VeriSim_Compiler :以Logisim为目标平台的verilog编译器

姓名	学号	主要任务
孟繁瑞	ZY2006239	设计思路，编译器前端（词法、语法、语义），相关文档撰写
郝 浩	ZY2006202	编译器后端，接口设计，相关文档撰写

设计背景

《计算机组成原理》是计算机类各专业本科生必修的硬件课程中的重要核心课。在北航，**计算机组成原理**、**操作系统**、**编译原理**，因其**高标准、高难度、高强度**的课程设计，一直是六系众人肩上的三座大山。自有记载以来，无数仁人志士日以继夜、夜以继日，挑灯夜读、埋头苦干。
说回《计组课程设计》，作为目前专业分流后的第一门核心专业课的动手环节，目前采用的是Logisim-verilog-mips的理论预习顺序 & logisim-verilog 的CPU构建顺序实现的。作为面向特定领域的语言（Domain Specified Language），verilog实现了将特定的语言对应到硬件模型上。如果学习了verilog，并且有一定的实践经验的编程人员能强烈地感受到，verilog和C/C++等编程语言有着本质且明显的差别。但对于初学者而言，仅在课堂上听高老板一句高屋建瓴的“编程时做到心中有电路”，难以对应到实践中。

因此，先用logisim完成简单电路的设计、单周期CPU的设计帮助学生建立电路构造的 “feel” ，再进行verilog的较复杂CPU设计。这一安排是很合理且成功的，但是在担任计组朋辈助教期间，我负责辅导的同学的课程进度多卡在**Logisim_CPU——Verilog_CPU**之间，说明对于一些同学而言，有限时间内思维方式的快速转换仍有难度。是否有其他方式可以帮助解决这一问题，在这门课上我想到一个解决方案，**反其道而行之**，基于可综合的HDL生成Logisim电路。

设计思路及相关工作调研

在选取基础语言方面，结合开发需求和当前领域内的主要工作进行调研。

高层次综合（Hhigh Level Synthesis）是通过编写C++高级语言代码实现RTL级硬件功能。可以方便已有的C算法代码映射到FPGA上，有点事开发快速，效率高。缺点是综合出来的结果质量落后于RTL的质量，使用门槛较高，只适用于硬件开发的特定领域。

Chisel作为基于Scala的语言，在UCB研究团队开发和推动下，目前广泛应用于RISC-V的开发当中。Chisel语言在设计时以Scala为基础，因为Scala中有许多特性适合描述电路，比如它是静态语言，以编译期为主，适合转换成Verilog/VHDL。再比如它的操作符即方法、柯里化（Currying）、纯粹的面向对象、强大的模式匹配、便捷的泛型编写、特质混入、函数式编程等特性，使得用Scala开发DSL语言很方便。Chisel语言不等同于HLS，因为语言没有EDA工具支持，其选择借助verilog，再交付EDA工具实现电路生成。通过Firrtl编译器，将chisel文件转换成firrtl文件，这是一种标准的中间交换格式，方便地让各种高级语言转换到VHDL。其重要特性，一是引入了面向对象的特点，二是减少了不必要的语法，并利用类继承等的特性迅速改变电路结构。缺点是入门门槛极高，调试极复杂，可读性极差。

考虑本项目的实现流程，需要将所用语言对电路做一层抽象，做成Logisim库中提供的元器件。Logisim元器件中一部分也为抽象表示，需要在设计时充分考虑元器件的逻辑和逻辑特性。再考虑到入门门槛和开发难度，决定以System-Verilog为蓝本，裁剪不必要的语言特性，重点实现可综合语法的语言到门电路的转换。

VeriSim语言特性：

1、语法较简单，入门门槛低。

2、以可综合电路为目标，完备的语法支持。

3、实现硬件语言到可编辑电路图的转换。

词法描述

保留字

VeriSim中的保留字，部分是对硬件结构的修饰，例如端口的input/output属性、保留的门电路（and、or、not等），另一部分是时序逻辑、组合逻辑的抽象，如always块、generate块。而initial、real等保留字多用于HDL的验证流程，不可综合，VeriSim不考虑实现。语言的保留字如下：

module	**endmodule**	**input**	**output**	**wire**	**signed**
reg	**integer**	**and**	**or**	**xor**	**not**
generate	**endgenerate**	**genvar**	**for**	**if**	**else**
case	**endcase**	**default**	**begin**	**end**	**assign**
always	**posedge**				

运算类型

VeriSim支持多种运算符，Logisim本身提供了常见的加减乘除运算器件，可以直接通过VeriSim的对应运算符映射。而对于门电路，也可以支持多位直接比较。赋值运算较为不同。对于组合逻辑，应当通过ASSIGN关键字和“=”进行赋值。但对于时序逻辑，考虑到非阻塞赋值对于代码整体可读性、错误易排查性、构成电路的效率影响，VeriSim仅支持阻塞赋值——即在Always块中使用“<=”对reg变量赋值，无法对wire变量赋值。

VeriSim支持的运算类型（EBNF）如下：

```
  unary_binary_op : +=|-=|\*=|/=|%=|&=|\|=|^=|<<=|>>=|**|==|<=|>=|<<|>>|[<>%^&\?
+/=~\-\*]
```

其他符号

数字：VeriSim支持两类数字，一种为普通十进制数字类型，另一种为格式化数字，即 **位数 ' 进制数值** 表示，如32'h6789ABCD

标识符：除保留字外的变量名

构成语法的其他符号`[@:,.`;]`、各种括号`[(){}[\]]`

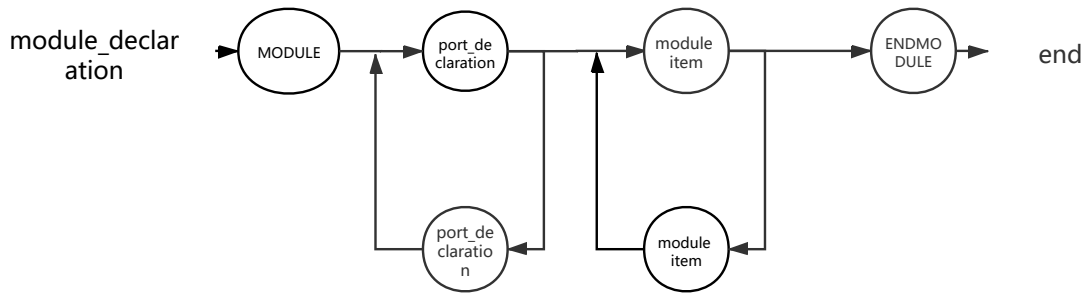
语法描述

本节主要用EBNF描述VeriSim语法，并针对主要语法表达式解释其语法含义，并给出其语法图。**加粗**代表终结符，其它为非终结符。

- identifier:
  - **ident**
- system_name:
  - sysname
- natural_number:
  - **natural**
- string:
  - **string**
- unary_operator:
  - **+|-|!|~|&|~&| | |~| |^|~^|^~**
- binary_operator:
  - **+|-|*|/|%| =\= | != | === | !\== | && | || | ** |< | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<**

模块声明

电路图的声明从<translation_unit>开始，<module_declaration>声明了模块的名字、输入输出端口，<module_item>是对模块内部电路结构的描述。

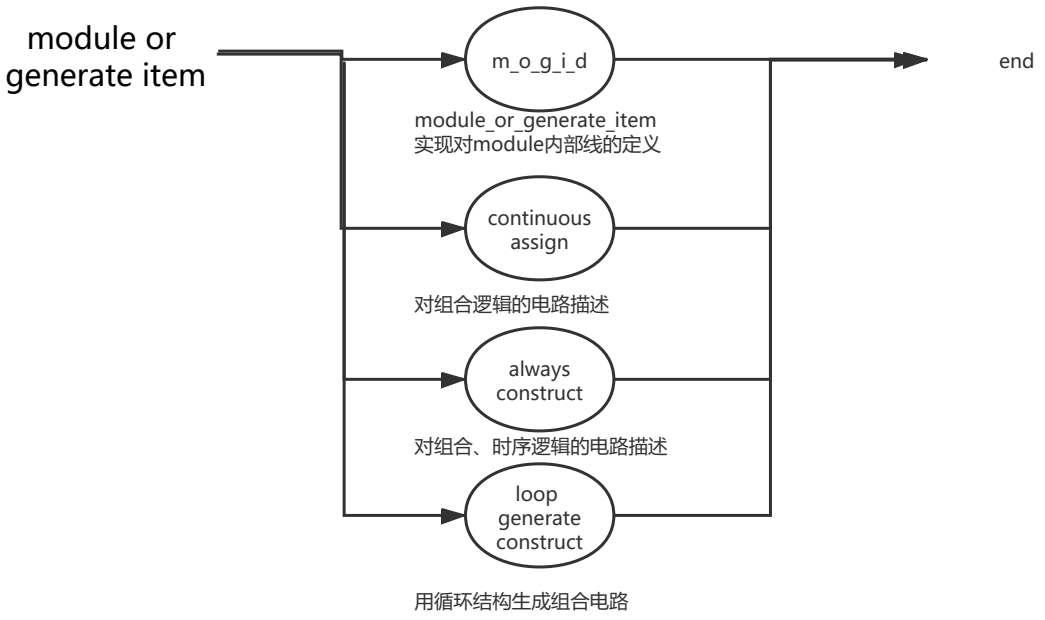


- translation_unit :
  - module_declaration
- module_declaration :

    - **module** module_identifier [**(** [list_of_ports | list_of_port_declarations] **)**] **;** { module_item } **endmodule**
- list_of_ports:
    - port{ **,** [port]}
- list_of_port_declarations:
    - port_declaration { **,** port_declaration }
- port:
    - port_expression | **.**port_identifier **(** [port_expression] **)**
- port_expression:
    - port_reference | **{** port_reference { , port_reference } **}**
- port_reference :
    - port_identifier[**[** constant_range_expression **]**]
- port_declaration:
    - input_declaration | output_declaration
- input_declaration :
    - **input** [**wire**] [**signed**] [range] list_of_port_identifiers
- output_declaration :
    - **output** ([**wire**] [**signed**] [range] list_of_port_identifiers | **reg** [**signed**] [range] list_of_variable_port_identifiers)

模块组成

当模块名、输入输出端口声明完成后，规定在模块内部不能再进行端口相关说明，仅能进行内部线/寄存器的定义。<module_item>实现对主要电路结构的描述，如时序逻辑、组合逻辑。



- module_item :
    - non_port_module_item
- non_port_module_item :
    - module_or_generate_item | generate_region

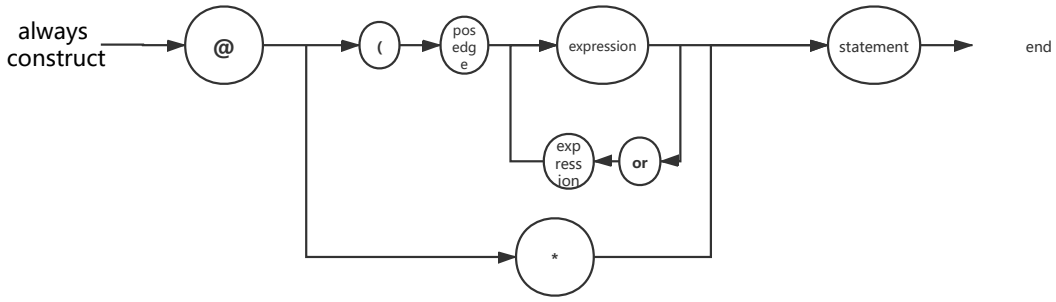
- module_or_generate_item :
  - module_or_generate_item_declaration
    | continuous_assign
    | gate_instantiation
    | always_construct
    | loop_generate_construct
- module_or_generate_item_declaration :
  - net_declaration | reg_declaration | integer_declaration | genvar_declaration
- integer_declaration :
  - **integer** list_of_variable_identifiers **;**
- list_of_net_decl_assignments_or_identifiers :
  - net_identifier [dimension { dimension } | **=** expression] { **,** net_identifier [dimension { dimension } | **=** expression] }
- net_declaration :
  - **wire** [**signed**] [range] list_of_net_decl_assignments_or_identifiers **;**
- reg_declaration :
  - **reg** [**signed**] [range] list_of_variable_identifiers **;**
- genvar_declaration :
  - **genvar** list_of_genvar_identifiers **;**
- variable_type :
  - variable_identifier ({ dimension } | **=** constant_expression)
- list_of_port_identifiers :
  - port_identifier { **,** port_identifier }
- list_of_real_identifiers :
  - real_type { **,** real_type }
- list_of_variable_identifiers :
  - port_identifier [**=** constant_expression] { **,** port_identifier [**=** constant_expression] }
- dimension :
  - **[** dimension_constant_expression **:** dimension_constant_expression **]**
- range :
  - **[** msb_constant_expression **:** lsb_constant_expression **]**
- gate_instantiation :
  - n_input_gatetype n_input_gate_instance { **,** n_input_gate_instance } **;**
    | n_output_gatetype n_output_gate_instance { **,** n_output_gate_instance } **;**
- n_input_gate_instance :
  - [name_of_gate_instance] **(** output_terminal **,** input_terminal { **,** input_terminal } **)**
- n_output_gate_instance :
  - [name_of_gate_instance] **(**expression { **,** expression } **)**
- output_terminal :
  - net_lvalue
- input_terminal :
  - expression
- name_of_gate_instance :
  - gate_instance_identifier **[** range **]**
- n_input_gatetype :

    - **and**
      | nand
      | or
      | nor
      | xor
      | xnor
- n_output_gatetype :
    - **not**
- generate_region :
    - **generate** { module_or_generate_item } **endgenerate**
- list_of_genvar_identifiers :
    - genvar_identifier { **,** genvar_identifier }
- loop_generate_construct :
    - **for (** genvar_initialization **;** genvar_expression **;** genvar_iteration **)** (generate_block | module_or_generate_item)
- genvar_initialization :
    - genvar_identifier **=** constant_expression
- genvar_expression :
    - [unary_operator] genvar_primary genvar_expression_nlr
- genvar_expression_nlr :
  [binary_operator genvar_expression genvar_expression_nlr
  | **?** genvar_expression **:** genvar_expression genvar_expression_nlr]
- genvar_iteration :
    - genvar_identifier **=** genvar_expression
- genvar_primary :
    - constant_primary
- conditional_generate_construct :
    - if_generate_construct
      | case_generate_construct
- if_generate_construct :
  if (constant_expression **)** generate_block_or_null [**else** generate_block_or_null]
- case_generate_construct :
    - **case (** constant_expression **)** case_generate_item { case_generate_item } **endcase**
- case_generate_item :
    - constant_expression { **,** constant_expression } **:** generate_block_or_null
      | **default** [**:**] generate_block_or_null
- generate_block :
  begin [**:** generate_block_identifier] { module_or_generate_item } **end**
- generate_block_or_null :
  generate_block
  | module_or_generate_item
  | **;**
- continuous_assign :
    - **assign** list_of_net_assignments **;**
- list_of_net_assignments :

    - net_assignment { **,** net_assignment }
- net_assignment :
    - net_lvalue **=** expression　　//lvalue 即为 left_value，等式左值

时序逻辑

always块主要实现了对寄存器的赋值连线，其中event_control可视为对寄存器trigger的定义，而always块内部的statement又可以通过条件语句（conditional_assign)、选择语句（switch/case)进行描述，可以视作寄存器的enable位。这一点在指称语义部分进行描述。需要注意的是，这里提供的语法图是多个语法句式组合后的结果，定义大量语法句式，是考虑到编译器实现上的需要，虽略显臃肿，但在使用上难度不大。

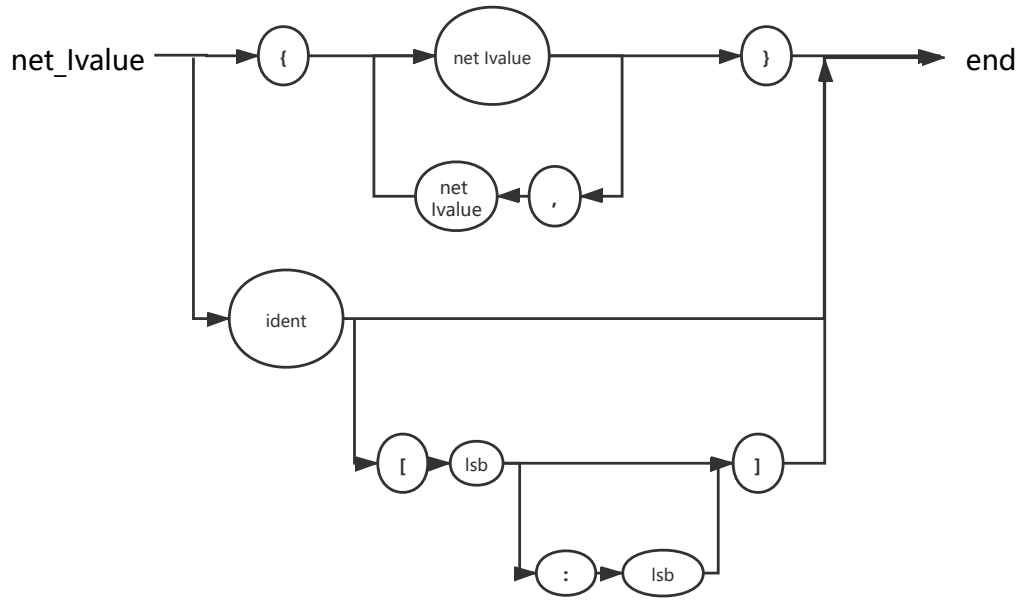


- always_construct :
    - **always** event_control statement
- procedural_continuous_assignments :
    - **assign** variable_assignment
- variable_assignment :
    - variable_lvalue **=** expression
- seq_block :
    - **begin** [**:** block_identifier { block_item_declaration }] { statement } **end**
- nonblocking_assignment ：
  variable_lvalue [**(** expression { **,** expression } **)**] [**<=** [expression]] **;**
- statement :
    - nonblocking_assignment
      | case_statement
      | conditional_statement
      | loop_statement
      | procedural_continuous_assignments ;
      | seq_block
      | ;
- event_control :
    - **@** (**(** (event_expression | *****) **)** | *****)
- event_expression :
    - **posedge** expression event_expression_nlr |
      expression event_expression_nlr
- event_expression_nlr :
    - [**or** event_expression event_expression_nlr
      | **,** event_expression event_expression_nlr]

- conditional_statement :
  if (expression **)** [statement] [**else** [statement]]
- case_statement :
  case (expression **)** case_item { case_item } **endcase**
- case_item :
  expression { **,** expression } **:** [statement]
  | **default** [**:**] [statement]
- loop_statement :
  - **for (** variable_assignment **;** expression **;** variable_assignment **)** statement
- constant_expression :
  - [unary_operator] constant_primary { constant_expression_nlr }
- constant_expression_nlr:
  - binary_operator constant_expression
    | **?** constant_expression **:** constant_expression
- expression :
  - [unary_operator] primary { expression_nlr }
- expression_nlr :
  - binary_operator expression
    | **?** expression **:** expression
- constant_primary :
  - number
    | string
    | (identifier | system_name) [**[** constant_range_expression **]** | **(** constant_expression { **,** constant_expression } **)**]
    | **{** constant_expression [**,** constant_expression { **,** constant_expression } | **{** constant_expression { **,** constant_expression } **}**] **}**
- primary :
  - hierarchical_identifier_range [**(** expression { **,** expression } **)**]
    | number
    | **{** expression [**,** expression { **,** expression } | **{** expression { **,** expression } **}**] **}**
- hierarchical_identifier_range :
  - identifier { **.** identifier [**[** range_expression **]**] | **[** range_expression **]** }
- range_expression :
  - expression [**:** lsb_constant_expression]

赋值语句

连线左值，既可以是限制对某条定义线特定位数的赋值，也可以是将多条线按照从左至右、从高至低的顺序“组合”在一起的“变量“。而variable_net_lvalue与net_lvalue的主要区别是，前者左值用于寄存器，后者左值用于连线。



- net_lvalue :
  - hierarchical_identifier_range_const
    | **{** net_lvalue { **,** net_lvalue } **}**
- hierarchical_identifier_range_const :
  - identifier { **.** identifier [**[** constant_range_expression **]**] | **[** constant_range_expression **]** }
- variable_lvalue :
  - hierarchical_identifier_range
    | **{** variable_lvalue { **,** variable_lvalue } **}**
- variable_or_net_lvalue :
  - hierarchical_identifier_range
    | **{**variable_or_net_lvalue { **,** variable_or_net_lvalue } **}**
- number :
  - real_number
    | natural_number [based_number | base_format (base_value | natural_number)]
    | sizedbased_number
    | based_number
    | base_format (base_value | natural_number)
- based_number :
  - **BASEDINT**
- base_value :
  - **BASEVAL**
- sizedbased_number:
  - **SIZEVAL**
- base_format :
  - **BASEFMT**
- constant_range_expression :
  - constant_expression [**:** lsb_constant_expression]
- list_of_variable_port_identifiers :
  - port_identifier [**=** constant_expression] { **,** port_identifier [**=** constant_expression] }

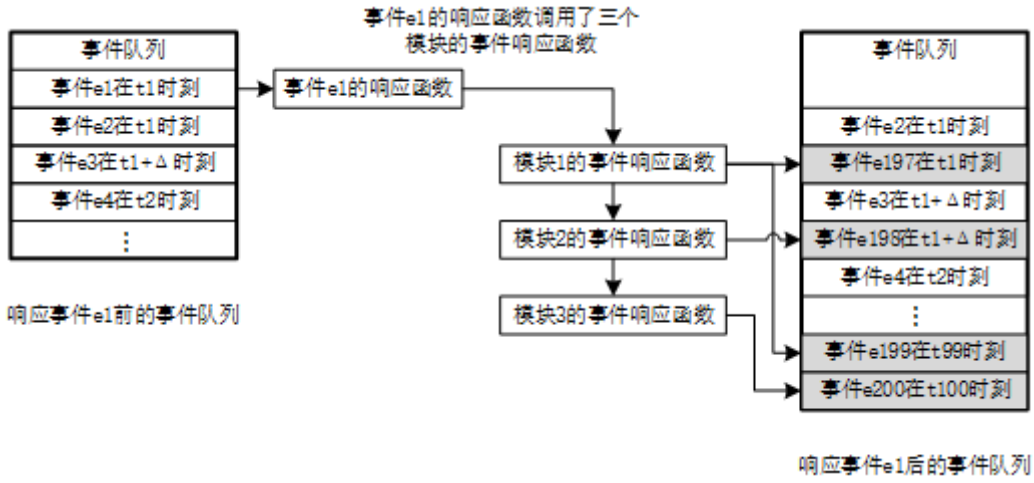
标识符

下面的与标识符相关的语法句式，虽然都归结为同样的终结符，但他们的语义是不一样的，这样定义有利于在编译器的语法分析阶段生成抽象语法树，语义阶段进行语义检查和语义分析。

- module_identifier:
  - identifier
- port_identifier :
  - identifier
- net_identifier :
  - identifier
- gate_instance_identifier :
  - identifier
- genvar_identifier :
  - identifier
- block_identifier :
  - identifier
- generate_block_identifier :
  - identifier
- dimension_constant_expression :
  - constant_expression
- msb_constant_expression :
  - constant_expression
- lsb_constant_expression :
  - constant_expression

指称语义

在分析VeriSim的指称语义之前，我们通过查阅资料，寻找HDL语义论述的相关文献。例如，在文献[1]中讨论了Verilog语义的抽象状态机模型，其思考方式和仿真器类似。当前主流的仿真器，都是接收Verilog源文件并将其转化为可执行的仿真源文件（c++/c）等。根据Verilog的语法规范，将语法结构转换为仿真器中的事件响应函数。这些函数和仿真器的框架一起成为可执行仿真程序的源文件。运行时，程序维护仿真时间，并在相应触发点触发进程执行原语（响应事件）。可以认为，是将**语法结构**映射为**事件队列**。



在VeriSim中，我们实现的是解析语法结构，最终生成Logisim可编辑的电路图，剥离掉时间这一维度，即无需考虑运行时状态，将**语法结构**映射为各种**元器件**，构成一张**"平面图"**。接下来，我们尝试根据编译器实现过程中前后端的运行流程，对VeriSim的指称语义进行描述。

语义域

```
Width                W = value
----- Width      域，用于声明Port域的位宽
Pin                   pin
Gate                 G = and + or + xor + nor + not + and
Arithmetic           A = add + diff + mult + div + split + bit extender
Mux                  M = Multiplexer + Demultiplexer + Decoder + Bit selector
Component            C = Gate + Pin + Arithmetic + Mux + Register
----- Component 域，代表元器件本身。
Port                 P = port
----- Port       域，代表元器件上的端口。
Single               S = Component X Port
----- Single     域，单个元器件最后的表现形式是 元器件类型+端口绑定。
Link                 L = Port X Port+
----- Link       域，决定了元器件间的连接情况，不局限于一对一，可以一对多。

Store                Sto = Single X Link
----- Store      域，代表元器件和连线的组合。
Environ              Env = Identifier->(bound new_Component + unbound )
----- Env        域，代表标识符到元器件的转换
```

语义

由于语法结构较为繁复，很难逐条对应写出EBNF的指称语义，这里针对关键的部分语法作说明。

输入输出声明（input_output_dec)

<input_declaration> ::= **input** [**wire**] [**signed**] [< range >] < list_of_port_identifiers >

<input_declaration> ::= **output** [（**reg** | **wire**）] [**signed**] [< range >] < list_of_port_identifiers >

< range > ::= **[** <msb_constant_expression > **:** <lsb_constant_expression > **]**

<lsb_constant_expression > ::= <constant_expression>

<msb_constant_expression > ::= <constant_expression>

这里假定有辅助函数 Evaluate ，可以实现 Evaluate（<constant_expression>) 计算出值，故从constant_expression不再往下分解。

构造辅助函数:

1、exist(< grammar >) 判断语句成分是否存在

2、check_range(< range >) 用于检查是否有范围说明，若有，则需要进一步语义分析并得出范围值。若无，则认为位宽为1 。

```
check_range : Value -> ( ( Value , Port ) -> Env  )
check_range(<range>) = let env = set_width(res , ports) in
                         let ports = get_ports() , res in
                       if not exist(<range>)
                       then res = 1
                       else
                        then res = Evaluate(msb_constant_expression -
lsb_constant_expression  )
```

3、set_width(value , ports) ， 用于将位宽属性绑定到每个端口声明标识符上 。

```
set_width(value,ports) = Value , Port -> (Width X  Ports ）->Env
```

4、add_reg() ， 用于检查是否声明为Reg变量。

5、Evaluate(<constant_expression>) ， 用于计算常量定义的位宽

6、set_sb(value,port) ,用于表示Pin实际中的最高最低位的数值

```
set_x_sb(value,port) = Value ->(Value X Port)
```

7、get_ports(ports = <list_of_port_identifiers>)，从后面的<list_of_port_identifiers>中得到声明的端口标识符

8、new_Comp(xxx) ， 用于在Store中新建一个元器件，并在Env中存在其名字（标识符）

9、add_attr(input/output),添加输入输出属性

指称语义如下：

```
lsb_constant_expression env : value-> ( value ->   Value X Port )

declare msb_constant_expression env =  let set_msb(Res, ports) in
                            let Res = value , ports =get_ports() in
                               value = Evaluate(<cosntant_expression>)
declare lsb_constant_expression env = let set_lsb(Res, ports) in
                            let Res = value , ports =get_ports() in
                               value = Evaluate(<cosntant_expression>)
```

```
range env : env -> env
declare  <input_declaration> env = let sto`,env` = new_Comp(Pin) ,env` in
                                     env` = {add_attr(input) env,
                                        check_range(<range>) env,
                                        declare msb_constant_expression env,
                                        declare msb_constant_expression env }

declare  <output_declaration> env = let sto`,env` = new_Comp(Pin) ,env` in
                                   let env` = {add_attr(output) env,
                                        add_reg(ports, flag ) env,
                                        check_range(<range>) env,
                                    declare msb_constant_expression env,
                                    declare msb_constant_expression env }in
                            ports = get_ports(<list_of_port_identifiers> ),
                            flag = exist(<REG>)
```

模块内的数据定义

模块内定义的连线类型与端口声明类似，只不过不包含input/output属性，Reg 声明可以直接作用于Sto域， 而Wire作用域 sto域需要通过 assign语句实现，声明阶段仅作用于Env域。其指称语义如下：

```
declare <net_declaration> env =    env` = { check_range(<range>) env,
                                            declare msb_constant_expression env,
                                            declare msb_constant_expression env
}
declare <reg_declaration> env = let sto`,env` = new_Comp(Register) ,env` in
                                    let env` = { add_reg(regs, flag ) env,
                                            check_range(<range>) env,
                                        declare msb_constant_expression env,
                                        declare msb_constant_expression env }in
                              regs = get_ports(<identifiers> ), flag = True
```

组合逻辑的指称语义

组合逻辑在VeriSim中有多种类型体现，这里进行部分列举：

1、分裂和聚合（split 、combine）

分裂和聚合都用到了Splitter元件，该元件既可以实现多条线从高到低组合，也可以实现单条线取部分位。这里进行举例说明：

构造辅助函数：

set_link(Comp1,port1,Comp2,port)， 实现两个元件的对应port相连,并在连接前进行位数检查

allocate env 构造一个临时wire变量。

```
allocate ENV : Env→ Env
```

cal_port_counter 计算splitter需要提供的端口数量，每个端口的位数

set_direction(Splitter，target=Combine), 可视作分线器的朝向决定了分线器的用途（聚合、分裂）

check_wire(< ident >) 检查ASSIGN的 左值是否为 wire类型对象，assign语句仅可以对wire进行赋值，对右值没有限制

聚合:

assign : Store X Env → Store X Env

assign 【A = {B,C} 】sto env = **if** check_wire(A) **then**

let sto' , env' **in**

let sto' = sto + Splitter + tmp + links , env' = env + tmp **in**

let links = {set_link(B, B.port1， Splitter， Spitter.port1)，

set_link(C, C.port1， Splitter， Spitter.port2)，

set_link(tmp, tmp.port， Splitter， Spitter.port3) }，

counter = cal_port_counter(Splitter) , tmp

in tmp = allocate env , Splitter= new_Comp(Splitter) sto

else ⊥

分裂:

```
assign 【{A,B}=C】 sto env = if check_wire(A,B) then
                                let sto' = sto+ Splitter + links in
                                let links = {set_link(B, B.port1 , Splitter , Spitter.port1) ,
                                            set_link(C, C.port1 , Splitter , Spitter.port2) ,
                                            set_link(A, tmp.port , Splitter , Spitter.port3) } ,
                                    set_direction(Splitter,Split) ,
                                    counter = cal_port_counter(Splitter) in
                                Splitter = new_Comp(Splitter) sto
                            else ⊥
```

数据选择（Mux）：

```
assign【A=< MUL1 > ? B： C 】 sto env = if check_wire(A) then
                                let sto' = sto+ Multiplexer+links in
                                let links = {set_link(B, B.port1,Multiplexer,Multiplexer.port1 ),
                                            set_link(C, C.port1,Multiplexer,Multiplexer.port2 ),
                                            set_link(A, A.port1,Multiplexer,Multiplexer.portout ),
                                            set_link(mul1, mul1.port1 ,Multiplexer,Multiplexer.enable)},
                                    Multiplexer in
                                Multiplexer = new_Comp(Multiplexer) sto
```

时序逻辑的指称语义

时序逻辑主要通过Always块实现，实现了对reg的赋值描述，包括trigger，enable位，input 三种port。这里用带posedge clk , conditional_blocking_assign 的always块举例说明：

辅助函数声明：

check_reg(< net >) 用于检查被赋值变量是否为Reg 。

```
check_reg 【net】 = ident → Bool（辅助域）
```

add_enable(a , b) sto 用于将变量绑定到寄存器的enable位。

```
add_enable 【a,b】 sto = let sto' = sto + link in
                         link = set_link(a,a.port1 , b , b.enable)
```

add_clock(a，b，c=Posedge) sto ,用于将trigger绑定到寄存器的trigger位，考虑到设计规范和现实的具体实现，默认都为posedge 。

```
add_clock 【a,b】 sto = let sto' = sto + link + set_posedge(b) in
                         link = set_link(a,a.port1 ,b , b.trigger)
```

```
<event_expression>  : Env → Env
event_expression 【a】 env = allocate a env
```

```
<conditional_b_assign>  :  Sto → Sto
<conditional_b_assign> 【 if (C) begin A <= B end 】 sto =
                    if check_reg【A】 then
                        let sto' = sto + links in
                            links = {set_link(B,B.port1,A,A.in_port),
                                     set_enable(C,A) }
                     else ⊥
```

Always块的指称语义如下：

always　sto env : Sto,Env → Sto，Env

Always【 (<event_expression>【D】, <conditional_b_assign> 【 if (C) begin A <= B end 】】 =

{ sto' = sto + add_clock【D,A】+ <conditional_b_assign> 【 if (C) begin A <= B end 】

,env' = env+event_expression【D】}

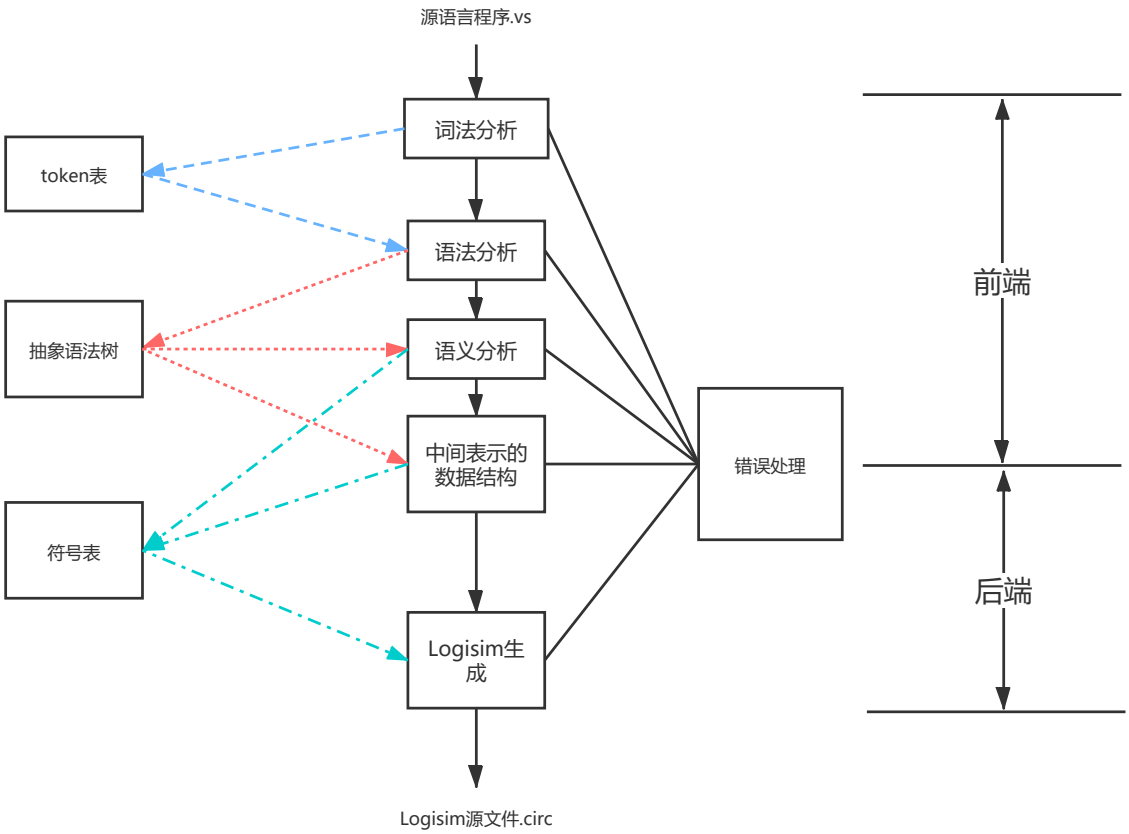
需要注意的是，时序逻辑中同样存在很复杂的情况，例如对每个变量取特定范围进行赋值，但实现的Logisim中寄存器无法部分赋值，因此在简单例子基础上的复杂变种不做讨论，敬请见谅。

编译器实现说明

本项目托管在Github上，项目目前共计约3000行python代码， https://github.com/jdmfr/Verisim-Compiler。

项目需求依赖包为PYPI的 spark-parser[2]，在Python 3.8.3 , 64-bit 环境下可以顺利编译运行。Spark-parser（与Spark无关）使用Earley算法解析上下文无关文法，类似于python中的ply包，利用正则表达式实现词法分析，还可以通过其提供的语法分析自定义抽象语法树，并通过抽象语法树解析框架自定义语义分析。该框架的使用特性，将分别在每一小节叙述。

利用该框架，实现的编译器框架如下图



编译器前端

词法分析

编译器的词法分析代码文件为`veriSim_scanner.py`，实现了从源文件到token表的转换。

首先，需要定义词例类，在`veriSim_token.py`中定义了`veriSimToken`类，包含两个属性，`kind`、`name`

```
class veriSimToken(GenericToken):
    def __init__(self, kind, name):
        self.kind = kind
        self.name = name

    def __str__(self):
        return 'Token %s: %r ' %( self.kind, self.name)
```

通过继承Spark-parser的`GenericScanner`类自定义词法分析器。若解析到非法词例，则通过`error`函数返回错误信息和位置：

```
    def error(self, s, pos):
            """Show text and a carot under that. For example:
            x = 2y + z
                ^
            """
            print("Lexical error:")
            print("%s" % s[:pos+10])  # + 10 for trailing context (尾部上下文)
            print("%s^" % (" "*(pos-1)))
            for t in self.rv: print(t)
            raise SystemExit
```

词法的解析规则，则通过`t_xxx`自定义函数，按照从上到下匹配第一个满足的正则表达式的规则调用解析函数。例如，解析括号字符的函数：

```
    def t_paren(self, s):
            r'[(){}[\]]'          #正则表达式规则
            self.add_token(BRACKET2NAME[s], s)
            #匹配后的语法动作，将kind赋为括号的类型，name为匹配的括号字符本身
```

所有解析到的词例都会通过`add_token`函数加入到`self.rv`这一列表中，当调用`tokenize`方法解析源文件且不发生错误后，会将`rv`列表返回。

```
    def tokenize(self, string):
        self.rv = []
        GenericScanner.tokenize(self, string)
        return self.rv
        # self.rv中按顺序存储了全部的token
    def add_token(self, name, s):
        t = veriSimToken(name, s)
        self.rv.append(t)
```

注意到，在前述`VeriSimToken`中定义了每个词例的字符串表示。因此，可以对返回的列表逐个打印，得到`token`列表。对照打印得到的列表，可以方便下一步语法分析模块的代码调试。

语法分析→语法抽象树

编译器的语法分析代码文件为`VeriSim_Parser.py`，定义了继承于`GenericParser`的`VeriSimParser`类，实现自定义的抽象语法树生成。抽象语法树的结点为`AST`，kids中包含子AST结点及其语法成分，保存在userlist中，kind定义为AST的种类，通过定义不同种类，可以在语义分析阶段解析为各种语义，关于对语法抽象树进行语义分析，在语义分析阶段进行描述。

```
class AST(UserList):
    def __init__(self, kind, kids=[]):
        self.kind = intern(kind)
        UserList.__init__(self, kids)
```

当语法分析出错时，会通过`GenericParser`类中的`error`函数定位到出错的`token`并进行信息返回，虽然无法知道当前解析到什么语法结构，但可以通过分析语法EBNF、错误的token位置找到大致的错误点。

```
def error(self, tokens, index):
    print("Syntax error at or near token %d: `%s'" % (index, tokens[index]))

    if "context" in self.debug and self.debug["context"]:
        #这里的context 为定义的debug开关。在debug结构中定义context后即可返回报错信息
        start = index - 2 if index - 2 >= 0 else 0
        tokens = [str(tokens[i]) for i in range(start, index + 1)]
        print("Token context:\n\t%s" % ("\n\t".join(tokens)))
    raise SystemExit
```

Spark-parser框架利用的解析算法为Earley算法，在编译器实现中我们不需要过多关注语法解析形式，只需要注意定义的语法规则即可（例如应实现为左递归，消除所有右递归）。并不直接支持EBNF文法，所以在定义语法规则时做了相应的修改。

解析语法的功能函数，其名字需要定义为`p_xxx`，解析顺序依旧是从上到下，匹配第一个相符的规则。在实际操作中，总结了以下几条要点：

1、每个解析函数的名字唯一定义，如果出现重复定义，例如在语法解析类内定义了两个同名函数，虽然它们实现的解析功能不同，但可能会产生无法预测的错误。

2、parser识别的是token的kind，因此需要将每个终结符大写，符号需要转化为对应的kind，不能用token的name表示。如：

```
def p_net_reg_dec(self,args):
    '''
    reg_declaration ::= REG signed_opt range_opt list_of_variable_identifiers
SEMICOLON
    '''
    return AST('REG',[args[2],args[3],AST('dec_reg_flag',[None])])
    ## '''注释为定义的语法规则， REG SEMICOLON 代表token中的 保留字'reg' 和 特殊符号';'
    ## return 则是返回了根据该条语法规则生成的AST节点
```

3、针对EBNF中的可选元素`[ < xxx > ]`，除非语法规则的右边仅有一个 终结符或非终结符（此时可以将其表示为 `::= xxx?`），否则不能直接表示，需要拆分。 这样制定规则是方便语法树的迭代时，args列表中每个单元都有对应的语法要素。

为了维持统一的代码风格，这里将其转换为 `xxx_opt` ，并新增对`xxx_opt`的语法分析。例如：

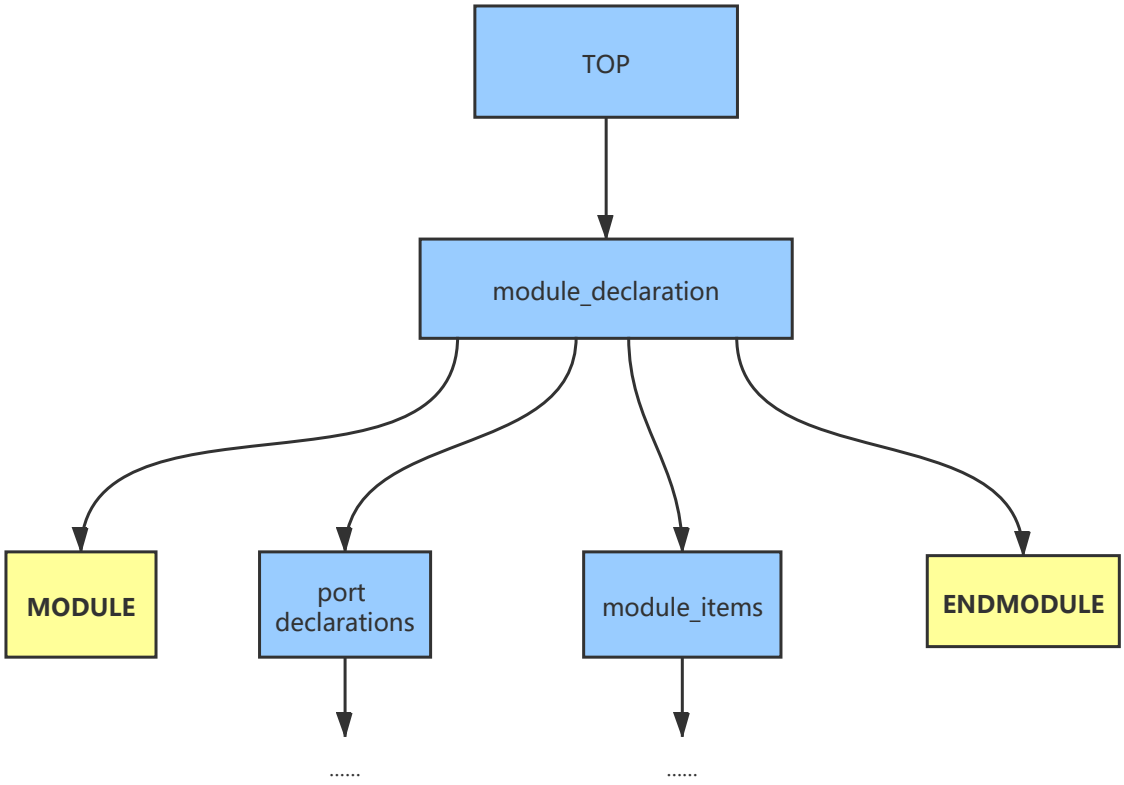
```
    def p_port_output_decla(self,args):
        '''
        output_declaration ::= OUTPUT wire_opt signed_opt range_opt
list_of_port_identifiers
        '''
        # ......
        return # .......

    def p_port_range_decla(self,args):
        '''
        range_opt ::= range
        range_opt ::=
        '''
        tmp = None
        if len(args) ==0 :
            tmp = AST('WIDTH', [None] )
        else :
            tmp = AST('WIDTH', [args[0]])
        return tmp
```

4、针对EBNF中的重复元素 `{ < xxx > }`， 除非语法规则的右边仅有一个 终结符或非终结符（此时可以将其表示为 `::= xxx*`），否则不能直接表示，需要将其拆分。这样设置的原因同上，同样，为了维持统一良好的编码风格，这里将其转换为 `xxxs` ，并增加对 `xxxs` 的语法分析，这里需要将xxxs转换为左递归文法。例如：

```
def p_m_g_items(self,args):
    '''
    module_or_generate_items ::= module_or_generate_items
module_or_generate_item
    module_or_generate_items ::=
    '''
    if len(args)>0:
        return AST('More',[args[0],args[1]])
```

解析为第一句时，参数表长度为2，解析为第二句时，参数表长度为0。这样做的好处是，将相同的语法规则根据参数表长度进行区分，统一处理，方便查看，并减少代码复杂度。

语法解析的结果，是生成了一棵根节点类型为TOP的树，结构较为臃肿。通过定义不同类型的节点，我们可以绑定对应的解析函数。**好处**是方便为语言添加新的语言特性，只需要添加新的节点类型，并在语义分析器中定义对应节点的解析方式即可；**缺点**是节点类型太多，不方便维护。以translation_top为起点的解析树是如下结构：



注：黄色元素代表终结符、蓝色元素代表非终结符

语法抽象树→语义分析→中间形式的数据结构

抽象语法树到中间形式，需要经过语义分析单元。值得注意的是，在语法分析阶段我们只是建立了抽象语法树，而对标识符含义没有解析。符号表是在语义分析阶段产生的。总地来说，语义分析主要进行三方面工作： 建立符号表、生成"元件-连线"列表、完成部分错误处理。

符号表的相关数据结构代码在`Verisim_dor_ST.py`中，`SignTable`类为符号表，提供部分查询接口，用于错误处理等，`Sign`类为符号表项。

```
class SignTable(GenericToken):

    def __init__(self):
        self.rv = []
    # ......
class Sign:
    def __init__(self, kind , name ,upper=None ,typer=None):
        self.kind = kind
        self.name = name
        self.type = typer
        self.upper = upper
        self.msb = 0
        self.lsb = 0
        self.size = 1
```

Spark-parser同样提供了语义分析框架`GenericASTTraversal`，可以实现抽象语法树的遍历。我们可以自定义语义分析器`Interpret`。

根据指称语义的分析结果，在类中可以自定义辅助函数，例如：针对需要生成临时变量（wire），通过函数`gen_tmp`可以得到。

`GenericASTTraversal`原生提供了两种遍历方式： 先序遍历、后序遍历。考虑到我们仅生成了一棵树，且树结构较为繁杂，最后决定采用先序遍历的方式解析语法树，遍历与原生稍有不同，在`Interpret`中重写了该函数：

```
class Interpret(GenericASTTraversal):
        #......
    # Override
    def preorder(self, node=None):
        #......
```

根据AST节点类型进行解析，类内提供了方法`n_xxx`,即可解析kind=xxx 的节点。对于中途解析完毕，不需要再解析子节点的情况，可由`self.prune()`进行剪枝。例如：

```
def n_cal_op(self,node):
        self.cal_op = node.data[0].name
        self.op_l = self.cur_name
        self.prune()
```

而遍历也有不方便之处，例如，在树上遍历时，端口的声明不知道何时停止。这时可以在语法分析阶段增加一个kind为FLAG的节点，作为port_declare的最后一个子节点。在解析该部分子树的过程中，逐渐收集声明所需信息，直到遇到FLAG结点，跳转进入标识符定义的辅助函数。例如：

```
def n_dec_reg_flag(self,node): # in semantic
        self.cur_kind = 'NORMAL'
        self.wire_type='REG'
        self.dec_flag = True
        self.SignTable.check_dup(self.cur_name)
        self.new_decla()
#------------
def p_net_reg_dec(self,args): # in parser
    '''
    reg_declaration ::= REG signed_opt range_opt list_of_variable_identifiers
SEMICOLON
    '''
    return AST('REG',[args[2],args[3],AST('dec_reg_flag',[None])])
```

在new_decla函数中，可以进行标识符重复声明的错误类型检查，检查通过后完善符号表，这里不再赘述。

完成语义分析后，就可以得到后端需要的数据列表。

后端生成

运行实例

[1] Verilog语义的ASM表示方法研究

[2] Spark-parser https://pypi.org/project/spark-parser/

[3] 领域专用语言实战/(美) Debasish Ghosh著 郭晓刚译