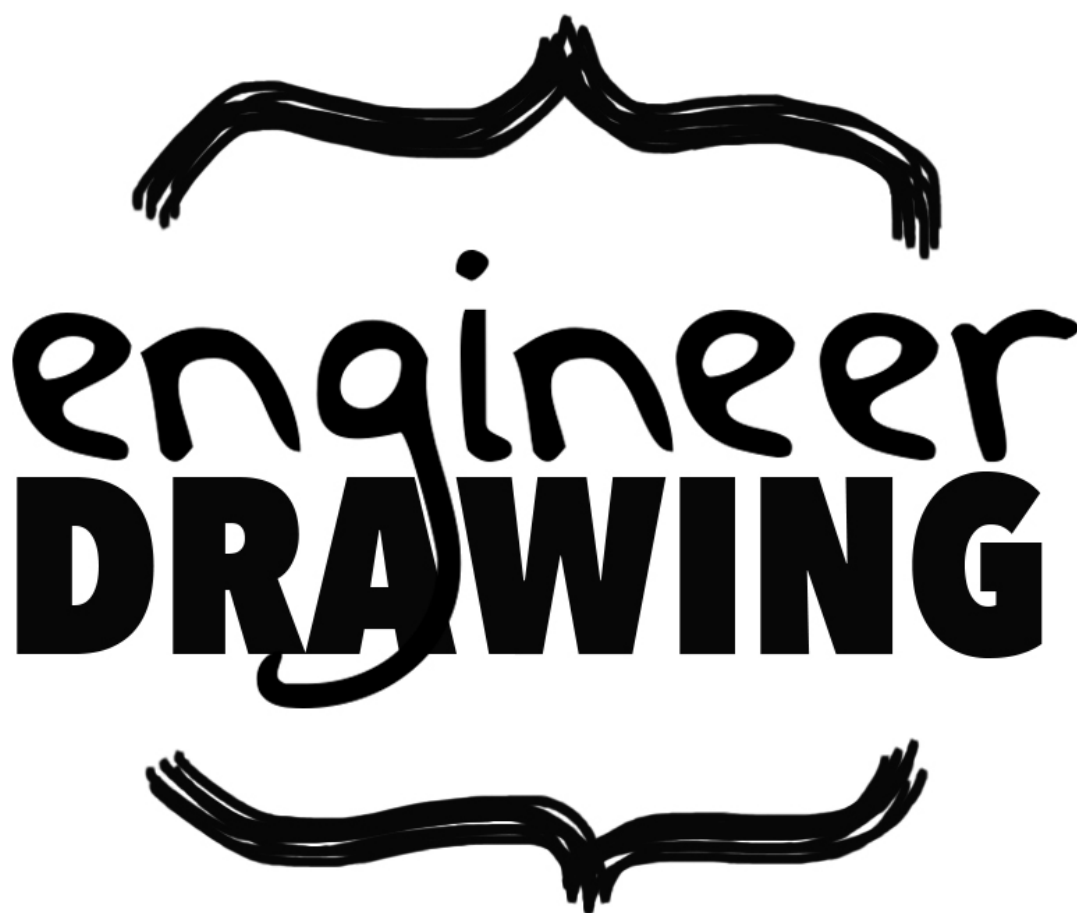


# 数字电路在线试验系统

分析设计文档



潘立航

李凝云

张鹿颂

周先达

# 目录

<b>1. 项目基本情况 .....</b>	<b>3</b>
1.1. 开发环境.....	3
1.2. 团队成员与分工 .....	4
1.3. 部署说明.....	5
<b>2. 架构设计 .....</b>	<b>6</b>
2.1. 开发环境.....	6
2.2. 数据库 .....	6
2.3. VHdl 代码与芯片信息 .....	6
2.4. 用户类别.....	7
2.5. 数字电路实验工程介绍 .....	8
<b>3. 技术细节 .....</b>	<b>12</b>
3.1. 数据库部分.....	12
数据库内容 .....	12
3.2. vhdl 代码翻译、生成模块 .....	18
74 系列芯片电路图翻译与代码生成 .....	18
cpu 组件电路图翻译和代码生成 .....	21
输入波形翻译与激励文件生成模块 .....	26
3.3. 后台管理模块.....	29
用户管理模块 .....	29
作业布置模块 .....	30
3.4. 登录界面.....	33
3.5. 编译仿真.....	35
使用 modelsim 进行 vhdl 文件的编译仿真 .....	35
vcd 文件解析 .....	35
波形可视化展示 .....	36
波形编辑.....	37

3.6. 代码编辑功能.....	39
文本编辑器.....	39
3.7. 电路拖拽编辑.....	44
Konva 的概念 .....	44
电路板 .....	44
芯片/元件 .....	46
导线 .....	49
其他 .....	54

# 1. 项目基本情况

本项目为数字电路在线实验系统，在软件工程课程中隶属于 PRJ7。

本系统可以为学生提供在线数字电路实验平台，学生用户可以自行创建工程，使用可视化芯片拖拽或者编辑硬件描述语言的方式来构建电路；通过可视化编辑波形或者编辑硬件描述语言的方式构建激励文件；将电路信息和激励文件提交后台进行编译仿真，得到可视化的输出波形。管理员则可以布置作业，并上传标准解答用来实现自动评分。

## 1.1. 开发环境

本项目使用的框架如下：

类别	名称
前端界面与 JavaScript	Bootstrap、jquery 等
数据库	Mongodb，使用 mongoose 框架
后端网络框架	后端网络框架

类别	名称
Nodejs + express	Nodejs + express

## 1.2. 团队成员与分工

项目的开发小组为 engineerDrawing。组员共 4 人，分工如下：

	分工	电子邮箱
潘立航 (组长)	<ul style="list-style-type: none"><li>• 后端实现</li><li>• 管理员后台实现</li><li>• 单元测试</li></ul>	kinnplh@126.com
李凝云	<ul style="list-style-type: none"><li>• 编译仿真模块</li><li>• 可视化波形编辑与展示</li></ul>	f6ce@163.com
张鹿颂	<ul style="list-style-type: none"><li>• 编辑器功能与界面实现</li></ul>	zhuanglusongg@gmail.com
周先达	<ul style="list-style-type: none"><li>• 电路图拖拽功能与界面实现</li></ul>	koocor@qq.com

### 1.3. 部署说明

为了运行代码，您需要：

#### 配置服务器

我们假定您的部署环境是 Ubuntu14.0 以上版本的操作系统

#### 下载源码

[http://166.111.131.70:8002/gitlab/2014011404/prj7\\_engineerdrawing](http://166.111.131.70:8002/gitlab/2014011404/prj7_engineerdrawing)

#### 安装相关软件

您需要安装 modelsim 和 nodejs 软件。前者需要在 modelsim 官网下载。后者也需要在官网下载最新的版本。注意，在以清华源为代表的一大批软件镜像中，nodejs 的版本过低，无法使用。

您还需要安装 mongodb。具体的操作请参照官网的指南。

您还需要安装包管理程序 npm。执行命令 `sudo apt-get install npm`

#### 安装第三方库

在项目源代码根目录下执行 `npm install`

#### 启动数据库和服务

启动服务器，执行 `sudo mongod`

启动数据库，在项目源代码的根目录下执行 `node ./bin/www`

然后可以在 <http://localhost:3000> 访问<sup>1</sup>

---

<sup>1</sup> 由于应用场景的特殊性，本系统不提供注册服务。如果需要，请手动对数据库进行修改

## 2. 架构设计

### 2.1. 开发环境

网站使用 nodejs 搭建，并使用 express 框架。采用 mongoose 框架操作 mongodb 数据库。使用 modelsim 进行硬件描述语言的编译与仿真。

views 文件夹存放了网站的所有网页，静态图片存放在 public/imgs，用于前端的 JavaScript 代码和 css 文件分别存放在 public/js 和 public/stylesheets 中。

routes 文件夹中存放了路由信息和相应的后端实现。

unitTest 文件夹中是本项目的单元测试。

### 2.2. 数据库

这一部分主要在 dataBase/\*下。

- *mongodb.js* 是服务器与数据库连接的脚本
- 在文件夹 *model* 中定义了若干需要被使用的 *schema*

### 2.3. VHDL 代码与芯片信息

分成 74 系列芯片的相关文件和 cpu 相关文件。

- VHDL/chipEntity 中定义了描述 74 系列芯片的相应的类
- VHDL 文件夹下的文件用来通过电路图或者波形图生成 vhdI 代码或者激励文件代码
- chipInfo/vhdls/中内置了默认的 cpu 组件的 chdl 代码
- chipInfo/cpu\_comps.js 用于对 vhdI 源代码进行解析，生成对应的可视化芯片

### 2.4. 用户类别

我们将网站的使用者分成两类：普通用户（学生用户）和管理员用户（老师用户）

#### 学生用户

学生用户可以进行的操作是

- 使用用户名和密码登录系统
- 创建、编辑、删除自定义项目，通过编辑器或拖拽界面进行电路的编辑
- 查看自定义项目的提交列表
- 对自定义项目进行编译仿真，查看输出波形
- 提交作业项目，获得系统评分
- 对于 cpu 类型的项目，支持上传本地 vhdI 代码生成对应的可视化芯片

本系统不支持学生用户的注册；所有的学生用户必须由管理员用户进行信息的导入才能生成。对于在本地运行想体验功能的用户，请自行对数据库做出修改。

#### 管理员用户



管理员用户可以进行的操作有

- 批量导入、生成学生账号
- 查看学生信息，包括学生的工程列表和每次提交的结果
- 查看已经布置的作业信息
- 布置作业

### 2.5. 数字电路实验工程介绍

数字电路实验工程经过普通自行创建项目或者管理员布置作业后进入数据库。学生只能删除自行创建的工程，不能删除作业。

#### 工程类型

普通用户自定义的工程有以下三种

- 编辑器工程：用户自行使用在线的编辑器进行源代码和激励文件的新建、修改、重命名和删除。也支持用户上传本地文件
- 拖拽工程：用户使用拖拽连线的方式搭建电路，并通过可视化的波形编辑器来生成激励文件。系统提供固定数量和种类的 74 系列芯片作为限制。
- cpu 工程：用户使用拖拽连线的方式搭建电路，并通过可视化的波形编辑器来生成激励文件。系统提供的芯片通过 vhdl 代码自动创建，支持用户新建、修改、删除 vhdl 代码，也支持从本地上传 vhdl 代码。系统默认提供 cpu 基本组件的 vhdl 代码。

管理员布置的作业有编辑器工程和拖拽工程两种，要求学生通过相应的界面完成作业。对于作业工程，系统不提供仿真的模块，用户在截止日期之前之前只能知道提交的作业是否正确，不能知道在标准输入下的输出结果。

### 工程状态

工程一共分为 4 种状态

- 这四种状态未曾编译
- 曾经尝试编译但是未曾成功
- 上一次编译成功
- 曾经编译成功但是上一次编译失败

这四种状态的工程分别对应于不同的颜色加以区分

### 工程编辑

参考“技术细节”中的相关部分

### 工程提交

系统会记录每一次工程的提交，包括

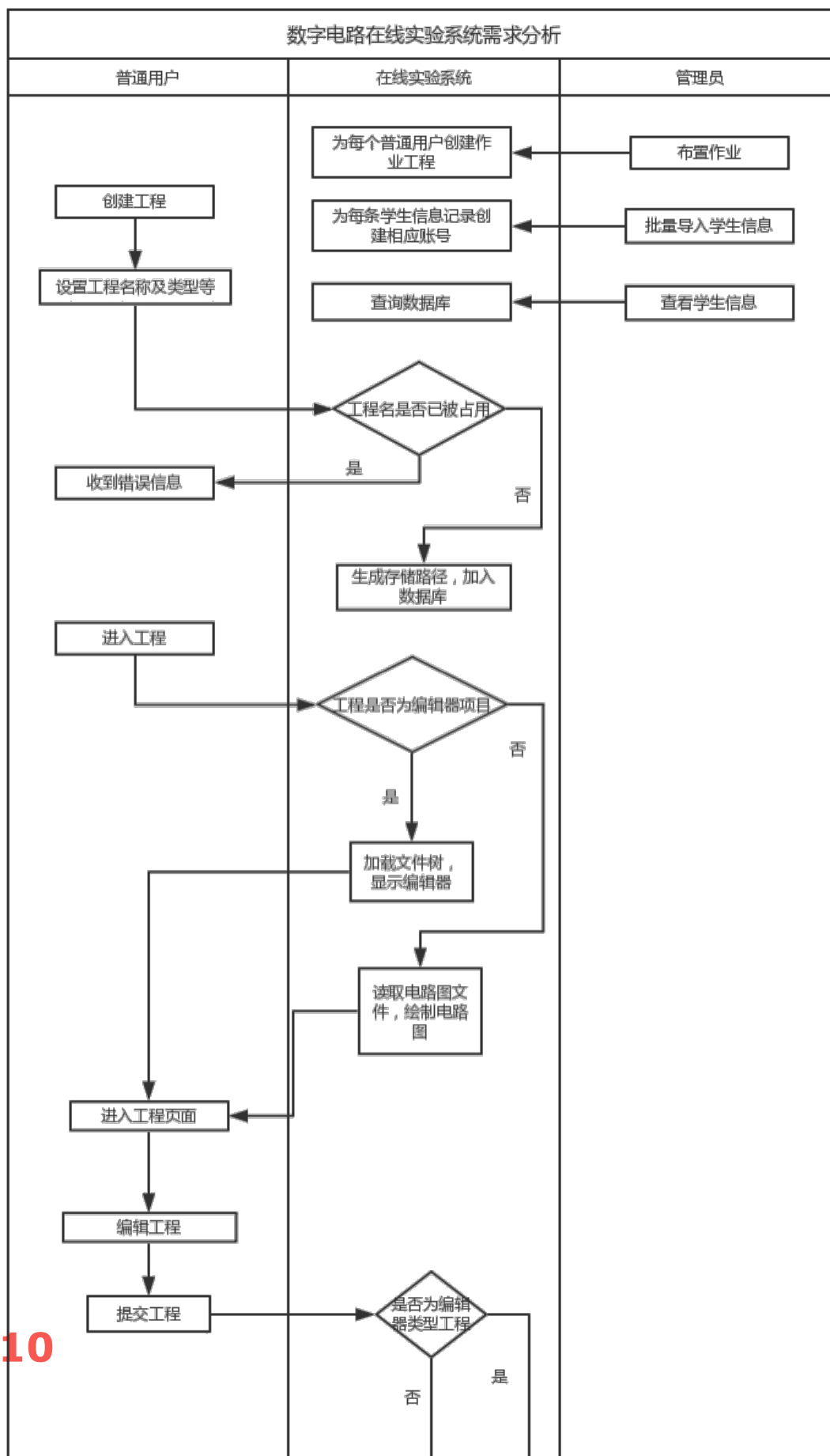
- 提交时间
- 编译信息
- 提交的源文件

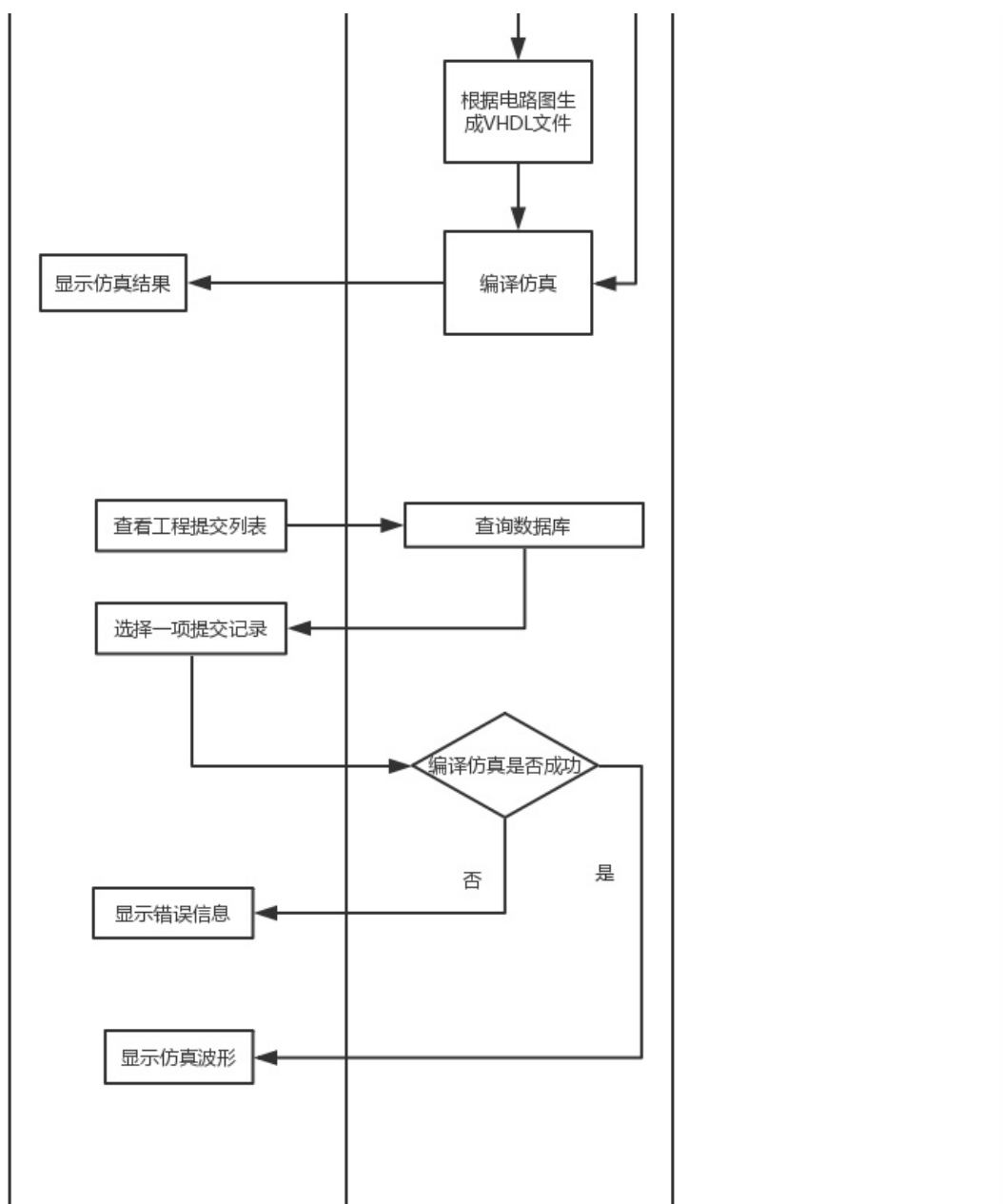
如果编译仿真成功的话，还会记录

- 编译仿真结果文件（以 vcd 文件格式存储）
- 编译仿真波形

所有的文件均支持用户下载到本地。

## 2.6. 需求分析（泳道图）





## 3. 技术细节

### 3.1. 数据库部分

我们使用 Mongodb 作为服务器使用的数据库，并使用 Mongodb 的 ORM 框架——Mongoose 来管理数据库。Mongodb 是一个非关系型的数据库，但是为了便于理解，在说明的时候仍然借鉴了关系型数据库的相关概念。另外，Mongodb 会为每一条记录增加“\_id”，作为主键；在后续的表格中不再显示。

#### 数据库内容

##### User

表示一个用户，包括学生用户和管理员用户

域列表		
域名	描述	类型
<b>userName</b>	用户名	字符串
<b>password</b>	密码	字符串
<b>projectBox</b>	自建工程列表	列表
<b>authority</b>	权限	数字

<b>group</b>	组别，将用户分组	字符串
<b>homeworkBox</b>	作业列表	列表

附加说明：

- *userName* 用于用户的登录，需要保证唯一；
- *projectBox* 中的每个元素是 *project* 记录的主键\_id，用来记录用户创建的所有工程
- *authority* 具体的取值是 0 和 1。0 代表普通用户，1 代表用户具有管理员权限；
- *group* 用来对用户进行筛选；
- *homeworkBox* 中的每个元素是 *project* 记录的主键\_id，要求 *project* 的 *homework* 域不能为空。用来记录用户的所有作业。

### project

表示一个工程，是用户自建工程、作业和 cpu 工程的统一。

域列表		
域名	描述	类型
<b>projectName</b>	工程名称	字符串
<b>author</b>	作者	user 记录的主键
<b>type</b>	工程类型	数字
<b>deleted</b>	是否已经被删除	布尔值
<b>createTime</b>	工程创建时间	时间戳
<b>lastModifiedTime</b>	上一次修改时间	时间戳

<b>filePath</b>	工程文件路径	字符串
<b>inputFile</b>	输入文件路径	字符串
<b>submitBox</b>	提交列表	列表
<b>compileStatus</b>	编译状态	数字
<b>topEntityName</b>	顶层实体名称	字符串
<b>homework</b>	从属的作业类型	homework 记录的主键
<b>score</b>	作业得分	数字
<b>hwSubmitBox</b>	作业提交列表	列表
<b>input</b>	工程的输入输出端口 <sup>2</sup>	列表
<b>output</b>		列表
<b>lastSimulationTime</b>	上一次仿真时间	时间戳
<b>entityPath</b>	存储实体代码的路径	字符串

附加说明：

- 值得注意的是，*mongodb* 是一个非关系型的数据库，这意味着上面所有的域在一条记录中不必全部出现。事实上，在 *project* 中，有很多的域是不会同时出现的，比如 *hwSubmitBox* 和 *submitBox*，根据是不是一个作业工程，在记录中仅会出现一项；
- *type* 域用一个数字表示工程的类型。0 表示是拖拽工程，1 表示是编辑器工程，2 表示是 *cpu* 工程；

---

<sup>2</sup> 在代码生成的时候确定，即代码生成接口中的 *portInfo* 参数

- *submitBox* 和 *hwSubmitBox* 都是元素是 *submit* 记录主键\_id 的列表，用来记录一项工程的提交记录；
- *compileStatus* 用一个数字来表征当前该工程的编译状态。0 表示上次编译是成功的；1 表示还没有进行过任何的编译；2 表示曾经编译成功，但是上一次编译失败；3 表示每次的编译都失败了；
- *entityPath*：存储 *vhdl* 代码的路径。这一个域只会出现在 *cpu* 工程中。*cpu* 工程会读取相关路径中的所有 *vhdl* 代码，进行词法分析和语法分析，根据分析的结果创建用于连线的芯片。

### submit

表示一次提交的记录。

域列表		
域名	描述	类型
<b>time</b>	提交时间	时间戳
<b>project</b>	所属的工程	project 记录的主键
<b>stdMsg</b>	编译仿真的标准输出	字符串
<b>errMsg</b>	编译仿真错误信息输出	字符串
<b>filePath</b>	本提交对应的文件路径	字符串
<b>inputFile</b>	本提交对应的输入文件	字符串
<b>simulateRes</b>	仿真结果文件名	字符串
<b>xtime</b>		字符串
<b>lastlist</b>		数组
<b>changelist</b>		数组



<b>signalname</b>	用于存储仿真波形 <sup>3</sup>	数组
<b>type</b>	提交类型	数字
<b>score</b>	本次提交分数	数字
<b>simulationTime</b>	仿真次数	数字

附加说明：

- *type* 表示提交的类型，0 或者未定义表示是自己创建的工程的提交（不需要计算分数）；1 表示是一次作业的提交（需要计算分数）；
- *score* 表示本次提交的分数，提交的分数是二值的，即 0 和 1，分别对应于正确和错误。

## homework

用来记录布置的一项作业的信息

域列表		
域名	描述	类型
<b>hwName</b>	作业名称	字符串
<b>type</b>	作业类型	字符串
<b>topEntityName</b>	顶层实体名称	字符串
<b>inPortName</b>	输入端口名称	字符串
<b>outPortName</b>	输出端口名称	字符串

<sup>3</sup> 这四项具体见 vcd 文件解析模块的说明。下同。

<b>deadline</b>	截止日期	时间戳
<b>filePath</b>	作业标准输入文件路径	字符串
<b>inputFile</b>	作业测试用例输入文件	字符串
<b>simuateRes</b>	作业标准输出文件	字符串
<b>xtime</b>	作为作业评判的依据	字符串
<b>lastlist</b>		数组
<b>changelist</b>		数组
<b>signalname</b>		数组
<b>describe</b>	作业描述	字符串
<b>correspondProject</b>	对应的工程	列表

附加说明：

- *type* 表示作业类型，即管理员要求你使用什么形式来完成这项作业。0 表示使用拖拽的方式，1 表示使用编辑器的方式；
- *correspondProject* 的元素是 *project* 类型记录的主键\_id，便于对布置的某一项任务的统计。

## 3.2. VHDL 代码翻译、生成模块

vhdl 代码生成模块负责将拖拽形成的电路图和输入波形翻译成对应的 vhdl 代码用于编译仿真。在本实验平台中，共有两种类型的电路图：一是 74 系列芯片对应的电路图；二是 cpu 对应的电路图。前者和后者相比，不仅在功能上比较简单，而且不需要实现 inout 类型的输入输出端口以及不需要实现对 vector 的支持。为了更好地适应两种电路图的不同特点，以及考虑到代码生成模块的效率等因素，这两种类型的电路图会使用不同的翻译与代码生成的模块。然而，对于两种电路图的输入波形的翻译工作，可以用同样的模块完成；这也是十分正常的事情：测试程序的实现本来就应该和程序的具体实现方式无关。

### 74 系列芯片电路图翻译与代码生成

在 VHDL/ VHDLCodeGenerator.js 定义了函数 VHDLCodeGenerator 函数来完成电路图的翻译和代码生成功能，具体接口如下

VHDLCodeGenerator (savePath, connectionInfo, topEntityName, portInfo, callback)

参数名称	含义
<b>savePath</b>	生成的 VHDL 代码的存储位置
<b>connectionInfo</b>	电路图的连接信息 <sup>4</sup>
<b>topEntityName</b>	顶层实体的名称
<b>portInfo</b>	引用输入，用于在代码生成之后获得输入输出信息
<b>callback</b>	回调函数

<sup>4</sup> 具体含义见“电路图描述相关模块”。下同。

函数内部相关变量的解释说明

变量名称	含义
<b>idToEntity</b>	将电路图中某个芯片的编号映射为相应实例
<b>idToInput</b>	将电路图中某个输入的编号映射为相应实例
<b>idToOutput</b>	将电路图中某个输出的编号映射为相应实例
<b>idToVcc</b>	将电路图中某个 vcc 的编号映射为相应实例
<b>idToGnd</b>	将电路图中某个 gnd 的编号映射为相应实例
<b>nameToEntityExample</b>	将芯片的名称映射到某一个作为代表的实例
<b>connectionArray</b>	描述连接信息的数组

函数中用到的一些类的说明：

### connection

用来描述两个芯片的连接关系，也就是说描述一根连线的信息。对于本程序中实现的 74 系列芯片而言，一根导线上连接的两个端口，必定一个是输入，一个是输出。

成员变量	描述
<b>src</b>	输出信号名称
<b>dest</b>	输入信号名称

在文件夹 VHDL/chipEntities 文件夹中定义了一系列描述芯片的类，这里以 Chip74LS00 类为例进行说明

## Chip74LS00

类型	名称	描述
成员变量	id	用来标志芯片的唯一 id
	pinType	静态常量，表明芯片每个端口的信息
	pinMap	将每个端口映射到一个信号
	name	该芯片实例的名称
	code	静态常量，该芯片对应的 VHDL 代码
成员函数	getPortMapString	返回对该芯片实例进行端口映射时对应代码
	getComponentString	返回该芯片作为一个组件时对应代码

附加说明：

- *pinMap* 实际上是一个 *Object* 对象，其键是端口名称，命名规则是字符 *p* + 编号；值是一个整数表示端口的性质，含义如下
  - 0: 输出 *output*
  - 1: 输入 *input*
  - 2: 接高电平 *vcc*
  - -2: 接低电平 *gnd*
  - -1: 无效端口 *nc*

这个接口先完成对电路图的两趟扫描，并根据获得的信息进行代码的生成。具体的行为通过下面的表格生成，在具体的实现中从上到下依次执行。

类型	名称	行为描述与实现
----	----	---------

电路扫描	第一趟扫描	对于每一个电路元件，生成相应的实例，并将其编号映射到该实例
	第二趟扫描	对于每一根导线，生成并存储相关的连接信息
代码生成	声明输入输出端口	遍历 idToInput 和 idToOutput
	声明所有信号	遍历 idToEntity，对于每一个芯片又遍历其 pinMap，对于非 vcc 和 gnd 的端口，声明该端口对应的信号名
	组件声明	遍历 nameToEntityExample，直接调用芯片的 getComponentString 函数
	vcc 与 gnd 的常量赋值	遍历 idToVcc 和 idToGnd 进行赋值
	信号赋值	遍历 connectionArray
	端口映射	遍历 idToEntity，直接使用函数 getPortMapString

## cpu 组件电路图翻译和代码生成

在 VHDL/ CPUCodeGenerator.js 定义了函数 CPUCodeGenerator 函数来完成电路图的翻译和代码生成功能，具体接口如下：

CPUCodeGenerator (savePath, connectInfo, cpuName, portInfo, entityPath, callback)

参数名称	含义
<b>savePath</b>	生成的 VHDL 代码的存储位置
<b>connectInfo</b>	电路图的连接信息
<b>cpuName</b>	顶层实体的名称

<b>portInfo</b>	引用输入，用于在代码生成之后获得输入输出信息
<b>entityPath</b>	工程引用组件的 VHDL 代码存储路径
<b>callback</b>	回调函数

函数内部相关变量的解释说明

变量名称	含义
<b>chipInfo</b>	根据 entityPath，解析 vhdI 文件生成的芯片信息
<b>nameInPort</b>	输入输出端口名称的集合
<b>chipNameToChipClass</b>	将芯片的名称映射到对应的类
<b>idToEntity</b>	将电路图中某个芯片的编号映射为相应实例
<b>idToInput</b>	将电路图中某个输入的编号映射为相应实例
<b>idToOutput</b>	将电路图中某个输出的编号映射为相应实例
<b>nameToEntity</b>	将芯片的名称映射到一个作为代表的实例
<b>connectionArray</b>	描述连接信息的数组，仅在涉及整个 cpu 输入输出的时候使用

函数中用到的一些类的说明<sup>5</sup>

### **portInfo**

用来描述一个端口的性质。实际上是一个结构体，只有成员变量，没有成员函数。

---

<sup>5</sup> 和 74 系列芯片电路图翻译和代码生成模块中类似的类不再赘述

成员变量名称	描述
<b>internalName</b>	内部信号的名称 <sup>6</sup>
<b>startNum</b>	开始编号
<b>endNum</b>	结束编号
<b>width</b>	端口位数
<b>order</b>	编号顺序
<b>type</b>	端口种类

附加说明：

- 本端口区分普通端口和总线型 (vector) 端口，并且要求，如果是总线型的端口，其端口名称是`< internalName >(<startNum> <order> <endNum>)`或者`< internalName >(<startNum> <order> <endNum>)`，其宽度 `width` 通过 `startNum` 和 `endNum` 计算。对于普通的端口，`startNum`、`endNum` 和 `order` 都被定义为 `undefined`，`width` 是 1。
- `type` 表征端口的类型，分为以下三种
  - 0: 输出 `output`
  - 1: 输入 `input`
  - 10: 输入输出总线 `inout`

匿名类（用来描述芯片信息）

---

<sup>6</sup> 具体的含义请参考对描述芯片的类的附加说明



类型	名称	描述
成员 变量	id	该芯片实例的编号
	name	该芯片实例的名称
	internalNameToExternalName	内部端口名称到端口外部名称的映射
	typeName	芯片类型名称
	internalNameList	静态常量，内部端口名称列表
	code	静态常量，该芯片的 vhdl 代码
	portIdToPortInfo	静态常量，本芯片中端口编号到端口信息的映射
	notInSignal	该芯片是否需要参与信号声明
成员 函数	getComponentString	返回对该芯片实例进行端口映射时对应代码
	getPortMapString	返回该芯片作为一个组件时对应代码

附加信息：

- 有内部端口名称和端口外部名称的解释

考虑到 cpu 端口信息的复杂性以及生成代码的效率，对于 cpu 芯片，芯片之间的连接关系不再通过信号的复制来体现，而是通过端口映射来体现。对于每一个芯片，其端口内部名称是确定的，也就是说，成员变量 `internalNameList` 长度固定并且每个元素的值固定。对于每一个芯片，其端口外部名称是不确定的；每次实例化一个芯片，必须明确该芯片每个端口的内部名称和外部名称之间的对应关系。外部名称的来源有两个方面：如果是输入端口，那么该端口的外部名称就是该输入对应的输出的外部名称；如果是输出端口或者输入输出端口，其外部名称是根据芯片名称和端口 `id` 自行确定的，与连接无关。

这两种名称主要是应用在 `getPortMapString` 函数中。根据上面的设定，在进行芯片端口映射的手，映射参数表只要简单地将外部名称进行罗列即可。

- 对于 CPU 项目，所有的芯片都是通过 VHDL 代码来动态加载的，不能像 74 系列芯片一样，事先将描述一种芯片的类通过硬编码的方式写在代码中。函数 `getFunctionAccordingToName` 就是通过对相应源代码解析的结果来动态生成相应的类的函数，其接口<sup>7</sup>如下：

```
getFunctionAccordingToName (chipName, internalNameList,
entityPath)。
```

本接口先对代码进行三趟扫描，并根据获得的信息进行代码的生成。具体的行为通过下面的表格描述，在具体的实现中从上到下依次执行。

类型	名称	描述
扫描	第一趟扫描	构建芯片符号表，并根据 vhdl 源代码动态生成类
	第二趟扫描	确定每一个芯片实例的名称、端口外部名称（先不考虑连接信息），并生成相应实例
	第三趟扫描	整理连接关系，加入到数组中（与 cpu 的输入输出相关）或者修改芯片的某一个端口外部名称
代码生成	声明输入输出端口	遍历 <code>idToInput</code> 和 <code>idToOutput</code>
	声明信号	遍历 <code>idToEntity</code> ，声明所有的端口外部名称
	声明组件	遍历 <code>nameToEntity</code>
	信号赋值	遍历 <code>connectionArray</code>

<sup>7</sup> 参数名称在上文中已经出现多次，此处不再赘述

	端口映射	遍历 idToEntity，使用 getPortMapString 函数
--	------	--------------------------------------

输入波形翻译与激励文件生成模块

在 VHDL/simulationCodeGenerator.js 文件中实现  
simulationCodeGenerator 函数完成相应的工作，具体接口如下  
simulationCodeGenerator (prj, signal, editSignal, callback)

参数名称	描述
prj	所属的工程，这里包括工程的完整信息
signal	所有的输入信号名称
editSignal	描述输入信号波形的列表 <sup>8</sup>
callback	回调函数

函数中用到一些变量的说明

变量名称	解释说明
inPortInfo	输入输出端口信息
outPortInfo	
currentIndexForSignals	辅助使用，记录当前关注的信号跳变的索引
jumpInfoAll	随着时间的推移，所有的信号中每发生一次信号的跳变就会向数组中增加一项

<sup>8</sup> 具体的格式参见波形编辑模块

函数中使用到的一些类的说明

### **jumpInfo**

记录跳变的信息

成员变量名称	含义
<b>signalName</b>	信号名称
<b>valueAfter</b>	跳变之后的信号值

### **jumpInfoPerTime**

记录某一个时刻所有信号的跳变信息

成员变量名称	含义
<b>time</b>	时间从开始仿真的时刻计算，单位是皮秒
<b>jumpList</b>	跳转信息的列表

在实现上，本接口先对所有的输入信号的信息进行归并，然后生成代码。具体的行为通过下面的表格描述，在具体的实现中从上到下依次执行。

行为	描述
跳转信息整理	参考归并排序中归并的实现，实现列表 jumpInfoAll，其中列表的每一项表示这某一个时刻的跳变信息。并且在列表中是根据时间升序的
声明组件	遍历 inPortInfo 和 outPortInfo
声明信号	

端口映射	
描述时序信息代码生成	遍历 jumpInfoAll

### 3.3. 后台管理模块

后台管理模块分成用户管理，信息管理和作业布置三个模块。其中信息管理模块主要涉及用户信息的展示，包括创建工程的信息、作业完成的情况等，是简单地对数据库进行读取和展示，这里不再赘述。下面，将分别介绍用户管理和作业布置两个模块。

#### 用户管理模块

用户管理模块实现的是对于用户信息的批量导入。用户信息的文件是 excel 表格，并且使用“网络学堂”的学生信息管理规范。根据表格默认生成的 user 记录如下所示

域名	默认值
<b>userName</b>	用户学号
<b>password</b>	用户学号
<b>projectBox</b>	空列表
<b>authority</b>	0（普通权限）
<b>group</b>	用户院系+班级
<b>homeworkBox</b>	空列表

具体的实现定义在 routes/admin/table.js，由“/importStudent”的路由函数完成相应的功能。我们使用了第三方库 excel-paser 来实现 excel 文件内容的解析。相应的接口如下：

parse (inFile, workSheet, skipEmpty, callback)

参数名称	描述
<b>inFile</b>	输入文件路径

<b>workSheet</b>	工作簿编号，这里只使用 1
<b>skipEmpty</b>	是否跳过空白，这里使用 false
<b>callback</b>	回调函数，第一个参数是错误信息，第二个参数是读取得到的每一条记录。这里，对于每一条记录，都加入到数据库中即可。

### 作业布置模块

作业布置模块分成三个部分，分别是作业基本信息填写，作业参考解答文件和标准输入文件上传，作业仿真及结果确认，分别介绍如下。

#### 作业基本信息填写

前端将填写的相关数据封装到 post 请求中，发送给后台；后台将这些信息封装成一个 homework 记录，并暂时存储在 session 中。该记录的每个域的值如下所示：

域名	默认值
<b>hwName</b>	req <sup>9</sup> .body.homeworkName
<b>type</b>	req.body.homeworkType
<b>topEntityName</b>	req.body.topEntityName
<b>inPortName</b>	req.body.inputPortName，并使用逗号分隔
<b>outPortName</b>	req.body.outputPortName，并使用逗号分隔

---

<sup>9</sup> req 是指一次 http 请求，其中 body 以键值对的形式存储 post 请求的内容；query 以键值对请求的形式存储 get 请求的内容。下同。

<b>deadline</b>	req.body.deadline
<b>filePath</b>	空
<b>inputFile</b>	req.body.simulateFileName
<b>simuateRes</b>	空
<b>xtime</b>	空
<b>lastlist</b>	空
<b>changelist</b>	空
<b>signalname</b>	空
<b>describe</b>	req.body.describe
<b>correspondProject</b>	空

### 作业文件上传

使用第三方的库 `multipart` 进行文件的上传。通过 `Form()` 函数，并以存储路径为参数生成 `form` 对象。然后调用 `form` 对象的 `parse` 函数进行上传文件的保存，最后在 `parse` 文件的回调函数中，将相应文件重命名为原来的文件名。

`parse (req, callback)`

参数名称	含义	
<b>req</b>	文件上传请求	
<b>callback</b>	参数名称	含义
	err	错误信息
	fields	这这里没有用到
	files	存储所有上传文件的信息



`fs.rename (originName, newName, callback)`

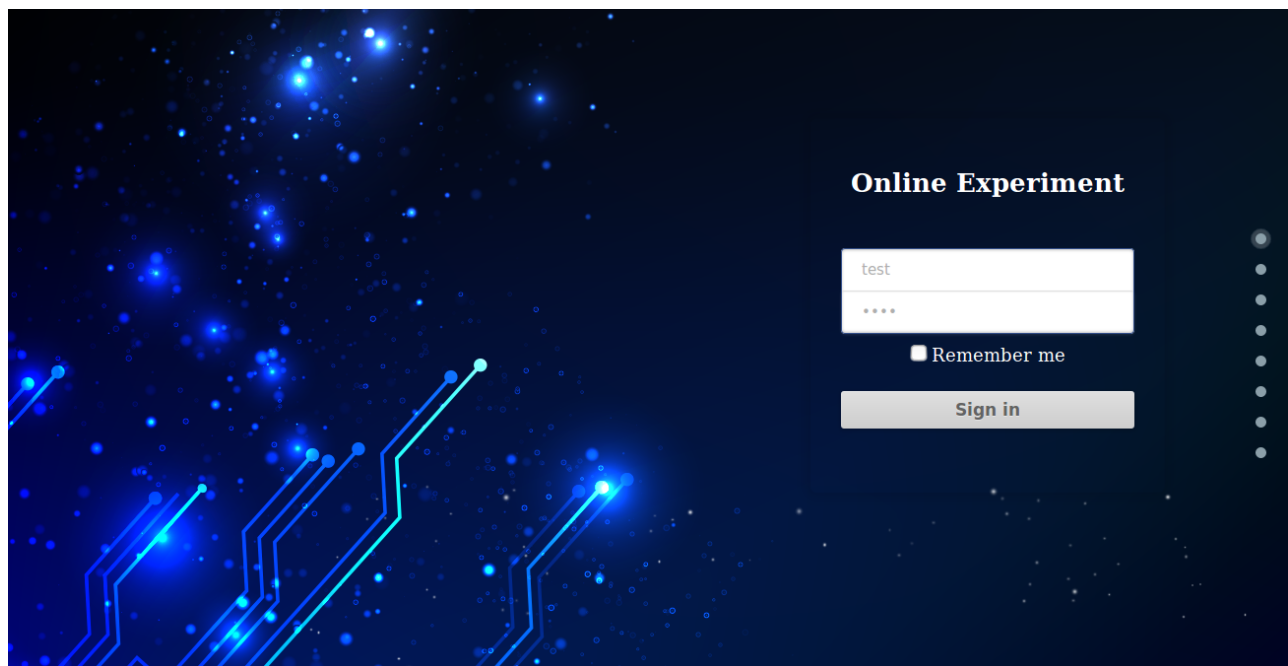
参数名称	含义
<b>originName</b>	原来的名称
<b>newName</b>	重命名之后的名称
<b>callback</b>	回调函数，只有一个参数 <code>err</code> 表示错误信息

### 作业仿真及结果确认模块

和普通的工程一样，调用编译仿真模块，并将仿真的结果图形化地表现出来。管理员如果对仿真的结果不满意，可以返回之前的步骤进行修改。否则，新的作业会被加入到数据库中，并对于每一个用户生成相应的作业工程。这里只是对数据库的简单操作，不再赘述。

### 3.4. 登录界面

在登录界面设计方面，本项目使用了 jQuery fullpage 的框架实现了登录界面的全屏滚动。登录界面效果图如下：



下面将列举各个主要的 css class 及其作用。

Class 名称	描述	图例
<b>.fullPage-tooltip</b>	页面右侧导航栏 点击圆点会跳转到教程的对应部分	
<b>.section</b>	页面主体部分 每一个 section 对应登录界面中的一页 其中还定义了 bg 和 bg img 分别对应页面的背景和背景图片	

<b>.active</b>	页面动画 每一个页面的每一部分 都 有 定 制 的 动 画 如.active.bg82 就是第 八页背景图片的动画样 式	无
<b>.ltie10</b>	为适配 IE 10 浏览器的 页面动画样式	无

### 3.5. 编译仿真

#### 使用 modelsim 进行 vhdI 文件的编译仿真

我们通过脚本调用 modelsim 进行 vhdI 文件的编译仿真。脚本文件位于 public/files 中，文件名为 model.sh

脚本命令：

命令	描述
<b>filename=\$1 jiliname=\$2</b>	vhdl 源文件名和激励文件文件名分别为参数 1 和参数 2
<b>vlib -unix work</b>	创建工作库 work
<b>vcom       \${filename}.vhd           \${jiliname}.vhd</b>	编译源文件和激励文件
<b>vsim -c -do "vcd add wave /\${jiliname}/*" -do "run 3000ns" -do "quit -sim" -do "q" \${jiliname}</b>	通过激励文件对源文件进行仿真，仿真时间 3000ns

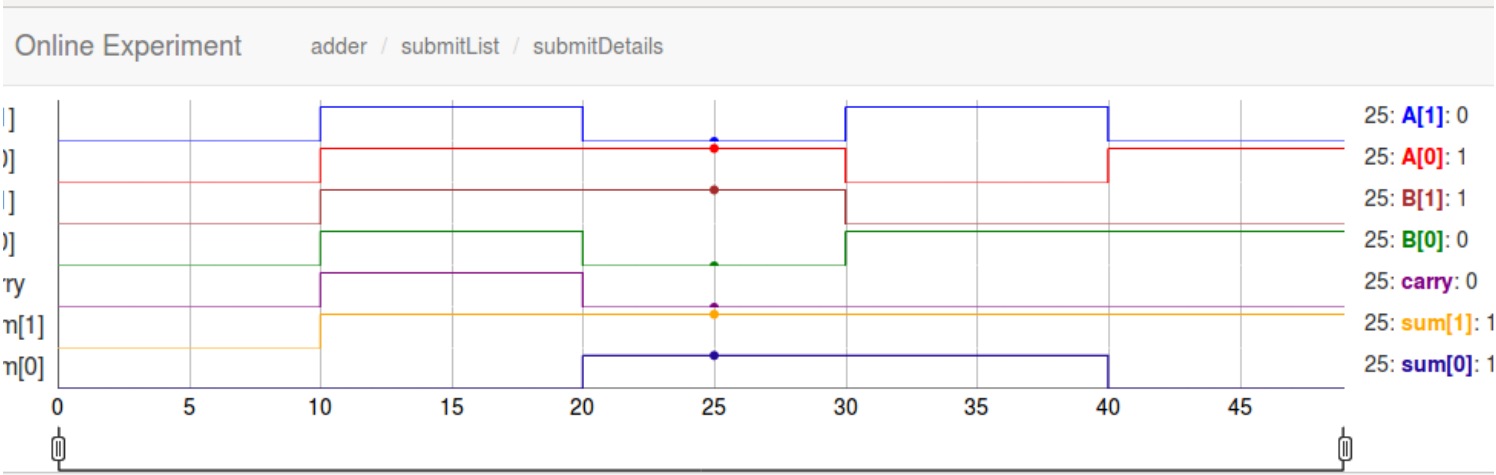
#### vcd 文件解析

modelsim 仿真会生成.wlf 文件，通过脚本文件可以将.wlf 文件转化为通用的.vcd 文件。为了生成波形图形所需的数据，我们需要对.vcd 文件进行解析。我们使用 node.js 中的 Duplex 流对文件进行了处理。主要函数列表如下：

函数名称	说明	返回值
<b>state (initial, props)</b>	将.vcd 文件分为不同的部分，将文件的每一行标注为不同的状态（是否为波形数据、是否是仿真时间）	这一行文件的状态
<b>vcdStream (opts)</b>	根据文件的不同部分	文件流

波形可视化展示

我们通过 dygraphs 图表库对解析出的.vcd 文件数据进行可视化展示，还实现了波形图像的拖拽、放缩以及波形数据的显示，还支持了 vector 的显示。波形可视化展示效果如下图：



Dygraph 的使用文档参见 [dygraphs.com](https://dygraphs.com)。波形图的主要参数如下：

参数	值	描述
labels	[xt,signallist[k]]	横 纵 坐 标 的 标 签， 分 别 为 时 间 和 波 形 值
connectSeparatedPoints	True	将 离 散 的 点 连 成 线
drawPoints	False	将 离 散 点 标 出
drawYAxis	True	显 示 Y 轴 刻 度 线
drawYGrid	False	不 显 示 X 轴 刻 度 线

<b>showLabelsOnHighlight</b>	False	在鼠标高亮时不显示标签
<b>height</b>	80	图表高度为80px
<b>Width</b>	900	图标宽度为900px
<b>panEdgeFraction</b>	0.000000001	图表拖拽边缘为原图的0.00000001倍，用于避免图像的无线拖拽
<b>showRangeSelector</b>	True	显示滑动条
<b>rangeSelectorHeight</b>	30	滑动条高度30px
<b>showInRangeSelector</b>	False	滑动条中不显示任何内容
<b>strokeWidth</b>	1.2	线条宽度 1.2px
<b>legend</b>	Never	不在鼠标旁显示标签波形数据
<b>highlightCallback</b>	function(e,x,pts,row)	在鼠标选中某一时刻时统一将波形数据显示在图表右方

## 波形编辑

我们通过重新定义图表的交互模式实现了波形的图形化编辑。用户可通过设置周期和鼠标选中两种方式对于输入波形进行编辑。

wave.js 中各个函数功能如下：

函数名称	说明
<b>downV4(event, g, context)</b>	定义鼠标按下事件，鼠标按下时调用 startEdit()函数开始进行编辑
<b>moveV4(event, g, context)</b>	定义鼠标移动事件，鼠标移动是调用 moveEdit()函数进行波形编辑
<b>upV4(event, g, context)</b>	定义鼠标放开事件，放开鼠标时调用 endEdit()函数结束波形编辑
<b>startEdit (event, g, context)</b>	开始波形编辑，设置相应标志
<b>moveEdit (event, g, context)</b>	进行波形编辑，计算鼠标移动距离等数据
<b>endEdit(event, g, context)</b>	结束波形编辑，根据 moveEdit 中得到的数据，将鼠标选中的部分进行翻转并记录波形变化的时间点
<b>init()</b>	初始化波形数据
<b>createGraph()</b>	初始化波形图
<b>onclickStart()</b>	设置仿真时间
<b>onclickReset()</b>	将波形图重置
<b>onclickPlot()</b>	生成周期性波形
<b>highlightSignal(ref)</b>	在鼠标选中拖拽界面输入信号时高亮相应波形
<b>disHighlightSignal(ref)</b>	取消对于波形的高亮
<b>changeName(ref)</b>	在用户修改输入信号名称时修改对应波形的名称
<b>reloadJili(projectId)</b>	将编辑好的波形数据传至后台进行处理

## 3.6. 代码编辑功能

### 文本编辑器

文本编辑器采用了开源的 ace-editor, 可为 VHDL 代码实现高亮功能。在这个基础上, 我们又实现了一些小功能来方便用户的编辑。

#### 代码补全

支持 VHDL 语言自带的标识符, 如 begin、end、process, 当出现匹配的下拉框时, 摁下回车键即可补全。除此之外, 用户自定义的变量均可以补全, 该功能有效地提高了用户的编辑速度。

该功能实现文件为\TextEditor\src-nonconflict\mode-vhdl.js

保留字的代码:

```
var keywords = "access|after|ailas|all|architecture|assert|attribute|"+
    "begin|block|buffer|bus|case|component|configuration|"+
    "disconnect|downto|else|elsif|end|entity|file|for|function|"+
    "generate|generic|guarded|if|impure|in|inertial|inout|is|"+
    "label|linkage|literal|loop|mapnew|next|of|on|open|"+
    "others|out|port|process|pure|range|record|reject|"+
    "report|return|select|shared|subtype|then|to|transport|"+
    "type|unaffected|united|until|wait|when|while|with";

var storageType = "bit|bit_vector|boolean|character|integer|line|natural|"+
    "positive|real|register|severity|signal|signed|"+
    "std_logic|std_logic_vector|string||text|time|unsigned|"+
    "variable";

var storageModifiers = "array|constant";

var keywordOperators = "abs|and|mod|nand|nor|not|rem|rol|ror|sla|sll|sra|"+
    "srl|xnor|xor";

var builtinConstants = (
    "true|false|null"
);
```

#### 代码折叠



当编辑较长代码时，过高的代码量可能会使调试变得困难。而代码折叠功能完美地解决了这个问题。用户可以轻松地将暂时不用查看的部分折叠，只显示自己想查看的部分。目前支持的折叠有 case-end case, if-end if, 还有各种的 begin end 段。在编辑区行号的旁边会有小箭头出现以示可以折叠，还可以通过摁下快捷键 F2 进行当前区域的折叠。

调用的函数有

函数名称	说明	返回值
<b>getFoldWidget(session, foldStyle, row)</b>	从某行号开始，由“折叠模式”生成折叠类	一个用来表示折叠的类
<b>getBeginEndBlock (session, row, column, matchSequence)</b>	从某个位置开始，搜索 begin end 或类似的表示开始结束的字符串，并返回位置	返回一个开始位置、结束位置的类

适当的改写该函数，并改写相应的开始、结束的正则表达式即可完成：

```
this.foldingStartMarker = /([^\n][^\d\s|^.{0,3})\bCASE\b|\bBEGIN\b|\bIF\b)/i;
this.startRegionRe = /^s*(\/\*|--)#?region\b/;
this.foldingStopMarker = /^[^\[\{\]*\(\)|\^[^\s\*]*\(\*\|\/\)|(\bEND\b(\w+;)?)/;
this.singleLineBlockCommentRe = /^s*(\/\*).*\*\/s*$/;
this.tripleStarBlockCommentRe = /^s*(\/\*\*\/s*).*\*\/s*$/;
```

### 跳转到指定行

Ctrl+L 实现跳转到指定行，只需输入行号，即可瞬间完成跳转。

### 文件树

采用开源工程 ez-filetree，用于显示文件列表。打开的文件将显示在多文本编辑器中，可支持新建、删除、重命名、保存等功能。

文件树部分主要复写以下函数：

函数名称	说明	返回值
<b>getFileTree()</b>	调用后端数据库生成前端的文件列表，并更新	无返回值
<b>activate(file)</b>	打开某个文件时进行的操作。包括维护当前打开的文件 activate_file（若已经打开，则跳转到那一页），通知多标签多开一个标签。	无返回值

### 多标签

经过大家的讨论，认为任务中存在多个 VHDL 文件共同完成一个工程的情况，因此需要一个多标签文本编辑器。这里采用了开源项目 hhEditor 的代码。重命名、删除、新建、保存均在这里完成。

### 保存

Ctrl+S 键即可完成保存。

### 新建

若文件不是从左侧文件树打开的（即为加号按钮新建出来的标签），此时 Ctrl+S 保存，即会新建文件。

### 重命名

双击标签，文件名即会出现输入光标，更改名字再保存即可完成重命名。

### 删除

打开想要删除的文件，windows 摁下 ctrl+shift+D，mac 摁下 option+D。

以上用到的函数有：

函数名称	说明	返回值
<b>uploadText()</b>	保存当前打开的文件，将其上传到后端，并给用户显示成功或失败信息。	无返回值
<b>deleteFile()</b>	删除当前打开的文件，将其上传到后端，并给用户显示成功或失败信息。如果成功关闭当前文件。	无返回值
<b>editor_setting(editor)</b>	对 editor 进行初始设置的函数，包括字体、高亮设置，以及各种快捷键的绑定。	无返回值

### 编译信息显示

如果在编辑区添加一个显示编译结果的列表，将会大大提高用户调试的速度。不仅如此，我们还实现了单击某个错误信息的自动跳转，该功能将会自动打开错误的 vhdl 文件，并将光标定位到出错的那一行。

当然，这个列表有可能对界面的整洁造成一定影响。因此，我们默认将其隐藏，想查看的时候单击的左下角即可显示，再单击按钮也可隐藏。提交的时候将会自动弹出该编译信息列表。

以下是设置隐藏与显示的代码：

```
$(document).ready(function(){  
    $("#closeButton").click(function(){  
        $("#listContainer").slideToggle();  
        $("#openButton").show();  
    })  
    $("#openButton").click(function(){  
        $("#listContainer").slideToggle();  
        $("#openButton").hide();  
    })  
    $("#listContainer").hide()  
});
```

### 3.7. 电路拖拽编辑

采用 Konva.js 绘图库操作 HTML5 <canvas> 标签的 SVG 矢量绘图功能，完成电路拖拽编辑功能。搭配 jQuery，控制网页 DOM 元素，完成电路试验工具箱的功能。

#### Konva 的概念

有了 Konva.js 的封装，我们可以用对象的方法创建/编辑 HTML5 的矢量图形，并为之赋予鼠标悬浮/点击/拖拽等事件。利用

Konva.js 绘图，首先创建一个 stage 对象，在这个对象上可以创建多个 layer（图层）对象，然后在不同的 layer 中绘图。

# Konva

在我们的项目中，为节省性能，只设置了两个 layer：

Layer 名称	描述
layer	所有图形对象都将被创建在这个图层。
dragLayer	电路图上某个元素（而非整个电路图）被拖动时，他将先上浮到 dragLayer。拖动完毕后再回去。这是 Konva.js 提升拖拽性能的一个技巧。

下面介绍各个 feature 及其实现：

#### 电路板

网格点：在背景上填充绘制一个屏幕大小的矩形，在矩形内平铺网格点图片。其中网格宽度设置为 15 像素，使用的图片是长宽为 100 像素，四个角点上

1/4 圆，如图所示：

背景图 background 是一个 Konva 矩形，参数如下。

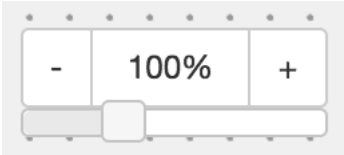
参数	值	描述
x	0	对齐屏幕窗口
y	0	
height	windowHeight	
width	windowWidth	
fillPatternRepeat	'repeat'	在 x,y 轴方向上平铺
fillPatternScale	0.15	将格点距离设置为 15 像素
perfectDrawEnabled	False	非精细绘制，提高性能

拖拽

按住 Shift+鼠标单击拖动，之前没有出现在频幕上的区域会提前加载出来。所以我们的电路图可以无尽缩放。

缩放

利用 jQuery 原生的 slider 控制电路图缩放。同时按住 Shift+鼠标滚轮滑动可以滑动 slider，从而缩放电路图。



电路图缩放很重要的一点是要以鼠标中心缩放，同时缩小时应注意之前没有出现在频幕上的区域要提前加载出来。另外设置了『100%』按钮方便用户还原。相应参数如下：

名称	值	描述
<b>MIN_SCALE</b>	0.5	最小 50%
<b>MAX_SCALE</b>	2.0	最大 200%
<b>SLDR_GRAIN</b>	2000	将 MIN_SCALE 到 MAX_SCALE 分成 2000 个区域离散地进行缩放（这是因为 slider 只支持离散地拖动）

## 芯片/元件

### 基本信息

用 Konva Circle 实现芯片端口，Rect（或者 PolyLine）实现芯片/元件主体，Line 实现端口到芯片主体的连线，Text 实现芯片名、端口名和端口编号，将他们装入 Konva Group 容器中，成为一个整体的芯片/元件。

每个芯片/元件有一组输入端口和一组输出端口，分别在芯片/元件的左侧和右侧。芯片有如下属性：

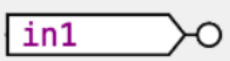
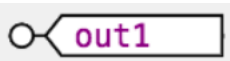
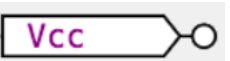

名称	描述
<b>name</b>	名称（显示在元件主体上）
<b>type</b>	类型
<b>number</b>	数量限制（-1 表示数量不限）
<b>description</b>	元件或芯片描述，其中芯片的描述会显示在工具箱中
<b>inputNodes</b>	输入端口集合
<b>outputNodes</b>	输出端口集合
<b>height</b>	高度（height 个 15 像素）

其中输入端口有如下属性：

名称	描述
----	----

y	在芯片主体上的垂直坐标
name	名称
num	编号
specieName	[可选]特殊属性名称，有'vcc_i'、'vcc_o'、'gnd_i'、'gnd_o'、'nc'，与 Vcc/GND 相关端口的连线逻辑有关。

共有五种芯片/元件：

属性名称	inputChip	outputChip	vcc	gnd	74 系列芯片
name	'input'	'output'	'Vcc'	'GND'	与具体芯片有关
type	'io'	'io'	'power'	'power'	
number	-1	-1	-1	-1	
description	'输入'	'输出'	'Vcc'	'GND'	
inputNodes	(空)	{y:0, name:' ', num:0}	(空)	{y:0, name:' ', num:0, specieName:'vcc_o'}	
outputNodes	{y:0, name:' ', num:0}	(空)	{y:0, name:' ', num:0, specieName:'vcc_i'}	(空)	
height	1	1	1	1	
图例					



一个 74 系列芯片举例：

属性名称	<b>74LS90</b>
<b>name</b>	'74LS90'
<b>type</b>	'counter'
<b>number</b>	3
<b>description</b>	'二-五-十进制异步计数器'
<b>inputNodes</b>	[ {y:1, name: 'Vcc', num: 5, specieName: 'vcc_i'}, {y:2, name: 'CPA', num: 14}, {y:3, name: 'CPB', num: 1}, {y:4, name: 'R1', num: 2}, {y:5, name: 'R2', num: 3}, {y:6, name: 'NC', num: 4, specieName: 'nc'}, {y:7, name: 'S1', num: 6}, {y:8, name: 'S2', num: 7}, {y:9, name: 'GND', num: 10, specieName: 'gnd_i'} ]
<b>outputNodes</b>	[ {y:2, name: 'NC', num: 13, specieName: 'nc'}, {y:4, name: 'Q0', num: 12}, {y:5, name: 'Q1', num: 9}, {y:6, name: 'Q2', num: 8}, {y:7, name: 'Q3', num: 11}, ]
<b>height</b>	10

注意芯片的 Vcc 和 GND 都放置在左侧，只能和相应的 vcc\_o/gnd\_o 端口连接。

### 高亮

为芯片/端口设置事件。点击后芯片主体的边框会变粗，以示高亮，此时可以删除芯片，删除芯片的同时也会删除从该芯片引出的所有导线。当鼠标悬浮到端

口上方时，端口内部会相应变色，当导线要从某端口引出时，该端口周围会有闪烁的呼吸灯。

端口类型	鼠标悬浮高亮颜色	周围呼吸灯颜色
输入	绿	绿
输出	绿	绿
空端口	(无)	(无)
Vcc 相关端口	红	红
GND 相关端口	灰	红

### 拖拽

拖拽完成后会吸附到格点上，通过 Group 的位置判断。group 有 dragDistance 属性，设置为 2，意思是当鼠标挪动超过 2 像素才被识别为滑动。

### 导线


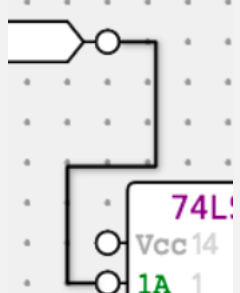
用 Konva Line 实现导线，一条导线是一根横平竖直的折线，多种颜色可选。导线右下方有阴影，阴影深度可以提示该区域导线堆叠数目。具体参数如下：

属性名称	值	描述
shadowColor	(多种颜色可选)	阴影颜色（与导线颜色相同）
shadowOffset	x: 1, y: 1	阴影位移
stroke	(多种颜色可选)	颜色
strokeWidth	1.3	导线宽度
lineJoin	'bevel'	结点连接方式

<b>shadowOpacity</b>	0.5	阴影不透明度
<b>shadowBlur</b>	5	阴影模糊程度

## 导线连接

在想要连接起来的两个端口上分别单击即可创建一条将这两个端口连接起来的导线。根据两个端口的相对位置，生成的导线有三种曲折方式：

名称	条件	图例
<b>LINE_SELF</b>	两个端口在同一个芯片上	
<b>LINE_TRI</b>	输入端口的横坐标在输出端口的右侧	
<b>LINE_QUIN</b>	两个端口在不同芯片上，且输入端口的横坐标在输出端口的左侧	

芯片/元件的端口之间有一套导线连接逻辑，不符合连线逻辑的导线连接将不会被执行，具体的逻辑如下：

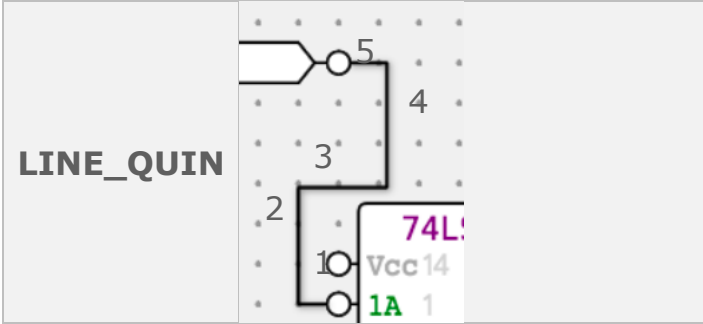
端口属性	允许你连接的端口的属性	是否允许引出多条导线
输入	vcc_o / gnd_o / 输出	否
输出	输入	是
空端口	(无)	-
vcc_o	vcc_i / 输入	是
gnd_o	gnd_i / 输入	是
vcc_i	vcc_o	否
gnd_i	gnd_o	否

## 导线编辑

为导线的每一段进行编号，用户和可以拖动某些段，对导线进行编辑。如将某一段拉长、将 LINE\_TRI 类导线折断为 LINE\_QUIN 类导线、将 LINE\_QUIN 类导线合并为 LINE\_TRI 类导线等等。拖动芯片/元件时也会改变导线的长度和形状。

编号如下

名称	图例
LINE_SELF	
LINE_TRI	



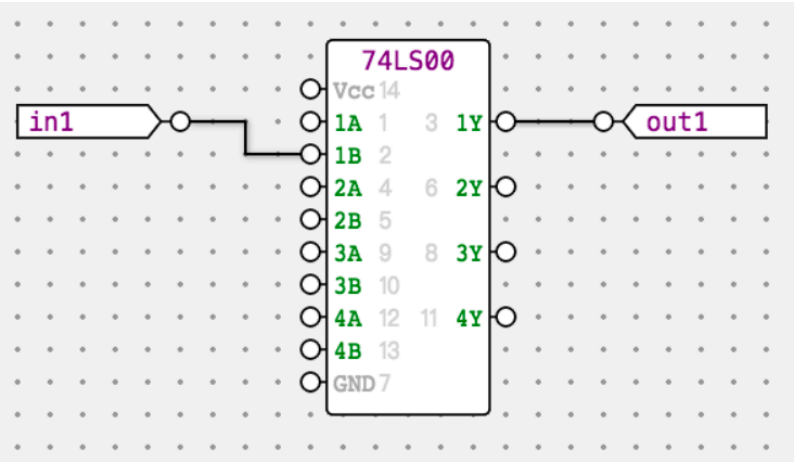
工具箱

工具箱如图所示，实现了芯片/元件/导线的删除，导线的颜色更改，芯片添加等功能。

电路描述

为了实现电路的保存和加载，以及电路的 VHDL 代码翻译。必须用简单的对象语言来描述并储存电路，需要记录下电路的连接信息和电路中的各元件的位置。

下面这个简单电路的描述文件如下：



操作 ✕删除

导线颜色 

1 2 3 4 5 6

其他元件 

Vcc GND 输入 输出

芯片

- 二输入与门
  - 74LS86 还剩 3 个 ?
  - 四-二输入异或门
    - 74LS90 还剩 2 个 ?
    - 二-五-十进制异步计数器
      - 74LS161 还剩 2 个 ?
      - 4位二进制同步计数器

made with by engineerDrawing

描述文件	解释
<pre>{  "7": {     "x": 525, y": 195,     "name": "74LS00",     "inputs": {</pre>	芯片/元件 id (id 对用户透明) 位置坐标 芯片/元件名称 输入端口编号列表

<pre> "1": [], "2": [ {     "parent": 66,     "num": 0,     "points": [525, 240, 495, 240, 495, 225, 465, 225],     "color": "black",     "shape": 1 } ], "4": [], "5": [], "7": [], "9": [], "10": [], "12": [], "13": [], "14": [] } }, "66": {     "x": 375, "y": 225,     "tag": "in1",     "name": "input",     "inputs": {} }, "79": {     "x": 660, "y": 225,     "tag": "out1",     "name": "output",     "inputs": {         "0": [{             "num": 3,             "parent": 7,             "points": [660, 225, 637.5, 225, 637.5, 225, 615, 225], </pre>	<p>1 号端口为空</p> <p>2 号端口连出了一根导线 连接到的芯片/元件 id 连接到芯片/元件的哪个端口 折线坐标</p> <p>导线颜色 导线形状类型 (LINE_TRI/等)</p> <p>4~12 号端口都为空</p> <p>输入元件的编号 输入元件只有输出端口</p> <p>输出元件的编号</p>
---	--

<pre>        "color": "black",         "shape": 1     }}     } }</pre>	
--	--

### 其他

在电路拖拽界面上整合了波形编辑功能，点击右下角按钮可以呼出相应的面板。面板中可以编辑输入波形，每一个输入元件对应一个输入波形，输入的在电路图上的位置对应面板中波形的顺序。每一次调整输入的位置、或者增删输入，都会调用 `updateInputList` 函数，通知面板中显示波形 DOM 进行调整/重绘。相应的还实现了输入/输出的名称自定义功能。

电路拖拽界面的左上角整合了保存/仿真/查看结果列表等按钮。与编辑器界面中的按钮功能相同。