1 Verilog 复习

本课程实验要求所有设计采用 Verilog 语言进行描述。我们这样选择是因为 Verilog 是一种常见的硬件描述语言，当前很多厂家的设计都采用 Verilog 语言开发，而且初学者学习 Verilog 不会比学习 VHDL 花费更多时间。有的同学可能会问为什么不采用 SystemVerilog 语言。在我们看来，Verilog 语言和 SystemVerilog 语言之间的关系有点类似于 C 语言和 C++语言，两种语言都有适合其应用的地方，并没有谁一定替代谁的趋势，并且学习 Verilog 语言之后能帮助学习 SystemVerilog 语言，所以我们还是选择了 Verilog 语言。我们也没有采用脚本化语言的设计风格，脚本化语言的设计风格的优势在于提高设计效率，但是考虑到课程实验的设计规模并不复杂，同时我们希望同学们能够对于设计的电路实现有更清晰的认识和掌控，所以我们还是坚持采用 Verilog 语言，且在 RTL 级进行设计建模。

目前已经有很多优秀的关于 Verilog 语言方面的书籍和文献，理论上来讲同学们学会使用 Verilog 语言应该没有任何问题，但是从我们的实际教学体会来看，情况并不是这样。这是因为 Verilog 的书籍主要是讲 Verilog 语言本身的，而同学们碰到的很多问题其实是使用 Verilog 的工程经验，所以这里我们的复习就围绕这方面展开。

1.1 怎么学习 Verilog 语言

我们把 Verilog 语言的学习要求分为两档，低一档的叫“学会”，能满足本课程实验任务，高一档的叫“学好”，能够熟练掌握 Verilog 语言进行工程开发。要想学好 Verilog 语言，重点已经不在语言本身，更多的涉及电路设计、系统思维、编程风格、项目管理，需要在实践中不断总结提高。将来有志于从事 IC 前端设计开发的同学们，可以把“学好 Verilog 语言”作为一个长期任务执行下去，这里就不展开进行讲述。接下来我们主要告诉同学们如何“学会”Verilog 完成实验。

本课程实验中有两个地方用到 Verilog 语言，一个地方是要求同学们用 Verilog 语言进行设计，另一个地方是我们用 Verilog 语言给大家写的测试平台（testbench）。前一个地方要求大家会看会写，后一个地方只要求大家看得懂。testbench 的 Verilog 代码中我们已经添加了相应的注释，所以各段代码干什么的同学们应该是知道的，如果有什么关键字、系统调用的不知道具体什么含义，就请自行查阅 Verilog 参考资料。那么同学们真正需要花力气学习的是用设计部分所需的 Verilog 语言。

我们建议同学们采用 RTL（Register Transfer Level，寄存器传输级）设计，且要求设计可以被 EDA 工具综合成最终的电路，那么大家实际开发过程中只会用到 Verilog 语言的一个子集（即所谓的可综合子集）。

国科大B02009计算机体系结构研讨课17-18秋季

同时由于 Verilog 语言自身用法比较自由，导致容易出现一些易混淆和易出错的地方，出现竞争条件、前后仿真不一致。为了那么为了避免同学们在实验过程中在这些语言方面花费过多的调试时间，我们又会对可综合子集的使用方式做出限制，从形式上降低代码出错的风险。在上述这一系列限制之下，同学们实际需要学习的 Verilog 语言要素已经非常非常少。为了帮助同学们迅速掌握这些必须的 Verilog 语言要素，我们会在后面给出一系列具体的代码示例，同学们只要把这些例子中涉及的语言要素都掌握了，则至少完成本实验的设计就不是问题了。事实上，我们在进行龙芯 CPU 的前端设计时，也就只用到 Verilog 的这些语言要素。

总之，语言只是个工具，是手段不是目的。即使有一把绝世名剑，也并不意味着你就是个好剑客。反之，真正的高手，草木竹石皆可为剑。我们花费了很多心思就是希望同学们不要花费过多的时间去学习 Verilog 语言本身，掌握最有用的一小部分就可以开始实践了，其余部分视你的学习、工作需要，可以在日后继续学习。我们要把精力更多的花在那些提到能力的方面，所以下面我们先讨论设计思路和代码风格，然后再介绍可综合 Verilog 代码并进行代码示例讲解。

1.2 面向硬件电路的设计思路

Verilog 语言的很多语法要素与 C 语言很像，而且 Verilog 语言也支持行为级建模。这就使得很多同学习惯于用 C 程序开发的思路去使用 Verilog 语言，即总是有个过程的概念在脑海里挥之不去。教学中常听到一些同学这样对我描述它的代码——“先调用 X 模块，然后如果条件 C1 成立，就调用 Y 模块，如果条件 C2 成立，就调用 Z 模块……”——是不是还是 C 程序编程那个调调。这种**串行的过程化的思维对于 Verilog 语言设计来说是有很大弊端的**。我们是用 Verilog 语言来设计硬件电路的，而硬件电路不是串行化过程化的，它是并行并发的。所以 Verilog 语言里面，一前一后两句 assign 语句，不是写在前面的先执行写在后面的后执行，在忽略底层电路延迟因素的情况下，这两句对应的逻辑是同时并发执行的。在使用 Verilog 语言设计电路时，无论你是采用门级建模、数据流建模还是行为级建模，都是对电路建模，不是对算法建模。这样说可能有些抽象，我们来举个例子。同学们都写过单周期的 MIPS CPU，知道取指的 PC，如果没有跳转（taken）的分支（branch）指令，那么下一个周期填入的 PC 是当前 PC+4；如果有 taken 的 branch，那么下一个周期填入的 PC 是当前 PC+Offset（这里我们暂时先不考虑 MIPS ISA 中分支延迟槽的问题）。就上面这个功能，有些同学的 Verilog 代码可能是这样写的：

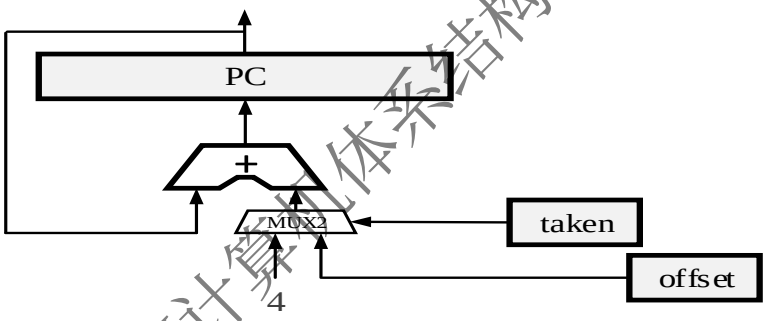
```
reg  [31:0] pc;


always @(posedge clk) begin
   if (inst_is_br && br_taken) begin
      pc <= pc + {{14{br_offset[15]}}, br_offset[15:0], 2'b00};
   end
```

国科大2020H计算机体系结构研讨课17-18秋季

```
    else begin
        pc <= pc + 3'h4;
    end
end
```

上面的代码并没有错，但是我不推荐。这种写法是面向算法功能描述的，不是面向硬件电路的。

怎么让同学们从 C 的那套思维方式转到面向硬件电路的设计思路呢？实践中我们总结出了一条经验，要求必须采用这样的设计步骤：**先进行电路结构设计，再进行 Verilog 代码编写**。这样就从根本上避免了学生们被 Verilog 语言中的行为级建模方法带偏了方向。当你在考虑电路的结构及其逻辑的时候，自然会产生并行并发的意识，也会采用起“数据通路+状态机”的设计思路。当电路设计已经被清晰地分解为结构图中的各个模块和模块之间的连接、模块内部的数据通路和状态机、数据通路中的电路逻辑以及状态机中的状态转换图，那么接下来的 Verilog 代码设计就只是一个简单的“翻译”而已。继续前面的例子。在组成原理或体系结构的教科书里面通常都会看到关于取指 PC 的数据通路结构图是如下这样的。



看着上面这个图，你会很清楚地知道，PC 是用 32 个触发器构建的一个 32 位寄存器向量，这组向量的 Din 输入来自于一个 32 位的加法器，加法器的一个输入永远是当前的 PC 寄存器中的值，另一个输入又来自于一个 32 位二选一，根据分支的 taken 信号选择是 branch 指令的 offset 还是 4。所以说，当你已经想清楚了电路结构，那么剩下来的事情真的就只是“描述”一下了。上面的例子用面向硬件电路的代码风格写出来是这样的：

```
reg  [31:0] pc;
wire [31:0] next_pc;
wire [31:0] pc_adder_a, pc_adder_b;

assign pc_adder_a = pc[31:0];
assign pc_adder_b = is_br_taken ? {{14{br_offset[15]}}, br_offset[15:0], 2'b00} : 32'h4;
assign next_pc    = pc_adder_a + pc_adder_b;

always @(posedge clk) begin
```

国科大B62009H计算机体系结构研讨课17-18秋季

```
    pc <= next_pc;
end
```

这种面向硬件电路的代码设计风格并不意味着让同学们必须是用逻辑门去构建设计。上面给的例子用到了“?:”、“+”这样的行为化描述，我们是推荐采用这种写法的。那么同学们可能会困惑了，到底怎么把握这个度呢。请注意，所谓面向硬件电路的设计风格，重点是在于代码描述要对于底层电路结构可控。你写出来一段行为化描述的代码，你需要知道其实最终的实现还是电路，是 EDA 综合工具（在我们实验环境下就是 Vivado 中的综合步骤）完成了从功能描述到底层电路实现的推导。你关注的仍然是电路。只有当你对于一个行为化描述会被综合工具综合成什么样的电路时，你才能够放心大胆地使用这种描述方式。

采用上述设计步骤设计一个较为复杂的电路系统时，同学们可能会觉得电路结构设计的工作量很大，不知道从何处下手。我们建议同学们采取**“自顶向下、模块划分、逐层细化”**的设计步骤。划分过程中，我们推荐**优先沿着数据流的路径进行划分**。一个实用的电路系统，肯定是有输入也有输出的，自然也就存在从输入到输出的数据流路径，沿着这个路径进行切分总是不会有大偏差的。具体如何划分要根据实际情况，总的来说，**功能上相对集中的逻辑应该放在一个模块内，高层次模块与模块之间的数据交互尽可能呈现单方向流动，最底层模块的规模控制在千行代码之内**。

上述电路结构设计过程通常不可能只考虑一遍就能获得很好的设计方案，需要反复迭代，多次调整。所谓“**谋定而后动**”，我们强烈建议同学们一定要先把电路结构设计想妥当了，再开始动手写 Verilog 代码。有的同学喜欢“写一点，试一下，改一点，再试一下”这种所谓的增量代码开发方法，尽管听上去很好，但从我们自身的实践来看，这种设计方式不太适合电路系统的设计，至少不适合 CPU 的设计。这主要是因为 CPU 中各部分之间的相互关系十分紧密，常出现牵一发而动全身的情况，所以往往一改就要前前后后改很多地方。改的地方越多，出错的概率也就越大，导致改的地方更多，这样代码开发过程就很容易呈现发散的状态，不容易在可控的时间内收敛到一个稳定状态。本实验课程需要大家设计的 CPU 相对简单，大概不到两千行 Verilog 代码就能实现，测试程序集也很少很不充分，同学们用试错的代码开发方法在规定时间内说不定也能搞定，但我们还是希望同学们养成“谋定而后动”的设计习惯，把迭代放到设计方案的制定阶段而不是放到代码编写阶段。记得有一个学生曾和我聊他做实验的过程，总是要“纠结”好几天甚至一两周，等到都想顺了，不“纠结”了，真正动手写代码一天不到就写完了，调起来也快，主要是一些笔误引入的语法错，基本没有什么逻辑错误，偶尔有一两个逻辑错误那也确实是因为之前就没有想到这种情况的组合。我觉得这个学生的设计开发方法就是我们极力推荐的，也希望大家都朝这个方向努力。

国科大UC620计算机体系结构研讨课17-18秋季

1.3 浅谈代码风格

代码风格是个很重要的问题，关乎代码质量、生产效率，是区分普通设计人员和优秀设计人员的重要指标之一。代码风格也是个见仁见智的问题，没有唯一正确，只有相对合理。我们安排实验的首要任务是培养同学们的能力，所以有必要在这里谈一下代码风格的问题，让大家多少对这个问题重视一下。

代码风格之所以是个问题是因为，代码写出来不仅仅是给机器看的，更多的时间里代码是给人看的。如果只是给机器看，数万行代码可以放在一个文件，不加任何缩进和换行，而且所有的变量名都可以命名为“vzi101dba3832”这种形式。这种风格机器看起来一点困难都没有，但是人看起来就是一本“天书”。所以**务必记住：代码写出来，要让自己过了一年半载还能看得懂改得了，要让同项目组的其它成员能看得懂改得了，要让项目无关的本领域设计人员在看了你的设计文档之后也能看得懂改得了。**这就是好的代码风格的最基本的评价标准。

好的代码风格有很多细节，篇幅有限，这里主要提一下变量命名和代码对齐。

变量命名

变量命名要注意在名字中体现出该变量的含义，最好能够让人“望文知义”。如果变量的名字定义的好，很多时候没有注释和设计文档都可以把代码看得差不多明白。例如，进行五级流水 CPU 设计的时候，按照每一级的功能，把模块的名字定义为 fe_module、dec_module、exe_module、mem_module 和 wb_module 就比定义为 pipe1_module、pipe2_module、pipe3_module、pipe4_module 和 pipe5_module 要好的多。五级流水 CPU 设计中译码流水级中的相关判断、将执行级、访存级、写回级各级的目标寄存器号分别定义为 exe_dest、mem_dest 和 wb_dest 就比定义为 dest1、dest2、dest3 要好的多。

可以看出来好的变量名通常是一个短语而不是一个单词，那么一个短语中若干单词之间如何区分呢。常见的做法有两种，一种是下划线，一种是大小写。我个人是习惯于下划线的，这可能与我对英文大写字母不敏感有关，大小写区分的写法我一眼看过去划不出单词。当然用下划线隔开只是我个人喜好，同学们可以自行选择自己喜欢的方式。

将变量名定义为一个短语会碰到的问题是变量名容易过长，这就需要我们采用一些缩写形式。如 counter 可以缩写为 cnt，forward 可以缩写为 fwd，source 可以缩写为 src，dest 可以缩写为 dst，buffer 可以缩写为 buf，XX_queue 可以缩写为 XXq，等等。缩写没有一个所谓的通行标准，一般来说一家公司内部的缩写是统一的。同学们至少要保证一个设计中的缩写是一致的，譬如，如果一个设计中有的同学用 wb 代表 write back，而其它同学用 wb 代表 write buffer，那么这样的设计代码阅读起来就容易让人产生困惑。此外建议大家对于采用了缩写的变量名，最好在变量声明的地方把缩写前的短语记录在注释里面，这样会大大帮助别

国科大R6200计算机体系结构研讨课1/16秋季

人理解你的缩写风格。

代码对齐

代码对齐主要有两个目的：一是容易规避语法错误，二是美观。

同学们在学习 C 语言程序设计的时候，可能已经接触过缩进的概念。同一缩进层次的代码的起始列对齐有助于我们一眼区分出代码的逻辑层次。此外像 Verilog 中的 begin…end、case…endcase、module…endmodule 都是配对出现，如果将这些关键字对齐在其所在的层次上，那么就比较容易检查出来有没有出现配对确实的情况，否则即使仿真综合工具报错，你定位错也要比较长的时间。

美观这件事情，很多同学可能不以为然，觉得只要写的代码是对的不就可以了吗，看嘛要看得好看。还是最初那句话，代码更多的时候是让人看的，而人看待一件事物的时候是抛不开主观感受的。同学们高考的时候，语文老师一定向你们强调过作文部分字迹要工整，道理是一样的。再精巧的设计，代码写得七零八落、参差不齐，别人也看不出个好来，甚至会曲解你的设计意图。又常说“字如其人”，对于程序员来说，代码反映了你的思维状态。作为我来说，我很难相信一段排列混乱的代码背后会是一套清晰的思维，我主观上倾向于怀疑这是一段不断“打补丁”之后的代码。所以美观这件要求，不是为了美观而美观，首要条件还是你的思路清晰了，在此基础之上稍加注意，则代码自然就体现出清晰简洁之美。

此外强调一个小细节，上下行**对齐的时候不要使用 Tab，而要用空格**。因为不同的文本编辑器对于 Tab 算作多少个空格可能是不一样的，这就有可能导致你写的代码在你自己看来已经排列得很整齐了，但是别人打开来看的时候就完全不是那么回事了。

Verilog 代码风格中还有很大一部分是用于强调可综合性、前后仿一致性和跨平台一致性的，也是很重要的。我们把其中一些关键内容融入到接下来关于可综合性的讨论和代码示例中，详细的原因我们不在此展开解释，感兴趣的同学可以自行阅读 Verilog 专家 Clifford E. Cummings 的一系列经典论文：

- A Proposal To Remove Those Ugly Register Data Types From Verilog
- full_case parallel_case, the Evil Twins of Verilog Synthesis
- Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill
- RTL Coding Styles That Yield Simulation and Synthesis Mismatches
- Verilog Nonblocking Assignments With Delays,Myths & Mysteries

国科大B6209H计算机体系结构研讨课17/18秋季

1.4 可综合代码

Verilog 和 VHDL 这类硬件描述语言设计最初是用于大型数字电路的建模、仿真，由基于 HDL 的设计转换为逻辑门相互连接的电路图的工作仍是由设计人员手工完成的，这个过程既费时费力又容易出错。后来逻辑综合工具的出现和发展改变了这一状况。设计者可以使用 HDL 在 RTL 级对电路进行描述，然后选定标准单元库并定义相关设计约束，那么逻辑综合工具就会自动的将 HDL 语言转换为门级网表，这个转换的过程称为逻辑综合。

在逻辑综合过程中，工具能够支持的语言要素是 Verilog 语言的一个子集，也就是说并不是所有的 Verilog 语言要素都可以进行逻辑综合。表 1-1 列举了 Verilog-1995 中可综合的语言要素。表 1-2 列列举了 Verilog-2001 中可综合的语言要素。

表 1-1 Verilog-1995 中可综合语言要素（来源于 Stuart Sutherland）

Verilog 结构	描述
模块声明 **module**, **endmodule**	完全支持
端口声明 **input**, **output**, **inout**	完全支持，且支持任意向量大小
线网数据类型 **wire**, *wand*, *wor*, *supply0*, *supply1*	完全支持，且支持标量和向量
变量数据类型 **reg**, **integer**	可以是标量、向量或变量数组 变量赋值只能来自于一个结构化过程语句（procedure） integer 类型缺省为 32 bits
参数常量（parameter constants）	仅限于 integer 类型
整型数（literal integer numbers）	完全支持，且支持任意大小和进制
模块实例化	完全支持； 同时支持位置相关和名字相关两种端口实例化方式
原语实例化 *and*, *nand*, *or*, *nor*, *xor*, *not* *buf*, *bufif1*, *bufif0*, *notif1*, *notif0*	完全支持
连续赋值语句 **assign**	完全支持； 且同时支持显式连续赋值和隐式连续赋值
过程赋值语句	完全支持； 但不支持 deassign 关键字
函数定义 *function*	仅支持部分结构
任务定义 *task*	仅支持部分结构
always 结构化过程语句 **always**	必须有敏感列表

国防大B6209计算机体系结构研究课17-18秋季

Verilog 结构	描述
begin ... **end** 块	完全支持； 命名块和非命名块都支持； 但是不支持 fork ... join 形式的块
阻塞赋值（=）和非阻塞赋值（<=）	完全支持； 同一个变量的所有赋值只能采取同一类型的赋值
条件判断语句 **if**, **if**...**else**, **case**, *casex*, *casez*	X and Z 均只视作“无关”位
for 循环 *while 循环*	步进量的赋值必须为递进和递减
disable statement group	must be used within the same named block that is being disabled
操作符 **&** **~&** | **~|** ^ ^~ ~^ == != < > <= >= **!** **&&** || << >> {} {{}} **?:** + - * /	操作数可以是变量或向量，可以是常量或变量 不支持=== 和 !== 操作符
向量比特选择 向量部分选择	出现在赋值语句右侧时，完全支持； 出现在赋值语句左侧时，仅支持选择量为常量。

表 1-2 Verilog-2001 中新增的可综合语言要素（来源于 Synopsys 文档）

Verilog 结构	描述
敏感列表中用逗号分隔	完全支持
组合逻辑敏感列表用@*	完全支持
将端口和数据类型集成在一起的声明方式	完全支持
ANSI C 形式的端口声明	完全支持
连续赋值中的隐藏线网	完全支持
多维数组	完全支持
数组位选或部分选择运算符+: 和 -:	完全支持
有符号数据类型	完全支持
有符号数字	完全支持
算术移位<<<, >>>	完全支持
幂运算符 **	完全支持
sized parameters	完全支持
以名字相关传递的参数（parameter）	完全支持
本地参数（local parameter）	完全支持
编译制导**`ifndef, `elsif, `undef**	完全支持
自动化任务和函数（automatic tasks and functions）	完全支持

国科大EB2009计算机体系结构研讨课17-18秋季

Verilog 结构	描述
常数函数（constant functions）	完全支持
generate 表达式	完全支持

针对同学们完成课程实验的实际需要，我们对表 1-1 和表 1-2 的 Verilog 可综合语言要素做进一步精简，所有*斜体下划线标记*的部分也**禁止**同学们在 RTL 设计中使用。这里进一步禁止是为了避免作为初学者的同学们遇到一些“设计陷阱”。譬如 function、task、casex、casez 虽然是可综合的，但是如果使用不当，则容易出现前后仿真不一致的情况；又譬如 Verilog-2001 中的有符号数据类型虽然对于混合有大量有符号、无符号数运算的 datapath 设计很有帮助，但是如果对于运算表达式的符号判定规则不是特别明确，则容易出现设计错误。我们希望同学们首先把 Verilog 中最基本的可综合结构彻底掌握好以后，有余力的情况下再去掌握其它内容，在某些情况下提到编码效率。大家不要小看了这些最基本的，龙芯处理器 RTL 设计所用到的 Verilog 语言要素也没有超过我们推荐给同学们的范围。

1.5 推荐的可综合代码风格

上面介绍了 Verilog 语言的可综合语言要素。那么在进行课程实验的时候，我们对于如何使用这些语言要素写代码又有一个建议的代码风格，其中主要内容列举如下：

1. 建议对于数据通路（datapath）上的组合逻辑采用数据流建模的方法，通俗一点来说就是数据通路上的组合逻辑用 assign 语句写，禁止用 always 语句写。
2. 用 always 语句写组合逻辑只允许出现在生成状态机 next state 的时候，该语句中只允许出现阻塞赋值（=）。
3. 写时序逻辑的 always 语句中只允许出现非阻塞赋值（<=）。
4. 寄存器堆封装成单独的模块，以实例化方式使用。
5. case 语句任何情况下都要有 default 分支。
6. 模块实例化时的参数和端口只允许用名字相关的方式进行赋值和连接。
7. 数据通路的组合逻辑中，1 比特的逻辑运算用&、|、~、^这类位运算符；控制信号的组合逻辑中，1 比特的逻辑运算用&&、||、!这三个逻辑运算符。
8. 如果你对于表达式中运算符的优先级没有确切把握的话，要么立刻去找一本参考书明确它，要么按照你所需要的优先级层次加上括号；我们推荐前者，因为括号太多影响代码的可阅读性。
9. 不要在 RTL 代码中加入用于仿真的延迟信息；时序是否满足的问题通过综合阶段的 STA 保证，不

国科大B6209H计算机体系结构研讨课1/12秋季

是通过仿真来保证。

10. 禁用//synopsys parallel case、//synopsys full case。

1.6 Verilog 示例

在本小节中，我们将给同学们提供一些 Verilog 的示例。这些示例都是一些基本的电路结构，如译码器、编码器、多选一等。一个简单 CPU 中所需的电路逻辑，除去加法器、乘法器这些运算部件外，余下都可以分解为这些基本的电路结构。

模块声明和实例化

```
module bottom #
(
    parameter A_WIDTH = 8,
    parameter B_WIDTH = 4,
    parameter Y_WIDTH = 2
)
(
    input  wire [A_WIDTH-1:0] a,
    input  wire [B_WIDTH-1:0] b,
    input  wire [        3:0] c,
    output wire [Y_WIDTH-1:0] y,
    output reg                z
);
  ......
endmodule

module top;

wire [15:0] btm_a;
wire [ 7:0] btm_b;
wire [ 3:0] btm_c;
wire [ 3:0] btm_y;
wire        btm_z;

bottom #(
    .A_WIDTH (16),
    .B_WIDTH ( 8),
    .Y_WIDTH ( 4)
    )
   inst_btm(
    .a   (btm_a),  //I
```

国科大B6209H计算机体系结构研讨课17-18秋季

```
    .b   (btm_b),  //I
    .c   (btm_c),  //I
    .y   (btm_y),  //O
    .z   (btm_z)   //O
    );

endmodule
```

这个例子里面，主要提醒大家注意模块实例化时 port 连接要采用名字相关的方式。无论是声明还是实例化时，我倾向于每个 port 单独成行，这样便于代码维护。同时，如果你想写一个端口宽度可参数配置的模块，那么这里的代码也可以参考。所举的例子中采用了 ANSI 风格的 port 声明方式，这也是我们推荐的。

基础逻辑门

```
wire [7:0] a;
wire [7:0] b;

assign y1 = ~a;          //反相器
assign y2 = a & b;       //与
assign y3 = a | b;       //或
assign y4 = a ^ b;       //异或
assign y5 = ~(a & b);  //与非
assign y6 = ~(a | b);  //或非
```

译码器

3-8 译码器。

```
module decoder_3_8(
  input  [2:0] in,
  output [7:0] out
);
assign out[0] = (in == 3'd0);
assign out[1] = (in == 3'd1);
assign out[2] = (in == 3'd2);
assign out[3] = (in == 3'd3);
assign out[4] = (in == 3'd4);
assign out[5] = (in == 3'd5);
assign out[6] = (in == 3'd6);
assign out[7] = (in == 3'd7);
endmodule
```

国科大B6209H计算机体系结构研讨课17-18秋季

相信有了上面的例子，其它各种译码器你就都会写的。

编码器

8-3 编码器，写法一：

```
module encoder_8_3(
  input  [7:0] in,
  output [2:0] out
);
assign out = in[0] ? 3'd0 :
             in[1] ? 3'd1 :
             in[2] ? 3'd2 :
             in[3] ? 3'd3 :
             in[4] ? 3'd4 :
             in[5] ? 3'd5 :
             in[6] ? 3'd6 :
                     3'd7;
endmodule
```

上面这样写，功能上是没问题的。但是它实际上实现得有点冗余了，因为这其实是一个优先级编码器。如果你能保证设计输入 in 永远至多只有一个 1，即所谓 at-most-1-hot 向量，那么可以像下面这样写：

```
module encoder_8_3(
  input  [7:0] in,
  output [2:0] out
);
assign out = ({3{in[0]}} & 3'd0)
           | ({3{in[1]}} & 3'd1)
           | ({3{in[2]}} & 3'd2)
           | ({3{in[3]}} & 3'd3)
           | ({3{in[4]}} & 3'd4)
           | ({3{in[5]}} & 3'd5)
           | ({3{in[6]}} & 3'd6)
           | ({3{in[7]}} & 3'd7);
endmodule
```

多路选择器

我们先说 select 信号还没有译码的例子，而且这里故意将选择的路数设为不是 2 的幂次方，当 select 信号超出范围时输出全 0。

```
module mux5_8b(
  input  [7:0] in0, in1, in2, in3, in4,
  input  [2:0] sel,
  output [7:0] out
```

国科大B62009H计算机体系结构研讨课17-18秋季

```
);
assign out = (sel==3'd0) ?  in0 :
             (sel==3'd1) ?  in1 :
             (sel==3'd2) ?  in2 :
             (sel==3'd3) ?  in3 :
             (sel==3'd4) ?  in4 :
                            8'b0;
endmodule
```

同样，上面的例子引入了不必要的优先级关系，譬如 sel 是 1 的时候自然不是 0。所以，这个多路选择器还可以这样写。

```
module mux5_8b(
  input  [7:0] in0, in1, in2, in3, in4,
  input  [2:0] sel,
  output [7:0] out
);
assign out =  ({8{sel==3'd0}} &  in0)
            | ({8{sel==3'd1}} &  in1)
            | ({8{sel==3'd2}} &  in2)
            | ({8{sel==3'd3}} &  in3)
            | ({8{sel==3'd4}} &  in4);
endmodule
```

上面这种写法一定注意不要忘了写{8{}}中的 8，因为这种笔误工具不会报错，但是功能不对。

如果 select 信号已经是译码后的向量的形式，也很容易写出来。

```
module mux5_8b_1hot(
  input  [7:0] in0, in1, in2, in3, in4,
  input  [4:0] sel,
  output [7:0] out
);
assign out =  ({8{sel[0]}} &  in0)
            | ({8{sel[1]}} &  in1)
            | ({8{sel[2]}} &  in2)
            | ({8{sel[3]}} &  in3)
            | ({8{sel[4]}} &  in4);
endmodule
```

其实回顾一下前面译码器的写法。可以看到第二个例子是把译码器和基于译码后向量的多路选择器合并在一起写出来的。

简单 MIPS CPU 的 ALU

我们用一个简单 MIPS CPU 中的 ALU 作为一个较复杂的组合逻辑设计的例子。整个 ALU 的控制信号

国科大B6200H计算机体系结构研讨课17-18秋季

alu_control 采用 12 位的独热码，暂不考虑溢出例外的判定。

```
module alu(
    input  [11:0] alu_control,
    input  [31:0] alu_src1,
    input  [31:0] alu_src2,
    output [31:0] alu_result
);
wire alu_add;   //加法操作
wire alu_sub;   //减法操作
wire alu_slt;   //有符号比较，小于置位，复用加法器做减法
wire alu_sltu;  //无符号比较，小于置位，复用加法器做减法
wire alu_and;   //按位与
wire alu_nor;   //按位或非
wire alu_or;    //按位或
wire alu_xor;   //按位异或
wire alu_sll;   //逻辑左移
wire alu_srl;   //逻辑右移
wire alu_sra;   //算术右移
wire alu_lui;   //高位加载

assign alu_add  = alu_control[11];
assign alu_sub  = alu_control[10];
assign alu_slt  = alu_control[ 9];
assign alu_sltu = alu_control[ 8];
assign alu_and  = alu_control[ 7];
assign alu_nor  = alu_control[ 6];
assign alu_or   = alu_control[ 5];
assign alu_xor  = alu_control[ 4];
assign alu_sll  = alu_control[ 3];
assign alu_srl  = alu_control[ 2];
assign alu_sra  = alu_control[ 1];
assign alu_lui  = alu_control[ 0];

wire [31:0] add_sub_result;
wire [31:0] slt_result;
wire [31:0] sltu_result;
wire [31:0] and_result;
wire [31:0] nor_result;
wire [31:0] or_result;
wire [31:0] xor_result;
wire [31:0] sll_result;
wire [31:0] sr_result;
wire [31:0] lui_result;
```

国科大B62009H计算机体系结构研讨课17-18秋季

```
wire [63:0] sr64_result;

assign and_result = alu_src1 & alu_src2;
assign or_result  = alu_src1 | alu_src2;
assign nor_result = ~or_result;
assign xor_result = alu_src1 ^ alu_src2;
assign lui_result = {alu_src2[15:0], 16'd0};

wire [31:0] adder_a;
wire [31:0] adder_b;
wire        adder_c;
wire [31:0] adder_result;
wire        adder_cout   ;
assign adder_a    = alu_src1;
assign adder_b    = alu_src2 ^ {32{alu_sub}};
assign adder_cin = alu_sub;
assign {adder_cout, adder_result} = adder_a + adder_b + adder_cin;

assign add_sub_result = adder_result;

assign slt_result[31:1] = 31'd0;
assign slt_result[0]    = (alu_src1[31] & ~alu_src2[31])
                        | (~(alu_src1[31]^alu_src2[31]) & adder_result[31]);

assign sltu_result[31:1] = 31'd0;
assign sltu_result[0]    = ~adder_cout;

assign sll_result  = alu_src2 << alu_src1[4:0];

assign sr64_result = {{16{alu_sra&alu_src2[31]}}, alu_src2[31:0]} >> alu_src1[4:0];
assign sr_result   = sr64_result[31:0];

assign alu_result = ({32{alu_add|alu_sub}} & add_sub_result)
                  | ({32{alu_slt       }} & slt_result)
                  | ({32{alu_sltu      }} & sltu_result)
                  | ({32{alu_and       }} & and_result)
                  | ({32{alu_nor       }} & nor_result)
                  | ({32{alu_or        }} & or_result)
                  | ({32{alu_xor       }} & xor_result)
                  | ({32{alu_sll       }} & sll_result)
                  | ({32{alu_srl|alu_sra}} & sr_result)
                  | ({32{alu_lui       }} & lui_result);
endmodule
```

接下来，我们介绍一些时序逻辑相关的代码例子。

国科大B6209班计算机体系结构研讨课17-18秋季

寄存器

例子 1，最普通的上跳沿触发的 D 寄存器。

```
module Dflipflop(
    input          clk,
    input          din,
    output  reg    q
);
always @(posedge clk) begin
    q <= din;
end
endmodule
```

例子 2，带复位的 D 寄存器。写法一：

```
module Dflipflop_r(
    input          clk,
    input          rst,
    input          din,
    output  reg    q
);
always @(posedge clk) begin
    if (rst)  q <= 1'b0;
    else      q <= din;
end
endmodule
```

写法二：

```
module Dflipflop_r(
    input          clk,
    input          rst,
    input          din,
    output  reg    q
);
always @(posedge clk) begin
    q <= ~rst & din;
end
endmodule
```

上面的两种写法，对应的底层电路相同吗？这需要看最终实现的平台是否包含带复位端的寄存器这种类型的器件。如果所实现的平台没有带复位端的寄存器这类器件，那么两种写法最后的电路时一样的。也就是说，不是你采用了写法一，综合工具就一定能给你推导出带复位端的寄存器，如果只有 D 寄存器这类器件，那么最终的电路会和写法二中描述的逻辑相似，即 rst 信号是参与到 D 寄存器输入端组合逻辑生成中，

国科大B6209H计算机体系结构研讨课17-18秋季

当 rst 为 1 时该组合逻辑一定为 0。那么，哪一种好呢？我推荐写法一。原因两点：（1）rst 清 0 操作通常具有最高优先级，用 if…else…这种写法优先级一目了然，不容易犯错；（2）综合工具有机会尝试推导出带复位端的寄存器这类器件，从而节省逻辑资源。

例子 3，带使能端的 D 寄存器。写法一：

```
module Dflipflop_en(
    input          clk,
    input          en,
    input          din,
    output  reg    q
);
always @(posedge clk) begin
    if (en) q <= din;
end
endmodule
```

写法二：

```
module Dflipflop_en(
    input          clk,
    input          en,
    input          din,
    output  reg    q
);
always @(posedge clk) begin
    q <= en ? din : q;
end
endmodule
```

这个例子和上面的带复位端的例子有相似之处，两种写法是否会综合出相同的电路取决于最终实现平台上是否有带使能端的寄存器器件。其实，严格意义上讲，没有所谓带使能端的寄存器，而是有带门控时钟的寄存器器件，即仅当 en 信号有效时，寄存器内部的时钟才会开启，才会看到时钟沿，才能将 din 的输入值锁存起来。同样的问题，这两种写法都是对的，写那种好呢？我推荐前一种。原因有两个：（1）特别直观，一看就明白怎么回事；（2）使得综合工具有可能自动插入门控时钟或者引入带门控的寄存器。

这里举的例子都是单比特寄存器，同学们可以很自然地将其推广到多比特的情况。

MIPS CPU 中的寄存器堆

寄存器堆是采用二维组织形式的“一堆寄存器”。在一个单发射五级流水简单 MIPS CPU 中，GR 对应一个 32 项，每项 32 位的寄存器堆，同时为了支持流水，意味着寄存器堆要能支持每周期读出两个 32 位的数、写入一个 32 位的数。同时，MIPS 还有一个特殊之处，即 0 号寄存器恒为 0。

国科大62009H计算机体系结构研讨课17-18秋季

```
module regfile(
   input        clk,
   input  [ 4:0] raddr1,
   output [31:0] rdata1,
   input  [ 4:0] raddr2,
   output [31:0] rdata2,
   input        we,
   input  [ 4:0] waddr,
   input  [31:0] wdata
);
reg  [31:0] rf [31:0];
//WRITE
always @(posedge clk) begin
   if (we) rf[waddr] <= wdata;
end
//READ OUT 1
assign rdata1 = (raddr1==5'b0) ? 32'b0 : rf[raddr1];
//READ OUT 2
assign rdata2 = (raddr2==5'b0) ? 32'b0 : rf[raddr2];
endmodule
```

在上面的例子中，rf[waddr]、rf[raddr1]、rf[raddr2]这种写法，综合工具实际上会推导出针对写地址和读地址的译码电路，实际上，上面例子实际上对应下面的这种表达。这里我们假设 decoder_5_32 是一个 5-32 译码器的模块。

```
module regfile(
   ……
);
reg  [31:0] rf [31:0];
wire [31:0] waddr_dec, raddr1_dec, raddr2_dec;
decoder_5_32 U0(.in(waddr), .out(waddr_dec));
decoder_5_32 U1(.in(raddr1), .out(raddr1_dec));
decoder_5_32 U2(.in(raddr2), .out(raddr2_dec));

//WRITE
always @(posedge clk) begin
   if (we & waddr_dec[ 0]) rf[0] <= wdata;
   if (we & waddr_dec[ 1]) rf[1] <= wdata;
   ……
   if (wen & waddr_dec[31]) rf[31] <= wdata;
end
//READ OUT 1
assign rdata1 = ({32{raddr1_dec[ 1]}} & rf[ 1])
              | ({32{raddr1_dec[ 2]}} & rf[ 2])
```

国科大B62009H计算机体系结构研讨课17-18秋季

```
                ……
                | ({32{raddr1_dec[31]}} & rf[31]);
//READ OUT 2
assign rdata2 = ({32{raddr2_dec[ 1]}} & rf[ 1])
                | ({32{raddr2_dec[ 2]}} & rf[ 2])
                ……
                | ({32{raddr2_dec[31]}} & rf[31]);
endmodule
```

两种写法，第二种只是为了加深各位同学对于 rf[addr]这种写法的理解，要知道虽然代码很短，但是背后的逻辑量是很大的。在平时的设计中我们推荐各位采用第一种写法。

这里顺带问一个问题，如果写有效时（we=1），写地址和读地址相同时，这一拍读出的结果是寄存器中的旧值还是新写入的值？各位不知道有没有看过 Patterson 的《计算组成与设计》那本教材，那里面在五级流水设计中提到了寄存器堆前半周期写、后半周期读，很显然我们的寄存器堆没有这个特性。所以记住，如果你按照那本教材里面的说法去添加 forward 逻辑，会有问题。后面做到五级流水实验的时候，我还会再提起这个问题，所以这里请留意。

流水线

首先要说明的一点是，流水线电路各位在数字电路课程中就应该学习的，这不是一个需要放到组成原理或体系结构课程才开始说明的概念。我们先来看一个完全不会被阻塞、停顿的流水线电路怎么写。

```
module non_stall_pipeline #
(
    parameter WIDTH = 100
)
(
    input              clk,
    input  [WIDTH-1:0] datain,
    output [WIDTH-1:0] dataout
);
reg  [WIDTH-1:0] pipe1;
reg  [WIDTH-1:0] pipe2;
reg  [WIDTH-1:0] pipe3;

always @(posedge clk) begin
    pipe1 <= datain;
end

always @(posedge clk) begin
    pipe2 <= pipe1;
end
```

国科大B6209H计算机体系结构研讨课17-18秋季

```
always @(posedge clk) begin
   pipe3 <= pipe2;
end

assign dataout = pipe3;
endmodule
```

实际电路设计中，并不总是上述这种无阻塞、停顿的流水线。下面给出一个具有阻塞、停顿功能的流水线的代码。

```
module stall_pipeline #
(
   parameter WIDTH = 100
)
(
   input              clk,
   input              rst,
   input              validin,
   input  [WIDTH-1:0] datain,
   input             out_allow,
   output            validout,
   output [WIDTH-1:0] dataout
);
reg             pipe1_valid;
reg  [WIDTH-1:0] pipe1_data;
reg             pipe2_valid;
reg  [WIDTH-1:0] pipe2_data;
reg             pipe3_valid;
reg  [WIDTH-1:0] pipe3_data;

// pipeline stage 1
wire           pipe1_allowin;
wire           pipe1_ready_go;
wire           pipe1_to_pipe2_valid;
assign pipe1_ready_go = ……;
assign pipe1_allowin = !pipe1_valid || pipe1_ready_go && pipe2_allowin;
assign pipe1_to_pipe2_valid = pipe1_valid && pipe1_ready_go;
always @(posedge clk) begin
   if (rst) begin
      pipe1_valid <= 1'b0;
   end
   else if (pipe1_allowin) begin
      pipe1_valid <= validin;
   end
```

国科大B6200H计算机体系结构研讨课17-18秋季

```
    if (validin && pipe1_allowin) begin
        pipe1_data <= datain;
    end
end

// pipeline stage 2
wire          pipe2_allowin;
wire          pipe2_ready_go;
wire          pipe2_to_pipe3_valid;
assign pipe2_ready_go = ……;
assign pipe2_allowin = !pipe2_valid || pipe2_ready_go && pipe3_allowin;
assign pipe2_to_pipe3_valid = pipe2_valid && pipe2_ready_go;
always @(posedge clk) begin
    if (rst) begin
        pipe2_valid <= 1'b0;
    end
    else if (pipe2_allowin) begin
        pipe2_valid <= pipe1_to_pipe2_valid;
    end
    if (pipe1_to_pipe2_valid && pipe2_allowin) begin
        pipe2_data <= pipe1_data;
    end
end

// pipeline stage 3
wire          pipe3_allowin;
wire          pipe3_ready_go;
wire          pipe2_to_pipe3_valid;
assign pipe3_ready_go = ……;
assign pipe3_allowin = !pipe3_valid || pipe3_ready_go && out_allow;
always @(posedge clk) begin
    if (rst) begin
        pipe3_valid <= 1'b0;
    end
    else if (pipe3_allowin) begin
        pipe3_valid <= pipe2_to_pipe3_valid;
    end
    if (pipe2_to_pipe3_valid && pipe3_allowin) begin
        pipe3_data <= pipe2_data;
    end
end

assign validout = pipe3_valid && pipe3_ready_go;
assign dataout = pipe3_data;
```

国科大B62009计算机体系结构研讨课17-18秋季

```
endmodule
```

上面的写法，是把流水线的 control logic 和 data path 写到了一个模块内部。你也可以把两者分开来写。教科书上在描述流水线 CPU 时，通常会在流水线之外画一个叫作 pipeline control logic 的椭圆形框，从这个框引出控制信号去往各级流水所在触发器的使能端。这就是把 control logic 和 data path 分离的写法。

这里面 pipeX_valid 为 1 表示第 X 级流水上当前存有有效数据，为 0 表示第 X 级是空的。引入控制位的好处是，清空流水线的时候不用刷各级流水线 data 域的值，可以节约逻辑资源，但是需要注意的是如果根据流水级的 data 域信息产生控制信号时，有时候不要忘了考虑这一级的 valid 信号。

pipeX_allowin 为 1 表示第 X 级可以接收前一级流水送来的数据。

pipeX_ready_go 为 1 表示第 X 级的信息可以传递到后一级流水级了。例如，假设在执行流水级用迭代法运算除法，需要多拍才能完成，那么在迭代没有完成前，执行流水级的_ready_go 将置为 0。

我这里给出的写法在关键控制信号上采用了链式的写法。这样写的好处是，如果发生流水级功能调整时，譬如将一级切为两级，或者某一级的处理周期数改变了，只需要修改局部，不会涉及全局性的调整。有的同学可能会觉得我这样写很麻烦，觉得自己写的更简单一些。请你们针对具体情况分析一下，是不是对于你的流水线设计，我上面的代码中，有些 pipeX_allowin 和 pipeX_ready_go 恒为 1，把级联的控制信号自行展开并把这些 1 带入化简，是不是就是你写的控制信号。

1.7 小结

基于我们在工程实践中的一些体会，我们给同学们讲述了一些 Verilog 使用中的注意事项，并且给出了一些典型电路的代码实例，供同学们参考。

国科大B5209H计算机体系结构设计课17-18秋季