CPU设计(Tomasulo)大作业报告

致远学院ACM班　柏钧文　张曦虎

2015 年 6 月 16 日

1　前言

本次计算机系统课大作业是CPU设计，本学期课上讲过的CPU设计主要是single cycle machine，5-stage pipeline machine，scoreboard和Tomasulo。本次课程大作业要求原本是要写一个5-stage pipeline machine，但我们想挑战一下自己，也想检验和运用自己所学的知识，于是，我们决定写Tomasulo，这个与现代处理器最接近的结构。

Tomasulo总的来说是比较麻烦的，主要要考虑以下三点：

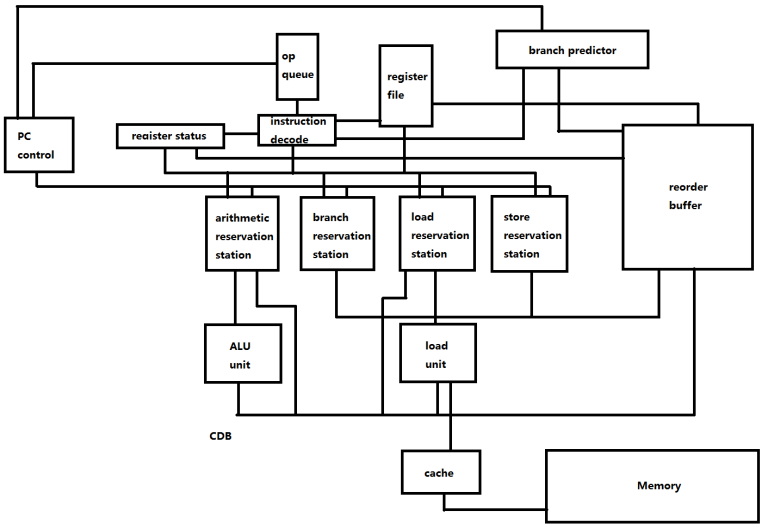
1. Tomasulo的模块设计
2. Tomasulo的out-of-order execution和in-order commit的实现
3. Branch Predictor

实际上，上述三点结合上课所学的知识就基本表明了我们所要写的模块。

总的来说，我们要写的模块基本有以下几个：operation queue，instruction decode, reservation stations, common data bus, reorder buffer, branch predictor。

当然，一些基本的必要的模块，比如register file，cache，memory等也都是要考虑并实现的。

下面这张图是我们设计的Tomasulo的一个简要示意图：



图中为了简洁起见，很多线都合并成了一根线，因此，图中的很多线兼具输入信号和输出信号的功能（实际并非如此，只是类似于将多根线包成了一根线而已），就不一一在这里赘述了。后面几个section会对各个组件进行详细的介绍。

此外，整个Tomasulo的过程按照上课所讲的内容那样，主要分为issue、execution、write back三个阶段。issue部分主要是对instruction进行解析，获得运算符以及运算数等等。execution阶段是对运算数进行运算，并监测CDB，实时接收广播值，赋给Reservation stations中一些并没有ready的运算数。运算结束后的结果广播到CDB上。最后一个write back 阶段主要就是通过CDB写到Reorder buffer和各个Reservation stations中。此外，Recorder buffer还有一个commit的过程，我们设计的Tomasulo中，一个cycle会commit一条指令（如果可以commit），具体细节会在后面描述。

2 使用方法

首先安装Icarus Verilog包。

在windows下使用命令行进入根目录下的cpu-design目录。

cpu-design目录下的testcase目录中包含3个cpp文件，generate_100_nums.cpp用于生成100个随机数字放入data.txt中，以16进制的形式保存。trans.cpp是一个简易的assemblier，用于将mips.s中的汇编代码转成对应机器码存入instruction.txt中。而calc.cpp则是将data.txt中的数字求和并输出，用于检验正确性。

将生成的instruction.txt与data.txt放置于cpu-design目录下：

编译命令：iverilog *.v

执行命令：vvp a.out

波形查看指令：gtkwave cpu.vcd

3 operation queue & instruction decode

首先说一下我们所写的mips指令和汇编器。我们写的mips指令是标准的mips指令。之后的汇编器会将mips指令翻译为机器码。我们所写的汇编器（trans.cpp）用c++语言，汇编器翻译的机器码为严格按照mips标准的机器码。举例来说，有一条add指令（add $1, $2, $3），31-26为opcode（000000），25-21位表示寄存器$1（00001），20-16位表示寄存器$2（00010），15-11位表示寄存器$3(00011)，10-6位表示mips中的shift amount部分，在这个例子中并没有用（但在其他指令诸如涉及到immediate的mips 指令中会作为立即数的一部分。最后，5-0位表示function code（100000）。由于标准mips指令所涉及到的运算非常多，但是大部分是我们这次大作业所不要求也不需要的运算。所以，我们写的汇编器处理的运算符中，主要是几个这次大作业所需要的，诸如add,addi,sub,mul,lw,sw,sll,srl,bne等。

下面讲解一下operation queue和instruction decode 部分。

operation queue和instruction decode主要是对指令进行抓取和分析的工作。在这次我们写的verilog各个组件中，operation queue对应的组件就是instruction fetch。一开始，我们将汇编器翻译的指令放入内存（instruction memory）中。之后，在operation queue 中，我们一开始会从instruction memory里面抓指令到instruction cache中。cpu开始运转后，operation queue会有一个输入的pc值，对于这个pc值，在cache中找到对应的instruction，并作为这个组件的output输出。

operation queue输出的instruction作为instruction decode组件的输入。

在instruction decode组件中，会对输入的instruction进行解析。经过汇编器翻译的机器码为一个32位的01串，我们需要通过读入这32位的01串，知道要处理的是怎样的一条指令。比如还是举上面所述的那个例子（add $1, $2, $3），我们现在读入了一个32位的01串（就是像上面所述生成方式生成，00000000010000110000100000100000），我们要知道我们做的是add $1, $2, $3。那这个过程实际上上面汇编器的一个逆过程。首先看前6位是不是000000，再看后六位是不是100000，发现是，那么就知道我们做的是add操作。再解析三个寄存器，发现要写的目标寄存器是$1（00001），另外两个要加起来的寄存器为$2（00010）和$3（00011）。其他诸如mul,addi,lw等也都是按照mips标准解析。

经过解析后，我们就知道了我们要做什么。解析出来的operatorType（add，addi等）、reg1、reg2、destreg、immediate等等作为instruction decode阶段的输出。

4 register file & register status

解析出来运算符和要读、要写的寄存器只是第一步，抓取具体的数值才是关键。这就需要到register file和register status里面走一遭了。

register file兼具读和写的接口，我们先介绍读的部分。读入的时候，输入主要为4个数据，运算符operatorType，两个运算数所在的寄存器reg1和reg2，还有可能会有的立即数immediate（addi指令中）。输出主要是3个数据，data1、data2和offset，表示从寄存器中抓的两个数。register file组件中有32个32位的register，首先，我们直接将寄存器阵列中的对应于reg1和reg2的register中存的值作为data1和data2输出。但是，可能有些运算符会涉及到immediate，如果运算符是addi，immediate作为data2输出，data2此时并不是reg2对应的数值（也就是说在这条指令中，reg2实际上是用不到的），如果运算符为lw或sw，immediate作为offset输出。

当然，由于Tomasulo的架构，并不是所有去register file里面抓的值都是正确的或者说我们想要的。但不管怎样，我们先抓了值再说。对不对由register status说了算。

register file还有写的接口。一般是在Reorder buffer的commit操作时写，如果register file中的input接口write enable为1，那么我们就可以把输入值writedata写到对应寄存器中。

那么下面就要说一说register status了。之前说过，register file抓的值合不合法由register status说了算。register status里面存放的本来应该是每个register正在被哪个reservation station写，但加入reorder buffer后，我们可以

不记录reservation station的编号，而是记录对应要写的reorder buffer中的编号。这两种方法实际上都是renaming的方式。

初始时，所有register的status都设为非法值，说明每个register都没有在被任何一个reservation station 写。在开始处理指令后，register的status可能会有所变化。

register status组件主要输入为reg1和reg2。对于任意一个输入的reg来说，如果reg对应的status是非法值，那么我们在register file中读取出来的值就是合法值。否则，reg对应的status为最新要写这个reg的reservation station要写的reorder buffer的编号，也说明我们现在从register file中读取出来的数据实际上并不是合法的。我们将两个register对应的status 作为q1和q2输出。

register status中register的status也可能会被修改。我们issue一条指令后，如果要写一个寄存器，我们就会抓取这条指令对应的reorder buffer的编号，并把寄存器的编号和reorder buffer的编号作为输入给到register status组件中，并把register status的write enable置为1，表示可以写，然后改变register的status即可。

至此，第一个阶段issue算是基本完成。

5 Arithmetic Reservation Station

Arithmetic Reservation Station（对应组件为addRS）的输入就是上面说的几个组件的输出：operatorType，data1，data2，q1，q2。另外还要从reorder buffer获取信息知道这个指令要写到哪个reorder buffer的单元上（就是一个编号）。当然，不光Arithmetic Reservation Station，其他reservation staions也是一样。

如果新的数据进来reservation station时发现reservation station已经满了，那么我们就要将输出接口availble置为0，表示满了，那么pc就不应该变化了。这是一个structure hazard，不可避免（除非reservation station够大）。否则，我们找到一个空闲的reservation station，将这5个数据扔进去，并把这个reservation station的busy bit 置为1，表示这个单元被占用。

reservation station还要有一个抓CDB上的数据的功能。对于每个cycle，CDB广播数据之后，Arithmetic reservation station需要扫一遍所有的单元，如果有单元上的q1或者q2与CDB广播出来的reorder buffer的编号相同，那么q1或q2就可以置为非法值，并把CDB上广播的data写入单元中对应的位置。

在每个cycle的上升沿，我们在Arithmetic reservation station里面扫一下

所有的单元，如果有单元的q1和q2都为非法值，且这个单元是busy的，那么就可以用ALU来处理运算数。并广播要写的reorder buffer编号和计算结果到CDB上。

6 Load Reservation Station & Store Reservation Station

Load Reservation Station与上面Arithmetic Reservation Station一样，几个输入就是operatorType，data，q 以及reorder buffer number，因为只有一个寄存器要读。另外，还要加入一个输入offset，作移位用。

它与Arithmetic Reservation Station的区别主要在于它的每个单元存储的信息实际上是地址，而且暂时不带offset。类似地，找一个空闲的单元，将上述几个数据写入Reservation Station即可。

CDB广播值的接收与Arithmetic Reservation Station 一样，如果广播了对应数值，那么这个单元就是ready 的了。

同样地，每次时钟上升沿，我们都将看一下有没有已经可以操作的数值（即q是否为非法值），如果有，那么将这个从寄存器中读取的数值加上对应的offset输出到load unit作为地址。

load unit是一个类似于ALU的组件，它的功能主要是接收地址，并去data cache中抓数据，关于data cache，后文会有详细解释。每个cycle我们都只会这样处理一个指令。如果hit了，那么就要到CDB上去广播一下。

Store Reservation Station也是类似，但输入的就得是operatorType，data1，data2，q1，q2了，因为有一个我们要写到内存中的数值要从寄存器里面拿。Store Reservation Station的存储就不说了，也是类似Load Reservation Station。

Store Reservation Station与上面几个Reservation Stations主要的区别在于它并不需要写到CDB上，而只要往Reorder buffer上写就行了，因为写回内存的部分由Reorder buffer来执行。

7 branch Reservation Station

为了处理分支语句，我们专门写了个组件branch Reservation Station，总的来说，它与Arithmetic Reservation Station一样，区别仅在于它不会写到CDB上去，而是把判断结果传给Reorder buffer。

8 CDB

common data bus是用来广播的组件。实际上CDB就是几根wire而已，这个组件内部并不计算什么。

唯一要注意的几点是，我们设计的Tomasulo中，每个Reservation Station都有一个属于自己的CDB，每个Reservation Station也要有与CDB个数相同的对应的几个接口（reorder buffer num 和广播的data等）。对于每一组接口，每次广播时，都要扫一遍Reservation Station，看能不能把一些数值的状态置为ready。

9 PC control

PC control这个组件主要是用来控制pc的。一般情况下，pc每个cycle加1即可。但是，会遇到两种特殊情况。

第一种，遇到了structure hazard，即有一个指令进Reservation Station时，它满了，不能再进了。那么就要让pc stall一下（pc保持不变）。这种情况的处理就是给每个Reservation Station一个available的状态记录它到底满没满，如果有条指令碰壁了，那么这个Reservation Station的available就应该置为0。所有的available状态都传给PC control，如果有一个available为0，那么就要stall。

第二种，branch predictior预测错了，要跳指令。这个时候，Reorder buffer会传过来一个pc跳转的enable 信息，将pc值改为要跳转的new pc就行了。

更新过的pc传给instruction fetch组件，让它抓取指令。

10 Cache & Memory

数据主要存在Memory中，而在实际应用中Memory的读取和写入速度是非常慢的。（在我们的实现中，Memory一次读写被认为需要花费200个cycles才能实现）因此，为了优化Memory的读取速度，我们实现了一个64KB的level 1 cache。该cache的block大小为64Bytes，即16 个words。

10.1 Memory

Memory module的实现源代码存放在dataMemory.v中。我们提供了read和write的端口来传递memory读写所需要和产生的信息。read操作相关的端口包括readAddress, readEnable, data out。其中readEnable来表示data out端口的数值是否是我们提供的readAddress所对应存储位置的值。当我们需要

更换地址作读取操作时（如果更换后地址与之前的地址相同则不作任何操作），module 会因为收到新的读操作而将readEnable置为0，告诉外部设备现在memory正在读取对应位置的数据，data out位置所输出的数据并没有被更新。当200 个cycle过去后，memory 会将从readAddress对应抓取位置的数据放置于data out处并且将readEnable置为1。表示读取完毕。

因为我们实现了cache，所以memory与cache之间的交互单位是一个block（在此我们使用512位的reg类型模拟），由于对象汇编代码仅最后一条是sw操作。我们采用的write policy是write through，因此我们仅为memory提供了一个32位整数的写入操作。（可以添加512位block写入操作）write端口有writeAddress, writeRequest, writeData以及writeDone四个。其中writeAddress与writeData用于存放write的对应地址及其所写的值。当外部设备处理好writeAddress 和write Data的值后，会通过writeRequest端口发射一个先上升后下降的脉冲。而当memory检测到writeRequest 的脉冲后，它会知道writeAddress 和write Data是它现在要处理的值。然后memory会开始模拟write操作：等待200个cycles，等待完毕后再将数据写入memory。

10.2 Cache

Cache module源代码存放在dataCache.v中。我们所使用的是64KB大小，block大小64B的direct map cache，并提供read和write端口支持cache的read 和write操作。但read和write 端口又分为cache 与Memory 交互的端口和cache 与外界（loadUnit & reorderbuffer）交互的端口。

对于loadUnit对cache的read操作。cache提供readAddr、hit、readdata端口与loadUnit相连接，为loadUnit提供read操作。loadUnit可以实时更改readAddr上的值。而cache 也会实时反应，将readAddr的值分解成tag、index与offset三段值。找到与其index所对应的block。并将该block中offset 位置的值读出放入readData 上。（无论所读数据是否是readAddr所需数据）并将block的tag与readAddr的tag相比较。如果比较成功则hit置为1，否则hit置为0，即为miss。loadUnit可以通过hit的值来作出是否使用该数据的配合行动。因此在任意时间都可以读cache中的数据。（不考虑是否hit的话）进一步地，因为从cache中读数据不保证总是hit。cache会在每次clock的上升沿处查看hit的状态，如果为1，则不作任何操作，否则它会通过将readAddr传给memoryReadAddr，触发Memory 的read来获得新的block data并等待memory搬运数据，替换其原block来读取正确的数据，搬运完成后再次刷新外界读操作的结果。由于搬运后block中数据与tag均已改变。hit会很自然地更新为1，从而完成整个read miss操作。

对于Reorderbuffer中commit sw的操作，cache提供有writeEnable、writeAddr、writeData的input接口与writeDone的output接口与reorderbuffer 作交互。 write Done的output接口用于告知reorder buffer其write的完成情况。0为未完成、1为已完成。 write过程先由reorderbuffer填写writeAddr与writeData的值，然后通过writeEnable 发射脉冲来驱动write。当cache收到writeEnable的脉冲信号后。它会先检测cache中的对应block是否为writeAddr 所指向的地址，如果是，则先改写cache中该地址的值。然后cache将writeAddr复制到memoryWriteAddr接口，将writeData复制到memoryWriteData接口，并通过memoryWritePulse向memory module发射脉冲驱动memory的write操作。当memory写入完成后，cache会将writeDone置为1表示写入操作执行完毕。

11 Reorderbuffer

ReorderBuffer源代码存放在reorderBuffer.v中。由于与ReorderBuffer 交互的单元特别多，为了尽量保证正确性，ReorderBuffer 尽量为每个单元提供的独占的读写接口，因此ReorderBuffer总共使用了54个端口，但每个端口的功能分配都比较清晰。在这里，reorderbuffer具备最多16个位置储存等待commit的指令。并且所有修改机器状态的指令都需要通过reorderbuffer执行。reorderbuffer同时具备处理branch misprediction的能力。

端口issue_opType, issue_data2, issue_pc, issue_destReg, issueValid与ISSUE阶段所涉及的instruction decode Unit、PC control Unit相连接。获取pc 位置信息，operation type信息，destination register信息，以及与jump相关的destination信息。而issueValid接由instruction decode Unit传过来的脉冲，发现脉冲后Reorder Buffer认为现在opType、data2、pc、destReg端口所提供的信息是一条新的指令，因此会将这些信息存入buffer队列中。

端口*IndexIn, *Result, *ReadyOut为reservation station查询并获取reorder buffer信息的接口。其中*IndexIn表示对应reservation station所需要读取的位置。*ReadyOut表示该指令的结果是否已经计算完成。*Result为该位置指令计算所得结果（即使没有ready也可以提供）。

端口available用于表示reorderbuffer中是否还有富余的指令存储空间，PC control接收该端口的值并且决定是否继续ISSUE。

端口statusWriteEnable、statusWriteIndex和statusWriteData使用脉冲触发register status Unit 改写其内部状态。由于register status Unit的写操作由reorderbuffer独占，因此不会产生write冲突。

端口cacheWriteEnable、cacheWriteData、cacheWriteAddr 和cacheWriteDone用于SW指令对cache与memory 的改写。同样使用脉冲触发。cacheWriteDo ne用于告知reorderbuffer其cache是否已经完成write 操作。

端口branchPrediction、branchAddr、branchWriteAddr、branchWriteEnable、branchWriteData、issueNewPC、issueNewPCEnable、resetAll用于辅佐reorderbuffer完成branch prediction 的correction 操作，即纠正misprediction。当reorder buffer在顺次commit指令时，如果遇到branch jump指令，它会检测来自branch Unit的结果。如果branchUnit所给的taken与not taken 的结果和branchPredictor所给得结果不同。它会将branch之后的指令全部清除。将register status中的内容通过write操作清空。并向所有reservation station和execution unit通过端口resetAll发射脉冲，重置其当前所有操作防止对reorderbuffer进行进一步地影响。并通过对存储在reorder buffer中的pc值与jump destination 以及branchUnit所给出的正确结果进行比较计算得出下一步pc 应该在的位置。将该位置写入issueNewPC，并将issueNewPCEnable 置为1，于下一个时钟上升沿时会将issueNewPCEnable重新置为0，防止每次PC的值都被重置。而每次计算完branch后，reorderbuffer通过branchWriteAddr、branchWriteEnable 、branch WriteData驱动branch predictor Unit改写其数据，更新prediction。

端口storeEnable、storeRobIndex、storeDest、storeValue 是由store reservation station所独占的接口。用于通知reservation该store指令在reorderbuffer中的位置（storeRobIndex），其存储至内存的地址（storeDest）和值（storeValue）。reorderbuffer将这些值存储并等待commit，到commit时再执行SW操作。store信息写入操作仍由storeEnable发射脉冲驱动。

端口CDBisCast、CDBrobNum、CDBdata分别表示CDB广播的驱动脉冲信号，CDB中data所对应填写的reorderbuffer位置以及CDB所提供的data。其中CDB1所给出结果来自ALU Unit，CDB2所给出的结果来自load Unit。

端口space用于向ISSUE阶段提供reorderbuffer队列的尾位置，是之后更新reorderbuffer中指令状态的必不可少的凭证。

端口regWriteEnable、regWriteIndex、regWriteData 是reorderbuffer独占的r egister file Unit的write操作接口。当一条指令被commit后，reorderbuffer会通过这些接口更新register中的data。由脉冲驱动。

端口cataclysm仅测试用。用于接收外部信号来表示整个verilog的simulation是否终止。

端口bneWriteResult、bneWriteEnable、bneWriteIndex由branch Unit独占。用于传递branch Unit的计算结果并更新reservation station。由脉冲驱动。

端口worldEnd仅测试用，用于向部分Unit发射信号来通知verilog simulation结束。

12 Branch Predictor

Branch Predictor的源码储存在BranchPredictor.v 中。我们设计的branch Predictor使用了Two-bit saturating counters与Two-level Branch predictor的方式来作prediction，即对于每个branch，考虑其前两个branch的结果后从4 条存储空间中选择出一条，并读取其对应位置的prediction数据。

端口branchWriteEnable、branchWriteData、branchWriteAddr 为reorder buffer独占的prediction改写端口，用脉冲驱动。更新prediction数据并刷新其它读入操作结果。

端口branchPCReadAddr、branchPCPredict是PC独占的读取prediction端口。用branchPCReadAddr来指定读取位置并实时更新。

端口branchROBReadAddr为reorderbuffer独占的prediction读取接口。

13 总结

我们这次大作业从开始准备做，到最终基本完成大概花了5天左右，应该来说还是比较快的。但其中的几天都通了宵，总体来说感觉工作量还是很大的，尤其是debug阶段，调得还是蛮辛苦的。

但通过这样子的磨练，我们现在可以说基本掌握了verilog这样一个硬件编程语言。也切实地检验了一下我们对上课所学知识的掌握。通过这样一个大作业，我们对Tomasulo的细节有了更加深入的认识，对整个结构也有了更加清晰的认识。

这个大作业也是比较锻炼人的思维能力和抗压能力的。这次大作业，初始的架构是很重要的，这就要求一开始就要对整个框架有个非常清晰的认识，甚至是细节部分。其次，很多时候，verilog 代码查错没有像c++、c、java这样来的方便，verilog 查错很考验耐心和分析错误能力。也间接地考验着我们的抗压能力。总的来说，对我们还是很有帮助的。

此外，这次大作业是两人合作项目，从效果来看，我们两个人的配合还是很不错的。基本上两个人分别写了一半的代码，debug的时候也是一起调，所以速度比较快。通过这样一次合作，我们的团队意识有所提升，也掌握了一些与别人合作的经验教训。相信这些在今后的学习生活科研中都是很有帮助的!

14 鸣谢

感谢梁晓峣老师一学期来的课程教授。梁老师的讲课是非常清晰的，所以我们对Tomasulo的理解从一开始就比较正确，只有几个细节部分要自己多想一想。其次，感谢我们身边的同学，很多时候我们走了弯路，犯了错误，都能从他们那里获得帮助和灵感。其次，还要感谢助教一学期的课后指导和作业批改。

15 Reference

《Verilog HDL程序设计实例详解》张延伟
《Verilog SOPC高级实验教程》夏宇闻
《基于Verilog HDL的数字系统应用设计》王钿
《CPU自制入门》水头一寿 米泽辽 藤田裕士
《计算机体系机构：量化研究方法》John L.Hennessy David A.Patterson