

<rhFlow Spec>

[Document Subtitle]

RyanHunter

[Company Name]

Revision History

v1.00 - initial version - November 22, 2021

1. rhMain architecture

The main tool has following features

* provides basic APIs for project or subFlow to create contents;
* capable to manage the multi-thread actions for accelaration;
* different flow steps can be executed separately by calling specific exe options
* log report control, different actions may have different log files
* build flow, to build source files into standard language files, such as verilog
* support loading project files, the mian entry is <PROJECT\_HOME>/node.rh
* support loading other flows, by loading its main node.rh entry

1. buildFlow architecture

buildFlow is a flow to build up design/env, from self-defined source contents into a unified file, such as verilog. The buildFlow will have following features for easy build and collecting:

* support to publish a specified file and all its dependences (mostly its instances);
* buildFlow can support design/env build for now;
* assemble a component by many of other information within it;
* propose a feature concept to describe all of the design logics;
* support to instantiate a third vendor standard design file
  1. buildDesign architecture
     1. entry node

file architecture of this flow: buildFlow/<version>/node.rh is the main entry, will load other files through this node file.

* + 1. signal.rh

This file only declared a Signal class in it.

* + - 1. class Signal

Signal class is an object presents all signal features of circuit. the class provides APIs as fllowing shows.

Signal class is a class to describe every nets/wires/inputs/outpus, so it has many attributes for representing those features and so as APIs to access these attributes.

* + - * 1. direction

Indicates the direction type of this signal, the direction can be input/output/inout/reg/wire. Later we can add more types to support full RTL features.

* + - * 1. width

Indicate the valid width of this signal, currently we only support signal bits start from bit0

* + - * 1. driver

driver is a hash indicates whom its every bit was driven by, and here the signal will store it's logical driver only, which means if this signal is from another named wire signal, then it will store that wired signal as its driver.

By using of this feature, we can detect multi-driven at a very early phase of the whole IC development.

example of driver hash: driver['3:0'] = 'signalA[3:0]', driver['15:4'] = 'signalB['11:0']

* + - * 1. loader

Similar with driver, stores all its loaders, one signal can have many of the loaders, it's also a hash.

* + - * 1. reset

This is only available for register type, indicates its reset value.

* + 1. action

The term action used within chapter 2 is specifically for design action.

By calling different action APIs, a user-defined feature can be created.For easily call those actions, every DesignAction will be stored in Feature class.

We don't recommand users to add self-defined actions even the flow provides this funcionality.

* + - 1. builtin actions

**condition**

For generating conditional blocks according to configures while calling this API.

**FSM**

To generate a FSM, need configurations while calling.

**connect**

This is for combinational connect 2 nets, configuration of this are:

* from
* to

**seqstart**

The action to start a sequential block, configuration of this are:

* reset: if this has sync reset/async reset/no reset; if has reset enabled, then the reset signal is required;
* sensitive list: kinds of pos/neg or both of a signal

**seqend**

That's the simple action to end the sequential block

* + 1. component

Component is a class defined in this file, and this class stores component relative APIs and information for publishing RTL code. And so as global component function.

By calling that global component API, users can add a new component with a specific name, and configurations within do-end block.

This API will call a singleton component method in Component class, and collect the configurations in do-end block, store those configurations into an instance of Component. So, key configures of a component are: ports, instances, features.

Besides core configurations above, we're aiming to establish a component building mechanism that can support variaty of component creation. So we'll let users add their specifc configures into a Component instance (this can be a further enhancement since we now have no any existing flow)

* + 1. feature

Features are actually logical blocks of RTL. By loading specific features with a bunch of configurations users can simply assemble a new component. So there're three major steps for using a feature:

* create a feature API
* create and load a new feature by calling the feature API
* set up feature instances in a component

While creating a new feature, it'll also create some configurations for instantiating, so the flow should allow developer to create new configurations.

* + - 1. feature creation

A new feature is created by calling the global feature API, which will collect actions within do-en block, and define a method named as the <arg::name>. A new feature's key requirements are: signals and actions.

Signals is an array storing Signal typed instances used in current feature, the flow provides APIs for easily adding these signals while instantiating a feature.

Actions is defined while creating a new feature, it specifies what behaviors will be done within this feature. Action is a class type, it can be added by user and also has some pre-defined actions.

* + - 1. feature instance

After defining or loading a named feature, users can call to use that feature by the feature name as a defined method, and with several configurations within do-end block.

* + 1. database

The database is the storage that contains information about components, features, signals and all other information. And can be accessed by other parts of the flow.

* 1. rhvlog.rh

This is a standalone lib for generating verilog language, it provides APIs for abstraction layer user, and will generate all possible verilog sources. Not only for verilog source, there also some of RTL generators.

**ifBundle <&block>**

This API will publish a verilog block with a format of if-elseif-else. The <&block> is used by the caller to setup the conditional bundles, act like a configuration.

* + 1. class RhVlog
       1. ifBundle <&block>
  1. buildEnv(TBD)
     1. features for env building

1. buildDesign implementation

This chapter will describe detailed implementation based on the buildDesign architecture. It's being a suppliment of that corresponding architecture. While writing this chapter, we may reconstruct the architecture or some proposals decided before.

By this action, user can publish a bunch of conditional logic, once calling this action, one if-elseif-else-end block will be generated. For using this action, users just need to add configurations within condition calling block.

**setup condition**

This section evaluates the behavior within code block of the add method. Steps of the block within:

* variables initialization, such as signals, and condition bunches
* call the &block from calling the condition method, in this block, initials should be done
* format and generate RTL code

**configure condition action**

code block is the configuration to setup a customized condition block, key configurations of this action is: TODO

* 1. global APIs

By loading the buildFlow, some of the global APIs should be ready to be invoked by project creator. In perspective of design, there're global APIs listed:

* + 1. def component

This global component method is defined in component.rh file.

This is created to assemble a standalone module of RTL (verilog).

It's a global function that can be called by project owner to setup a component. Some key information should be determined within the code block when calling this API: ports, direct connections, instances, and features.

**ports**

While declaring a component, we need port information to tell generator tool that which part of signals that are designate to be connected outside this module, and so its direction and width.

**directConnect**

This is a simple action, to connect 2 nets directly, no any other logic.

**instance**

The instance feature allow users to instantiate other components or standard RTL module, with some of the connection description. Here we need to find out an easy way instantiating other components/modules. (TBD)

**feature**

Any logical operations are encapsulated as features, no matter sequential or combinational logics, we assume to use feature to describe those in a component, by that way we can reuse existing features by only a few of configurations, and can add more actions to make coding more flexible compared to the raw RTL languange (verilog).

* + - 1. arguments

**name**

the name argument specifies the name definition while declaring a new component, it can be a string or a symbol, but finally it will stored as a symbol.

**&block**

the block argument here is kind like a handler that points to a bunch of declaration of this newly created component. Those desclarations are more like:

* addPort :iClk, :i, 1, 0
* aNamedFeature do xxxx end
* connect xxx
* instance xxx do xxx end
  + - 1. procedures

major steps:

* get name argument, and create a new Component instance and register it to database
* store the &block into that newly created Component instance
  + 1. def feature

This provides users APIs to establish a new logical feature block for further building a design.

For users to create a new feature, call a global function: feature. By doing that, a new <class::Feature> will be added to the feature database. Besides, multiple feature blocks with the same name can be added separately and the tool will automatically collect those and executed when in publish.

How to distinguish a call of feature is aiming to add a feature or use a feature?

By calling a feature like <feature :aFeature do-end>, users are going to establish a new feature as a name of :aFeature. Then, users can use that feature by calling the method named as <aFeature>.

* + - 1. arguments

**name**

indicates the name of feature, which will be called as a method when instantiating a new feature.

**&block**

a code block to setup a customized feature. So as configurations of a feature.

* signals, all signals that will be used in this feature block, A hash of Signal class type, stores regs/wires that used by this feature.
* actions, A hash of DesignAction class type, stores signals behavior, some of them are built-in actions, and user-defined actions are also supportive. Built-in actions, such as conditional block, direct connection, FSM are defined within the DesignAction class. Detailed information can be found in Section: design\_action.rh. Actions are defined within Feature class.
  + - 1. procedures
* create a new instance of Feature class,
* call block to initial all signals and invokate different actions
  + - 1. examplify

using ref <PROJ>/metaFlowVersion/demos/example.feature

* + 1. def action

This is a global def for building different actions, this method will create a new method of named action within the DesignAction class, and the class will stored within a Feature class.

* + - 1. arguments

**name**

name of action, used for the name of the newly created action.

**&block**

configure of the new action.

* + - 1. procedures

This action method first executes <block> in which calls a bunch of DesignAction's configure methods to setup a new instance of DesignAction; then this method will call DesignAction's add metho to add a new action.

* 1. class DesignAction

This is the class type for all legal design actions, will provide APIs and attributes for easy developing an action.

Action of design is eventually to generate RTL code, the only difference is the way to generate it.

* + 1. @@actions

A static hash of DesignAction that stores all available actions

* + 1. def add <name> <&block>

To add a new action and stored in @@actions of this class, the add method will define a new method named as <name>, and its block, is just what the new method <name> do.

* + - 1. procedures

This method is called after the configures of this DesignAction instance is setup; then in this method, we'll create a new <name> method with a block from using the <name> action; after defining the new method, we should also register this method into DesignAction class.

* 1. class Component

rtlLanguage

Indicates which language to publish, default is verilog. As a further feature, this config can be set to other language type.

signals

Stores interfaces of this component, all I/O and nets are stored here, is of Signal type. But for a component, only ports like input/outpu/inout should be listed in signals hash, other internal signals will not manually listed, but should be generated by component tool automatically.

params(TBD)

This is of parameters and macro defines of a verilog based feature. Mostly we're not recommand to use RTL parameters since the source code are already support parameters and configurations.

subInstances(TODO)

subInstance stores a vlnv+connection list of components that will be instantiated within this component. By using API of instance to add components and connections, detail description of instance will be list below within Component class.

third vendor RTL supports(TODO)

Sometime we may instantiate a third vendor RTL module, to support this, we add a normal instantiation with

1. simFlow architecture

simFlow is the flow for simulation, such as running UVM tests. Features of this flow contains:

* support simulation only, can support different EDA tools
* separated phases for simFlow: compile phase/elab phase

1. testFlow
2. usage examples