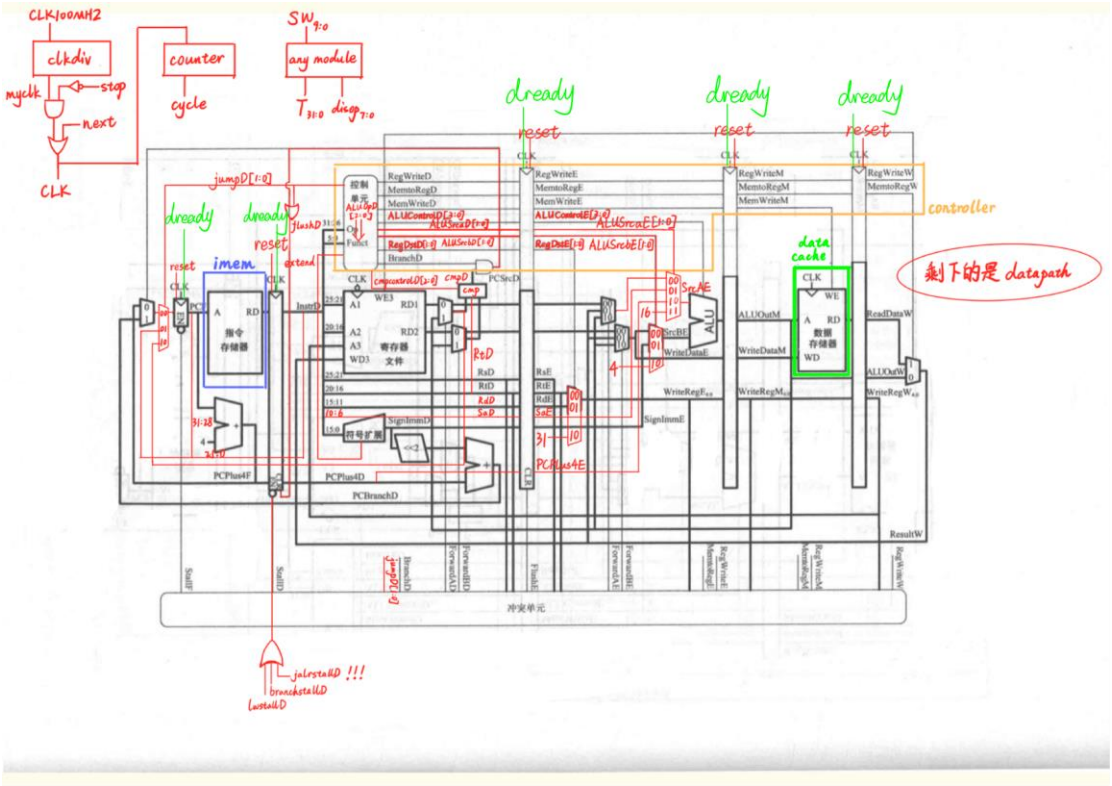
3.9计数

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：

```
//count
always@(posedge readen|writeen) total<=total+1;
always@(*) hitcount<=total-replacecount;
always@(posedge clean) savecount<=savecount+1;
always@(posedge replace) replacecount<=replacecount+1;
```

4 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。

5 仿真测试

5.1cache 测试情况

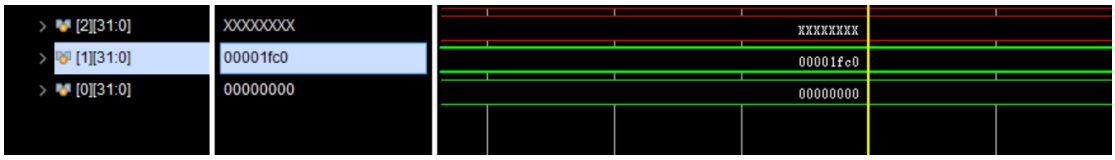
cache 的测试以下方面要进行测试：

- 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。

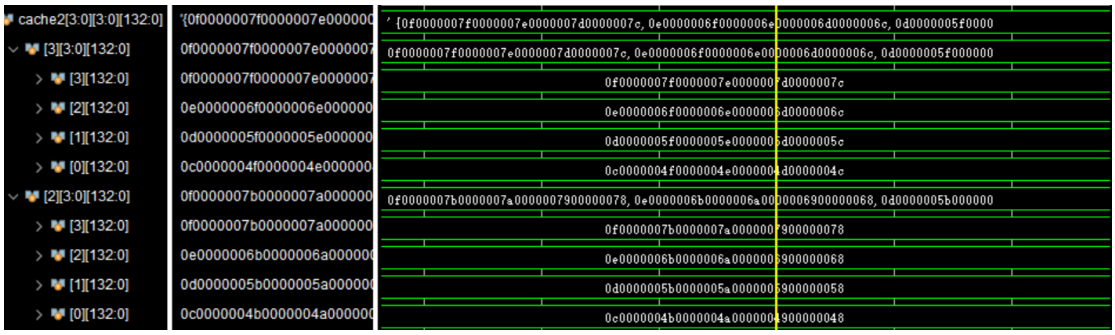
如果把读和写的情况分开，那么就一共是 2*3=6 种情况，将在 cache.txt 代码中测试。此外还需要对 cache 的替换策略进行单独测试，将在 LRU.txt 中进行测试。那么 cache 的全部方面测试完了。

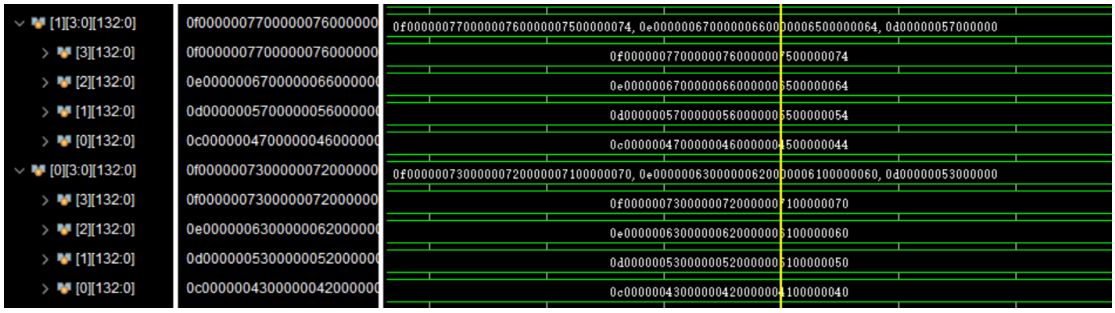
5.2cache 读写测试

- 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





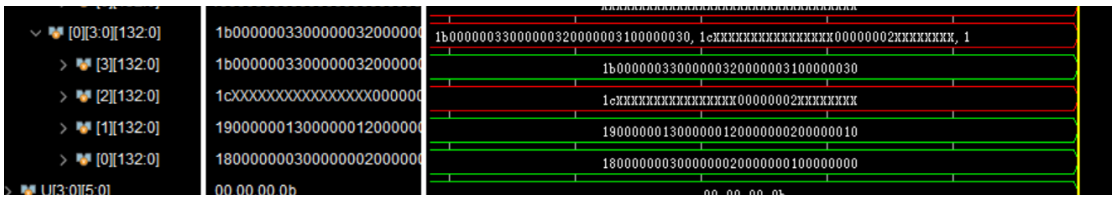
- 运行过程中的 cache 状态也是对的，就不一一进行截图了。

5.3替换策略 LRU 测试

- 在 LRU.txt 中进行测试：

```
# in this program:
# 1.sw 0~3,16~19,32~35,48~51 into address 0~3,16~19,32~35,64~67
# 2.read 2(0~3), write 17(16~19)
# 3.sw 65(64~67)
# 4.result: 32~35 is replaced by 64~67
```

- 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：



6 性能分析

从上到下分别是：单周期、多周期、流水线、cache

6.1时钟占用

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7.437 ns	Worst Hold Slack (WHS):	0.324 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	19	Total Number of Endpoints:	19	Total Number of Endpoints:	20

All user specified timing constraints are met.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7.654 ns	Worst Hold Slack (WHS):	0.265 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	19	Total Number of Endpoints:	19	Total Number of Endpoints:	20

All user specified timing constraints are met.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.979 ns	Worst Hold Slack (WHS):	0.114 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	45	Total Number of Endpoints:	45	Total Number of Endpoints:	46

All user specified timing constraints are met.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	7.195 ns	Worst Hold Slack (WHS):	0.305 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	45	Total Number of Endpoints:	45	Total Number of Endpoints:	46

All user specified timing constraints are met.

6.2资源占用

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1057	63400	1.67
LUTRAM	136	19000	0.72
FF	74	126800	0.06
IO	45	210	21.43
BUFG	2	32	6.25

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1751	63400	2.76
LUTRAM	136	19000	0.72
FF	249	126800	0.20
IO	46	210	21.90
BUFG	2	32	6.25

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	1771	63400	2.79
LUTRAM	200	19000	1.05
FF	432	126800	0.34
IO	48	210	22.86
BUFG	3	32	9.38

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	9260	63400	14.61
LUTRAM	78	19000	0.41
FF	6792	126800	5.36
IO	48	210	22.86
BUFG	7	32	21.88

7 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。