**深 圳 大 学 实 验 报 告**

**课 程 名 称： 计算机系统(3)**

**实验项目名称： 加法器和乘法器实验**

**学 院： 计算机与软件学院**

**专 业： 计算机与软件学院所有专业**

**指 导 教 师： 罗秋明**

**报告人： 刘俊楠 学号： 2017303010 班级： 01**

**实 验 时 间： 2021.11.12**

**实验报告提交时间： 2021.11.12**

**教务处制**

## 实验目的

了解chisel编程语言，掌握FPGA开发工具vivado的基本操作，能够完成简单组合逻辑的设计。

### 工作

1. 给出加法器的chisel文件，并配备简要说明文档。并准备两个现成文档：给出chisel语言的规范（两页纸那个）和chisel的入门pdf作为辅助阅读。
2. 设计一个两位乘法器
3. 给出vivado的操作步骤文档，并说明每隔步骤完成什么工作。这个请截屏并撰写说明文档。
4. 准备mini chisel开发环境的容器或虚拟机镜像一个，准备vivado的虚拟机镜像一个，放到我们校内服务器上。使得学生直接下载镜像后就可以开展实验。

学生要完成一个ALU的工作，我们需要设计该部件的外部引脚和功能要求。初步定为能做加法、减法、高低位互换三个功能。我们提供实现样例

\*\*逻辑部件，功能测试的输入用按钮，输出用LED灯

## 一、 实验环境的搭建

操作系统：Ubuntu 20.4 LTS

**1. 安装JDK、git、make、gtkwave、verilator、curl**

sudo apt install default-jre git make gtkwave verilator curl

**2. 安装sbt**

echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" | sudo tee /etc/apt/sources.list.d/sbt.list  
echo "deb https://repo.scala-sbt.org/scalasbt/debian /" | sudo tee /etc/apt/sources.list.d/sbt\_old.list  
curl -sL "https://keyserver.ubuntu.com/pks/lookup?op=get&search=0x2EE0EA64E40A89B84B2DF73499E82A75642AC823" | sudo apt-key add  
sudo apt-get update  
sudo apt-get install sbt

**3.修改sbt源**

由于众所周知的原因，sbt默认源在国内使用时下载资源非常缓慢，并且在后续的使用中时常无法访问导致问题的出现。因此在这里将sbt默认的源修改为[国内镜像](https://www.huaweicloud.com/zhishi/SBT.html)以提高访问速度。

创建 ~/.sbt/repositories 目录与文件并将如下内容写入到repositories中

[repositories]  
local  
maven-aliyun-public: https://maven.aliyun.com/repository/public  
maven-aliyun-central: https://maven.aliyun.com/repository/central  
ivy-huawei: https://repo.huaweicloud.com/repository/ivy/, [organization]/[module]/(scala\_[scalaVersion]/)(sbt\_[sbtVersion]/)[revision]/[type]s/[artifact](-[classifier]).[ext]  
typesafe: https://repo.typesafe.com/typesafe/ivy-releases/, [organization]/[module]/(scala\_[scalaVersion]/)(sbt\_[sbtVersion]/)[revision]/[type]s/[artifact](-[classifier]).[ext], bootOnly  
sonatype-oss-releases  
maven-central  
sonatype-oss-snapshots

**4.验证开发环境**

获取[chisel-template](https://github.com/freechipsproject/chisel-template)，其中包括了chisel开发基本的目录结构以及配置文件([build.sbt](https://github.com/freechipsproject/chisel-template/blob/main/build.sbt))，并且包含了一个GCD模块的示例代码以及对应的测试模块([ChiselTest](https://www.chisel-lang.org/chiseltest/)，这里不涉及)。我们可以基于这个基本的项目文件来开发自己的模块。

git clone https://github.com/freechipsproject/chisel-template.git

在chisel-template目录中执行如下命令

sbt test

当看到终端上显示如下信息时，即代表开发环境配置无误

[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0  
[info] All tests passed.  
[success] Total time: 5 s, completed Dec 16, 2020 12:18:44 PM

## 二、ALU的实现

参考资料：

[Digital Design with Chisel](https://github.com/schoeberl/chisel-book)

[chisel-cheatsheet](https://github.com/freechipsproject/chisel-cheatsheet/releases)

[Scala Document](https://docs.scala-lang.org/zh-cn/tour/tour-of-scala.html)

[Chisel教程汇总-CSDN](https://blog.csdn.net/qq_34291505/article/details/86744581)

在这个部分我们实现一个简单的ALU，其功能是对两个32整数进行操作，并输出一个32位整数。这里支持的操作有加法、减法以及乘法。

想要实现一个硬件模块，首先需要确定其输入输出端口。这这个ALU中需要两个32位信号以及一个2位的控制信号作为输入，输出一个32位信号。

                ALU\_OP  
                  │  
                  │2bit  
                  │  
          ┌───────▼───────┐  
  32bit   │               │  
A──────────►│               │ 32bit  
          │     ALU       ├───────►out  
  32bit   │               │  
B──────────►│               │  
          └───────────────┘

由于Chisel是基于Scala语言实现的，因此首先需要导入chisel3包。

// src/main/scala/ALU/ALU.scala  
package ALU  
​  
import chisel3.\_

首先定义ALU的操作码，在这里可以使用一个[单例对象](https://docs.scala-lang.org/zh-cn/tour/singleton-objects.html)来存储其对应关系，代码如下。

object ALUConst {  
 val ALU\_ADD = 0.U(2.W)  
 val ALU\_SUB = 1.U(2.W)  
 val ALU\_MUL = 2.U(2.W)  
}

随后需要定义模块的端口，在Chisel中定义端口需要一个[Bundle](https://www.chisel-lang.org/api/3.4.3/chisel3/Bundle.html)类型(作用类似于C中的结构)的对象。在这里定义一个继承自Bundle的类，类成员包含所需要的三个输入端口以及一个输出端口。

class ALUIO extends Bundle{  
 val A = Input(UInt(32.W))  
 val B = Input(UInt(32.W))  
 val alu\_op = Input(UInt(2.W))  
 val out = Output(UInt(32.W))  
}

最后定义我们的ALU模块，Chisel中的模块继承自Module类。在模块内部首先定义io，这里传入上面定义的ALUIO。随后根据输入数据以及操作码确定输出数据。这里使用Chisel提供的[MuxLookup](https://www.chisel-lang.org/api/3.4.3/chisel3/util/MuxLookup$.html)来完成，其功能就是一个多路选择器，根据传入的alu\_op来选择输出的数据。若要使用MuxLookup首先需要引入chisel3.util.MuxLookup 包。这里推荐使用[IntelliJ IDEA](https://www.jetbrains.com/idea/)进行开发，配合[scala](https://plugins.jetbrains.com/plugin/1347-scala)插件可实现自动导入包以及代码补全的功能，很方便。

import chisel3.\_  
import chisel3.util.MuxLookup

MuxLookup的第一个参数为key，第二个参数为默认值，第三个参数为一个映射列表。ALU模块的代码如下。注意这里的乘法操作在Chisel中直接使用"\*"运算符来进行乘法操作，最终"\*"运算在综合时会生成为一个由组合逻辑构成的单周期的乘法器，注意与使用移位和加法操作组成的多周期乘法器做区别。

import ALUConst.\_ //引入ALUConst以便于使用其中定义的变量  
class ALU extends Module{  
 val io = IO(new ALUIO)  
  
 io.out := MuxLookup(io.alu\_op,0.U,Seq( //default = 0.U  
 ALU\_ADD -> (io.A + io.B),  
 ALU\_SUB -> (io.A - io.B),  
 ALU\_MUL -> (io.A \* io.B)  
 ))  
}

完成ALU模块的设计以后即可生成对应的Verilog文件。首先创建一个继承自App类的Main对象，并在当中执行如下函数即可，注意这里传入ALU对象。

// src/main/scala/ALU/Main.scala  
package ALU  
  
object Main extends App {  
 (new chisel3.stage.ChiselStage).emitVerilog(new ALU)  
}

在终端中执行sbt run即可生成对应的Verilog文件。

szu@szu-VirtualBox:~/chisel-template$ sbt run  
[info] welcome to sbt 1.4.9 (Ubuntu Java 11.0.11)  
[info] loading settings for project chisel-template-build from plugins.sbt ...  
[info] loading project definition from /home/szu/chisel-template/project  
[info] loading settings for project root from build.sbt ...  
[info] set current project to %NAME% (in build file:/home/szu/chisel-template/)  
[info] compiling 2 Scala sources to /home/szu/chisel-template/target/scala-2.12/classes ...  
[info] running ALU.Main   
Elaborating design...  
Done elaborating.  
[success] Total time: 19 s, completed 2021年9月3日 下午1:45:45

工程目录下的ALU.v文件即为生成的Verilog文件

module ALU(  
 input clock,  
 input reset,  
 input [31:0] io\_A,  
 input [31:0] io\_B,  
 input [1:0] io\_alu\_op,  
 output [31:0] io\_out  
);  
 wire [31:0] \_io\_out\_T\_1 = io\_A + io\_B; // @[ALU.scala 25:24]  
 wire [31:0] \_io\_out\_T\_3 = io\_A - io\_B; // @[ALU.scala 26:24]  
 wire [63:0] \_io\_out\_T\_4 = io\_A \* io\_B; // @[ALU.scala 27:24]  
 wire [31:0] \_io\_out\_T\_6 = 2'h0 == io\_alu\_op ? \_io\_out\_T\_1 : 32'h0; // @[Mux.scala 80:57]  
 wire [31:0] \_io\_out\_T\_8 = 2'h1 == io\_alu\_op ? \_io\_out\_T\_3 : \_io\_out\_T\_6; // @[Mux.scala 80:57]  
 wire [63:0] \_io\_out\_T\_10 = 2'h2 == io\_alu\_op ? \_io\_out\_T\_4 : {{32'd0}, \_io\_out\_T\_8}; // @[Mux.scala 80:57]  
 assign io\_out = \_io\_out\_T\_10[31:0]; // @[ALU.scala 24:12]  
endmodule

## 三、乘法器的实现

在该部分通过移位和加法操作实现一个乘法器，接口定义如下。由于该乘法器并非组合电路，需要多个周期才能计算出一个结果。因此对于输入输入分别增加的Valid信号以表示计算过程的开始和完成。这里定义xlen表示乘法器的位数。

object MulConst{  
 val xlen = 4  
}  
  
import MulConst.\_  
  
class MultiplierIO extends Bundle{  
 val multiplier = Input(UInt(xlen.W))  
 val multiplicand = Input(UInt(xlen.W))  
 val product = Output(UInt((xlen\*2).W))  
 //valid signal  
 val inputValid = Input(Bool())  
 val outputValid = Output(Bool())  
}

在乘法器内部需要准备四个寄存器，分别用于存储被乘数、乘数、结果以及一个计数。计数寄存器在完成计数的同时也表明了乘法器输入输出的状态，当计数值为0时候表示一次乘法完成，计算的结果已经输出并且可以接收新的输入。在Chisel可以使用 RegInit()方法来定义一个具有初始值的寄存器。

val multiplierReg = RegInit(0.U(xlen.W))  
val multiplicandReg = RegInit(0.U((xlen\*2).W))  
val productReg = RegInit(0.U((xlen\*2).W))  
val cntReg = RegInit(0.U)

进行乘法的过程代码如下，当计数值为0时可接收新的运算数，当io.inputValid为真时对各寄存器进行初始化。初始化完成后正式开始计算，根据除数寄存器的最低位判断是否需要执行加法运算，同时更新计数值以及对被乘数寄存器和乘数寄存器进行移位操作。

when(cntReg =/= 0.U){  
 when(multiplierReg(0) === 1.U){  
 productReg := productReg + multiplicandReg  
 }  
 multiplierReg := multiplierReg >> 1.U  
 multiplicandReg := multiplicandReg << 1.U  
 cntReg := cntReg - 1.U  
}.elsewhen(cntReg === 0.U){  
 when(io.inputValid){  
 multiplicandReg := Cat(Fill(xlen,0.U),io.multiplicand)  
 multiplierReg := io.multiplier  
 productReg := 0.U  
 cntReg := xlen.U  
 }  
}

对于输出端口的值，当计数值为0时表示结果运算完成，并且productReg中的数值即为结果。

io.outputValid := (cntReg === 0.U)  
io.product := productReg

## 四、Verilator仿真

[Verilator](https://www.veripool.org/verilator/)是一个开源的硬件仿真器，其原理是将Verilog源码编译成单/多线程的C++源代码来进行仿真，其将所需要仿真的DUT(device under test)编译为一个类，DUT的IO口则被编译为类成员。Verilator不单单只是简单的把Verilog编译为C++，Verilator还会将代码进行优化，编译成优化过的C++模型来进行仿真。

参考资料：

[Verilator User’s Guide](https://verilator.org/guide/latest/index.html)

[跨语言的Verilator仿真：使用进程间通信](http://www.sunnychen.top/2019/07/25/%E8%B7%A8%E8%AF%AD%E8%A8%80%E7%9A%84Verilator%E4%BB%BF%E7%9C%9F%EF%BC%9A%E4%BD%BF%E7%94%A8%E8%BF%9B%E7%A8%8B%E9%97%B4%E9%80%9A%E4%BF%A1/)

[Verilator仿真器入门](https://hubohan.space/2020/08/30/Verilator_note/)

[复旦大学体系机构课程实验——Verilator仿真](https://fducslg.github.io/ICS-2021Spring-FDU/)

### 1、编译C++等价类模型

通过chisel即可生成对应的Verilog源码，要使用Verilator进行仿真首先即要将Verilog源码编译成等价的C++类模型。这里以刚生成的ALU.v为例。执行如下指令即可将chisel生成的ALU.v转换为等价的C++类模型。执行完毕后会在当先目录下生成obj\_dir的文件夹，里面含有编译得到的头文件和C++源文件。

# --cc 后跟所需要编译的verilog源代码文件   
# --trace 添加生成.vcd波形文件的功能   
verilator --cc ALU.v --trace

通过查看生成的VALU.h即可查看模块对应的端口

szu@szu-VirtualBox:~/chisel-template$ cat obj\_dir/VALU.h

可以发现端口包含了io\_alu\_op、io\_A、io\_B、io\_out以及chisel自动加入的clock和reset信号(在该模块中未使用)

…………  
// PORTS  
// The application code writes and reads these signals to  
// propagate new values into/out from the Verilated model.  
VL\_IN8(clock,0,0);  
VL\_IN8(reset,0,0);  
VL\_IN8(io\_alu\_op,1,0);  
VL\_IN(io\_A,31,0);  
VL\_IN(io\_B,31,0);  
VL\_OUT(io\_out,31,0);  
…………

### 2、编写C++ harness

Verilator通过C++ harness文件来规定仿真的过程。对于一个基本的仿真需要包含如下几个部分。

1. 包含Verilator核心头文件：verilated.h。如果要加上生成.vcd波形文件的支持，还需要包含verilated\_vcd\_c.h。同时将需要进行仿真的Verilog代码编译得到的头文件包含进来，如这里的VALU.h。
2. 初始化模块对象以及波性文件对象，在这里分别对应VALU以及VALUC对象。并对Verilator做适当的初始化工作。
3. 编写仿真流程，其核心则是DUT顶层对象的eval()方法。
4. 清理工作，包括指针指向空间释放以及文件流关闭等工作。

编写harness.cpp，其功能是分别给ALU模块的端口a与端口b赋值0-10，alu\_op赋值0-3

// chisel-template/harness.cpp  
  
#include <verilated.h> // 核心头文件  
#include <verilated\_vcd\_c.h> // 波形生成头文件  
#include <iostream>  
#include <fstream>  
#include "VALU.h" // ALU模块类  
using namespace std;  
  
VALU \*top; // 顶层dut对象指针  
VerilatedVcdC \*tfp; // 波形生成对象指针  
  
vluint64\_t main\_time = 0; // 仿真时间戳  
  
int main(int argc, char \*\*argv){  
 // 一些初始化工作  
 Verilated::commandArgs(argc, argv);  
 Verilated::traceEverOn(true);  
  
 // 为对象分配内存空间  
 top = new VALU;  
 tfp = new VerilatedVcdC;  
  
 // tfp初始化工作  
 top->trace(tfp, 99);  
 tfp->open("VALU.vcd");  
  
 // 仿真过程  
 top->reset = 0;  
 for (int a = 0; a <= 10; ++a){  
 for (int b = 0; b <= 10; ++b){  
 for (int op = 0; op <= 3; ++op){  
 top->io\_A = a;  
 top->io\_B = b;  
 top->io\_alu\_op = op;  
 top->eval(); // 仿真时间步进  
 tfp->dump(main\_time); // 波形文件写入步进  
 main\_time++;  
 }  
 }  
 }  
   
 // 清理工作  
 tfp->close();  
 delete top;  
 delete tfp;  
 exit(0);  
 return 0;  
}

### 3、执行仿真

执行如下指令，即可在当前目录下生成VALU.vcd波行文件，使用gtkwave打开波形文件即可观察仿真产生的波形。

verilator --cc ALU.v --trace --exe harness.cpp   
make -j -C ./obj\_dir -f VALU.mk VALU   
./obj\_dir/VALU

可以从波形中观察到输入输出的情况，当alu\_op变化时io\_out便会产生相应的结果。当alu\_op为3时候输出default(0.U)

## 五、对乘法器进行仿真

我们编写的乘法器与前面编写的ALU所不同，是一个时序电路，因此在仿真过程当中还需要涉及到对时钟信号的设定。这里需要注意的是寄存器会在时钟的上升沿更新数值，因此在仿真过程中给定信号需要在时钟信号处于低电频时设置，以免造成输入数据的不稳定。

/\*\*  
 \* +--1--+ +--1--+ +--1--+  
 \* A | B | A | B | A | B | A  
 \* clk --0--+ +--0--+ +--0--+ +--0--  
 \* ---------->|---------->|---------->|---->  
 \* tick() tick() tick() ...  
 \* ---------------------------------->| ...  
 \* ticks(3)  
 \*/

对于乘法器，可以使用如下的harness代码进行仿真验证。这里定义了一个test函数用于设定每个时钟周期的输入数值，并且记录其状态。仿真完成后通过波形图即可观察模块是否正确给出了计算的结果。

#include <verilated.h> // 核心头文件  
#include <verilated\_vcd\_c.h> // 波形生成头文件  
#include <iostream>  
#include <fstream>  
#include "VMultiplier.h"  
using namespace std;  
  
void test(int multiplier, int multiplicand, bool inputValid);  
  
VMultiplier \*top;  
VerilatedVcdC \*tfp;  
int main\_time = 0;  
  
int main(int argc, char const \*argv[])  
{  
 Verilated::commandArgs(argc, argv);  
 Verilated::traceEverOn(true);  
  
 top = new VMultiplier;  
 tfp = new VerilatedVcdC;  
  
 top->trace(tfp, 99);  
 tfp->open("VMultiplier.vcd");  
  
 for (int i = 0; i < 16; i++)  
 {  
 for (int j = 0; j < 16; j++)  
 {  
 test(i, j, true);  
 while (!top->io\_outputValid)  
 {  
 test(0, 0, false);  
 }  
 }  
 }  
  
 tfp->close();  
 delete top;  
 delete tfp;  
 exit(0);  
 return 0;  
}  
  
void test(int multiplier, int multiplicand, bool inputValid)  
{  
 top->clock = 0;  
 top->io\_multiplier = multiplier;  
 top->io\_multiplicand = multiplicand;  
 top->io\_inputValid = inputValid;  
 top->eval();  
 tfp->dump(main\_time++);  
  
 top->clock = 1;  
 top->eval();  
 tfp->dump(main\_time++);  
}

## 六、FPGA烧写

在这个部分当中将我们编写的乘法器硬件烧写到FPGA中运行。在这个过程当中需要使用vivado，可以[xilinx官网](https://www.xilinx.com/support/download.html)中下载到vivado程序。vivado的安装过程可以参考 [A6 - Vivado 安装说明.pdf](A6%20-%20Vivado%20%E5%AE%89%E8%A3%85%E8%AF%B4%E6%98%8E.pdf) 或《[Installing Vivado, Vitis, and Digilent Board Files](https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-vitis)》，选择最新版本安装即可。vivado安装完成后需要[载入FPGA板对应的配置文件](https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-vitis#install_digilent_s_board_files)，将 [vivado-boards-Microblaze-MIG.zip](https://github.com/Digilent/vivado-boards/archive/refs/heads/Microblaze-MIG.zip?_ga=2.138338544.1162580199.1633153410-273144726.1633153410) 压缩包中new文件夹中的文件复制到vivado安装目录中(默认为C:/Xilinx/Vivado)的<version>/data/boards/board\_files文件夹(若没有则自行创建)当中即可。

vivado安装完成后我们即可进行烧写。首先创建一个新的工程项目，用于将我们编写的乘法器烧写到FPGA当中。这个过程可以参考 [A7 - Vivado 使用说明.pdf](A7%20-%20Vivado%20%E4%BD%BF%E7%94%A8%E8%AF%B4%E6%98%8E.pdf)

在新建项目的过程当中选择我们手上的FPGA板型号ARTY A7 35。

添加sources时添加由chisel生成的Verilog文件。

对于约束文件，我们新建一个约束文件以待稍后编辑。

接下来就需要编写约束文件，约束文件的作用在于确定我们自己编写模块的输入输出端口与FPGA板上端口的对应关系。我们可以基于[Arty-A7-35-Master.xdc](https://github.com/Digilent/digilent-xdc/blob/master/Arty-A7-35-Master.xdc)

以及[Arty A7 Schematic](https://digilent.com/reference/programmable-logic/arty-a7/schematic)进行编写。我们只需要在Arty-A7-35-Master.xdc中选择我们需要使用到的端口然后将其对应到我们自己编写模块的端口上即可。

在这个部分当中使用如下的约束文件，由于FPGA板上只有四个开关因此在这里将被乘数以及乘数输入的高2位映射到IO0-IO3端口上并且固定为上拉信号(高电平1)，而低2位则分别映射到四个开关当中。同时将乘法器的输出映射到8个LED当中，即可通过LED的亮灭来观察到乘法的结果。对于valid信号，输入valid对应到按钮1，而输出信号在这里并非必须，因此映射到IO4端口上。reset映射到按钮0中。

## Clock signal  
set\_property -dict { PACKAGE\_PIN E3 IOSTANDARD LVCMOS33 } [get\_ports { clock }];   
create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports { clock }];  
  
## Switches  
set\_property -dict { PACKAGE\_PIN A8 IOSTANDARD LVCMOS33 } [get\_ports { io\_multiplier[0] }]; #IO\_L12N\_T1\_MRCC\_16 Sch=sw[0]  
set\_property -dict { PACKAGE\_PIN C11 IOSTANDARD LVCMOS33 } [get\_ports { io\_multiplier[1] }]; #IO\_L13P\_T2\_MRCC\_16 Sch=sw[1]  
set\_property -dict { PACKAGE\_PIN C10 IOSTANDARD LVCMOS33 } [get\_ports { io\_multiplicand[0] }]; #IO\_L13N\_T2\_MRCC\_16 Sch=sw[2]  
set\_property -dict { PACKAGE\_PIN A10 IOSTANDARD LVCMOS33 } [get\_ports { io\_multiplicand[1] }]; #IO\_L14P\_T2\_SRCC\_16 Sch=sw[3]  
  
## RGB LEDs  
set\_property -dict { PACKAGE\_PIN E1 IOSTANDARD LVCMOS33 } [get\_ports { io\_product[0] }]; #IO\_L18N\_T2\_35 Sch=led0\_b  
set\_property -dict { PACKAGE\_PIN G4 IOSTANDARD LVCMOS33 } [get\_ports { io\_product[1] }]; #IO\_L20P\_T3\_35 Sch=led1\_b  
set\_property -dict { PACKAGE\_PIN H4 IOSTANDARD LVCMOS33 } [get\_ports { io\_product[2] }]; #IO\_L21N\_T3\_DQS\_35 Sch=led2\_b  
set\_property -dict { PACKAGE\_PIN K2 IOSTANDARD LVCMOS33 } [get\_ports { io\_product[3] }]; #IO\_L23P\_T3\_35 Sch=led3\_b  
  
## LED  
set\_property -dict { PACKAGE\_PIN H5 IOSTANDARD LVCMOS33 } [get\_ports { io\_product[4] }]; #IO\_L24N\_T3\_35 Sch=led[4]  
set\_property -dict { PACKAGE\_PIN J5 IOSTANDARD LVCMOS33 } [get\_ports { io\_product[5] }]; #IO\_25\_35 Sch=led[5]  
set\_property -dict { PACKAGE\_PIN T9 IOSTANDARD LVCMOS33 } [get\_ports { io\_product[6] }]; #IO\_L24P\_T3\_A01\_D17\_14 Sch=led[6]  
set\_property -dict { PACKAGE\_PIN T10 IOSTANDARD LVCMOS33 } [get\_ports { io\_product[7] }]; #IO\_L24N\_T3\_A00\_D16\_14 Sch=led[7]  
  
## Buttons  
set\_property -dict { PACKAGE\_PIN D9 IOSTANDARD LVCMOS33 } [get\_ports { reset }]; ] #IO\_L6N\_T0\_VREF\_16 Sch=btn[0]  
set\_property -dict { PACKAGE\_PIN C9 IOSTANDARD LVCMOS33 } [get\_ports { io\_inputValid }]; #IO\_L11P\_T1\_SRCC\_16 Sch=btn[1]   
  
## PULLUP  
set\_property -dict { PACKAGE\_PIN V15 IOSTANDARD LVCMOS33 PULLUP true} [get\_ports { io\_multiplier[2] }];#IO\_L16P\_T2\_CSI\_B\_14 Sch=ck\_io[0]  
set\_property -dict { PACKAGE\_PIN U16 IOSTANDARD LVCMOS33 PULLUP true} [get\_ports { io\_multiplier[3] }];#IO\_L18P\_T2\_A12\_D28\_14 Sch=ck\_io[1]  
set\_property -dict { PACKAGE\_PIN P14 IOSTANDARD LVCMOS33 PULLUP true} [get\_ports { io\_multiplicand[2] }];#IO\_L8N\_T1\_D12\_14 Sch=ck\_io[2]  
set\_property -dict { PACKAGE\_PIN T11 IOSTANDARD LVCMOS33 PULLUP true} [get\_ports { io\_multiplicand[3] }];#IO\_L19P\_T3\_A10\_D26\_14 Sch=ck\_io[3]  
  
## io\_outputValid  
set\_property -dict { PACKAGE\_PIN R12 IOSTANDARD LVCMOS33 } [get\_ports { io\_outputValid }];#IO\_L5P\_T0\_D06\_14 Sch=ck\_io[4]

约束文件配置完成后即点击generate bitstream进行综合、实现、产生比特流。生成比特流完成后将FPGA板连接到电脑上并且在vivado上进行连接。点击program device将我们生成的比特流文件(一般在\*.runs/impl\_1目录下)烧入到FPGA中。

烧写完成后即可通过波动开关设置输入值，设置完成后按下按钮1表示input valid，随后即可从led灯上看到乘法器的计算结果。

首先将开关全部拨到下方表示0，这样子按照我们之前高2位为0的情况表示1100b \* 1100b，按下按钮1，随后即可看到LED7与LDE4亮起，表示输出结果的第4和7位为1，即结果为10010000b，等于1100b \* 1100b的结果。

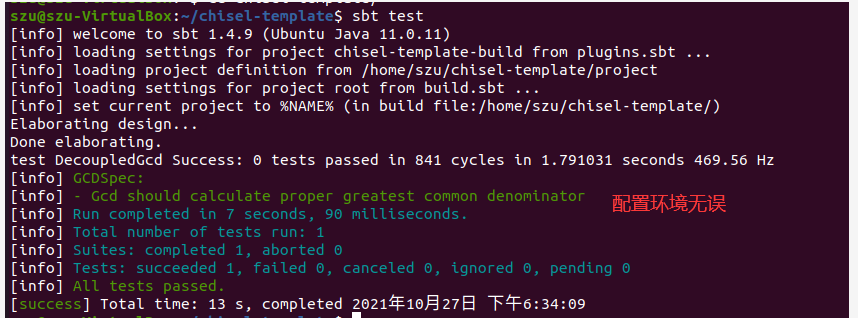
随后将按钮配置成1001，意味着输入1110b\*1101b，按下按钮1，即可观察到输出为 1011 0110，符合预期。

**五、实验结果**

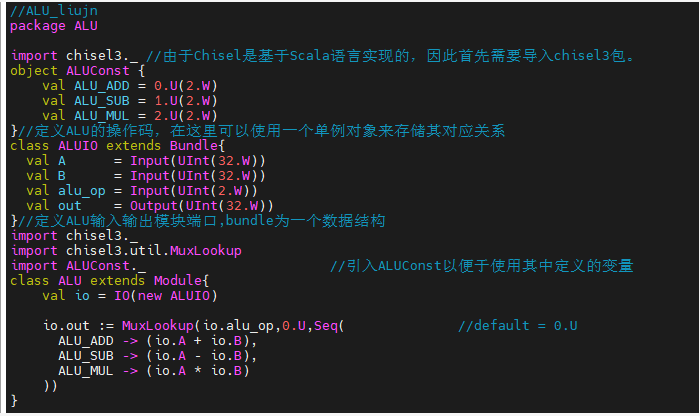
1、下载课程所给的虚拟机文件，安装完成后检查各个依赖版本



2、运行sbt test 检查配置环境是否安装完成



3、根据实验指导所给的代码，在对应位置写下ALU加法的代码（在mul.cpp文件中）。



4、在实验中遇到了问题，分析后发现

**六、实验总结与体会**

|  |
| --- |
| **指导教师批阅意见：**  **成绩评定：**  指导教师签字：  年 月 日 |
| 备注： |