NE-Core 设计报告

东北大学 1 队

吴浩宇、孙平炜、陈宇童、刘向峰

2022 年 8 月

目录

一、 设计简介

本项目实现了一个基于 MIPS32 指令集设计的 CPU，支持 MIPS32 指令集的部分指令。CPU 流水线采用顺序双发射六级流水结构，同时还实现了两路组相连的 ICache 和 Dcache 以及 TLB，并且经过封装后通过 AXI 接口和外设通信。

(一) CPU 结构

我们在传统五级流水结构的 CPU 基础上对其进行修改，添加指令缓冲队列等部件以实现顺序双发射的六级流水线结构。在我们目前的测试中频率可以达到 70M，为了实现更全面的功能我们还为其添加了 cache、TLB 等部件。

(二) CP0 寄存器与异常处理

现阶段的 CPU 支持功能测试所需的全部例外情况和相应的寄存器，为了能够在项目种运行操作系统，我们还额外实现了 cahce 和 TLB 相关的例外处理机制。在流水线上，我们采用了精确异常的处理方式，异常检测分布在流水线的各个阶段，但是得到的异常信息都会随着指令流传递到 MEM 段，并在 CP0 模块进行统一处理。

(三) 分支跳转

由于我们采用的是双发结构，由 ID 段发出跳转请求，因此当流水线运行起来时一旦产生跳转则势必会造成两周期的损失。我们在指令缓冲队列中添加了判断逻辑，针对相邻空间的跳转请求进行优化，减少跳转指令造成的损失。

(四) TLB

为了在项目中运行操作系统，我们添加了 TLB 模块并实现了相关的异常处理。为了不被很少出现的 TLB 指令拖慢核心频率，我们为 TLB 配备了转发模块，以减少 TLB 指令和访存地址映射造成的关键路径过长的问题。

(五) Cache

cache 采用两路组相联设计，写回，写分配策略。使用 LRU 算法调度。单路 cache 大小为 4KB，故两路 cache 总容量 8KB，指令 cache 和数据 cache 总体共 16KB。

(六) AXI

我们在实现 cache 的同时也将 CPU 进行封装，通过 AXI 接口能够解决指令 cache、数据 cache 和 uncached 类型访存的冲突问题，并能够实现读写并行。在读取新数据的同时，可以将旧数据写回，减少了大量花在访存上的时间。

二、 设计方案

(一) 总体设计思路

我们的项目中实现了官方要求的 57 条指令、4 条 TLB 指令、4 条非对其访存指令以及 MUL 指令并且实现了相应的异常处理和相关 CP0 寄存器。

我们以传统的五级流水线结构为基础，在其上添加了指令缓冲队列、双发判断模块来实现流水线的双发射需求。为了充分利用指令缓冲队列的结构，我们还将其视作一个简易的分支跳转预测器，当产生跳转请求时可以在其中查找是否有目标 pc 以及相应的指令内容。

在指令执行阶段我们针对性地配备了两个不同功能的计算单元以满足两条流水线的不同需求，同时还能够减少线路的复杂度。

对于流水线的访存需求，为了克服添加 TLB 模块以后访存关键路径过长的问题，我们在原有五级流水结构的基础上添加了一级 DT 流水段，这样可以有效地切断关键路径。

在 MEM 段我们一方面对访存发回的数据进行解耦分配，另一方面通过其内部设置的 CP0 模块对流水线上产生的例外情况进行精确异常处理。此外，考虑到 TLB 指令的低频性，我们将 CP0 和 TLB 的交互都设置在此段。

当指令执行到达 WB 段时，在 ID 段被调换位置的跳转指令和延迟槽指令会恢复原有执行顺序，在此阶段将结果写回通用寄存器堆和 HILO 寄存器组。

在核外部分，我们为 CPU 配置了对应的指令 cache 和数据 cache 以减少访存开销，同时对 CPU 进行了封装，对外采用 AXI 接口方便系统调试和功能适配。

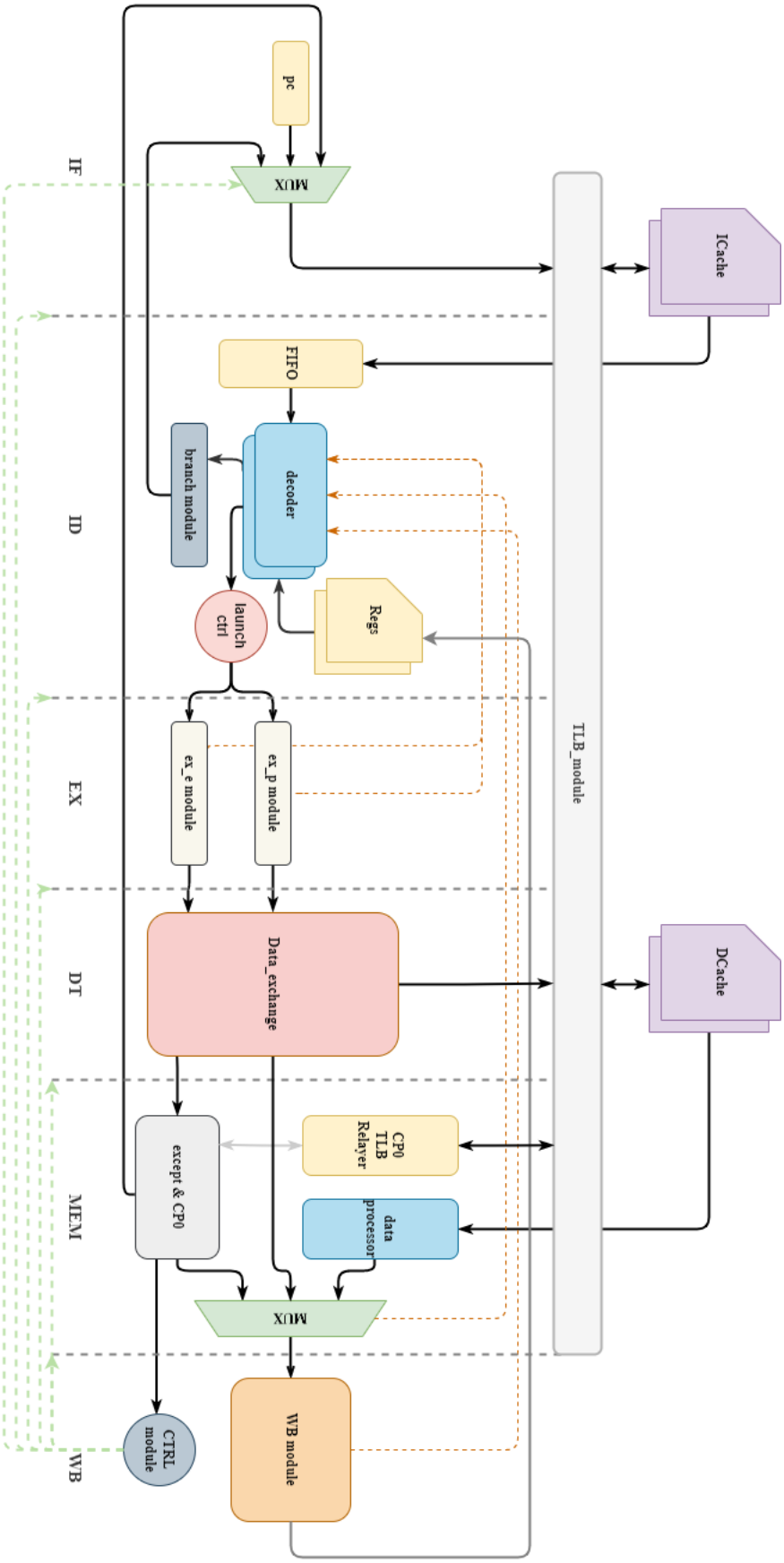


图 1 双发射六级流水线结构

(二) 流水线设计

我们的双发射六级流水线各部分的功能和设计方案如下：

1. IF 段

向指令 cache 发出读请求信号，默认访问地址为当前 pc+8，如果发生跳转或者需要进行异常处理则到相应的位置取址。当流水线全段暂停时也会暂停向 icache 发出取址请求。

需要注意的是，由于我们是双发射流水线，因此在取址阶段要做到 8 字节对齐，这样一来可能会不符合一些跳转和 CP0 指令的要求（pc[2]为 1）.这种情况我们在 IF 段同一按照 8 字节对齐方案发送取址请求，至于取回的指令是否都有效则会在指令缓冲队列入口判断。

2. ID 段

此段的主要功能是对指令进行译码和发射，依靠以下模块实现其功能：

- **FIFO 缓冲队列：**

两条指令能够双发射是有条件约束的，为了保存起暂时无法发射的指令，我们在 ID 段设置了指令缓冲队列。由此队列负责接收 icache 发来的指令，并提供给后端的译码器进行译码和发射。队列的容量为 32*4Byte，当队满时会向流水线的控制模块发出请求，暂停 IF 段的取址工作，直到队列腾出空间。

前文提到，IF 段发出的取址 pc 都是 8 字节对齐的，对于有些跳转和异常处理的情况，我们要在指令收容进 FIFO 前做具体的甄别。在我们的流水线中，一旦出现 pc 由于跳转或异常而不符合默认的八字节递增的情况时，目标 pc 会被送到 id 段并且让 FIFO 入口的对照机制触发，只有从 icache 取到的指令的 pc 和当前的目标 pc 一致时，该指令及其后续的指令才能被 FIFO 正常接收。

- **译码器**

在此模块中会读取 FIFO 队头的两条指令进行译码解析，进而确定指令类型、源寄存器地址、目标寄存器地址、是否跳转、跳转目标地址。从而完成运算控制信号生成、取操作数、跳转控制等功能。

- **发射控制和跳转指令处理**

此模块包含了双发的控制逻辑和跳转指令的发射控制逻辑。对于双发情况的选择，为了平衡设计复杂度和双发效率，我们的发射逻辑如下：

a) 跳转指令始终和延迟槽绑定，双发的同时跳转指令和延迟槽交换位置

b) 两条指令存在数据相关则单发

c) 乘除法，CP0 相关指令要单发

d) 两条访存指令同时出现则单发

上述没有涉及到的情况默认都进行双发射。

同时，我们选择在 ID 段进行跳转指令的发射以减少周期损失。具体而言，在此阶段会根据跳转指令要求进行判断，如果需要进行跳转则会更新 FIFO 入口的目标 pc、触发对照机制并向 IF 段发送跳转请求。同时我们也会对 FIFO 里写入过的指令进行匹配，如果在其中发现了跳转的目标 pc 则可以将其有效位恢复并在下一阶段进行译码和发射。这样一来可以一定程度上减少分支跳转的周期损失，尤其是对于近邻域的跳转操作而言。

3. EX 段

为了配合 ID 段的双发逻辑，我们在指令执行阶段也做出了有针对性的调整。从前文提及的发射逻辑中不难看出，第二条流水线上的计算单元功能需求是要比第一条流水线少的，乘除法任务只会出现在第一条流水线上。因此，我们对计算单元进行了差异化设置，在满足需求的同时降低结构复杂度，从而能够适配更高的频率。

总体来说我们的运算单元包含由 add、sub、slt、sltu、and、nor、or、xor、sll、srl、sra、lui 等 12 种基础运算部件组成的 ALU-base，2 周期 booth-wallace tree 乘法器，32 周期试商法除法器。

表 1 计算单元功能

名称	能够执行的指令或功能
Ex_p	ALU-base、访存、乘除法、CP0 相关指令
Ex_e	ALU-base、访存

4. DT 段

为了减少关键路径长度提高频率，我们选择在流水线中为数据访存请求多设置一级流水段，这样可以有效地将访存地址计算和 TLB 以及 cache 的查找逻辑阻断开来，有助于我们提升 CPU 频率。为了实现网口相关功能，我们还添加了指令集手册中的非对齐访存指令。

5. MEM 段

在此阶段，流水线会对 dcache 中取回的数据按照指令要求进行分配，同时对 CP0 寄存器进行访问并处理流水线上的异常情况。对于流水线上可能发生的异常情况，我们设计了专门的线路用来记录并且在 MEM 段统一进行精确异常处理。

表 2 实现的例外情况

例外标识符	ExcCode	检测位置
Int	0x00	MEM 段 CP0 模块
Mod	0x01	MEM 段 CP0 模块
TLBL	0x02	MEM 段 CP0 模块
TLBS	0x03	MEM 段 CP0 模块
Adel	0x04	ID 段译码器、EX 段运算单元
Ades	0x05	EX 段运算单元
Sys	0x08	ID 段译码器
Bp	0x09	ID 段译码器
RI	0x0a	ID 段译码器
Ov	0x0c	EX 段运算单元

在流水线上对异常记录线路进行更新时要注意其优先级，一条指令有可能在多个位置触发不同的例外，因此越靠后的检测点在更新时越要考虑之前的异常信息能否被覆盖。

6. WB 段

指令执行完毕后会在此阶段进行写回寄存器的操作，包括通用寄存器堆和 HILO 寄存器。在 ID 段被翻转过的两条指令在此阶段也会归位并按照原顺序写回。

(三) TLB 设计

为了能在我们的项目上运行操作系统，我们添加了 16 路的 TLB 模块，并且实现了相关的指令和例外处理。

对于流水线的正常取址和访存请求，流水线给出的虚地址会在 TLB 模块里经过查找和映射得到实地址并和 cache 或 AXI 进行进一步的交互。而对于 TLB 指令，我们选择让其统一在 CP0 模块与 TLB 进行交互。也就是说读写请求都会在 MEM 段完成，为了能够优化时序，我们为 TLB 和 CP0 的交互配备了转发模块，因此在 TLB 指令执行时流水线会全段阻塞 3 个周期来让 CP0 和 TLB 完成交互任务。

对于 TLB 相关的例外情况，其也会在转发模块中经过一周期后再发给流水线，这样由于 IF 段和 DT 段访存地址引发的例外情况也可以分别发给 ID 段和 MEM 段，这一方案与我们的例外处理逻辑也是十分贴合的。

(四) Cache 设计

我们的数据 cache 和指令 cache 都采用两路组相联设计，写回，写分配策略。使用 LRU 算法调度。单路 cache 大小为 4KB，故两路 cache 总容量 8KB，指令 cache 和数据 cache 总体共 16KB。

此外由于是双发机制，因此我们对指令 cache 和数据 cache 针做了差异化设计。其中指令 cache 单路设置了 64 个 cacheline，每 cacheline 内有两个 line 存储单元，每个单元可存储 8 条指令。故指令 cache 单路容量为 64*2*8*4Byte，即 4KB 大小。line 存储单元根据高低地址进行区分，高地址存储单元存储 pc[2]为高电平的指令，低地址存储单元存储 pc[2]为低电平的指令，在取址过程中二者共用同一个偏移量（offset），可在一次访问过程中读出八字节对齐的两条指令。数据 cache 共有 128 个 cacheline，每 cacheline 使用一个 line 存储单元。故数据 cache 单路容量为 128*8*4Byte，即 4KB 大小。

当访问命中时，可使 cpu 核心流水地完成读或写请求，无需阻塞主流水线。当检测到缺页时会立即阻塞流水线，使主流水线保持当前请求，并向 AXI 接口发出读请求。此时，如果预计要写回的 cacheline 中存在脏数据，则 cache 会在发送读请求的同时发送写请求，将 cacheline 中的脏数据写回。当 cache 模块接到 AXI 接口发回的新数据并写入后，主流水线的请求会转为命中状态。

另外，我们还实现了 uncache 模块用来访问不可缓存的数据，当主流水线发来访存请求时，默认为访存缺失，阻塞主流水线，并将请求转发给 AXI 接口进行访存。

为减少对主流水线的阻塞，在该模块中实现了一个简易的写缓冲。其作用是在出现写请求时，将该请求保存下来慢慢执行，不阻塞流水线，与主流水线同时进行。如果下一个请求到来时，该写请求尚未完成，则需阻塞流水线直到该请求完成后再判断新请求的类型来执行。该操作将一部分写请求所花费的时间隐藏到了其他指令执行的时候。有效地缩短了程序运行时间。

(五) AXI 设计

在设计过程中，我们将 CPU 封装为 AXI 接口，用于与系统环境和外设的交互。

其中，数据 cache 的请求和 uncached 类型的访存请求是互斥关系，不会同时出现。uncached 类型的读请求和写请求也是互斥关系，不会同时出现。如果同时出现指令读和数据读，则先完成指令读，然后完成数据读。写行为同时只会出现一种，且可以在读行为的过程中同时进行。写操作或者读操作完成后，需等待另一种行为完成才能再次回到检测状态，以保证二者的一致性。

表 3 访存请求处理顺序

优先级	类型
1	指令 cache 读请求
2	数据 cache 读请求和写请求
3	uncached 类型访存的读请求/写请求

三、 设计结果

(一) 设计交付物说明

目前，我们的 NE-Core 主频为 70M，内置 TLB 模块和两组 cache 模块，整体封装为 AXI 接口。在我们本地进行的测试中，NE-Core 可以上板通过四项测试。

根据比赛提交包的规范在发送到邮箱的压缩包内按以下方式对作品进行组织：

```
NEU_1_wuhaoyu.zip
    |---README.pdf
    |---score.xls
    |---design.pdf
    |---sran_src(我们的设计最终封装为 AXI 接口 因此没有提交 SRAM 代码)
    |---src
        |---mycpu
        |---perf_clk_pll.xci(cpu_clk 设定为 70M)
    |---bit
        |---func_test
            |---axi_mem_game_test/mem.bit
            |---soc_axi_func/axi_func.bit
        |---perf_test/perf.bit
        |---system_test/sys.bit
```

(二) 设计演示结果

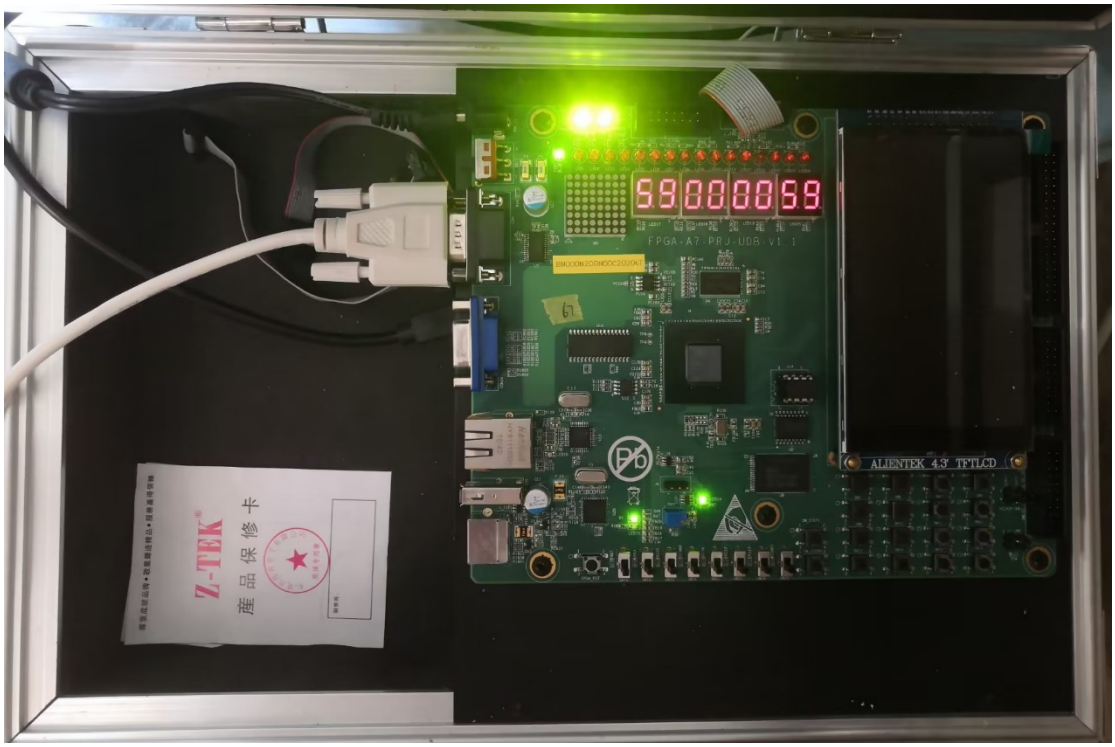


图 2 功能测试上板

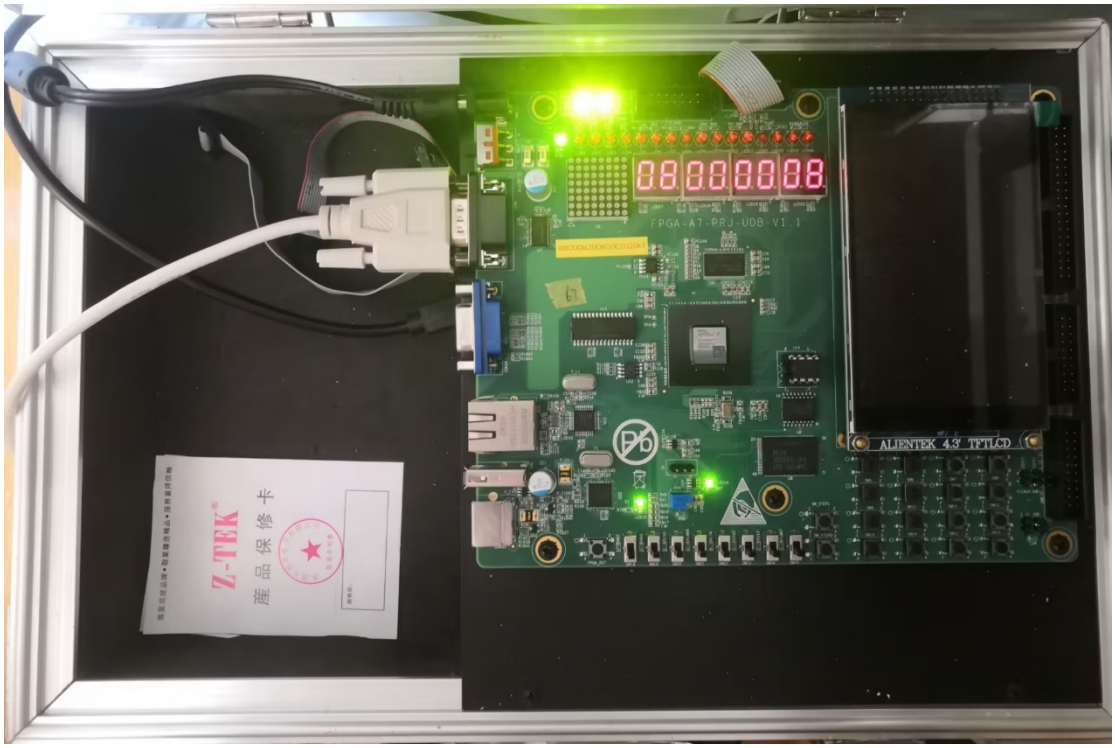
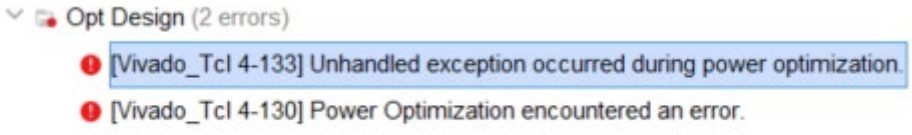
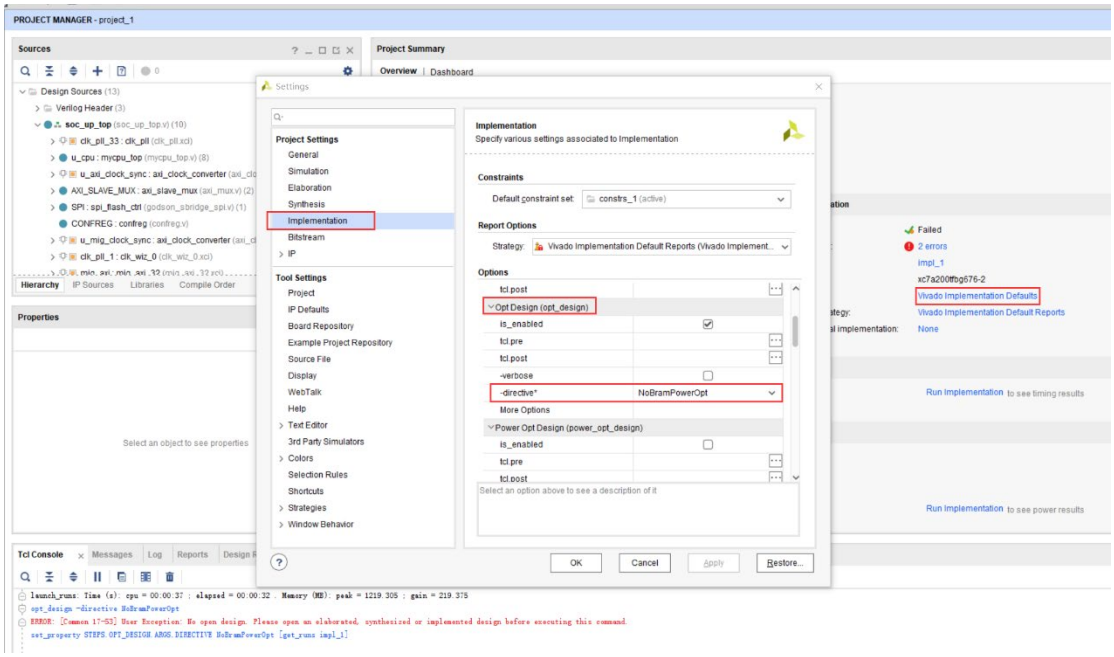


图 3 记忆游戏上板

我们的系统测试在 vivado 进行 implementation 时遇到了如下问题：



查阅资料后并未在代码方面找到有效方法。因此我们根据 Xlinx 社区的建议将 implementation 的 Opt Design 进行了如下调整：将 drective 项设置为 NoBramPowerOpt。



在我们的本地测试中，系统测试可以全部通过，但是有部分测试程序需要多尝试几次，如果在官方进行验证时遇到此问题还麻烦评委老师花时间进行一下复现。

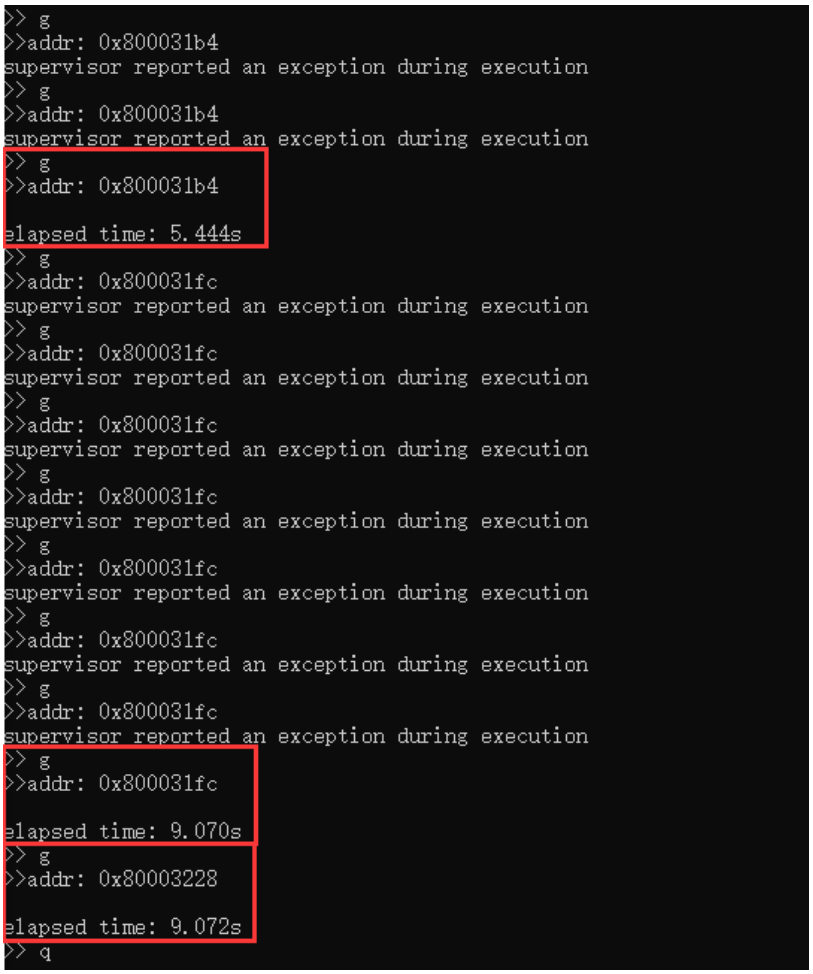
```
C:\Users\Administrator\Desktop\nscscc-group\system_test_v0.01\supervisor-mips32\term>python term.py -s com4 -b 57600
MONITOR for MIPS32 - initialized.
>> g
>>addr: 0x8000300c
supervisor reported an exception during execution
>> g
>>addr: 0x8000300c

elapsed time: 0.395s
>> g
>>addr: 0x8000303c
supervisor reported an exception during execution
>> g
>>addr: 0x8000303c

elapsed time: 0.368s
>> g
>>addr: 0x800030c4

elapsed time: 5.417s
>> g
>>addr: 0x8000315c
OK
elapsed time: 0.000s
>> g
>>addr: 0x80003180
supervisor reported an exception during execution
>> g
>>addr: 0x80003180
supervisor reported an exception during execution
>> g
>>addr: 0x80003180

elapsed time: 9.070s
```


>> g
>>addr: 0x800031b4
supervisor reported an exception during execution
>> g
>>addr: 0x800031b4
supervisor reported an exception during execution
>> g
>>addr: 0x800031b4
elapsed time: 5.444s
>> g
>>addr: 0x800031fc
supervisor reported an exception during execution
>> g
>>addr: 0x800031fc
supervisor reported an exception during execution
>> g
>>addr: 0x800031fc
supervisor reported an exception during execution
>> g
>>addr: 0x800031fc
supervisor reported an exception during execution
>> g
>>addr: 0x800031fc
supervisor reported an exception during execution
>> g
>>addr: 0x800031fc
supervisor reported an exception during execution
>> g
>>addr: 0x800031fc
supervisor reported an exception during execution
>> g
>>addr: 0x800031fc
elapsed time: 9.070s
>> g
>>addr: 0x80003228
elapsed time: 9.072s
>> q

四、 参考设计说明

我们项目中的 cpu 流水线参考了《自己动手做 cpu》[1]的传统五级流水结构，在其基础框架上进行模块的改良和整合实现双发并优化关键路径提高频率。双发的控制逻辑上我们综合考虑了往届大赛的一些双发射作品的发射策略并结合我们的设计做了相应的改良。

在 cache 和 axi 的实现中，我们在设计初期参考了基于计分板算法的乱序多发射处理器的设计与实现[3]项目中的设计模式以及我校去年的参赛作品，并对其进行了重构与优化以适应双发需求。

五、 参考文献

[1] 雷思磊. 《自己动手做 cpu》.
[2] 汪文祥、邢金璋. 《 CPU 设计实战 》.
[3] 倪仁涛 基于计分板算法的乱序多发射处理器的设计与实现. 东北大学 2022 年.
[4] MIPS32 官方文档.