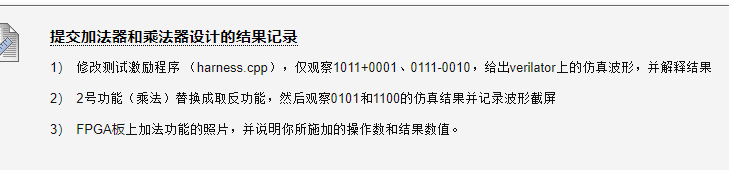
# 初级实验



初级实验首先包含一个简单加法器和乘法器的Chisel设计过程，以及相应的功能仿真和FPGA烧写实现，从而掌握Chisel语言的基本设计和vivado软件的使用方法。

其次，还对RISC-V mini处理器执行指令的细节进行观测，并用FPGA实现该处理器——能够运行简单的RISC-V程序。从而体验如何从处理器设计和硬件实现，并最终在编译工具链的配合之下，成功运行一个简单的C程序。

实验1——加法器和乘法器实验

本实验通过简单的加法器和乘法器的设计过程，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。

如果读者还没有了解过Chisel语言，请先阅读chisel语言的规范基本信息（chisel\_cheatsheet.pdf文档）和chisel的入门文档（《Digital Design with Chisel》 2nd edition, Martin Schoeberl）。

* + 1. 实验环境

本次实验需要两个环境，一个是Chisel设计开发环境，用于完成加法器和乘法器的Chisel设计，并输出各自的Verilog硬件描述语言文件；另一个是FPGA（Field Programmable Gate Array，现场可编程逻辑阵列）硬件开发环境，用于将加法器和乘法器的Verilog硬件描述语言文件，经过综合步骤后写入到FPGA中从而实现加法器和乘法器的硬件实体。

Chisel开发环境

Chisel开发环境可以在Linux系统中暗中，我们选择Ubuntu 20.4 LTS为例来说明其安装配置过程。主要包括：1）JDK、git、make、gtkwave、verilator、curl几个软件的安装；2）Scala的sbt安装；3）

##### 安装JDK、git、make、gtkwave、verilator、curl

sudo apt install default-jre git make gtkwave verilator curl

##### 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

##### 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

##### 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

Vivado开发环境

（李泽铭完善，现在直接提供虚拟机）

ALU和乘法器

下面我们来学习如何用Chisel语言来设计一个简单的ALU运算部件和一个简单的程法器。

* + 1. 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ALU部件的输入输出端口。这个ALU需要两个32位输入信号作为两个操作数，以及一个2位的输入信号用于控制ALU的功能（0：加法、1：减法、2：乘法、3：无效功能），以及一个32位输出信号表示输出结果。

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

##### ALU的Chisel描述

下面来看这个加法器的Chisel描述。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，使用起来很方便。由于Chisel是基于Scala语言实现的，因此首先需要导入chisel3包，才能在Scala基础之上使用Chisel的功能。

1. // src/main/scala/ALU/ALU.scala
2. package ALU
3. import chisel3.\_

然后我们定义ALU的控制信号的编码，每个编码对应一种运算操作。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。

1. **object** ALUConst {
2. **val** ALU\_ADD = 0.U(2.W)
3. **val** ALU\_SUB = 1.U(2.W)
4. **val** ALU\_MUL = 2.U(2.W)
5. }

随后需要定义加法器部件的端口，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，用于描述一组硬件信号线。在这里定义一个继承自Bundle的类ALUIO，类成员包含所需要的三个输入端口A、B和alu\_op，分别是两个操作数和一个运算功能选择控制信号，以及一个输出端口out用于表示计算结果。

1. class ALUIO extends Bundle{
2. val A = Input(UInt(32.W))
3. val B = Input(UInt(32.W))
4. val alu\_op = Input(UInt(2.W))
5. val out = Output(UInt(32.W))
6. }

最后可以定义我们的ALU模块（一个电路部件在Chisel中称为模块Module）的功能描述。在Chisel中，所有的电路部件都需要继承自Module类。在模块内部首先定义io端口，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的操作运算行为。其中加法、减法和乘法，在Chisel中就可以用“+”、“-”和“\*”直接表示，而无需关系其内部的逻辑电路如何实现，具体见代码 4‑1的6~8行。其中第6行表示将输入信号io.A和io.B作“+”然后将结果信号传递给ALU\_ADD信号，减法和乘法也类似。

代码 4‑1 ALU的核心功能描述

1. import ALUConst.\_ //引入ALUConst以便于使用其中定义的变量
2. class ALU extends Module{
3. val io = IO(new ALUIO)
4. io.out := MuxLookup(io.alu\_op,0.U,Seq( //default = 0.U
5. ALU\_ADD -> (io.A + io.B),
6. ALU\_SUB -> (io.A - io.B),
7. ALU\_MUL -> (io.A \* io.B)
8. ))
9. }

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。

这里同时计算了三种操作的结果，但是只能输出io.alu\_op所选定的那种运算结果到io.out信号上。地5行的代码使用Chisel提供的[MuxLookup](https://www.chisel-lang.org/api/3.4.3/chisel3/util/MuxLookup$.html)来选择不同的运算，MuxLookup就是一个多路选择器，根据传入的alu\_op控制信号来选择输出的数据。若要使用MuxLookup还需要先引入chisel3.util.MuxLookup 包。

1. import chisel3.\_
2. import chisel3.util.MuxLookup

MuxLookup的第一个参数为key，即选择控制信号，这里是io.alu\_op。第二个参数为默认值，此处为0，表示没有映射的控制信号时所输出的缺省信号值。其他选择控制信号的映射关系由第三个参数决定，它是为一个映射列表。从代码中可以看出——根据列表顺序，当选择控制信号为0时，io.out连接到ALU\_ADD加法运算结果；选择控制信号为1时，io.out连接到ALU\_SUB加法运算结果；当选择控制信号为2时，io.out连接到ALU\_MUL加法运算结果。列表中没有对应编号3的选择控制信号，此时复选器将输出第二个参数中指定的值，也就是0。

##### 产生Verilog文件

由于Chisel是硬件的高级描述语言，而硬件设计领域中所使用的是Verilog语言。只有Verilog语言才能够于各种硬件设计软件相配合，例如进行硬件仿真和进行IC流片生产、或者用FPGA实现。因此Chisel语言描述的硬简单路，需要转换输出成Verilog硬件描述语言的代码。

完成ALU模块的Chisel设计以后即可生成对应的Verilog文件，这也是有Chisel自带的功能实现的。首先，需要创建一个继承自App类的Main对象，并在当中执行emitVerilog()函数即可，这个函数将根据传入的ALU对象，转换输出相应的Verilog描述文件。

1. // src/main/scala/ALU/Main.scala
2. package ALU
3. object Main extends App {
4. (new chisel3.stage.ChiselStage).emitVerilog(new ALU)
5. }

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

上述命令执行相应的Chisel程序后，将在工程目录下成生一个ALU.v文件，即为生成的Verilog文件。如果查看该文件，可以看到如下内容：

1. module ALU(
2. input clock,
3. input reset,
4. input [31:0] io\_A,
5. input [31:0] io\_B,
6. input [1:0] io\_alu\_op,
7. output [31:0] io\_out
8. );
9. wire [31:0] \_io\_out\_T\_1 = io\_A + io\_B; // @[ALU.scala 25:24]
10. wire [31:0] \_io\_out\_T\_3 = io\_A - io\_B; // @[ALU.scala 26:24]
11. wire [63:0] \_io\_out\_T\_4 = io\_A \* io\_B; // @[ALU.scala 27:24]
12. wire [31:0] \_io\_out\_T\_6 = 2'h0 == io\_alu\_op ? \_io\_out\_T\_1 : 32'h0; // @[Mux.scala 80:57]
13. wire [31:0] \_io\_out\_T\_8 = 2'h1 == io\_alu\_op ? \_io\_out\_T\_3 : \_io\_out\_T\_6; // @[Mux.scala 80:57]
14. 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]
15. assign io\_out = \_io\_out\_T\_10[31:0]; // @[ALU.scala 24:12]
16. endmodule
    * 1. 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，而不是直接利用Chisel的“\*”来实现。这里定义的xlen表示乘法器的位数为4，即实现4位的乘法器。所实现的4-bit乘法器接口定义如代码 4‑2所示，其中包括两个输入multiplier被乘数和multiplicand乘数（其中的xlen.w用于表示信号的位数），输出信号为product乘积（信号的位数是乘数的两倍：xlen\*2）。由于需要多个周期才能计算出一个结果，该乘法器并非完全由组合电路构成。需要有寄存器保存其结果，并需要对于输入输出分别增加的Valid信号以表示计算过程的开始和完成。

代码 4‑2 乘法器的IO端口MultiplierIO

1. **object** MulConst{
2. **val** xlen = 4
3. }
4. **import** MulConst.\_
5. **class** MultiplierIO **extends** Bundle{
6. **val** multiplier = Input(UInt(xlen.W))
7. **val** multiplicand = Input(UInt(xlen.W))
8. **val** product = Output(UInt((xlen\*2).W))
9. *//valid signal*
10. **val** inputValid = Input(Bool())
11. **val** outputValid = Output(Bool())
12. }

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、结果以及一个计数用于记录乘法操作步骤的计数。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。

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

进行乘法操作的Chisel描述如代码 4‑3，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，根据乘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。

代码 4‑3 乘法操作的分布操作描述

1. when(cntReg =/= 0.U){
2. when(multiplierReg(0) === 1.U){
3. productReg := productReg + multiplicandReg
4. }
5. multiplierReg := multiplierReg >> 1.U
6. multiplicandReg := multiplicandReg << 1.U
7. cntReg := cntReg - 1.U
8. }.elsewhen(cntReg === 0.U){
9. when(io.inputValid){
10. multiplicandReg := Cat(Fill(xlen,0.U),io.multiplicand)
11. multiplierReg := io.multiplier
12. productReg := 0.U
13. cntReg := xlen.U
14. }
15. }

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。

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

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。

下面我们先用Verilator对上面设计的加法器和乘法器的Verilog描述文件进行功能仿真。

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所生成的对应Verilog源码，要使用Verilator进行仿真前需要做一次转化——将Verilog源码编译成等价的C++类模型。这里以刚生成的ALU.v为例来说明功能仿真如何进行。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可以了解该模块对应的端口，例如用cat命令来查看：

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等Chisel源代码中所描述的信号，以及chisel自动加入的clock和reset信号。它们是同步数字电路所必须的默认存在的信号，在加法器模块中未使用这两个信号，但是乘法器中的寄存器就使用了这两个信号。

1. …………
2. *// PORTS*
3. *// The application code writes and reads these signals to*
4. *// propagate new values into/out from the Verilated model.*
5. VL\_IN8(clock,0,0);
6. VL\_IN8(reset,0,0);
7. VL\_IN8(io\_alu\_op,1,0);
8. VL\_IN(io\_A,31,0);
9. VL\_IN(io\_B,31,0);
10. VL\_OUT(io\_out,31,0);
11. …………

### 2、编写C++ harness

为了验证电路功能是否正常，需要用特定的输入作为电路的激励，然后从电路模块的输出端口的响应信号来检查其是否达到预期设计效果。Verilator通过C++ harness文件来描述仿真过程中的输入和所期望的输出，并判定是否出现异常。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。

下面来编写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，通过三重循环来枚举上述取值——作为输入激励信号。

1. *// chisel-template/harness.cpp*
2. #include <verilated.h> *// 核心头文件*
3. #include <verilated\_vcd\_c.h> *// 波形生成头文件*
4. #include <iostream>
5. #include <fstream>
6. #include "VALU.h" *// ALU模块类*
7. **using** **namespace** std;
8. VALU \*top; *// 顶层dut对象指针*
9. VerilatedVcdC \*tfp; *// 波形生成对象指针*
10. vluint64\_t main\_time = 0; *// 仿真时间戳*
11. int main(int argc, char \*\*argv){
12. *// 一些初始化工作*
13. Verilated::commandArgs(argc, argv);
14. Verilated::traceEverOn(**true**);
15. *// 为对象分配内存空间*
16. top = **new** VALU;
17. tfp = **new** VerilatedVcdC;
18. *// tfp初始化工作*
19. top->trace(tfp, 99);
20. tfp->open("VALU.vcd");
21. *// 仿真过程*
22. top->reset = 0;
23. **for** (int a = 0; a <= 10; ++a){
24. **for** (int b = 0; b <= 10; ++b){
25. **for** (int op = 0; op <= 3; ++op){
26. top->io\_A = a;
27. top->io\_B = b;
28. top->io\_alu\_op = op;
29. top->eval(); *// 仿真时间步进*
30. tfp->dump(main\_time); *// 波形文件写入步进*
31. main\_time++;
32. }
33. }
34. }
36. *// 清理工作*
37. tfp->close();
38. **delete** top;
39. **delete** tfp;
40. exit(0);
41. **return** 0;
42. }

### 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。其中verilator命令参数—cc表示将ALU.v编译转换成C++模型，其中激励信号是由hardness.cpp给出的。

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：对于每一组io\_A和io\_B的组合，当alu\_op从0~3变化时io\_out便会产生相应运算的结果。其中alu\_op为3时候是未定义功能，输出default(0.U)。

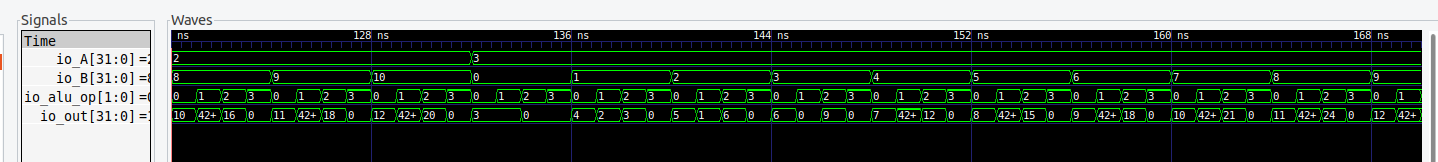


图 4‑1 ALU的Verilator仿真波形

## 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，它是一个时序电路，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平时设置，否则在始终上升沿进行修改地话将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。

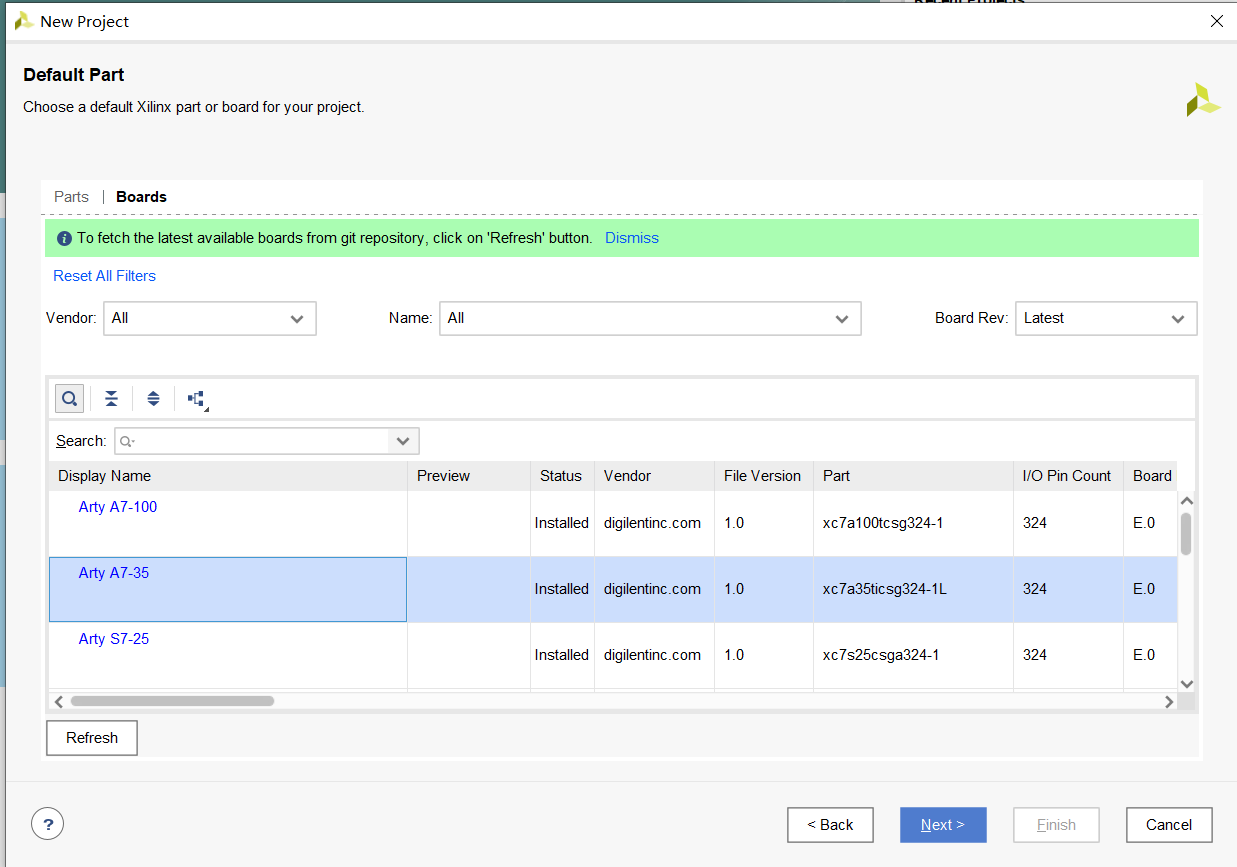
1. #include <verilated.h> *// 核心头文件*
2. #include <verilated\_vcd\_c.h> *// 波形生成头文件*
3. #include <iostream>
4. #include <fstream>
5. #include "VMultiplier.h"
6. **using** **namespace** std;
7. void test(int multiplier, int multiplicand, bool inputValid);
8. VMultiplier \*top;
9. VerilatedVcdC \*tfp;
10. int main\_time = 0;
11. int main(int argc, char const \*argv[])
12. {
13. Verilated::commandArgs(argc, argv);
14. Verilated::traceEverOn(**true**);
15. top = **new** VMultiplier;
16. tfp = **new** VerilatedVcdC;
17. top->trace(tfp, 99);
18. tfp->open("VMultiplier.vcd");
19. **for** (int i = 0; i < 16; i++)
20. {
21. **for** (int j = 0; j < 16; j++)
22. {
23. test(i, j, **true**);
24. **while** (!top->io\_outputValid)
25. {
26. test(0, 0, **false**);
27. }
28. }
29. }
30. tfp->close();
31. **delete** top;
32. **delete** tfp;
33. exit(0);
34. **return** 0;
35. }
36. void test(int multiplier, int multiplicand, bool inputValid)
37. {
38. top->clock = 0;
39. top->io\_multiplier = multiplier;
40. top->io\_multiplicand = multiplicand;
41. top->io\_inputValid = inputValid;
42. top->eval();
43. tfp->dump(main\_time++);
44. top->clock = 1;
45. top->eval();
46. tfp->dump(main\_time++);
47. }

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安装说明.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" \l "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/data/boards/board\_files文件夹（若没有则自行创建）当中即可。

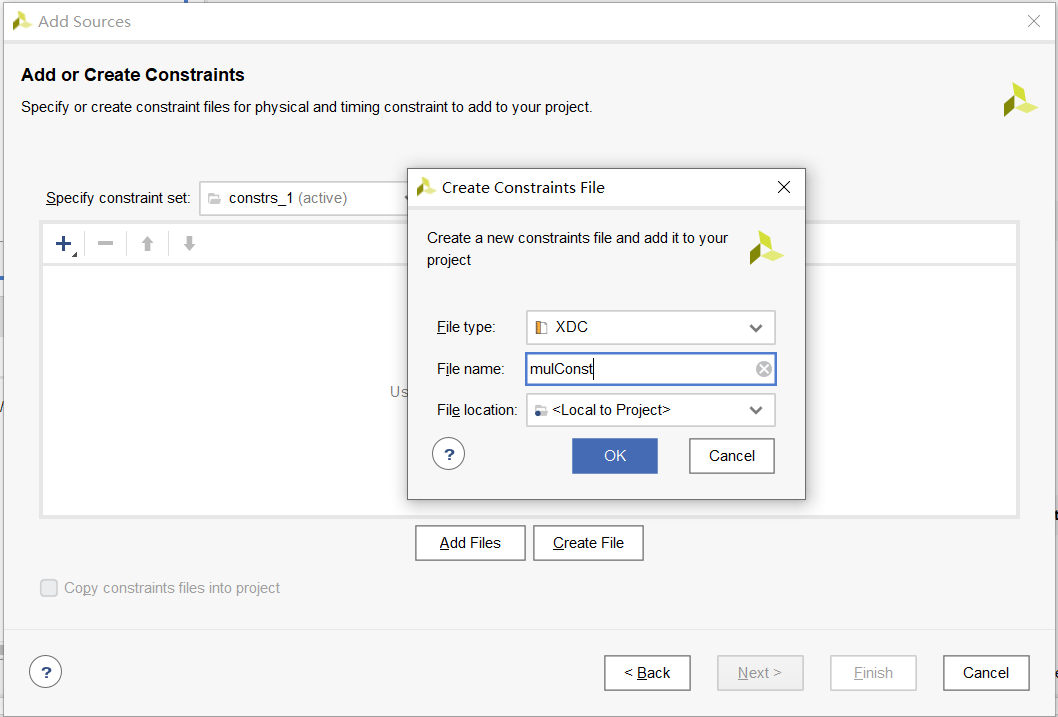
vivado安装完成后我们即可进行烧写。首先创建一个新的工程项目，用于将我们编写的乘法器烧写到FPGA当中。这个过程可以参考 [A7 - Vivado 使用说明.pdf](./A7%20-%20Vivado%20使用说明.pdf)

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。

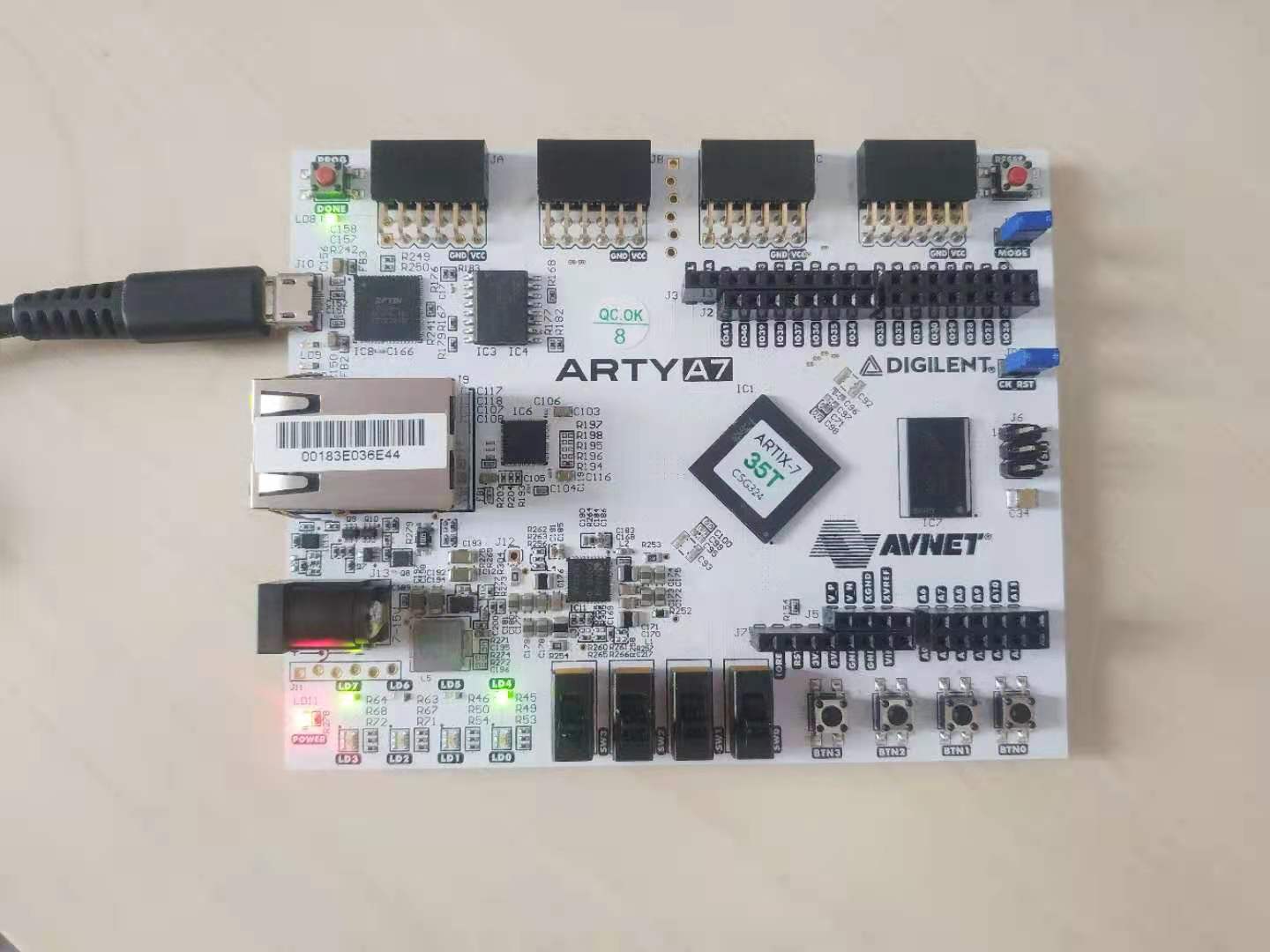
1. ## Clock signal  
   set\_property -dict { PACKAGE\_PIN E3 IOSTANDARD LVCMOS33 } [get\_ports { clock }];   
   create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports { clock }];  
     
   ## Switches  
   set\_property -dict { PACKAGE\_PIN A8 IOSTANDARD LVCMOS33 } [get\_ports { 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。

实验2——指令执行过程观测实验

实验目的：

了解RISC-V mini处理器架构，了解其Chisel设计，观察指令执行

工作：

1. 编写risc-V汇编程序样例一个，记录程序编译和生成bin过程作为实验说明
2. 完成带ram的risc-v mini的下载，记录步骤，作为实验说明
3. 展示如何通过观测执行过程中程序计数器的变化、寄存器文件的变化、内存竖直的变化。记录观测方法和过程

学生只需要找上面操作完成即可 这个可以chisel仿真完成，同时也配一个vivado ILA的版本。