毕 业 论 文

题　　目　基于深度学习的数据预取质量优化算法设计与实现

姓　　名　李芳达

学　　号　13070038

指导教师　蔡旻

日　　期　2018.5.30

北京工业大学

毕业设计（论文）任务书

题目 基于深度学习的数据预取质量优化算法设计与实现

专业 计算机科学与技术 学号 13070038 姓名 李芳达

主要内容、基本要求、主要参考资料等：

主要内容：

（1）了解数据预取的基本原理、算法实现和相关工作。

（2）了解深度学习的基本原理、算法实现和相关工作。

（3）熟悉多核体系结构模拟器的使用与扩展编程。

（4）在多核体系结构模拟器中实现基于深度学习的数据预取质量优化算法，并对其性能与功耗进行分析比较。

基本要求：

1．参与本课题的同学将根据用户需求，进行系统分析、系统设计、系统实现。

2．系统分析、设计、实现过程应遵循系统开发规范。

3．课题进行期间，每周保证不少于 40 学时从事课题研究工作；每周至少一次到校汇报课题进度及接受指导。

4．课题结束应整理出系统相应文档。

参考文献：

[1] Babak Falsafi and Thomas F. Wenisch, “A Primer on Hardware Prefetching,” in Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2014.

[2] Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman, and Bingjun Xiao, “Customizable Computing,” in Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2015.

完成期限：2018 年 06 月 10 日

指导教师签章：

专业负责人签章：

2018 年 02 月 01 日

独 创 性 声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京工业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签名：_____ 日期：_____

关于论文使用授权的说明

本人完全了解北京工业大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签名：_____ 导师签名：_____ 日期：_____

摘要

针对数据预取的优化旨在减少存储访问延迟产生的时间消耗。通过把即将被处理器访问的数据提前从主存移动到 cache，预取可以有效降低存储访问的延迟。现代处理器配有多个硬件预取器，每个预取器针对特定的存储层次，并且使用各自独立的预取算法。但是，为了使不同程序的运行性能达到最大，需要采用不同的预取器子集。启用所有预取器很难产生最佳的性能结果，并且在某种情况下，预取甚至会降低性能。

在本篇文章中，我们讨论了单线程代码的预取效果，使用硬件构建语言 Chisel 实现了带有 LFU 算法和 MSI 一致性协议的 cache 硬件模块，以及两种典型的 NextLine 预取器和 Stride 预取器，之后提出了一种利用感知机神经网络优化 Stride 预取器预测结果的设计方法，从而改善数据预取质量。使用 Chisel 生成了上述程序的 Verilog 代码，并在 vivado 中进行综合，进而得出了这些模块的硬件成本和功耗。结果表明，可以用较少的硬件资源和较低的功耗实现本系统。

关键词 数据预取 深度学习 多预取器控制

Abstract

Optimizations for data prefetching are designed to reduce the time consumed by storage access delays. By pre-fetching the data to be accessed by the processor from main memory to cache, prefetching can effectively reduce the latency of memory access. Modern processors are equipped with multiple hardware prefetchers, each prefetcher for a specific storage hierarchy, and use separate prefetching algorithms. However, in order to maximize the performance of different programs, different prefetcher subsets need to be used. Enabling all prefetchers is difficult to produce the best performance results, and in some cases, prefetching can even degrade performance.

In this article, we discussed the prefetching effects of single-threaded code. Using the hardware construction language Chisel, we implemented a cache hardware module with LFU algorithm and MSI consistency protocol, and two typical Next Line constant prefetchers and stride. The predictive pre-fetcher, afterwards, presents a design method that uses a perceptron neural network to optimize the predictor prefetcher's prediction results. Ideally, the results of this experiment will enable pre-fetching to achieve a certain speedup. Using Chisel to generate the Verilog code for the above programs and synthesizing them in vivado, the hardware costs and power consumption of these modules are derived. The results show that the system can be implemented with less hardware resources and lower power consumption.

Keywords： Data Prefetch, Deep Learning, Multiple Prefetcher Control

目录

1.绪论

1.1 课题背景及意义

为了降低处理器和内存之间因访问速度差距过大造成的延迟，在计算机系统中加入 CPU 高速缓存（CPU cache，下文简称 cache），使处理器访问数据的速度接近处理器本身的频率。同时，现代处理器还会配有多个硬件预取器，每个预取器针对特定的存储层次，并且使用各自独立的预取算法。预取通过监视并推断流访问模式，将数据超前预取到更高层次的缓存中来降低内存延迟。预取在现在的体系结构中是一种关键的技术转型，决定优化预取的参数存在多个挑战[1]。第一，预取器必须精确地预测存取模式。如果预测错误，就会增加存储访问负担，并且更重要的是，会在容量小且昂贵的缓存中造成冲突。第二，预取指令必须及时。如果预取造成数据早于需要之前被放置到更高层缓存中，可能会被那些更紧迫需要的数据覆盖掉。这些挑战在多线程程序中被更进一步地放大。L2 等更低层次的缓存可以被多个线程共享，每个线程可能需要不同位置的数据，准确地决定出读取顺序是一件困难的事情。

在本文中，我们设计并实现了一种基于深度学习技术的有效的预取策略。

1.2 CPU 高速缓存

1.2.1 结构和参数

在计算机系统中主要采用组相联结构。组相联缓存把缓存空间分为多个组，每组包含若干缓存块。通过建立内存数据和组索引的对应关系，一个内存块可以被载入到对应组内的任意缓存块上。本文中所使用的组相联缓存均表述为公式(1.1)。

$$\mathrm{C} = \mathrm{B} * \mathrm{N} * E_n \qquad (1.1)$$

其中，C 为缓存容量，B 为每个数据块的大小，N 为相联度（每组中有 N 个数据块），E_n为组数。当使用组相联时，在通过索引定位到对应组之后，必须进一步地与所有缓存块的标签值进行匹配，以确定查找是否命中。

1.2.2 性能指标

本文中对 cache 的主要性能评价指标有加速比和 cache 的命中率。加速比S_p一般表示为公式(1.2)。

$$S_p = \frac{1}{H_c\frac{T_c}{T_m}+(1-H_c)} \quad (1.2)$$

其中，H_c为 cache 的命中率，T_c为 cache 的访问周期，T_m为主存储器的访问周期。可推断出，当$H_c \to 1$时，$S_p \to \frac{T_m}{T_c}$。研究表明[2]，H_c的大小受到 cache 的预取算法影响，本文即着重于通过深度学习优化预取算法来提高 cache 性能。

1.2.3 替换算法

对于组相联缓存，当一个组的全部缓存块都被占满后，如果再次发生缓存失效，就必须选择一个缓存块来替换掉。存在多种算法决定哪个块被替换。

最简单的替换算法是随机法（Rand 法），即随机决定被替换的缓存块。而先进先出（FIFO）法替换掉进入组内时间最长的缓存块。这种方法虽然考虑了程序运行的历史状况，但无法正确地反映程序的局部性。最近最少使用法（LRU 算法）则跟踪各个缓存块的使用状况，并根据统计比较出哪个块已经最长时间未被访问。这种方法反映程序局部性规律，因为最近最少使用的块，很可能在将来的近期也很少使用，因此 LRU 算法的命中率比较高。但是这种方法比较复杂，硬件实现比较困难，对于 2 路以上相联，这个算法的时间代价会非常高[3]。

本文使用与 LRU 法技术思想相同的最久没有使用法（LFU），其实现方法为记录近期使用次数的多少，然后替换最少的那一个。

1.3 Cache 数据预取

1.3.1 背景及原理

尽管 cache 层级技术的应用有效地减少了那些最常用数据的访问延迟，在科学计算程序中花费超过一半的时间用于内存请求仍不少见[4]。大型且密集的矩阵操作是许多科学计算程序的基础，而这些操作往往使得 cache 的利用效率低下。处理器在发现 cache 缺失后必须等待 cache 访问内存获取数据，然后继续进行运算。这种数据获取策略使每一个首次访问的数据块都会成为一次缓存缺失（即强制失效）。如果被访问的数据是一个大型数组操作的一小部分，它很有可能在之后被替换出 cache，为数组后续的数据成员进入 cache 腾出空间。当同样的数据块再次被需要时，处理器必须重新将其从内存中提取出来，产生更高的访问延迟（即容量失效）。

因此，如果能在处理器还未用到某个数据块之前就提前将其放入 cache 中，便能进一步提高 cache 的命中率。这种操作与处理器运算同时进行，使得数据在处理器需要时刚好到达了 cache 中。这样既利用了空间局部性，又能覆盖传输延迟，这种技术即为 cache 的预取（Prefetch）。本文将讨论如何通过优化预取算法达到提高程序运行效率的目的。

1.3.2 预取技术类型

现代计算机使用的预取主要分为两类，一是软件预取，二是硬件预取。软件预取多由编译器进行。指令集会提供预取指令供编译器优化时使用。编译器则负责分析代码，并把预取指令适当地插入其中。这类指令直接把目标预取数据载入缓存。本文使用硬件预取进行优化，因此将着重讨论硬件预取技术的特点和可优化空间。

硬件预取在 cache 旁添加支持元件，可以实时动态地进行预取，并且不需要编译器介入。典型的硬件指令预取会在缓存因失效从内存载入一个块的同时，把该块之后紧邻的一个块也传输过来。第二个块不会直接进入缓存，而是被排入指令流缓冲器（Instruction Stream Buffer）中。之后，当第二个内存访问指令到来时，会并行尝试从缓存和流缓冲器中读取。如果该数据恰好在流缓冲器中，则取消缓存访问指令，并将返回流缓冲器中的数据。同时，发起起一次新的预取。如果数据并不在流缓冲器中，则需要将缓冲器清空。

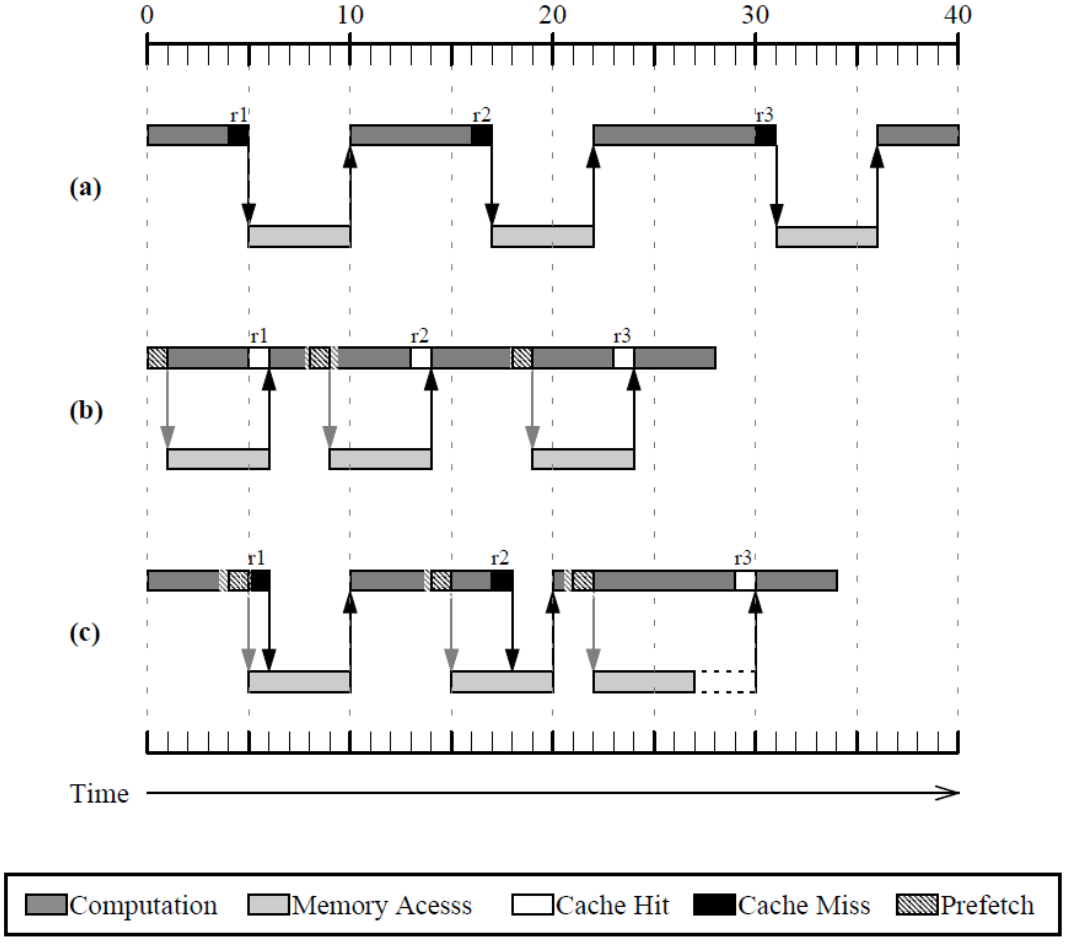

图 1.1 在 (a)不进行预取(b)完美预取(c)退化的预取三种情况下程序运行消耗的时间

在数据预取方面，图 1.1 展示了不同情况下典型的程序运行情况。当不使用预取时（图 1.1(a)），处理器在访问 r1, r2, r3 时发生了强制失效，并需要停止运算以等到相应的 cache 将数据从内存中获取之后才能继续执行运算。为了解决这个问题，一种越来越普遍的数据预取启动方式是由处理器明确发出 `fetch` 指令。一个 `fetch` 指令会最低限度指定被放入 cache 的数据块的地址。当指令执行时，此地址会直接传递给内存系统，而不会让处理器停滞并等待应答。Cache 以类似应答 `load` 的方式回应 `fetch`，但并不会在数据块进入 cache 后传入处理器。图 1.1(b)展示了这种方法如何通过并行执行访存和处理器运算从而隐藏内了存访问延迟。这种理想情况下，数据刚好在其被处理器需要之前被预取到 cache 中。图 1.1(c)则展示了一种不乐观的情况，在本图中对 r1 和 r2 的预取执行过晚，使处理器仍需要停止运算以等待数据到来，但预取操作的确减少了处理器等待的时间。而 r3 的预取则执行过早，这块数据将暴露在 cache 的替换候选之中，如果 r3 在被处理器引用前被替换掉，则需要重新进行访存操作。

其他几种硬件预取技术不需要使用 `fetch` 指令，这些技术使用特殊硬件对处理器进行监测来做出预取选择。尽管硬件预取不会造成指令上的额外负担，但它们往往都产生比软件预取更多的无效预取，从而产生更多缓存污染消耗内存带宽[5]。

1.3.3 优化目标

在使用预取技术时，必须妥善考虑进行时机和实施强度，并且仅产生少量的负担。如果过早地进行预取，则有可能在预取数据被用到之前就已经因为冲突置换被清除。如果预取得太多或太频繁，则预取数据有可能将那些更加确实地会被用到的数据取代出 cache。

cache 预取技术的优劣可以由三个指标评价[3]，一是覆盖率，表示因预取所减少的缺失数占总 cache 缺失数的比例，可表示为公式(1.3)。

$$\mathrm{Cov} = \frac{M_p}{M_c} \tag{1.3}$$

其中，Cov 为覆盖率，M_p为因预取所减少的缺失数，M_c为总 cache 缺失数。

二是准确率，表示有效预取所占比例，可表示为公式(1.4)。

$$\mathrm{Acc} = \frac{M_p}{P_{useless}+M_p} \tag{1.4}$$

其中，Acc 为准确率，$P_{useless}$为无效的预取。

三是及时性，及时性的定义为块预取的时间相对于块被使用的时间提早了多少。

1.4 深度学习

1.4.1 基本原理

深度学习是机器学习中一种基于对数据进行表征学习的算法，其基础是机器学习中的分散表示（distributed representation）。分散表示意为假定观测值是由不同因子相互作用而生成。在此基础上，深度学习进一步假定这一相互作用的过程可分为多个层次，代表对观测值的多层抽象。不同的层数和层次的规模可用于不同程度的抽象。

在深度学习方法中，更高层次的概念从低层次的概念学习得到。这一分层结构常常使用贪婪算法逐层构建而成，并从中选取有助于机器学习的更有效的特征。深度学习多使用非监督式，或半监督式的特征学习和分层特征提取高效算法来替代手工获取特征，成为其优于其他算法的一大特点[6]。

1.4.2 深度神经网络

本文使用的深度神经网络（deep neural network，DNN）是一种在输入与输出之间有多个隐藏层次的人工神经网络（artificial neural network，ANN）。深度神经网络能够为复杂的非线性关系提供建模，当对象表达为多层次基本数据类型时 DNN 构架生成复合模型，多出的层次可以从更低层次组合特征，因此与相似的浅层网络相比可以使用更少的单元达成对复杂数据的建模[7]。

深度结构在几种基础方法上有许多变种，每种结构在不同特定领域都有着显著的成效。不同的结构之间很难直接比较性能优劣，除非使用相同的数据集进行评估。深度神经网络一般是前馈网络，数据流从输入层传向输出层而不进行传回。主要的类型有递归神经网络（recurrent neural network，RNN），数据可以流向任何方向，常使用于语言建模中。卷积神经网络（convolutional neural network，CNN）用于计算机图像处理，目前也应用于声学模型以进行自动语音识别[8]。

1.4.3 感知机

感知机（perceptron）是一种人工神经元，由 Frank Rosenblatt 发明于 1950 至 1960 年代[9]。它的工作方式可以表述为，假设一个最简单的感知机，仅由一个神经元组成。它的输入是 n 个二进制x_1、x_2、x_3…，输出一位二进制，如图 1.2 所示。

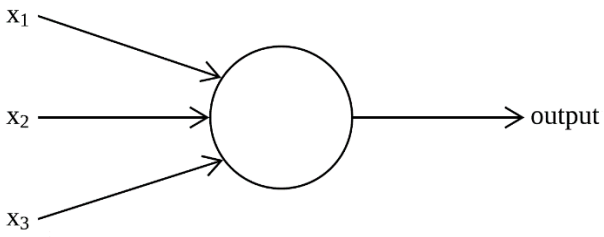

图 1.2 神经元示意图

在图 1.2 的神经元中有三个输入分别为x_1、 x_2、 x_3。Rosenblatt 提出了一种计算输出的简单规则，即引入权重（weight）的概念，分别表示为w_1、w_2、w_3，用来表示各个输入对于输出的重要程度。神经元的输出是 0 或是 1，取决于加权和$\sum_j w_j x_j$是否小于或者大于某一个阈值（threshold value）。此阈值也是感知机的一个实数参数。用代数形式则可以表示为公式(1.5)。

$$\text{output} = \begin{cases} 0 \ \ if \sum_j w_j x_j \leq threshold \\ 1 \ \ if \sum_j w_j x_j > threshold \end{cases} \tag{1.5}$$

通过调整权重和阈值的大小，可以得到不同的决策模型。权重越大，则表示本输入对决策结果的影响越大。一个简单的感知机虽然无法完成更加复杂的决策系统，但由其构成的网络则能够作出更加精细的决策，如图 1.3。

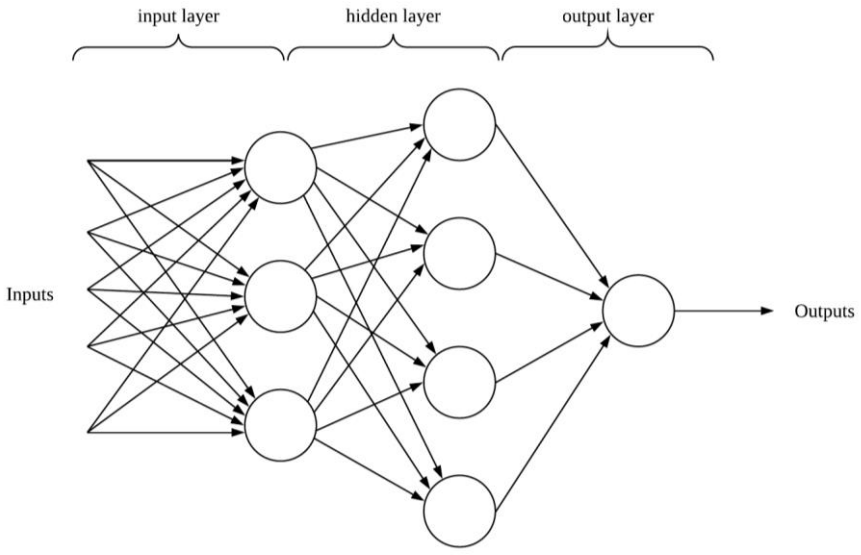

图 1.3 一个多层感知机的示意图

在图 3.1 展示的感知机中，共有三个层，左边为输入层，包含感知机的输入数据。中间为隐藏层，将输入层的输出作为输入，然后把输出传给输出层。右边为输出层，包含感知机的输出数据。

1.5 FPGA

1.5.1 FPGA 简述

FPGA 为 Field Programmable Gate Array 的缩写，即现场可编程逻辑阵列。是在原有的可编程逻辑器件的基础上发展而来的。以硬件描述语言描述的逻辑电路，可以利用逻辑综合和布局、布线工具软件，快速烧录到 FPGA 上进行测试，这一过程是现代集成电路设计验证的技术主流。它是作为专用集成电路（ASIC）领域中的一种半定制电路而出现的，可以实现任何 ASIC 上的逻辑功能，并且一次性工程费用很低（但元件费用更高）。尽管 FPGA 的速度要慢，无法完成更复杂的设计并且耗电量更大，但其具有高度的灵活性，内部逻辑可以被反复修改从而大幅降低了除错成本，既解决了全定制电路的不足，又克服了原有可编程逻辑器件门电路数有限的缺点[10]。

1.5.2 业界现状

目前世界上的两大厂商 Altera 和 Xilinx 占有将近 90%的市场，它们均成立于上个世纪 80 年代。Altera 在 1984 年推出了业界第一款可重复编程逻辑硬件 EP300。Xilinx 的联合创始 Ross Freeman 和 Bernard Vonderschmitt 在 1985 年发明了首个可商业化 FPGA——XC2064。目前 Xilinx 和 Altera 都提供 windows 和 Linux 平台的设计软件（ISE/Vivado 和 Quartus），设计者可以利用这些软件设计、分析、仿真和综合（编译）他们的应用。

1.6 硬件描述语言

本文在使用 FPGA 作为实验平台的基础上，选择硬件描述语言进行编程。在对目前多种语言进行比较后决定使用由加州大学伯克利分校开发的开源语言 Chisel3，并在下文阐述原因及其特点。

1.6.1 Verilog

Verilog 是电气电子工程师学会（IEEE）的 1364 号标准，它主要用于设计和验证抽象化的寄存器传输级的数字电路，也用于模拟电路和混合信号电路，以及生物合成电路[11]。Verilog 的基本语法和 C 语言相近，因此对熟悉 C 语言的设计人员来说可以很快掌握。

使用 Verilog 进行程序设计的基本思路是将复杂的电路划分为多个模块（module），模块作为提供简单功能的基本结构。工程师可采用自顶向下的思路进行模块分层、划分。

1.6.2 SystemVerilog

SystemVerilog 是一种由 Verilog 发展而来的硬件描述、硬件验证统一语言，前一部分基本上是 2005 年版 Verilog 的扩展，而后一部分功能验证特性则是一门面向对象程序设计语言。面向对象特性很好地弥补了传统 Verilog 在芯片验证领域的缺陷，改善了代码可重用性，同时可以让验证工程师在比寄存器传输级更高的抽象级别，以事务而非单个信号作为监测对象，这些都大大提高了验证平台搭建的效率[12]。

相较于 Verilog，SystemVerilog 定义了两种数据生存周期：静态和自动，添加了几种新的数据类型，添加了三种新的程序块类型，新增 interface 类型改善了 Verilog 原有 port 类型在多层次电路中大型模块间连接过于复杂时难以管理的问题。在硬件验证方面，SystemVerilog 拥有的多种功能一般用于协助创建扩展、灵活的 test bench 而非综合。它包含一些新的数据类型、支持面向对象程序模型等等[13]。

1.6.3 Chisel3

Chisel 是 Constructing Hardware In a Scala Embedded Language 的简称，它是嵌入在高级编程语言 Scala 中的硬件构建语言。换言之，Chisel 是遵循 Scala 使用规则的一系列特殊类定义、以及预定义的对象，设计者相当于使用 Scala 语言进行硬件图构建。Chisel 的目前已经更新到了第 3 个版本号，其主要具备的特征有：

- 抽象数据类型和接口；
- 面向对象编程和函数构建；
- 使用高度参数化的元编程（metaprogramming）；
- 自动生成可以在标准 ASIC 或 FPGA 上使用的 Verilog 程序。

例如，设计一个比较器，输入 2 个宽度为 8 的无符号整型数据，输出较大的结果，则可编写 Chisel 程序如图 1.4 所示。

```
    class Max2 extends Module {
  val io = IO(new Bundle {
    val in0 = Input(UInt(8.W))
    val in1 = Input(UInt(8.W))
    val out = Output(UInt(8.W))
  })
  io.out := Mux(io.in0 > io.in1, io.in0, io.in1)
}
```

图 1.4 Chisel3 示例程序

编译后在指定文件夹中输出的自动生成 Verilog 代码如图 1.5 所示。

```
  circuit Max2 : @[:@2.0]
module Max2 : @[:@3.2]
  input clock : Clock @[:@4.4]
  input reset : UInt<1> @[:@5.4]
  input io_in0 : UInt<8> @[:@6.4]
  input io_in1 : UInt<8> @[:@6.4]
  output io_out : UInt<8> @[:@6.4]

  node _T_11 = gt(io_in0, io_in1) @[Max2.scala 17:24:@8.4]
  node _T_12 = mux(_T_11, io_in0, io_in1) @[Max2.scala 17:16:@9.4]
  io_out <= _T_12
```

图 1.5 Chisel3 自动生成的 Verilog 代码

显然，Chisel 程序代码更加符合使用者的设计和阅读习惯，且直观易懂。另外 Chisel 提供的预定义类和函数也简化了编写过程，一些 Verilog 固定输入信号如 clock，reset 等不必再手动定义。

1.7 本章小结

本章我们讨论了本课题的技术背景，在存储墙造成的问题日益加剧的情况下[14]，如何通过有效的方法隐藏或者降低 CPU 和内存之间的数据传输速度延迟已变得十分重要。之后，本章介绍了 cache 及预取器在解决这方面问题使用的原理，以及本文为改善此问题将会使用到的人工智能算法、硬件平台和编程语言的简单介绍。

2.总体设计

2.1 需求分析

在前文中已经探讨过，缓存预取有可能导致性能降低，并造成许多不良后果。这说明预取仍存在很多优化空间，有许多研究已经在相关方面做出了进展。Cavazos 等人[15]发明了一种机器学习模式，能够找出 SPEC CPU2006 标准程序组的最佳优化配置，使程序得到性能提升。这证明了使用人工智能进行性能优化的可行性。但他们的工作目的是找出一组最优化编译，而本文将更注重于硬件优化。McCurdy 等人[16]使用性能事件描述了预取对系统应用程序的影响。他们试验了 AMD 处理器上的多种标准程序的组合。Liao 等人[17]提出了一种基于机器学习的方法，用于选择优化预取配置。这两项研究都为基于人工智能的硬件预取优化提供了思路。

受此启发，本文使用了深度神经网络的方法进行优化。考虑到添加额外的硬件成本可能与提升的性能相比并不划算，本文选择使用最基础的神经网络模型：感知机来预测未来预取器将要预取的内容。感知机模型的神经网络易于理解且得出的输出均由明显可见的权重计算而来，方便进行测试。

本文将构建一个精简的处理器-缓存-内存系统并展开实验。实验平台选择 FPGA 以利用其可以随时修改，重新烧录的高度灵活性特征。本实验中使用的 Zynq-7000 还集成有双核 AMD Cortex-A9 处理器[18]，可以在进行 FPGA 实验的同时直接输出处理结果。由于本系统采用多层次多模块设计，使用传统 Verilog 语言可能产生大量重复性工作，并因大量数据定义使得查错工作难以进行，这在上文 1.6.2 节中已经讨论过。因此实验决定采用高级程序设计语言 Scala 的硬件构建工具 Chisel3 进行编程，这样既减轻了设计难度，也提升了程序可读性。

2.2 系统模块设计

本实验的目的旨在通过优化预取器的预取行为达到提升程序性能，缩短运行时间的目的，可以看做提升 cache 的各项性能指标，因此系统设计集中于处理器-缓存-内存部分，不考虑其他因素和元件的影响。

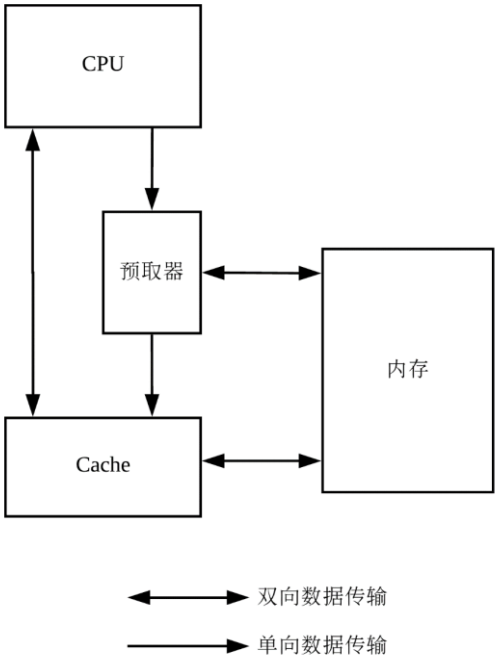

图 2.1 系统模块设计

如图 2.1 所示，本系统主要分为四个大模块：处理器模块、cache 模块、预取器模块和内存模块。

2.3 开发环境

本实验开发时使用的操作系统为 8GB 内存 64 位 Ubuntu16.04，软件开发工具为 IntelliJ IDEA，Chisel 版本 3.1.0，Scala 版本 2.11.12，sbt 版本 1.1.1。FPGA 综合及布线工具 Vivado。

实验平台的 FPGA 为 Xilinx 生产的 Zynq-7000 SoC。SoC 的特点是同时具有可编程的硬件 FPGA 和可编程软件的处理器系统两种功能。Zynq-7000[19]配备有 ARM Cortex-A9 双核处理器，并集成了基于 28nm Artix-7 或 Kintex®-7 的 FPGA 功能，有着极佳的性能/功耗以及灵活性优势。

2.4 测试方法

在 Chisel3 中提供了一套强大的测试机制，可用来进行电路的单元测试和系统测试[20]。测试器被绑定到一个模块中，然后使用者可以通过调用测试器的方法来测试电路。其中最常用的测试工具有以下四种：

- 指数操作（poke）：可以在输入端口和状态电路上放置数值；
- 单步操作（step）：可以让电路往前运行一个时钟单位；

- 窥探操作（peek）：可以窥探端口和状态电路的数值；
- 期待操作（expect）：可以用期望的参数来比较读取的电路数值。

假设要测试一个带有输入 A、B 和输出 C 的电路模块，则可以使用 poke 方法，对 A、B 赋予适当的输入数值，然后告诉仿真把这些数值放在正在测试的设备的输入上，并使用 step 向前运行电路一个时钟周期，接着使用 peek 或 expect 方法检查输出 C 是否是对应的已知数值，这样便完成了对此模块的功能测试。单步运行将更新寄存器以及寄存器驱动的组合电路，因此置数操作往往连接着 step 操作。但是，对于纯组合电路而言，置数操作本身就足够更新所有与被置数的输入相连接的组合电路。

最后，通过调用函数 runPeekPokeTester 来激活一个测试器。如图 2.2 所示。

```
  def main(args: Array[String]): Uint = {
 runPeekPokeTester(() => new MyModule()){
   (a,b) => new Tests(a,b)}
}
```

图 2.2 Tester 函数示例

2.5 参数设计

本系统中定义了 Params 接口负责指定所有相关参数，具体数据结构和功能如表 2.1 所示。

表 2.1 Params 接口设计

数据名称	数据类型	描述
addrWidth	UInt	内存地址宽度
memSize	Int（单位：MB）	内存容量
cacheSize	Int（单位：kB）	Cache 容量
blockSize	UInt	缓存块大小
assoc	UInt	Cache 相联度
numSets	UInt	Cache 组数

2.6 传输协议设计

在本系统使用的 zynq-7000 设计实验中，需要实现 PS-PL 互动，因此采用了与 AXI4 协议思想相同的数据传输办法。本文仿照常用于控制传输指令方面的 AXI4-Lite 进行设计和实现。AXI4-Lite 的优势在于，它是一个轻量级的地址映射单次传输接口，并且占用较少的逻辑单元。

2.6.1 握手机制

与 AXI4-Lite 相同，本文在系统模块之间通信时分为主、从两端，两者之间可以进行连续通信。我们所使用的是一种 READY-VALID 握手通信机制，在主从模块进行数据通信之前，需要根据不同情况对用到的数据、地址通道进行握手。具体的操作为发送方 A 等待接收方 B 的 READY 信号后，A 将 VALID 信号和数据同时发送给 B。其时序图如图 2.3 所示，其中 ACLK 为总线时钟。

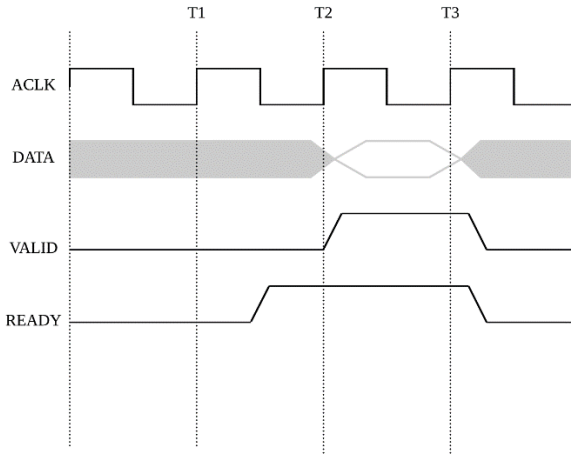

图 2.3 模块通信握手机制

基本上，本系统中的接口数据结构均为 Chisel 所提供的已经包装好的 DecoupledIO（包含 READY、VALID、DATA 信号）或 ValidIO（包含 VALID、DATA 信号）类型。

2.6.2 工作模式

以读操作为例，本文中的数据传输操作过程为：主从之间进行地址通道握手并传输地址内容，然后进行读数据通道握手并传输数据内容的回应，其过程如图 2.4 所示。

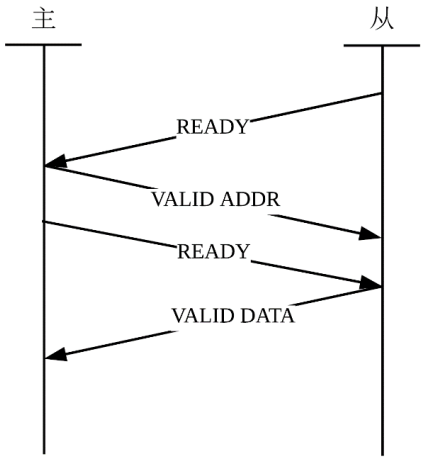

图 2.4 读操作过程

当发送方准备发送某一请求时，不会直接将请求发送给接收方，而是等待接收方的 READY 信号为真后，将请求和回应 READY 信号发给接收方。同样，接收方在收到请求后，会等待发送方的 READY 信号再传回应答数据。这种工作过程的目的是防止接收方在处理其他事务时发送方便将数据发出，导致接收方没有处理而丢失掉数据。

2.7 CPU 指令流生成功能

本系统中不包含具体的 CPU 硬件模块，但需要完成 CPU 对 cache 和预取器发出正常指令并接收 cache 应答的功能。因此我们定义了一套 CPU IO，由 Instruction Generation 代替 CPU 完成发送指令的功能，如图 2.5 所示。

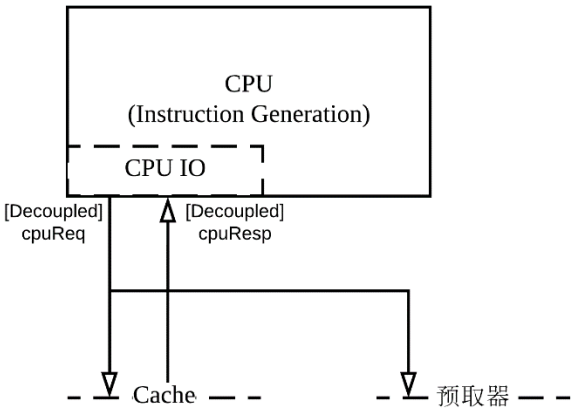

图 2.5 CPU 模拟功能设计

在单元测试程序中，一个假定的 CPU 通过 Instruction Generation 和 CPU IO

向 cache 发出读/写请求，并接收 cache 返回的数据应答，同时由预取器接收到发出的请求并进行预取判断。CPU IO 接口的具体数据结构如表 2.2、2.3、2.4 所示。

表 2.2 CPU IO 接口设计

接口名称	方向	数据类型	描述
cpuReq	输出	DecoupledIO	CPU 发送给 cache 的数据请求
cpuResp	输入	DecoupledIO	CPU 由 cache 接收的数据

表 2.3 cpuReq 具体结构

数据名称	方向	数据类型	描述
valid	输出	Bool	CPU 请求有效信号
ready	输入	Bool	Cache 接收有效信号
read	输出	Bool	请求是否为读数据
addr	输出	UInt[addrWidth]	请求数据的地址

表 2.4 cpuResp 具体结构

数据名称	方向	数据类型	描述
valid	输入	Bool	Cache 应答有效信号
ready	输出	Bool	CPU 接收有效信号
data	输入	UInt[blockSize]	Cache 返回的数据值

参数均由 2.5 节中的 Params 进行定义。

2.8 FPGA 实验流程概述

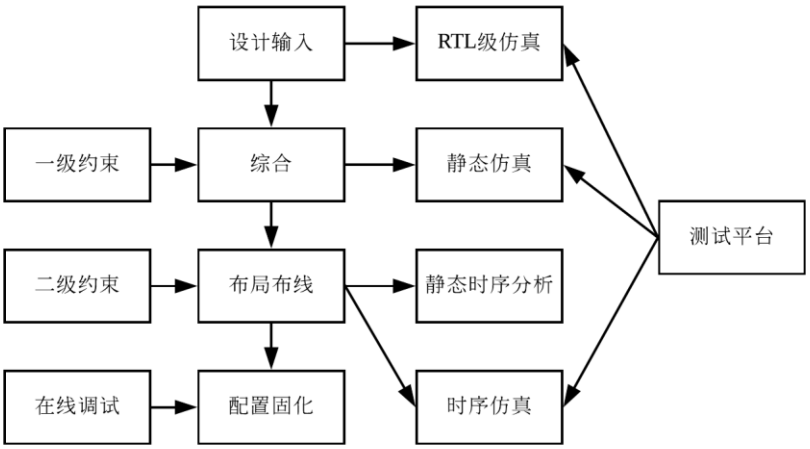

图 2.6 FPGA 实验流程图

如图 2.6 所示，FPGA 的开发流程可分为如下四个主要步骤：

1) 设计输入。传统的设计输入方式有 3 种：原理图、HDL 和 IP 核。目前原理图输入已经很少使用，常用的输入源有 Verilog 等硬件描述语言。本文使用硬件构建语言 Chisel 设计电路，其所具有的面向对象编程和自带测试仿真功能的特性使设计过程更为方便，并且设计输入和仿真阶段可以直接在 IDEA 中完成。Chisel 还可以自动生成所需 Verilog 或 VHDL 语言文件，可在进一步的综合过程中直接使用。

2) 综合。对设计输入进行综合，得到一个可以和 FPGA 硬件资源相匹配的描述。假设 FPGA 是基于 LUT 结构的，那么就得到一个基于 LUT 结构的门级网表。其中，一级约束即综合约束，用来指导综合过程，是小范围内实现运行速度和资源消耗平衡的一种方式。不同的约束，将会产生性能不同的电路。另外，静态仿真（或门级仿真）：是综合后 LUT 门级网表的仿真，目的是当工程用 LUT 门级描述时，从功能上验证工程的正确性。

3) 布局布线。布局考虑的问题是如何将这些逻辑上已连接的 LUT 及其他元素合理地放到现有的 FPGA 里，并且达到功能要求的同时保证质量。布线考虑的问题就是线路最优问题，具体来说就是如何让各部分连接起来，如何让输入输出信号到达相应的位置，且保证电路连接后的整体性能。其中，二级约束：即布局布线约束，可分为位置约束和时序约束。位置约束指布局策略，根据所选择的 FPGA 平台现有的硬件资源分布来决定布局。时序约束在很大程度上和布线有关，但是是先用软件默认的原则布线，然后对其结果进行静态时序分析，不满足

时序要求的，再对具体的问题路径做一些指导约束。另外，布线时时延问题的截获就可以通过时序仿真完成。将工程下载到 FPGA 芯片上，可通过在线调试（或板级调试）分析代码运行的情况。

4) 配置及固化。在配置模式和初始化模式下，FPGA 的用户 I/O 处于高阻态（或内部弱上拉状态），这两个模式相继结束后，进入用户模式，此时用户 I/O 就能够按照用户设计的功能工作。固化既是将程序固化到存储器中。

2.9 本章小结

本章首先论述了前人在相关工作方面所处的贡献和结论，以及在此基础上本文所实现的系统的设计思路。之后介绍了本文设计的系统的整体规划，以及一些全局控制参数、协议的详细设计。同时也介绍了针对系统的测试方法和硬件实验过程。

3.基于 Chisel 的存储模块设计与实现

3.1 Cache 模块设计

Cache 模块由三个主要部分构成：cache IO 、cache blocks、LFU 替换算法。Cache 与外部模块的数据传输以及内部模块关系如图 3.1 所示。

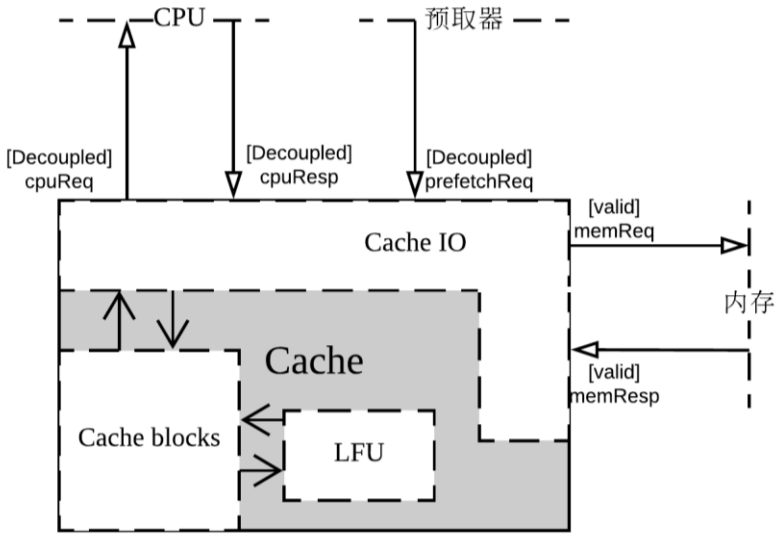

图 3.1 Cache 模块设计图

在与外部模块的数据传输上，cache 模块通过 cache IO 由 CPU 接收读/写指令并返回相应数据，向内存发送读/写指令并接收相应数据，同时接收预取器发来的 prefetch 指令。Cache IO 负责与 CPU、内存、预取器模块进行数据传输，具体数据结构设计如附表 1、2、3、4、5 所示。

为保持缓存一致性，在 cache blocks 模块中采用 MSI 一致性协议状态机维护每个缓存块的状态。Cache blocks 是 cache 的存储结构，cache 从内存获取的数据和缓存块的状态及目录标签均存储在 cache blocks 中。Cache 内各个子模块的详细设计将在下文叙述。

3.2 Cache blocks 模块设计

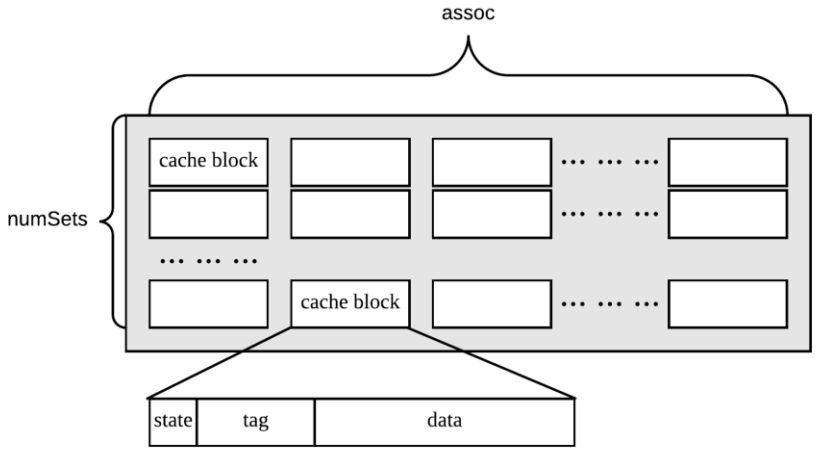

图 3.2 cache blocks 结构设计

Cache blocks 模块是 cache 的存储结构。本实验中采用了组相联结构，共有 numSets 个组、相联度为 assoc。其结构示意图如图 3.2 所示。其中每个缓存块包含状态、tag 和数据三个部分。具体数据结构如表 3.1 所示。

表 3.1 cache 存储结构设计

数据名称	数据类型	描述
cacheBlock ├ state ├ tag └ data	Module Reg UInt[tagWidth] UInt[blockSize]	缓存块模块 缓存块状态 缓存块查找标签 缓存块数据
blocks	Mem[numSets]*[assoc]	按组为单位定义的二维数组 cache blocks，组员数据类型为 data

每个缓存块的 state 由一个 MSI 一致性协议状态机进行状态维护。本协议包含三个稳态：

- I（Invalid） 该块目前不在缓存当中，或因总线的请求被标记为无效块。如果要对该块进行读/写操作，则需要从内存中获取该块。
- S（Shared） 该块在缓存中以只读状态存在，在进行替换操作时，可以直接替换而不进行数据写回操作。

- M（Modified） 该块在缓存中已因为 CPU 写操作被修改，缓存中的数据和内存中的数据不一致，在进行替换操作时必须将该块的数据写回内存。

另外，还包含 6 个暂态表示缓存块在稳态之间的过渡状态：IS（Invalid 转到 Shared 时的中间状态，下同）、SI、SM、MS、MI、IM。具体的状态转移图如图 3.3 所示。

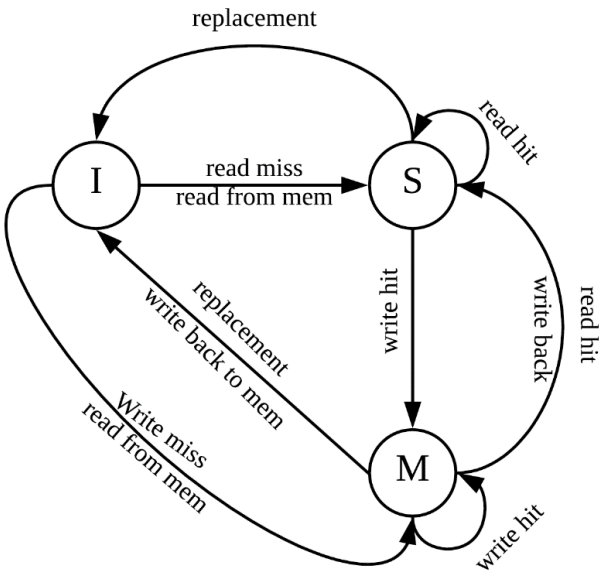

图 3.3 缓存一致性协议状态转移图

若 cache 收到了 CPU 的读请求，此时目标块处于“M”或者“S”状态，则可以直接提供数据给 CPU。如果块处于“I”状态，即尚未被装入缓存，则在进行装入操作之前，必须先保证改地址的数据在其他缓存块中不会处于“M”状态。否则必须把已被修改的数据写回内存并回到“S”或“I”状态。在数据写回后，缓存可以将该块装入并对 CPU 的读取请求进行应答。

3.3 LFU 替换算法设计

本系统使用与 LRU 算法技术思想相同的最久没有使用法（LFU），其实现方法是记录近期使用次数的多少。记录方法是为每个缓存块设置一个计数器，具体执行流程如下：

1) 被调入或者被替换的块，其计数器清零，其他计数器加 1；
2) 当访问命中时所有块的计数值与命中块的计数值进行比较，如果计数值小于命中块的技术值，则该块的计数值加 1。如果块的计数值大于命中块的计数值，则不进行变动。命中块的计数器清零；
3) 当出现缓存缺失时，选择计数值最大的缓存块进行替换。

在 cache 模块中，缓存块替换部分的执行流程可用伪代码表示为图 3.4 所示。

```
if hit :
  read hitblock
  call LFU(hit) //update counter
  send hitblock to CPU
  else
  victimID = LFU(miss)  //update counter
  newBlock = {read from mem}
  cacheBlocks(victimID) = newBlock
  send newBlock to CPU
```

图 3.4 LFU 替换执行流程

其中，LFU 的 hit 函数执行流程为，取命中缓存块的计数值，对本组缓存块的计数器进行历遍操作，将命中块的计数值和历遍块的计数值进行比较，如果历遍块的计数值小于等于命中块的计数值，则历遍块的计数值加一，否则不进行操作。最后，将命中块的计数值清零。

miss 函数执行流程为，首先历遍本组缓存块的计数器，找出数值最大的缓存块。然后将此块标记为替换块，将该块的计数值清零，其他块的计数值加一。

hit 函数和 miss 函数的详细过程表示在图 3.5、3.6 的伪代码中。

```
def hit(hitWay)
  hitCounterValue = counter(hitWay).value
  for each counter :
    if counter.value <= hitCounterValue :
      counter.value++
   else
     no action
    counter(hitWay).value = 0
```

图 3.5 LFU hit 函数设计

```
def miss
  for each counter :
    if counter.value > victimCounterValue :
     victimWay = counter.way
     victimCounterValue = counter.value
```

```
  else
    victimWay NO change
    victimCounterValue NO change
for each counter :
  if counter.way == victimWay :
    counter.value = 0
  else
    counter.value++
return victimWay
```

图 3.6 LFU miss 函数设计

3.4 内存模块设计

内存模块由两个部分组成：mem IO 负责与 cache 和预取器模块进行数据传输，mem blocks 为内存的存储结构。内存模块与外部模块之间的数据传输关系和内部子模块之间的结构如图 3.7 所示。

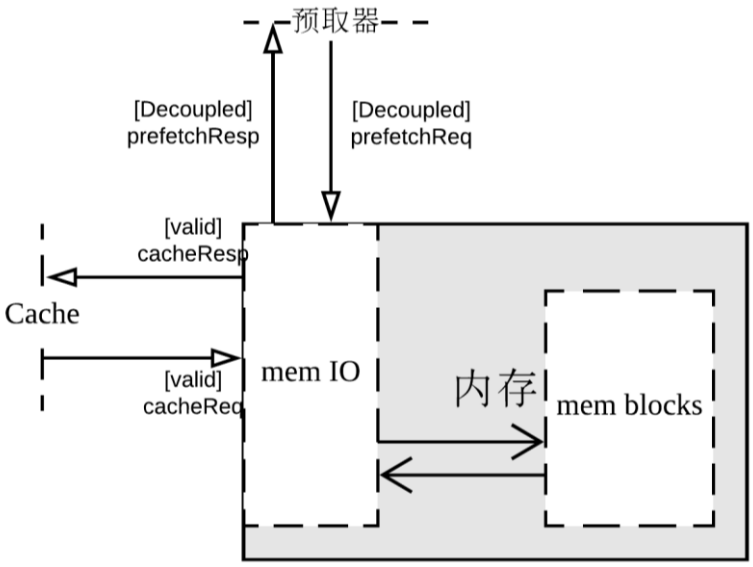

图 3.7 内存模块设计

Mem blocks 由 chisel 提供的 Mem 结构定义，大小为 memSize，每个数据块的宽度为 blockSize，与 cache 保持一致。参数均由上一章中的 Params 进行定义。

内存 IO 负责与外部模块进行数据传输，具体数据结构设计如附表 6、7、8、9、10 所示。

3.5 本章小结

本章介绍了系统中存储结构的具体设计方案和实现方法。其中 cache 主要分为三个字模块：cache IO、cache blocks、LFU 替换算法，并分别详细叙述了它们的设计与实现方案。最后，介绍了内存模块的数据传输接口和存储结构设计。

将 Chisel 自动生成的 Verilog 程序输入 vivado 中，可以进行电路连接和 zynq 的板上仿真测试，如图 3.8 所示。

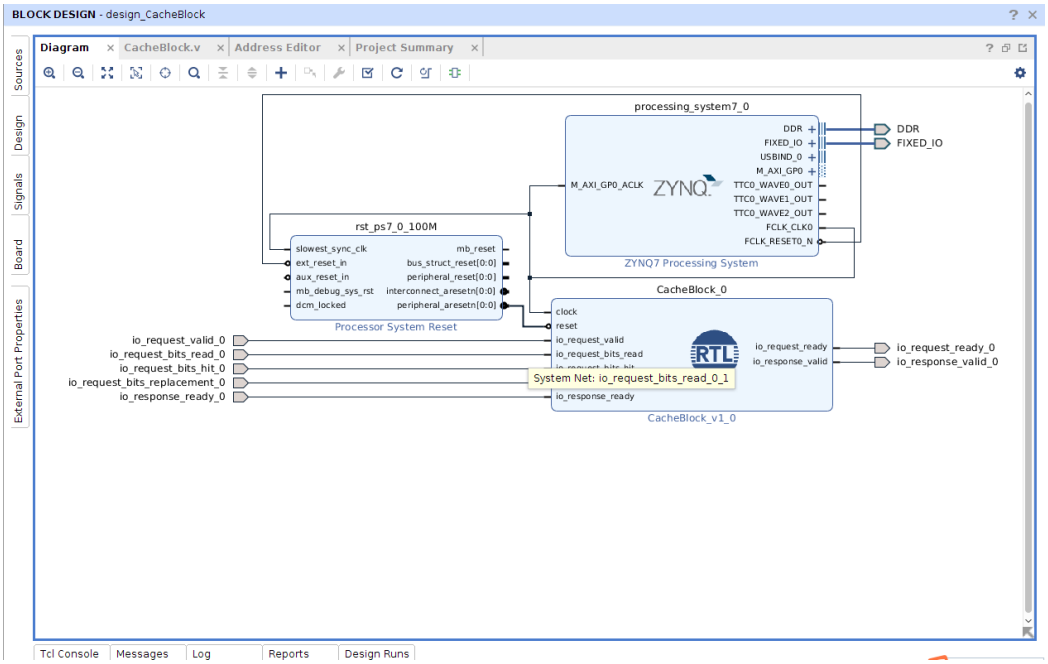

图 3.8 vivado 中 cache block 的 block 设计连接图

然后在生成结果中查看硬件资源和功耗相关数据，如图 3.9 所示。

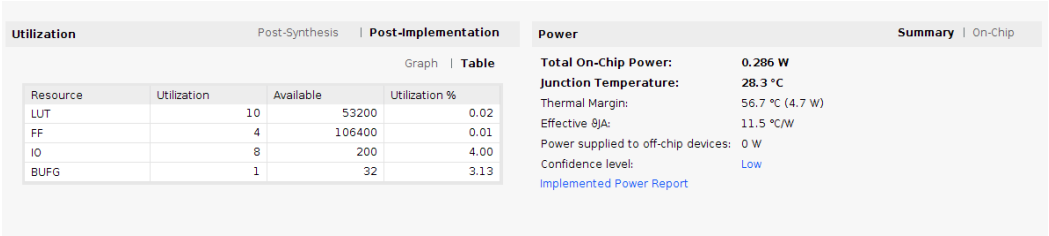

图 3.9 cache block 在 vivado 中的执行结果

利用以上数据便可以进行性能统计和对比等工作。

4.基于 Chisel 的预取器设计与实现

4.1 预取器 IO 设计

本系统中的预取器均使用相同的 IO 结构，具体设计如表 4.1 所示。

表 4.1 预取器总体 IO 设计

数据名称	方向	数据类型	描述
request ├valid ├pc └effectiveAddress	输入	ValidIO Bool UInt[addrWidth] UInt[addrWidth]	CPU 访问请求 CPU 访问有效信号 CPU 访问指令 CPU 访问地址
response ├valid └prefetchTarget	输出	ValidIO Bool UInt[addrWidth]	预取器应答 预取器应答有效信号 预取目标地址

作为预取器的输入信号，request 虽然是 CPU 发出的读/写请求指令，但因为预取器接收此请求仅作为监视和预测未来预取地址的作用，因此没有常规的对 CPU 应答信号。输出信号则是发给 cache 的预取指令，包含预取器所预测的目标地址信息。同样不用等待 cache 对此预取的应答。

4.2 Next Line Prefetcher 的设计与实现

在前文中我们已经详细叙述了本系统的 cache 设计方案，需要考虑到包含多个字的缓存块本身即为一种形式的数据预取，将连续的内存字打包为一个缓存块单位，cache 在这里利用了空间局部性的原理将被访问地址的数据字附近的数据一并预取到了缓存块中。

因此，一种最简单的预取器设计方案即是只要 CPU 进行了访问操作，不论 cache 是否命中，都发出预取命令，将被访问字所在块的下一块从内存预取到 cache 中。其工作流程可用图 4.1 的伪代码表示。

```
if CPU.request.valid == true :
  response.valid = true
  response. prefetchTarget = CPU.request.
effectiveAddress + addrWidth
  send response. prefetchTarget to cache
```

图 4.1 Next Line Prefetcher 工作模式

4.3 Stride Prefetcher 的设计与实现

连续块预取是一种简单有效的预取方式，但当程序出现大步长的数组引用时，这种方法便会造成无用预取。尽管同样符合空间局部性原理，这种访问模式需要新的方法来实现预取操作。

识别固定步长数组访问的最简单方法是在程序中明确声明将要执行此类操作，并将信息传递给硬件。当程序员设计了一系列向量和矩阵计算时便可以添加这种操作，而这在含有向量处理机的编程机中十分常用。预取器可以直接通过程序传递的这些信息计算需要预取的数据。但是，当程序员无法提供这类高级别的信息时，预取的优化则通过硬件监视处理器指令流或地址访问模式来达成。

本系统设计的 stride prefetcher 通过探测循环结构中存在持续的固定步长数组访问模式，来预测未来将要访问的数据地址。假设有内存访问指令m_i，在三个连续的循环迭代中访问地址a_1、a_2和a_3。当公式（4.1）的条件满足时，

$$(a_2 - a_1) = \Delta \neq 0 \tag{4.1}$$

将初始化针对m_i的预取。此时，假设Δ是后续一系列数组访问的步长值，可得下一个预取地址应该为公式（4.2）所得结果。

$$A_3 = a_2 + \Delta \tag{4.2}$$

注意A_3是实际下一次访问的地址a_3的假设值，若二者不相等则不再进行预取。否则预取器将持续按照此步长进行预取，直到$A_n = a_n$不再为真。

显然，将过去所有的内存访问历史进行记录是无法实现的。为了实现上述方法，将过去的访问地址存储在预取器中，需要在预取器中设置一个引用地址缓存表，用来存储最近被使用的几个内存访问指令，其设计如图 4.2 所示。

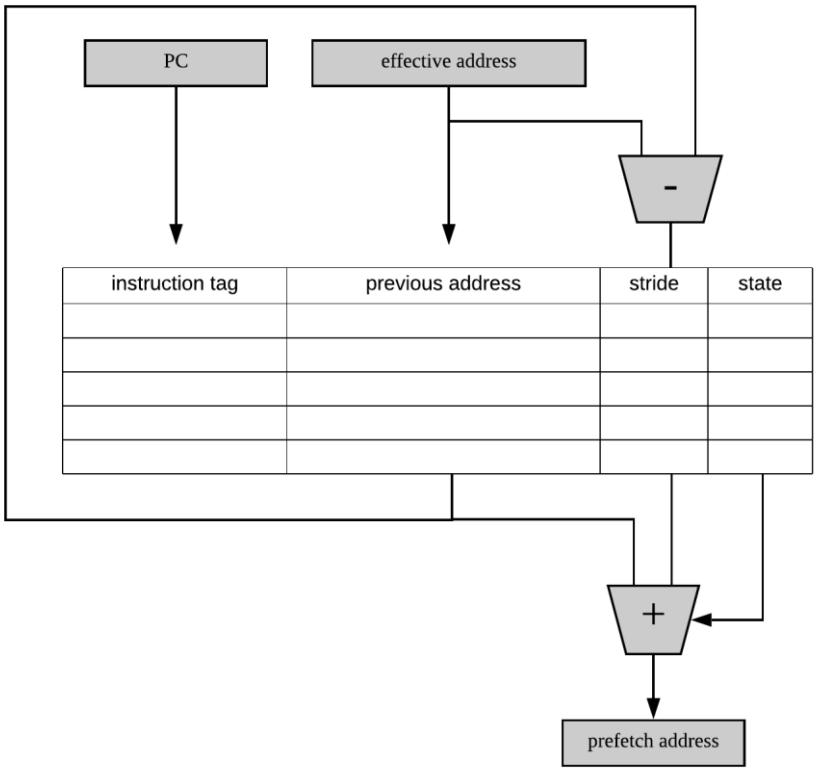

图 4.2 引用地址缓存表设计

在该表中，每个表项包含内存访问指令的地址（PC/instruction tag），此指令上一次访问的内存地址（previous address），计算出的步长值（stride）和记录表项状态的状态标签（state）。完整的状态转移图如图 4.3 所示。

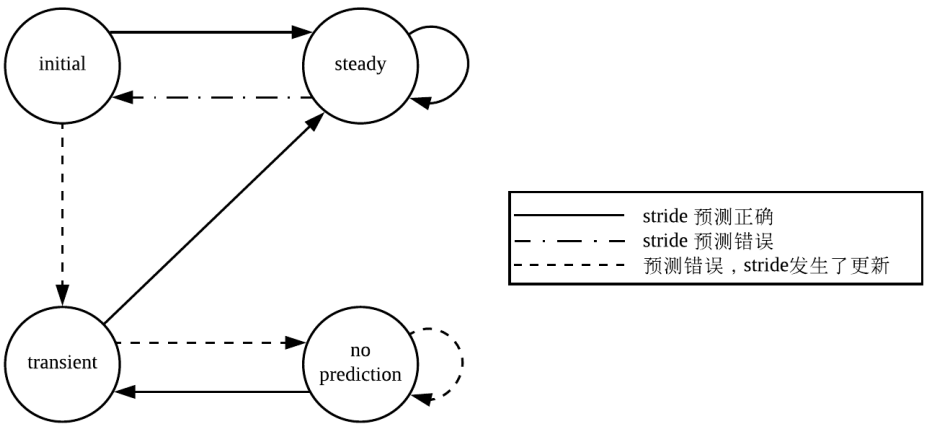

图 4.3 引用地址缓存表项的状态转移图

其中，initial 是表项的初始状态，此状态下不进行预取。 transient 状态为步长 stride 出现变化，此时进行试探性预取。 steady 状态为在持续的预取中步长

stride 保持不变且不为 0，此时将会一直进行预取直到状态发生变化，即发生了错误的步长预测。no prediction 状态为发生了多次不正确的预测，或预取器计算出的步长不断变化，意味着程序可能没有进行数组操作，此时不发出预取指令，但仍然进行缓存表的更新工作，直到发生了正确的步长预测。

4.4 基于感知机的 Stride Prefetcher 设计

在上节中，我们实现了传统 Stride Prefetcher 利用状态机维护预取表项，并决定是否进行预取的策略。为了引入深度学习优化预取的方法，可以考虑将状态机替换为一个输出预取步长的感知机神经网络，当输出为 0 时不进行预取，当输出不为 0 时进行相应步长的预取。

感知机的基本原理已经在前文中进行介绍，在本系统的设计中同样使用含有三个层的多层感知机神经网络，但输出不止一个。如图 4.4 所示，其中输入$x_1,\ldots,x_i$为预取器预测步长的历史记录，则x_i的大小均为字长的整数倍。输出$y_1,\ldots,y_m$代表不同步长值的预测概率，并且y_1代表的步长固定为0，表示不进行预取的情况。显然$\sum\{y_1,\ldots,y_m\}=1$。

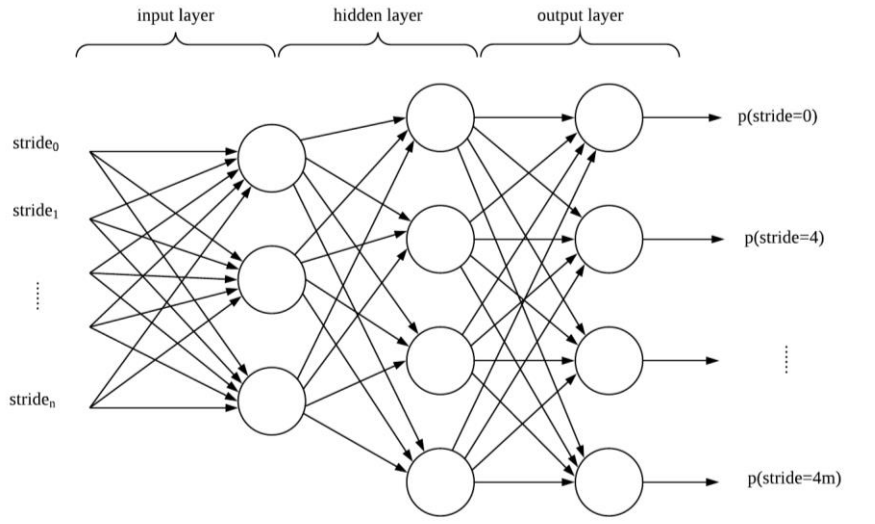

图 4.4 基于感知机的 stride 预取优化模型

为存储 stride 历史，我们设计了一个与 Stride Prefetcher 的预取表结构相似的记录表中，表按 PC 值进行索引，每一表项包含该指令的 PC 值、过去执行此指令时每一个产生的 stride 值以及最后一次执行此指令时访问的数据所在地址。其结构如图 4.5 所示。

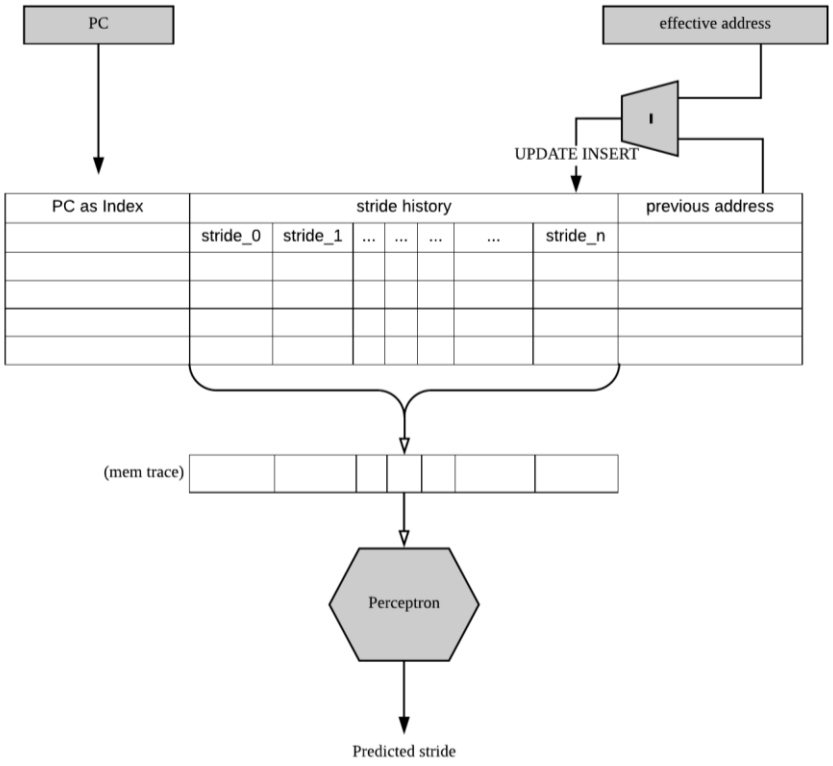

图 4.5 stride 记录缓存表结构设计

缓存表的更新步骤为：当 CPU 发出新的指令时，在表中建立新的表项并存储 PC 值作为索引，此时 stride history 中为空。随着程序继续运行，每一次访问都记录相应的 stride 值并放入缓存表中，直到 stride history 为满后，新产生的 stride 插入表的最后一个位置，然后将最早的 stride 记录值删除。

如果当前指令拥有完整的 stride history 表，便可以利用感知机进行预取判断。当 CPU 再一次发出相同指令时，将缓存表中 stride history 提取出来，作为输入放入感知机中，并得出本次预取的步长值，然后发出相应预取指令，同时将指令读取的数据地址与上一次的地址做差，用得出的步长更新记录表。

4.5 本章小结

本章我们首先定义了预取器使用的统一 IO 结构。然后用预取器中基本的恒预取机制实现了一个固定预取下一数据块的 NextLine Prefetcher。接着我们设计并实现了一种更为进阶的预取器：Stride Prefetcher。其利用引用地址缓存表和状态机预测 stride 值，从而实现在执行带有向量的循环迭代程序时，能够较为准确地预取下一访问数据的地址。之后在传统 Stride Prefetcher 的基础上，通过将原有的状态机替换为感知机，从而实现将人工智能技术嵌入预取器中并进一步提高预取的正确率。

结论

在本篇文章中，我们讨论了 cache 和预取等技术出现的历史背景，并考察了其他研究者在解决内存访问延迟等技术上给出的策略和结果。之后我们介绍了本篇文章中使用到的硬件基础、技术原理和工具。

在本文的系统设计与实现上，我们使用硬件构建语言 Chisel 实现了带有 LFU 算法和 MSI 一致性协议的 cache 硬件模块、内存存储结构。在下一章，我们设计并实现了两种典型的 NextLine 预取器和 Stride 预取器，之后提出了一种利用感知机神经网络优化 Stride 预取器预测结果的设计方法，从而改善数据预取质量。使用 Chisel 生成了上述程序的 Verilog 代码，并在 vivado 中进行综合，进而得出了这些模块的硬件成本和功耗。结果表明，可以用较少的硬件资源和较低的功耗实现本系统。

由于时间紧张，未能完成整个系统的集成工作，但保证了每个模块能够独立地正确工作。在之后的工作中，应该进行模块之间的连接测试工作、以及系统集成后的整体测试工作。为了进一步研究多预取器控制优化，还可以实现更多不同的预取器，并结合深度学习的优化方法进行测试。

参考文献

[1] Rahman, Saami; Burtscher, Martin; Zong, Ziliang; Qasem, Apan: “Maximizing Hardware Prefetch Effectiveness with Machine Learning.” Texas State University

[2] Solihin, Yan. “Fundamentals of parallel multicore architecture.” Boca Raton, FL: CRC Press, Taylor & Francis Group. 2016, p. 163.

[3] 方娟. 计算机系统结构. 清华大学出版社. 2011.3. ISBN 978-7-302-24395-3.

[4] Mowry, T.C., Lam, S. and Gupta, A., “Design and Evaluation of a Compiler Algorithm for Prefetching,” Proc. Fifth International Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, MA, Sept. 1992, p. 62-73.

[5] Steven, VanderWiel; David J, Lilja: “A Survey of Data Prefetching Techniques.” 1996.

[6] Song, H.A.; Lee, S. Y. “Hierarchical Representation Using NMF.” Neural Information Processing. Lectures Notes in Computer Sciences 8226. Springer Berlin Heidelberg. 2013: 466–473. ISBN 978-3-642-42053-5

[7] Schmidhuber, J. "Deep Learning in Neural Networks: An Overview". Neural Networks. 61, 2015, p.85–117.

[8] Nielsen, Michael A. “Neural Networks and Deep Learning: Determination Press.” .2015.

[9] dspace.library.cornell.edu. “Frank Rosenblatt - July 11, 1928-July 11, 1971”

[10] https://en.wikipedia.org/wiki/Field-programmable_gate_array

[11] Nielsen AA, Der BS, Shin J, Vaidyanathan P, Paralanov V, Strychalski EA, Ross D, Densmore D, Voigt CA. "Genetic circuit design automation". Science. 352 (6281): aac7341. Oct. 2016

[12] 钟文枫. SystemVerilog 与功能验证. 机械工业出版社. 2010. ISBN 978-7-111-31373-1.

[13] https://en.wikipedia.org/wiki/SystemVerilog

[14] Wulf, Wm. A.; McKee, Sally A: “Hitting the memory wall.” In SIGARCH Comput. Archit. News 23 (1), pp. 20–24. (1995). DOI: 10.1145/216585.216588.

[15] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O’Boyle, and O. Temam, “Rapidly selecting good compiler optimizations using performance counters,” in Code Generation and Optimization, 2007.

[16] C. McCurdy, G. Marin, and J. Vetter, “Characterizing the impact of prefetching on scientific application performance,” in International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS13), 2013.

[17] S. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, “Machine learning-based prefetch optimization for data center applications,” in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, 2009, p. 56.

[18] Xilinx. Zynq-7000 All Programmable SoC Data Sheet:Overview. DS190 (v1.11) June 7, 2017

[19] Xilinx. Zynq-7000 All Programmable SoC (Z-7030, Z-7035, Z-7045, and Z-7100): DC and AC Switching Characteristics. DS191 (v1.18) April 12, 2017

[20] Jonathan Bachrach, Krste Asanovi´c, John Wawrzynek. “Chisel 3.0 Tutorial”. UC Berkeley. May 8, 2017

附录

附表 1 Cache IO 接口设计

接口名称	方向	数据类型	描述
memReq	输出	ValidIO	Cache 发给内存的读取请求
memResp	输入	ValidIO	内存应答数据
cpuReq	输入	DecoupledIO	CPU 发送给 cache 的数据请求
cpuResp	输出	DecoupledIO	Cache 的应答数据
prefetchReq	输入	DecoupledIO	预取器的预取请求

附表 2 memReq 具体结构

数据名称	方向	数据类型	描述
valid	输出	Bool	Cache 请求有效信号
read	输出	Bool	读有效信号
Addr	输出	UInt[addrWidth]	请求的数据地址

附表 3 memResp 具体结构

数据名称	方向	数据类型	描述
valid	输入	Bool	内存应答有效信号

data	输入	UInt[blockSize]	内存返回的数据

附表 4 cpuReq 具体结构

数据名称	方向	数据类型	描述
valid	输入	Bool	CPU 请求有效信号
ready	输出	Bool	Cache 接收准备信号
read	输入	Bool	读有效信号
addr	输入	UInt[addrWidth]	请求数据的地址

附表 5 cpuResp 具体结构

数据名称	方向	数据类型	描述
valid	输出	Bool	Cache 应答有效信号
ready	输入	Bool	CPU 接收准备信号
data	输出	UInt[blockSize]	Cache 返回的数据值

附表 6 内存 IO 接口设计

接口名称	方向	数据类型	描述
cacheReq	输入	ValidIO	Cache 发给内存的读取请求
cacheResp	输出	ValidIO	内存返回给 cache 的应答

prefetchReq	输入	DecoupledIO	预取器发送给内存的请求
prefetchResp	输出	DecoupledIO	内存返回给 cache 的应答

附表 7 cacheReq **具体结构**

数据名称	方向	数据类型	描述
valid	输入	Bool	cache 请求有效信号
read	输入	Bool	读有效信号
addr	输入	UInt[addrWidth]	cache 请求数据地址
data	输入	UInt[blockSize]	写请求时的写入数据

附表 8 cacheResp **具体结构**

数据名称	方向	数据类型	描述
valid	输出	Bool	内存应答有效信号
data	输出	UInt[blockSize]	内存返回的数据

附表 9 prefetchReq **具体结构**

数据名称	方向	数据类型	描述
ready	输出	Bool	内存接收准备信号
valid	输入	Bool	预取器请求有效信号

addr	输入	UInt[addrWidth]	预取器请求数据地址

附表 10 prefetchResp **具体结构**

数据名称	方向	数据类型	描述
ready	输入	Bool	预取器接收准备信号
valid	输出	Bool	内存回应有效信号
data	输出	UInt[blockSize]	内存回应数据块

参数均由 2.5 节中的 Params 进行定义。

致谢

首先感谢我的指导老师蔡旻老师，如果说大学生活是人生中的一次精彩的旅行，蔡旻老师就是那个接机的人；如果说大学生活是人生中的一次大航海，蔡旻老师就是那个帮助我上岸的人。

在程序设计和论文撰写过程中，由于自己的专业知识和眼界的不足，常常专注于细节而看不清方向，见木不见林。蔡旻老师总是及时指出我在程序设计思路上的偏航，适时纠正，避免了许多弯路。有时我对老师的建议暂时不能理解，总想按照自己的思路进行。蔡旻老师也会小范围内允许我自行其是，等到结果出来后再深刻剖析，让我自行体会程序思路及建立模型的要旨，使我对专业的理解更加深刻。蔡旻老师不止是指导我完成了毕业设计，还让我学会了思考和研究的方法，所谓授人以渔也许就是如此吧。蔡旻老师常常和我们一起工作到很晚，有时双休日也放弃休息，指导我们的毕业设计，每当这时我常常想说：蔡老师，辛苦了。

我还要感谢我的父母，他们在我最困难的时候永远地帮助我、鼓励我，那即是我避风的港湾，也是我再次起航的冲锋号。世上的父母不都是这样吗？他们不需要我语言的感谢，他们希望看到我此生幸福。

我更应该感谢北京工业大学。在校期间，不论是欢笑还是迷茫、不论是顺利还是挫折，都将是我一生抹不去的记忆。