ACM 班体系结构暨计算机系统大作业

李嘉森

作业二是本学期的大作业，主要目标是使用 Verilog（或其它 HDL）完成一个能实现 32 位 RISC-V 指令集整数部分用户级别实模式的 CPU。RISC-V 是 UC Berkeley 设计的开源指令集架构，在设计上属于精简指令集。

为了方便调试 CPU 及运行程序，本次作业采用在板上实现内存，利用 UART 接口进行数据传输及调试的方式来支持 CPU 的运行。整个顶层架构由 CPU，通信接口模块，输入输出接口，板上内存，以及内存总线等模块构成。

我的文件提交有：

0.ALU.v 由所有的 CPU 共享，能够即刻得到运算结果。延时远小于 1 个周期。

1.文档中附了我的 cpu.v。模拟正确并且能够生成正确的 bitstream。

采用**三级流水**，有一个很小的 **cache**（256 字节），第一级流水是取指令，第二级是操作，第三级是存储。每一级流水在执行好后，向前一个流水发送 ready 信号（清空指令 Buffer），向后一个流水的指令 Buffer 发送指令和数据。

Cpu 和 alu 形式上分离。CPU 和 Cache 形式上一体。

Cache 是直接映射 write back。每个 block 有 4 字节。

CPU 向 Cache 输入指令，等待 Cache 执行，执行完后 Cache 向 CPU 返回 ready。之后 CPU 可以从里面读入数据。如果要访问的地址在 cache 中不存在，则 miss，从 ram 里面调数据。在调数据前把原本的内容写回 ram 里面，调数据后返回 ready。如果要访问的存在，则直接返回 ready。

内存使用流水访问，5 个周期可以读取 4 字节，4 个周期可以写回 4 个字节。

IO 输入用的是 12 周期读入，5 周期输出。原因是我认为 IO 比起内存操作少得多。

此 CPU 是我提交的所有 CPU 里面最慢但是最稳定的。

2.此外，文档中还附了模拟的 Cpu（模拟）.v。cache 是 16KB 的。

这个文档由于综合时间太长，没有综合，在模拟的时候表现很好。每条指令大约 2.5 个周期能够完成。

设计思路和上一个大致相同，但是 Cache 大一些，并行程度更高一些。但是综合时间比较长，我电脑慢，跑不出来。流水中通常能够有至少两个部件同时运行。此外允许操作少的指令提前结束：如果某一个指令不用第三个部件，可以由第二个部件抹去这条指令。此外采用了更加快速的 Cache 接口方式，但是 miss 的代价比较高。

此 CPU 是这三个 CPU 中最快的，但是只提供模拟。

3.此外，文档中还附了一个不带 reset 的 cpu.v 文件。Cache16KB。

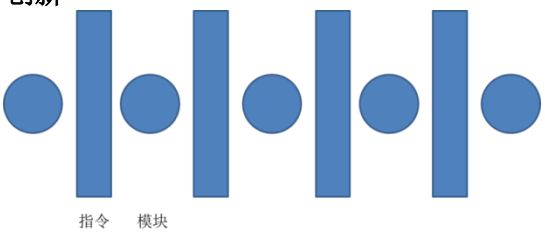
这个能够正确模拟，能够生成正确的 bitstream，但是没有 reset 功能。此 CPU 由同学帮忙烧制过，发现每次的第一个程序都能正常运行。

设计思路和第一个也相同，但是比第一个快。和第一个 CPU 仅仅两个区别：没有 reset 功能，cache 更大。

4.CPU1 和 3 的两个 bit 文件，非倍频版本

注：由于是异步的芯片，除了内存操作必须保障时序之外，**可以有任意的超频**，并且不必在意线路，因为任何除了内存的操作可以延续到下一个周期，或者说没有亚稳态的一大堆问题。最大尝试过**两倍的超频**。**瓶颈**之一主要是板子上的 RAM 和它的接口部分跟不上。

创新



这是一个流水的大致示意图，圆形的是模块的示意，方形的是指令的 Buffer。指令从左往右流动。

我们发现有些时候，流水的一个模块需要的时间可能会差别很大，比如 Instruction Fetch，可能是 1 个周期，也可能是 8 周期，而且每一个周期的时间可能都不同。在一个频率非常高的电路里面，**保持同步的时钟的代价是非常高的**，不论是在电量消耗还是在多余周期的损耗还是在难以倍频上都有很大的限制。

于是这个时候，为了避免每个模块都被安排上相同的周期，可以这样：每个模块都检查下一个指令的 Buffer 里面是不是还有空位置。如果有空位置，则这个模块可以运行。而 Buffer 是空的就是一个部件是 ready 的意思。

以上就是一个异步芯片的思路。异步芯片有很多**优点**：节电、节约等待时间等等。由时钟管理的芯片的运行速度不会比其最慢的部件快，它只有在它的每个部件都完成了自己的工作以后，才会产生响应。相反，在**异步芯片**上的晶体管之间可以**独立地交换信息**，而无需等待其他的任何事情，因而它可以**以所有部件的平均速度运行**。

比如 load 和 store 这两类语句，有可能前一个 miss 了，后一个 hit 了。如果模块全都是一个固定的周期，有可能会造成流水的阻塞或者每个模块的周期数都太大。这是我们不希望看到的。

为了实现每个模块都能取走前一个指令 Buffer 的指令，每个模块必须能够修改前一个指令 Buffer 的数据。为了实现每个模块都能把处理好的指令放进下一个指令 Buffer，每个模块必须能够修改下一个指令 Buffer 的数据。

所以我们的“模块”并不是形式意义上的模块，而是逻辑意义上的模块。

根据现代的 **ASIC 设计理念**，我采用的是**集合了同步设计和异步设计的设计方式**：模块内部进行同步，模块之间异步传输，**既有同步时序稳定可靠的优点，又有异步时序模块互相独立的优点**。

我的代码为了实现这一点，把所有的内容都摆在了一个形式上的 module 里面。包括 cache 也一样。

我有一个专门管理内存的模块组合：cache_XXX 寄存器

```
reg [4:0] cache_busy;          //表示占用 cache 的模块编号，0 表示 cache 空闲
reg [4:0] cache_ready;         //数字 31 表示 cache 已经就绪，可以吐出数据，也可以表示已经接受数据，
```

reg cache_io; //1 表示写回，0 表示读取
reg [17:0] cache_addr; //cache 地址
reg [31:0] cache_out; //写回的数据
reg [31:0] cache_in; //读取的数据
reg [4:0] cache_len; //操作的长度，只接受 2 的幂次(1,2,4)

此外，在指令向后流动的时候，我可以自动让下一条指令提前开始预读。我的流水允许一条指令在没到最后一个器件就提前消逝，避免造成拥堵。ALU 的运算不占任何周期，能够即刻得到结果。此外在 Cache 是空闲的时候允许 CPU 的器件绕开以上接口直接访问 Cache。

达到的效果：
模拟 CPU 的时候：
GCD 的模拟 CPI 是 3.6 左右（原因是过多的 Compulsory Misses）
Q-sort（N=1000）的模拟 CPI 是 2.5 左右
在局部能够达到平均 CPI 是 2
烧制板子的时候：
在非倍频下，实现速度为约 4.5 秒的 PI 程序运行。
两倍倍频时间减少一半左右。

创新点：异步芯片和 ready 机制。绕开接口窃听机制。指令提前消逝。预读指令。不等周期元件。内存流水访问。倍频。设计简易具有可移植性，方便修改。

异步的优势

异步设计中，由于控制信号本身就和数据信号的状态相关，可以比较容易地解决亚稳态的问题。

异步电路的速度可以比较好地适应变化的外部条件。如果条件发生变化，导致电路延时变大，异步电路本身就可以适应这种变化，而不会发生功能的错误。

同步芯片运行时，会被最差性能点限制，异步没有相关的缺点。

在转化为 ASIC 的时候，由于不必使用一个太强力的同步时钟，可以节约很多功耗，这种设计思路可以在不同的板之间实现比较高速的互联。

此思路可以避免太多的电磁干扰，为高倍数的倍频提供可能性，同时在实际电路的设计中可以增加集成度。

异步的芯片可以避免潜在的黑客下手。时钟本身为寻找一个信号提供了很大的便利。而异步则更散乱，没有明显的信号可供观察。潜在的黑客将无从下手。

由于异步的性质，在真实的 ASIC 环境中，性能将很可能会优于在 FPGA 上同时间级别的设计。

简易

全部代码 16KB，包含注释六百多行，用尽量少的代码和尽量少的板上资源实现了一个性能不错的芯片设计。这样的话，设计比较方便移植到其它地方。

遇到过的问题：

由于局部时序性，每个小部分必须要有自己的时钟（不必同步），为了简化电路设计，用了同一个时钟，延时造成的不同步并不会有任何影响，所以除了 RAM 外可以超频很多倍。

由于不熟悉烧制，于是请同学帮忙烧制过 FPGA。

两个 Cache 和 ram 接口险些出现抢总线的情况，于是改掉成了一个 cache。

流水读入 Ram 容易写错，纠正了。

Data Hazard 通过调整流水数目规避了。

Branch Hazard 通过强制不执行 branch 语句后面的语句规避了。

Structure Hazard 通过 cache_busy 来规避了。同时访问 cache 则后一个流水先访问到。

程序复读的问题目前没法解决，上板不会复读。

电路可以抗电磁干扰，并且实现优美的“弹性流水线”，但是弹性流水线有间距的缺点。

"Elastic" pipelines, which achieve high performance while gracefully handling variable input and output rates and mismatched pipeline stage delays.

设计思路来源：

“Biological neural networks are fundamentally asynchronous, as reflected by the absence of an explicit synchronization assumption in the continuous time SNN model given in the Spiking Neural Networks section. Accordingly, asynchronous design methods have long been seen as the appropriate tool for prototyping SNNs in silicon, and most published chips to date use this methodology. Loihi is no different, and, in fact, the asynchronous design methodology developed for Loihi is the most advanced of its kind. For rapid neuromorphic design prototyping, we extended and improved on an earlier asynchronous design methodology used to develop several generations of commercial Ethernet switches. In this methodology, designs are entered according to a top-down decomposition process using the CAST and CSP languages... Given a hierarchical design decomposition written in CSP, a pipeline synthesis tool converts the CSP module descriptions to Verilog representations that are compatible with standard EDA tools. ”

M. Davies et al., “Loihi: a Neuromorphic Manycore Processor with On-Chip Learning,” IEEE Micro, vol. 38, no. 1, 2018

生物神经网络从根本上说是异步的，因此将来的芯片使用异步电路也是很自然的选择。

我设计这样一个异步的基础芯片是为将来设计更加复杂的异步芯片打下基础。

如果这篇报告将来被学弟看到，可能下面的备注能够帮助在很短的时间里面学会一个 CPU 设计方案：
简单的实现教程参考我的 github 仓库中的<教程.pdf>，有关于如何实现异步芯片和为什么这样实现的介绍和阐述。
如果没有相关需要，此备注请忽略，谢谢。