Open Source Device Interface (OSDI) Specification - Working Draft

Version 0.3

July 27, 2022

Contents

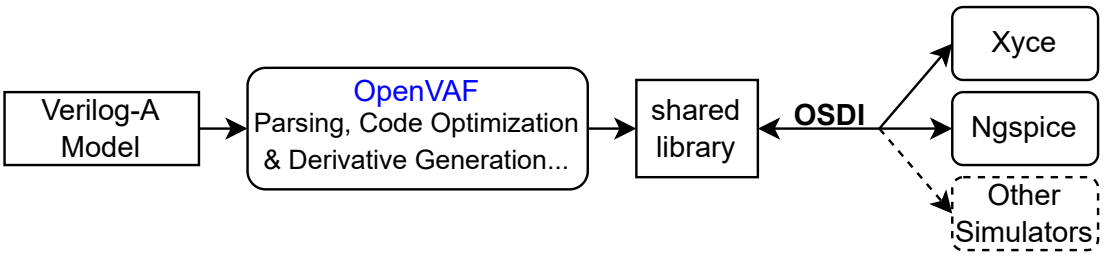
1 Introduction

All circuit simulators have their own unique interface for incorporating compact semiconductor device models. This slows down model integration, development, distribution and standardization. To remedy this situation, the Verilog-A language has been introduced as a means of having a unified description of compact models.

Most simulators are using the transpiler ADMS for incorporating Verilog-A models, yet ADMS's XML file-based transpilation approach has significant disadvantages compared to a true compilation approach with respect to execution time, compilation time and language standard support.

OSDI overcomes these disadvantages by defining a simulator-independent interface. Compact Verilog-A models are compiled to shared libraries that adhere to the OSDI interface as defined herein. Circuit simulators can then implement code that enables to interface with OSDI. The open-source Verilog-A compiler OpenVAF will serve as a back-end for generating shared libraries that adhere to OSDI from Verilog-A models.

Foremost, this positions OpenVAF and OSDI as the new open-source standard compiler solution for Verilog-A compact models, which in turn would be excellent for Verilog-A standardization. Furthermore, having a ready-to-use compiler will help circuit simulators obtain Verilog-A language support quicker, at reduced implementation and time effort.



2 General Overview

2.1 Exported Symbols

Each compiled shared library exports the following four symbols:

```
extern uint32_t OSDI_VERSION_MAJOR;
extern uint32_t OSDI_VERSION_MINOR;
extern OsdiDescriptor OSDI_DESCRIPTORS[];
extern uint32_t OSDI_NUM_DESCRIPTORS
```

The symbols `OSDI_VERSION_MAJOR` and `OSDI_VERSION_MINOR` indicate the OSDI version that the shared library was compiled with. Simulators that implement OSDI version `X.Y` must be able to load all shared libraries with `OSDI_VERSION_MAJOR = X` and `OSDI_VERSION_MINOR <= Y`. While OSDI is under development (Version `X = 0`) simulator implementations only have to support `OSDI_VERSION_MINOR = Y`.

`OSDI_DESCRIPTORS` is a list of device descriptors of length `OSDI_NUM_DESCRIPTORS`. A device descriptor is an instance of the `OsdiDescriptor` struct that contains all compiled information about a device model (Verilog-A module).

Compared to hard-coded models, OSDI models do not call simulator-specific functions (with a couple exceptions). Instead, `OsdiDescriptor` contains static metadata, enabling more flexibility for the simulator implementation. Compiler implementation is also simplified because static data is less complex to generate than executable code.

To store the metadata, OSDI defines several data structures. The static metadata is closely intertwined with the behavior of the compiled functions, hence both must be described together. Herein, first the overall working principle of the interface is roughly explained for each simulation stage. Thereafter, all data structures and routines are formally documented.

2.2 Model Parameter Input

During the first stage of simulation, the simulator reads the user configuration, creates the models / instances and populates these with parameters. After the correct device descriptor for each device is identified by the simulator, the size of the model and instance data can be obtained. The simulator must then allocate the required memory for storing this data.

A list of all model parameters is provided in the structure `param_opvar`. This list contains all static information about the parameters such as their names, types and so on. It does **not contain** the default values and bounds of the parameters, because these can depend on the value of other model parameters, simulator parameters and even temperature. Populating default values and checking parameter bounds are handled by OSDI during the circuit setup stage.

The values of parameters are stored within the model and instance data. These values can not directly be accessed by the circuit simulator since later simulation stages must know which parameters have been explicitly set, and which have remained at their default value. For accessing parameters, the `access` function must be used, it returns a pointer to a parameter and corresponding information. A set of flags is provided to this function that indicate how the simulator intends to use the pointer, so the OSDI model can update internal data correctly.

2.3 Circuit Setup

After the model / instance data is created and populated, the simulator must create the required nodes and respective jacobian entries. Furthermore, default values for all parameters that were not explicitly set during the parameter input stage are calculated. Finally, all calculations that only depend on parameter values and temperature are executed.

The `setup_model` routine handles model parameters (and dependent calculations). The `setup_instance` routine does the same for instance parameters. Furthermore, the `setup_instance` routine handles node collapsing and is therefore closely intertwined with the metadata that describes nodes and jacobian entries.

A list of all nodes and **non-zero** jacobian entries is provided by the `nodes` and `jacobian_entries` fields. The indices of these lists are very important as they are referring to jacobian entries and nodes elsewhere.

Multiple nodes may be collapsed into a single node, depending on the model and instance parameters. The `collapsible` field contains all pairs of nodes that **can possibly** be collapsed. Which node pairs are actually collapsed for a given instance is determined in the `setup_instance` routine, because node collapsing can not depend on the operating point. The results are stored within the instance data at offset `collapsed_offset`.

The simulator must use all this information to create a mapping from OSDI node IDs to circuit nodes. This mapping between node IDs must be stored at `node_mapping_offset` within the instance data. Using this mapping, all required matrix entries must be created by the simulator. The simulator is expected to write pointers to these entries into the instance data at `jacobian_ptr_resist_offset` and `react_ptr_off`.

Furthermore, the simulator must reserve for `num_states` values in the instance state. The indices of the reserved state entries must be stored in the instance data at

`state_idx_off`. During evaluation these indices will be used to access the state in the `prev_state`/`next_state` pointers.

2.4 Evaluation

During the evaluation stage the operating point dependent Verilog-A code and its derivatives are evaluated. The results of these calculations are then loaded into the simulator RHS and jacobian entries. For accommodating all use cases, the evaluation stage is very flexible as OSDI supports different simulation modes and different simulator implementations.

For the most part these tasks are performed by the `eval` function. This function calculates the resistive residuals / jacobian entries and the reactive residuals / jacobian entries and stores them in the instance data. The reactive residuals / jacobian entries contains values that depend on time via (`ddt`), i.e. charges/capacitances.

This formulation is well suited for simulators that implement harmonic balance simulations like Xyce. These simulators can simply use the `load_residual_resist`, `load_residual_react`, `load_jacobian_resist` and `load_jacobian_react` functions to copy the residuals and jacobian entries from the instance data into the global simulator state.

However, traditional simulators are not directly compatible with this approach. The `load_jacobian_tran` function is available to support loading the reactive and resistive jacobian entries into a single matrix. For DC and AC simulations the `load_jacobian_resist` and `load_jacobian_react` functions can be used.

SPICE-like simulators in particular use the formulation of the NEWTON method shown in (2.2), instead of the usual formulation shown in (2.1). Therefore, the RHS of SPICE-like simulators contains further terms in addition to the residual. To support these simulators the `load_spice_rhs_dc` and `load_spice_rhs_tran` functions are available.

$$J(x_k - x_{k+1}) = F(x_k) \tag{2.1}$$
$$Jx_{k+1} = Jx_k - F(x_k) \tag{2.2}$$

3 Routines

Information that can not be exposed as static metadata can be accessed using the following routines. This comprises mainly compiled behavioral Verilog-A code and access to heterogeneous data inside the model and instance data.

All routines require a pointer to the model and/or instance data as arguments. These pointers must be allocated by the simulator with correct size, alignment and must be initialized with zeroed bytes. This can be achieved as follows:

```
void* inst = calloc(1, descriptor->instance_size);
void* model = calloc(1, descriptor->model_size);
```

All functions that execute Verilog-A behavioral code must also be provided with the `void *handle` argument. This argument is used to invoke the `osdi_log` functions whenever Verilog-A emits messages. The contents of this pointer are entirely up to the simulator.

3.1 access

```
void *access(void *inst, void *model, uint32_t id, uint32_t flags)
```

This function allows the simulator to read and write parameters as well as operating point variables. A pointer to the data corresponding to `param_opvar [id]` is returned and can be accessed by the simulator.

In order to work, the OSDI shared library must know how the simulator intends to use the returned argument. To that end bit flags are set in the `flags` argument. Reading the pointer never has an effect on the internal data and therefore requires no special flags. When `ACCESS_FLAG_SET` is set, the pointer can be written as well.

By default, the `access` function can be used to access model parameters. To access the corresponding instance parameter instead, `ACCESS_FLAG_INSTANCE` must be provided. If an instance parameter is not set for an instance, the parameter is copied from the model during `setup_instance`.

Note that none of these flags apply to operating point variables.

3.2 setup_model

```
void setup_model(void *handle, void *model, OsdiSimParas *sim_params,
                                 OsdiInitInfo *res)
```

This function initializes all parameters that were not explicitly set using `access` and also performs a bounds check for all model parameters and instance parameters. Additionally, it executes all Verilog-A code that does not depend on:

- operating point
- analysis mode
- values of instance parameter
- `$param_given` with instance parameters as argument
- `$port_connected`
- `$temperature`
- `$simparam` and `$simparam$str`
- `@final_step` event

This function provides an instance of the `OsdiInitInfo` into the provided pointer. This struct contains a list of errors that occurred while checking the model parameters. Additionally, it contains any execution flags emitted by the behavioral Verilog-A code.

Whenever the model parameters change, this function must be called. Whenever this function is called for a model, the `setup_instance` routine must be called for all its instances as well. An `OsdiSimParas` struct is required by this function if any parameter default values invoke `$simparam`.

3.3 setup_instance

```
void setup_instance(void *handle, void *inst, void *model,
                                  double temperature, uint32_t num_terminals,
                                  OsdiSimParas *sim_params, OsdiInitInfo *res)
```

This function initializes all instance parameters that have not been explicitly set, and also performs bounds checks for all instance parameters. Instance parameters that were not set for the instance but are set for the model are copied into the instance. The routine executes all Verilog-A code that was not executed in `setup_model` and does not depend on:

- operating point
- `@final_step` event
- `$simparam` and `$simparam$str`

This function requires a `OsdiSimParas` struct in case any parameter default values invoke `$simparam`. It puts an instance of the `OsdiInitInfo` into the provided pointer. This struct

contains a list of errors that occurred while checking the model parameters. Flags emitted by the behavioral Verilog-A code are also contained in this struct.

A list of collapsible nodes within the instance data at `collapsed_offset` is also set-up by this function. When this function is called repeatedly (e.g. during a parameter sweep), the simulator must ensure that node collapsing is updated whenever `collapsed` changes. Alternatively, the simulator can disallow such sweeps producing an error whenever `collapsed` changes.

This function must be called whenever the model or instance parameters are changed.

3.4 eval

```
uint32_t eval(void *handle, void *inst, void *model, OsdiSimInfo *info)
```

This function evaluates the remaining behavioral code in the Verilog-A analog block and puts the calculated jacobian entries and residuals in the instance data. Therefore, it encapsulates the major part of the Verilog-A code. This function can be configured to calculate multiple results that can be accessed by the simulator with separate functions:

- operating point variables - read with the `access` function
- noise source arguments - read with the `load_noise` function
- resistive residual - read with the `load_residual_resist`, `load_spice_rhs_dc` or `load_spice_rhs_tran` function or directly by using `resist_residual_off`
- reactive residual - read with the `load_residual_react` or `load_spice_rhs_tran` function or directly by using `react_residual_off`
- resistive jacobian entries - read with the `load_jacobian_resist` or `load_jacobian_tran` function
- reactive jacobian entries - read with the `load_jacobian_react` or `load_jacobian_tran` function

This function does not calculate any numeric derivatives. Simulators that do not support a separate resistive and reactive RHS must calculate and load the numeric derivatives of the reactive residual themselves. This code will look roughly as follows:

```
if (descriptor->nodes[i].is_reactive) {
    rhs[node_mapping[i]] += numeric_derivative(reactive_residual[i]);
}
```

3.5 load_noise

```
void load_noise(void *inst, void *model, double freq, double *noise_dens,
                double *ln_noise_dens)
```

This function is the primary function to be called for noise analysis. It generates the frequency dependent noise densities using the (operating point dependent) results of the `eval` function. The results are written into `noise_dens` and `ln_noise_dens`. These two pointer-must each point to a list of doubles with length `num_noise_src`. The element `noise_dens[i]` is set to the noise density that corresponds to the noise source described in `noise_sources [i]`. `ln_noise_dens[i]` is set to `log(noise_dens[i])`.

3.6 load_residual_resist

```
void load_residual_resist(void *inst, void* model, double *dst)
```

This function adds the resistive residuals calculated by `eval` function to the global simulator RHS. The results are read from the instance data and added to the value at the provided pointer location. Using the positions provided in the instance data at `node_mapping_offset`, the values are written.

3.7 load_residual_react

```
void load_residual_react(void *inst, void* model, double *dst)
```

This function adds the reactive residuals calculated by `eval` function to the global simulator RHS. The results are read from the instance data and added to the value at the provided pointer location. Using the positions provided in the instance data at `node_mapping_offset`, the values are written.

3.8 load_limit_rhs_resist

```
void load_limit_rhs_resist(void *inst, void* model, double *dst)
```

This function adds the resistive residuals calculated by `eval` function to the global simulator RHS. The results are read from the instance data and added to the value at the provided pointer location. Using the positions provided in the instance data at `node_mapping_offset`, the values are written.

3.9 load_limit_rhs_react

```
void load_limit_rhs_react(void *inst, void* model, double *dst)
```

This function adds the reactive residuals calculated by `eval` function into the global simulator RHS. The results are read from the instance data and added to the value

at the provided pointer location. Using the positions provided in the instance data at `node_mapping_offset`, the values are written.

3.10 load_spice_rhs_dc

```
void load_spice_rhs_dc(void *inst, void* model, double *dst,
                double* prev_solve)
```

This function adds the resistive residuals calculated by `eval` function to the global simulator RHS. The results are read from the instance data and added to the value at the provided pointer location. Using the positions provided in the instance data at `node_mapping_offset`, the values is written.

Compared to `load_residual_resist`, this function adds the additional terms in (2.2) required for use within SPICE-like simulators. For calculating these terms, the `prev_solve` vector must be supplied to this function.

3.11 load_spice_rhs_tran

```
void load_spice_rhs_tran(void *inst, void* model, double *dst,
                double* prev_solve, double alpha)
```

This function adds the resistive residuals calculated by `eval` function to the global simulator RHS. The results are read from the instance data and added to the value at the provided pointer location. Using the positions provided in the instance data at `node_mapping_offset`, the values are written.

This function also adds the additional terms in (2.2) required for SPICE-like simulators. Compared to `load_spice_rhs_tran`, this function also considers the reactive jacobian entries while calculating the RHS. The reactive terms are multiplied with the `alpha` argument. `alpha` should be set to the factor that the reactive residuals are multiplied with during numeric differentiation. To calculate these terms the `prev_solve` vector must be supplied to this function.

This function does not add the reactive residuals to the simulator RHS. These terms must be numerically differentiated first which is simulator specific.

3.12 load_jacobian_resist

```
void load_jacobian_resist(void *inst, void* model)
```

This function adds the resistive jacobian entries calculated by `eval` function into the global simulator jacobian. The results are read from the instance data and added to the value at the provided pointer at `jacobian_ptr_resist_offset`.

3.13 load_jacobian_react

```
void load_jacobian_react(void *inst, void* model, double alpha)
```

This function adds the reactive jacobian entries calculated by `eval` function into the global simulator jacobian. The results are read from the instance data and added to the value at the provided pointers at `react_ptr_off`.

Every reactive (+jacobian entry) is multiplied with `alpha`. For traditional simulators without separate resistive and reactive jacobian entries this argument can be set to $2\pi f$ to obtain the complex matrix during AC analysis.

3.14 load_jacobian_tran

```
void load_jacobian_tran(void *inst, void* model, double alpha)
```

This function adds the resistive and reactive jacobian entries calculated by `eval` function into the global simulator jacobian. The results are read from the instance data and added to the value at the provided pointers at `jacobian_ptr_resist_offset`.

This function is intended for transient simulations in traditional simulators without separate resistive and reactive jacobian entries. Every **reactive** jacobian entry is multiplied with `alpha`. `alpha` should be set to the factor that the reactive residuals are multiplied with during numeric differentiation.

4 Callbacks

OSDI tries to avoid callbacks to simulator-specific functions. However, such callbacks can not be entirely avoided, hence additional symbols for these callbacks are exported by OSDI. These symbols are initiated by the simulator with appropriate function pointers. If the simulator does not populate one of these symbols undefined behavior will occur (unless specified otherwise). The compiler is allowed to omit generating these symbols if the appropriate functions are not used. An example for populating the`osdi_log` callback is shown below.

Every callback accepts a `handle` pointer. This pointer is passed by the simulator to any function that executes Verilog-A behavioral code. It is intended to allow the callbacks to access simulator-specific data.

```
void **osdi_log_ = (void**) dlsym(lib, "osdi_log");
if (osdi_log_){
  *osdi_log_ = (void*) osdi_log_impl;
}
```

4.1 Log Messages

```
extern void (*osdi_log)(void *handle, char* msg, uint32_t lvl);
```

Behavioral Verilog-A code can emit log messages with functions like `$strobe`. Handling of log messages is highly simulator specific, yet not performance critical. Therefore, OSDI delegates this functionality to a callback.

4.1.1 osdi_log

```
extern void (*osdi_log)(void *handle, char* msg, uint32_t lvl)
```

Formatting is closely intertwined with Verilog-A and needs to be handled by the compiler for proper standard compliance. The `osdi_log` callback receives the formatted message as an argument `msg`.

Furthermore, the `lvl` flag indicates what kind of message is emitted. This allows the simulator to format the message appropriately and emit when desired. The possible values of this argument are described in section 6.7.

Note: Ownership of the `msg` allocation is transferred to the simulator. It must free the `msg` to avoid memory leaks. However if `LOG_FMT_ERR` a string constant is passed to `osdi_log` that must not be freed.

A basic implementation of the function that writes each message immediately to stdout is shown below:

```
extern void osdi_log(void *handle, char* msg, uint32_t lvl){
    switch(lvl & LOG_LVL_MASK){
        case LOG_LVL_DEBUG: printf("VA debug: "); break;
        case LOG_LVL_DISPLAY: printf("VA: "); break;
        case LOG_LVL_INFO: printf("VA info: "); break;
        case LOG_LVL_WARN: printf("VA warn: "); break;
        case LOG_LVL_ERR: printf("VA error: "); break;
        case LOG_LVL_FATAL: printf("VA fatal: "); break;
        default: printf("VA unknown message: "); break;
    }

    if (lvl & LOG_FMT_ERR){
        printf("FAILED TO FORMAT \"%s\"", msg);
    }else{
        printf("%s", msg);
    }
}
```

4.2 Built-In `$limit` Functions

```
extern uint32_t OSDI_LIM_TABLE_LEN;
extern OsdiLimFunction OSDI_LIM_TABLE[];
```

Verilog-A allows using `$limit` to call simulator built-in functions such as `pnjlim` for improving the convergence properties of non-linear equations. These built-in functions are specified as a string and are therefore completely arbitrary. OSDI implements a dynamic look up table to support these functions.

These callbacks always have the following signature:

```
double lim_callback(bool init, bool *limit, double old_val, double new_val, double arg1, double arg2);
```

The value returned by this function is used as the result for `$limit`. `init` is true if `INIT_LIM` is set, the function should return an initial value in that case. `limit` should be set to `TRUE` by the function if the function does not return `new_val`. `new_val` is the value of the limited branch in the current iteration. `old_val` is the value of the limited branch in the previous iteration. The remaining arguments are `N` arguments of type `double`. Here `N` is the number of arguments specified in `num_args`

4.2.1 OSDI_LIM_TABLE_LEN

```
extern uint32_t OSDI_LIM_TABLE_LEN
```

This symbol is exported by OSDI shared libraries and specifies how many entries are present in `OSDI_LIM_TABLE`. If this symbol is missing from the loaded library, its value is assumed to be `0`.

4.2.2 OSDI_LIM_TABLE

```
extern OsdiLimFunction OSDI_LIM_TABLE[]
```

A list of length `OSDI_LIM_TABLE` that contains information about every built-in `$limit` function used in the OSDI library. When the OSDI library is loaded, the simulator should set the `func_ptr` field for the table entries. If a function is used that is unknown to the simulator, a warning should be printed to the user and the pointer should be set to `NULL`. In this is the case the voltage/potential supplied to the function is returned without change.

5 Data Structures

5.1 OsdiDescriptor

```
struct OsdiDescriptor {
  char *name;

  uint32_t num_nodes;
  uint32_t num_terminals;
  OsdiNode *nodes;

  uint32_t num_jacobian_entries;
  OsdiJacobianEntry *jacobian_entries;

  uint32_t num_collapsible;
  OsdiNodePair *collapsible;
  uint32_t collapsed_offset;

  OsdiNoiseSource *noise_sources;
  uint32_t num_noise_src;

  uint32_t num_params;
  uint32_t num_instance_params;
  uint32_t num_opvars;
  OsdiParamOpvar *param_opvar;

  uint32_t node_mapping_offset;
  uint32_t jacobian_ptr_resist_offset;

  uint32_t num_states;
  uint32_t state_idx_off;

  uint32_t bound_step_offset;

  uint32_t instance_size;
  uint32_t model_size;

  void *(*access)(void *inst, void *model, uint32_t id, uint32_t flags);
```

```
  void (*setup_model)(void *handle, void *model, OsdiSimParas *sim_params,
                                    OsdiInitInfo *res);
  void (*setup_instance)(void *handle, void *inst, void *model,
                                    double temperature, uint32_t num_terminals,
                                    OsdiSimParas *sim_params, OsdiInitInfo *res);

  uint32_t (*eval)(void *handle, void *inst, void *model, OsdiSimInfo *info);
  void (*load_noise)(void *inst, void *model, double freq, double *noise_dens,
                 double *ln_noise_dens);
  void (*load_residual_resist)(void *inst, void* model, double *dst);
  void (*load_residual_react)(void *inst, void* model, double *dst);
  void (*load_limit_rhs_resist)(void *inst, void* model, double *dst);
  void (*load_limit_rhs_react)(void *inst, void* model, double *dst);
  void (*load_spice_rhs_dc)(void *inst, void* model, double *dst,
                 double* prev_solve);
  void (*load_spice_rhs_tran)(void *inst, void* model, double *dst,
                 double* prev_solve, double alpha);
  void (*load_jacobian_resist)(void *inst, void* model);
  void (*load_jacobian_react)(void *inst, void* model, double alpha);
  void (*load_jacobian_tran)(void *inst, void* model, double alpha);
}
```

This struct contains the entire OSDI API for one compiled Verilog-A module. Instances of this struct are provided in the OSDI_DESCRIPTORS list.

5.1.1 name

```
char *name
```

The name of the Verilog-A module.

5.1.2 num_nodes

```
uint32_t num_nodes
```

The **total** number of nodes used within the device descriptor. During simulation, the actual number of nodes may be lower due to node-collapsing.

5.1.3 num_terminals

```
uint32_t num_terminals
```

The number of nodes that are device terminals.

5.1.4 nodes

```
OsdiNode *nodes
```

A list of size `num_nodes` that contains metadata for each node used within the device descriptor. Each entry is an instance of `OsdiNode`. The node's index in this list is used to represent a node in all parts of the interface.

The first `num_terminals` entries are filled with the model's terminals in the same order as defined in Verilog-A source. A specific defined order does not need to be followed by the remaining entries.

5.1.5 num_jacobian_entries

```
uint32_t num_jacobian_entries
```

The number of **non-zero** jacobian entries.

5.1.6 jacobian_entries

```
OsdiJacobianEntry *jacobian_entries
```

A list with length `num_jacobian_entries` that contains the **non-zero** jacobian entries. Each entry contains an `OsdiJacobianEntry`. Other parts of the interface refer to jacobian entries via the indices in this list.

5.1.7 num_collapsible

```
uint32_t num_collapsible
```

The number of node pairs that can be collapsed, typically indicated by `V(x,y) <+ 0` in Verilog-A.

5.1.8 collapsible

```
OsdiNodePair *collapsible
```

A list with length `num_collapsible` that contains `OsdiNodePair` that are collapsible, typically indicated by `V(x,y) <+ 0` in Verilog-A. The `setup_instance` routine tells the simulator which of these pairs to collapse for a specific instance. Note: The `node_2` field can bet set to `UINT32_MAX` to indicate a collapse into the global reference node.

5.1.9 collapsed_offset

```
uint32_t collapsed_offset
```

Provides the offset of the `collapsed` lists within the instance data in bytes. The `collapsed` list has length `num_collapsible` and contains `bool` elements.

`collapsed[i]` is set true if the `collapsible` node pair at `collapsible[i]` is collapsed for the given instance. The entries of `collapsed` are written in the `setup_instance` routine.

Example You can access the `collapsed` list of the instance data `inst` as follows:

```
bool *collapsed = (bool *) ((char *)inst) + collapsed_offset;
```

5.1.10 num_noise_src

```
uint32_t num_noise_src
```

The number of uncorrelated noise sources used in the model (`white_noise`, `flicker_noise`, `table_noise` and `table_noise_log` in Verilog-A).

5.1.11 noise_sources

```
OsdiNoiseSource *noise_sources
```

A list of all noise sources used within the device model with length `num_noise_src`.

5.1.12 num_params

```
uint32_t num_params
```

The number of Verilog-A model parameters.

5.1.13 num_instance_params

```
uint32_t num_instance_params
```

The number of Verilog-A parameters marked with the attribute `type="instance"`.

5.1.14 num_opvars

```
uint32_t num_opvars
```

The number of operating point variables (marked with `description` and `units` attribute).

5.1.15 param_opvar

```
OsdiParamOpvar *param_opvar
```

A list of metadata for each Verilog-A parameter with length num_params + num_opvars. Each element is an instance of the OsdiParamOpvar struct. The first num_opvars elements correspond to the model's operating point variables. The following num_instance_params elements correspond to the model's instance parameters. The remaining elements correspond to the model's parameters.

5.1.16 node_mapping_offset

```
uint32_t node_mapping_offset
```

The offset of the node_mapping within the instance data in bytes. This field can be used to calculate a pointer to the node_mapping as follows:

```
uint32_t *node_mapping = (uint32_t *) ((char *)inst) + node_mapping_offset;
```

The node_mapping contains the global offset of each node within the solution/rhs vector. The values must be provided by the simulator before the eval, load_residual_react, load_residual_resist, load_spice_rhs_dc or load_spice_rhs_tran function are called. The offsets must be stored in the same order as the nodes list.

5.1.17 jacobian_ptr_resist_offset

```
uint32_t jacobian_ptr_resist_offset
```

The offset of the jacobian_ptr_resist array within the instance data in bytes. This field can be used to calculate a pointer to the jacobian_ptr_resist array as follows:

```
double **jacobian_ptr_resist = (double **) ((char *)inst) + jacobian_ptr_resist_offset;
```

The jacobian_ptr_resist array contains pointers to resistive jacobian entries. These pointers must be provided by the simulator before the load_jacobian_resist / load_jacobian_tran functions are called. They must be stored in the same order as the jacobian_entries field.

The load_jacobian_resist and load_jacobian_tran functions uses these pointers to store the calculated jacobian entries stored in the instance data.

5.1.18 num_states

```
uint32_t num_states
```

The number of states required for each instance. During initialization, the simulator must ensure that num_states doubles storage is available for each instance.

5.1.19 state_idx_off

```
uint32_t state_idx_off
```

The offset of the `state_idx` array within the instance data in bytes. This field can be used to calculate a pointer to the `state_idx` array as follows:

```
uint32_t *state_idx = (uint32_t *) ((char *)inst) + state_idx_off);
```

The `eval` function will read/write `num_states` values from/to `prev_state`/`next_state` at the indices stored in this array. For example to read the value of the first state the `eval` function will perform the following:

```
double state_1 = prev_state[state_idx[0]]
```

5.1.20 bound_step_offset

```
uint32_t bound_step_offset
```

The offset of the `bound_step` field within the instance data in bytes. At this field the `eval` function stores the minimum step size required for this instance. This step size is determined by the call to `$bound_step` with the smallest value.

For devices that never call `$bound_step` this value is set to `UINT32_MAX`. If a device calls `$bound_step` conditionally and the condition is not true for this instance then the value `+inf` is stored here.

5.1.21 instance_size

```
uint32_t instance_size
```

The size of the instance data in bytes.

5.1.22 model_size

```
uint32_t model_size
```

The size of the model data in bytes.

5.1.23 access

```
void *(*access)(void *inst, void *model, uint32_t id, uint32_t flags)
```

A function pointer to the implementation of the `access` function for this descriptor.

5.1.24 setup_model

```
void (*setup_model)(void *handle, void *model, OsdiSimParas *sim_params,
                                  OsdiInitInfo *res)
```

A function pointer to the implementation of the `setup_model` function for this descriptor.

5.1.25 setup_instance

```
void (*setup_instance)(void *handle, void *inst, void *model,
                                  double temperature, uint32_t num_terminals,
                                  OsdiSimParas *sim_params, OsdiInitInfo *res)
```

A function pointer to the implementation of the `setup_instance` function for this descriptor.

5.1.26 eval

```
uint32_t (*eval)(void *handle, void *inst, void *model, OsdiSimInfo *info)
```

A function pointer to the implementation of the `eval` function for this descriptor.

5.1.27 load_noise

```
void (*load_noise)(void *inst, void *model, double freq, double *noise_dens,
                double *ln_noise_dens)
```

A function pointer to the implementation of the `load_noise` function for this descriptor.

5.1.28 load_residual_resist

```
void (*load_residual_resist)(void *inst, void* model, double *dst)
```

A function pointer to the implementation of the `load_residual_resist` function for this descriptor.

5.1.29 load_residual_react

```
void (*load_residual_react)(void *inst, void* model, double *dst)
```

A function pointer to the implementation of the `load_residual_react` function for this descriptor.

5.1.30 load_spice_rhs_dc

```
void (*load_spice_rhs_dc)(void *inst, void* model, double *dst,
                double* prev_solve)
```

A function pointer to the implementation of the load_spice_rhs_dc function for this descriptor.

5.1.31 load_limit_rhs_resist

```
void (*load_limit_rhs_resist)(void *inst, void* model, double *dst)
```

A function pointer to the implementation of the load_limit_rhs_resist function for this descriptor.

5.1.32 load_limit_rhs_react

```
void (*load_limit_rhs_react)(void *inst, void* model, double *dst)
```

A function pointer to the implementation of the load_limit_rhs_react function for this descriptor.

5.1.33 load_spice_rhs_tran

```
void (*load_spice_rhs_tran)(void *inst, void* model, double *dst,
                double* prev_solve, double alpha)
```

A function pointer to the implementation of the load_spice_rhs_tran function for this descriptor.

5.1.34 load_jacobian_resist

```
void (*load_jacobian_resist)(void *inst, void* model)
```

A function pointer to the implementation of the load_jacobian_resist function for this descriptor.

5.1.35 load_jacobian_react

```
void (*load_jacobian_react)(void *inst, void* model, double alpha)
```

A function pointer to the implementation of the load_jacobian_react function for this descriptor.

5.1.36 load_jacobian_tran

```
void (*load_jacobian_tran)(void *inst, void* model, double alpha)
```

A function pointer to the implementation of the load_jacobian_tran function for this descriptor.

5.2 OsdiNoiseSource

```
struct OsdiNoiseSource {
  char *name;
  OsdiNodePair nodes;
}
```

This struct describes a noise source. A single instance corresponds to a single call to one of the following Verilog-A functions:

- white_noise
- flicker_noise
- table_noise
- table_noise_log

Correlated noise sources are not directly supported at the moment. Instead, a correlation network should be used. Therefore, every noise source can only be used in single Verilog-A contribute statement and is only connected to a single pair of nodes.

5.2.1 name

```
char *name
```

This field contains the name of the noise source if given as a last argument in Verilog-A. If no name was provided this field contains a NULL pointer.

5.2.2 nodes

```
OsdiNodePair nodes
```

The pair of nodes that the noise source is connected to. node_1 is the node with a positive contribution.

5.3 OsdiParamOpvar

```
struct OsdiParamOpvar {
  char **name;
  uint32_t num_alias;
  char *description;
  char *units;
  uint32_t flags;
  uint32_t len;
}
```

Metadata that describes model parameters, instance parameters and operating point variables.

5.3.1 name

```
char **name
```

A list of identifiers with `1 + num_alias` entries. Its first entry corresponds to the canonical identifier while the remaining `num_alias` entries are aliases.

5.3.2 num_alias

```
uint32_t num_alias
```

The number of additional identifiers (aliases) that can be used for this parameter besides the canonical identifier.

5.3.3 description

```
char *description
```

Description of the parameter / op variable as given by the (standardized) Verilog-A `description` attribute.

5.3.4 units

```
char *units
```

Units of the parameter / op variable as given by the (standardized) Verilog-A `units` attribute.

5.3.5 flags

```
uint32_t flags
```

This field stores binary encoded information such as the data type. The bit patterns are documented in section 6.2

5.3.6 len

```
uint32_t len
```

This field is used to encode array types. The value of this field is 0 for scalar types. For arrays, it is set to the **total** length of the array: For arrays with multiple dimensions their lengths are multiplied.

Example A Verilog-A array `[10][5][2]` has a total length of 100.

5.4 OsdiNode

```
struct OsdiNode {
  char *name;
  char *units;
  char *residual_units;
  uint32_t resist_residual_off;
  uint32_t react_residual_off;
  uint32_t resist_limit_rhs_off;
  uint32_t react_limit_rhs_off;
  bool is_flow;
}
```

This struct contains metadata describing a node used within OSDI.

5.4.1 name

```
char *name
```

The name of the node. For net potentials this corresponds to the name of the Verilog-A net.

5.4.2 units

```
char *units
```

The units of the unknown quantity this node represents. The contents of this field is derived from the Verilog-A discipline/nature system and depend on what kind of unknown the node represents. For net potentials, this string corresponds to the contents of the `units` attribute of the `potential_nature`.

5.4.3 residual_units

```
char *residual_units
```

The units of the residual associated with the unknown this node represents. The contents of this field is derived from the Verilog-A discipline/nature system and depend on what kind of unknown the node represents. For net potentials this string corresponds to the contents of the `units` attribute of the `flow_nature`.

5.4.4 resist_residual_off

```
uint32_t resist_residual_off
```

The offset (in bytes) of this nodes' resistive residual within instance data. If this node has no resistive residual this value is set to `UINT32_MAX`. Simulators usually do not need to access this data directly. Instead, functions like `load_residual_resist`, `load_spice_rhs_dc` and `load_spice_rhs_tran` should be used.

5.4.5 react_residual_off

```
uint32_t react_residual_off
```

The offset (in bytes) of this nodes' resistive residual within instance data. If this node has no resistive residual this value is set to `UINT32_MAX`. Simulators (like SPICE) that require that every model implementation calculates numeric derivatives manually can read the charges using this offset. Instead, `load_residual_react` should be used.

5.4.6 resist_limit_rhs_off

```
uint32_t resist_limit_rhs_off
```

The offset (in bytes) of this nodes' resistive `$limit` residual within instance data. If this node has no resistive `$limit` residual this value is set to `UINT32_MAX`. Simulators usually do not need to access this data directly. Instead, functions like `load_limit_rhs_resist`, `load_spice_rhs_dc` and `load_spice_rhs_tran` should be used.

5.4.7 react_limit_rhs_off

```
uint32_t react_limit_rhs_off
```

The offset (in bytes) of this nodes' reactive `$limit` residual within instance data. If this node has no reactive `$limit` residual this value is set to `UINT32_MAX`. Simulators usually do not need to access this data directly. Instead, functions like `load_limit_rhs_react`, `load_spice_rhs_dc` and `load_spice_rhs_tran` should be used.

5.4.8 is_flow

```
bool is_flow
```

In OSDI nodes refer to any simulator unknown. Usually equations are formulated in terms of current-controlled current sources. In those cases all unknowns are node potential (hence the name). However, if other sources are used as unknowns for currents (flow) are required. In those cases this field is set to `true`.

5.5 OsdiJacobianEntry

```
struct OsdiJacobianEntry {
  OsdiNodePair nodes;
  uint32_t react_ptr_off;
  uint32_t flags;
}
```

This struct contains metadata describing a **non-zero** jacobian entry used within OSDI.

5.5.1 nodes

```
OsdiNodePair nodes
```

The nodes of this matrix entry. The first node corresponds to the row, the second node corresponds to the column in the jacobian.

5.5.2 react_ptr_off

```
uint32_t react_ptr_off
```

The offset of the pointer to the reactive value of this jacobian entry within the instance data in bytes. This field is set to `UINT32_MAX` if this jacobian entry has no reactive component. Otherwise, the field can be used to calculate a pointer to as follows:

```
double **jacobian_ptr_react = (double **) (((char *)inst) + react_ptr_off);
```

The pointer stored here must be provided by the simulator before the `load_jacobian_react` / `load_jacobian_tran` functions are called. The `load_jacobian_react` function uses these pointers to store the calculated reactive jacobian entries stored in the instance data.

5.5.3 flags

```
uint32_t flags
```

Various properties about the jacobian entry encoded as bit_flags. These flags are documented in section 6.4.

5.6 OsdiNodePair

```
struct OsdiNodePair {
  uint32_t node_1;
  uint32_t node_2;
}
```

This struct contains a pair of nodes. The nodes are referenced by their index in the `nodes` list.

5.7 OsdiInitInfo

```
struct OsdiInitInfo {
  uint32_t flags;
  uint32_t num_errors;
  OsdiInitError *errors;
}
```

This struct is returned by the `setup_model` and `setup_instance` functions. It provides feedback about the setup process to the simulator.

5.7.1 flags

```
uint32_t flags
```

Status flags that can be set by executed Verilog-A code to control the flow of the simulation. These flags are documented in section 6.6.

5.7.2 num_errors

```
uint32_t num_errors
```

The number of errors that occurred during setup.

5.7.3 errors

```
OsdiInitError *errors
```

A list of errors that occurred during setup. This last has a length of `num_errors`. The memory that is pointed to by this field is allocated by the setup routine with `calloc` (if `num_errors != 0`). The caller **must free** this memory to avoid memory leaks.

5.8 OsdiInitError

```
struct OsdiInitError {
  uint32_t code;
  OsdiInitErrorPayload payload;
}
```

If errors (usually caused by invalid user input) occur during the `setup_model` or `setup_instance` function, a list of `OsdiInitError` instances is allocated and returned in the `OsdiInitInfo` struct.

5.8.1 code

```
uint32_t code
```

This field contains an integer that represents the type of error. The error codes are documented in~section 6.8.

5.8.2 payload

```
OsdiInitErrorPayload payload
```

Additional information associated with the occurred error.

5.9 OsdiInitErrorPayload

```
union OsdiInitErrorPayload {
  uint32_t parameter_id;
}
```

This union contains additional information about an `OsdiInitError`. Which variant is contained within this union depends on the error code in the `code` field.

5.9.1 parameter_id

```
uint32_t parameter_id
```

This variant is used for errors associated with a parameter. The parameter is represented by its index with the `param_opvar` list.

5.10 OsdiSimInfo

```
struct OsdiSimInfo {
    OsdiSimParas paras;
    double abstime;
    double *prev_solve;
    double *prev_state;
    double *next_state;
    uint32_t flags;
}
```

This struct contains various information about the simulation required by the `eval` function.

5.10.1 paras

```
OsdiSimParas paras
```

This field requires a pointer to an instance of `OsdiSimParas` struct. This struct contains the simulation parameters returned by `$simparam` and `$simparam$str`.

5.10.2 abstime

```
double abstime
```

The value returned by the `$abstime` function. It should contain the current time (in seconds) during large signal simulations. During DC and small signal simulations this field should be set to `0`.

5.10.3 prev_solve

```
double *prev_solve
```

This field must point to the solution of the previous iteration (or initial values). The value of the unknown of each node will be read from this pointer. To that end the mapping stored at `node_mapping_offset` within the instance data is used.

5.10.4 prev_state

```
double *prev_state
```

This field must point to data that can be used for internal state of the model. The main use case is the implementation of `$limit` at the moment. Internal state can not be stored within the instance data, because large signal analysis might discard a timestep. In that case the internal state must be reset to the previous time step. The various internal state will be read from this pointer between `states[state_off]` and `states[state_off + num_states ]`. Here `state_off` is an argument to the `eval` function.

5.10.5 next_state

```
double *next_state
```

This pointer is used the same as the `prev_state` field except that internal state is written instead of read. For simulators where the internal state of the current and the next iteration are stored at the same place `next_state=prev_state` can be used.

5.10.6 flags

```
uint32_t flags
```

This argument indicates which values must be calculated by the `eval` function. It contains bitflags that each enable some calculations listed in the documentation for the `eval` function. These flags are documented in section 6.5.

5.11 OsdiSimParas

```
struct OsdiSimParas {
  char **names;
  double *vals;
  char **names_str;
  char **vals_str;
}
```

This struct contains lists of known simulation parameters that can be returned by the `$simparam` and `$simparam$str` Verilog-A function. What values are supported is up to the simulator. Primarily simulators should focus on supporting the parameters listed in Table 9-27 of the Verilog-AMS language reference manual (2.4.0). The `gmin` parameter in particular are commonly used by compact models. Additionally, the CMC recommends the parameter `minr` as a minimal resistance between two nodes, for smaller values the nodes are collapsed.

5.11.1 names

```
char **names
```

A list of names of all real-valued simulation parameters. The last entry in this list must contain a `NULL` pointer to indicate the end of the list.

5.11.2 vals

```
double *vals
```

A list of the values that correspond to the simulation parameters in `names` at the same index.

5.11.3 names_str

```
char **names_str
```

A list of names of all string-valued simulation parameters. The last entry in this list must contain a `NULL` pointer to indicate the end of the list.

5.11.4 vals_str

```
char **vals_str
```

A list of the values that correspond to the simulation parameters in `names_str` at the same index.

5.12 OsdiLimFunction

```
struct OsdiLimFunction {
  char *name;
  uint32_t num_args;
  void *func_ptr;
}
```

This struct contains information about a built-in `$limit` function. The simulator shall use that information to find the implementation of this function and write a pointer to this function into the `func_ptr` field.

5.12.1 name

```
char *name
```

The name of the function specified in Verilog-A.

5.12.2 num_args

```
uint32_t num_args
```

The number of additional `double` arguments received by this function. Because the compiler has no way to know the correct number of arguments, it is imperative that the simulator checks that the number of arguments is correct. In fact OSDI even allows a function with the same name used with multiple different numbers of arguments. This allows simulators to support optional arguments for `$limit` functions.

5.12.3 func_ptr

```
void *func_ptr
```

This field contains a pointer to the implementation of the function. The simulator must store this pointer here when it loads the library. If the simulator does not have an implementation of the function, this field should be set to `NULL`

6 Constants

6.1 Version number

```
#define OSDI_VERSION_MAJOR_CURR 0
#define OSDI_VERSION_MINOR_CURR 3
```

The current OSDI version number.

Simulators that implement the current OSDI version 0.3 must be able to load all shared libraries with `OSDI_VERSION_MAJOR = 0` and `OSDI_VERSION_MINOR = 3`.

6.1.1 OSDI_VERSION_MAJOR_CURR

```
#define OSDI_VERSION_MAJOR_CURR 0
```

Current major version of OSDI.

6.1.2 OSDI_VERSION_MINOR_CURR

```
#define OSDI_VERSION_MINOR_CURR 3
```

Current minor version of OSDI.

6.2 `OsdiParamOpvar` flags

```
#define PARA_TY_MASK 3
#define PARA_TY_REAL 0
#define PARA_TY_INT 1
#define PARA_TY_STR 2
#define PARA_KIND_MASK  (3 << 30)
#define PARA_KIND_MODEL (0 << 30)
#define PARA_KIND_INST  (1 << 30)
#define PARA_KIND_OPVAR (2 << 30)
```

Various properties about `OsdiParamOpvar` are encoded in its `flags` field as bit patterns. These bit patterns are documented herein

6.2.1 PARA_TY_MASK

```
#define PARA_TY_MASK 3
```

A bitmask that allows obtaining basic type of this parameter / op variable. `flags & OSDI_TY_MASK` will yield a number that that corresponds to one of the basic types (`PARA_TY_*`).

A basic type can never be an array but rather indicates the type of the scalar elements. Whether a type is an array is encoded with the `len` field instead.

6.2.2 PARA_TY_REAL

```
#define PARA_TY_REAL 0
```

A basic type (see `PARA_TY_MASK`) that corresponds to the `real` datatype in Verilog-A. Parameters and op variables with this type store their data as `double`.

6.2.3 PARA_TY_INT

```
#define PARA_TY_INT 1
```

A basic type (see `PARA_TY_MASK`) that corresponds to the `integer` datatype in Verilog-A. Parameters and op variables with this type store their data as `int32_t`.

6.2.4 PARA_TY_STR

```
#define PARA_TY_STR 2
```

A basic type (see `PARA_TY_MASK`) that corresponds to the `string` datatype in Verilog-A. Parameters and op variables with this type store their data as `char *` (null terminated UTF-8 strings).

6.2.5 PARA_KIND_MASK

```
#define PARA_KIND_MASK  (3 << 30)
```

Bitmask that allows obtaining the kind of object is represented by an `OsdiParamOpvar` instance. `flags & PARA_KIND_MASK` will yield a number that that corresponds to one of constants (`PARA_KIND*`) defined below.

6.2.6 PARA_KIND_MODEL

```
#define PARA_KIND_MODEL (0 << 30)
```

Represents an `OsdiParamOpvar` kind (see `PARA_KIND_MASK`). This bit pattern is used for model parameters. All Verilog-A parameters are model parameters unless specified otherwise.

6.2.7 PARA_KIND_INST

```
#define PARA_KIND_INST  (1 << 30)
```

Represents an `OsdiParamOpvar` kind (see `PARA_KIND_MASK`). This bit pattern is used for instance parameters. Verilog-A parameters must be marked with the `type="instance"` attribute to be treated as instance parameters.

6.2.8 PARA_KIND_OPVAR

```
#define PARA_KIND_OPVAR (2 << 30)
```

Represents an `OsdiParamOpvar` kind (see `PARA_KIND_MASK`). This bit pattern is used for operating point variables. Operating point variables are Verilog-A variables marked with the `description` and / or the `units` attribute. Their values are calculated by the `eval` function when `CALC_OP` is set in `flags` argument of the `eval` function.

6.3 access function flags

```
#define ACCESS_FLAG_READ 0
#define ACCESS_FLAG_SET 1
#define ACCESS_FLAG_INSTANCE 4
```

The `access` function must know how the pointer it returns is used. To that end the simulator must provide a set of bit flags that indicate the usage.

6.3.1 ACCESS_FLAG_READ

```
#define ACCESS_FLAG_READ 0
```

This flag indicates that the returned pointer will be read. Note: Reading the pointer is always allowed, and therefore this flag is simply `0`. Its only provided to improve the readability of simulator implementations.

6.3.2 ACCESS_FLAG_SET

```
#define ACCESS_FLAG_SET 1
```

This flag indicates that a value will be written to the returned pointer.

6.3.3 ACCESS_FLAG_INSTANCE

```
#define ACCESS_FLAG_INSTANCE 4
```

The access function always returns a pointer to model data for parameters. This allows the user to set a model default value for an instance parameter. When the ACCESS_FLAG_INSTANCE is set, a pointer to the instance value of the parameter is returned instead.

6.4 OsdiJacobianEntry flags

```
#define JACOBIAN_ENTRY_RESIST_CONST 1
#define JACOBIAN_ENTRY_REACT_CONST 2
#define JACOBIAN_ENTRY_RESIST 4
#define JACOBIAN_ENTRY_REACT 8
```

The jacobian entries are of central importance to the simulator. Therefore, various additional properties about each entry are exposed by flags

6.4.1 JACOBIAN_ENTRY_RESIST_CONST

```
#define JACOBIAN_ENTRY_RESIST_CONST 1
```

This flag Indicates that the resistive component of this jacobian entry is constant. In this context a jacobian entry is regard as constant if it's always independent of the operating point.

6.4.2 JACOBIAN_ENTRY_REACT_CONST

```
#define JACOBIAN_ENTRY_REACT_CONST 2
```

This flag Indicates that the reactive component of this jacobian entry is constant. In this context a jacobian entry is regard as constant if it's always independent of the operating point.

6.4.3 JACOBIAN_ENTRY_RESIST

```
#define JACOBIAN_ENTRY_RESIST 4
```

This flag Indicates that this jacobian entry has a resistive component.

6.4.4 JACOBIAN_ENTRY_REACT

```
#define JACOBIAN_ENTRY_REACT 8
```

This flag Indicates that this jacobian entry has a reactive component.

6.5 `eval` argument flags

```
#define CALC_RESIST_RESIDUAL 1
#define CALC_REACT_RESIDUAL 2
#define CALC_RESIST_JACOBIAN 4
#define CALC_REACT_JACOBIAN 8
#define CALC_NOISE 16
#define CALC_OP 32
#define CALC_RESIST_LIM_RHS 64
#define CALC_REACT_LIM_RHS 128
#define ENABLE_LIM 256
#define INIT_LIM 512
#define ANALYSIS_NOISE 1024
#define ANALYSIS_DC 2048
#define ANALYSIS_AC 4096
#define ANALYSIS_TRAN 8192
#define ANALYSIS_IC 16384
#define ANALYSIS_STATIC 32768
#define ANALYSIS_NODESET 65536
```

The `eval` function allows the simulator to specify which values can be calculated. This can improve performance by avoiding unneeded calculations. To that end the simulator can set bit flags within the `flags` argument of the `eval` function.

6.5.1 CALC_RESIST_RESIDUAL

```
#define CALC_RESIST_RESIDUAL 1
```

This flag instructs the `eval` function to calculate the resistive residual. The `eval` function must be called with this flag before the `load_residual_resist`, `load_spice_rhs_tran` and `load_spice_rhs_dc` functions are called.

6.5.2 CALC_REACT_RESIDUAL

```
#define CALC_REACT_RESIDUAL 2
```

This flag instructs the `eval` function to calculate the reactive residual. The `eval` function must be called with this flag before the `load_residual_react` and `load_spice_rhs_tran` functions are called.

6.5.3 CALC_RESIST_JACOBIAN

```
#define CALC_RESIST_JACOBIAN 4
```

This flag instructs the `eval` function to calculate the resistive jacobian entries. The `eval` function must be called with this flag before the `load_jacobian_resist`, `load_spice_rhs_tran` and `load_spice_rhs_dc` functions are called.

6.5.4 CALC_REACT_JACOBIAN

```
#define CALC_REACT_JACOBIAN 8
```

This flag instructs the `eval` function to calculate the reactive jacobian entries. The `eval` function must be called with this flag before the `load_jacobian_react`, `load_spice_rhs_tran` and `load_spice_rhs_dc` functions are called.

6.5.5 CALC_NOISE

```
#define CALC_NOISE 16
```

This flag instructs the `eval` function to calculate the arguments of noise sources. The `eval` function must be called with this flag before the `load_noise` function is called.

6.5.6 CALC_OP

```
#define CALC_OP 32
```

This flag instructs the `eval` function to calculate operating point variables.

6.5.7 CALC_RESIST_LIM_RHS

```
#define CALC_RESIST_LIM_RHS 64
```

This flag instructs the `eval` function to calculate the resistive components of the rhs that accounts for `$limit`. The `eval` function must be called with this flag before the `load_limit_rhs_resist` and `load_spice_rhs_dc` functions are called.

6.5.8 CALC_REACT_LIM_RHS

```
#define CALC_REACT_LIM_RHS 128
```

This flag instructs the `eval` function to calculate the reactive components of the rhs that accounts for `$limit`. The `eval` function must be called with this flag before the `load_limit_rhs_react`, `load_spice_rhs_tran` and `load_spice_rhs_dc` functions are called.

6.5.9 ENABLE_LIM

```
#define ENABLE_LIM 256
```

This flag instructs the `eval` function to use `$limit` functions. If this flag is not set calls to `$limit` simply return the value of the branch without any limiting.

6.5.10 INIT_LIM

```
#define INIT_LIM 512
```

This flag instructs the `eval` function to use the initial value for built-in limit functions by passing true to the `init` argument of the callback.

6.5.11 ANALYSIS_DC

```
#define ANALYSIS_DC 2048
```

If this flag is set, then `analysis("dc")` returns `1`. See the section 4.6.1 of Verilog-AMS language reference manual (2.4.0) for details.

6.5.12 ANALYSIS_AC

```
#define ANALYSIS_AC 4096
```

If this flag is set, then `analysis("ac")` returns `1`. See the section 4.6.1 of Verilog-AMS language reference manual (2.4.0) for details.

6.5.13 ANALYSIS_TRAN

```
#define ANALYSIS_TRAN 8192
```

If this flag is set, then `analysis("tran")` returns `1`. See the section 4.6.1 of Verilog-AMS language reference manual (2.4.0) for details.

6.5.14 ANALYSIS_IC

```
#define ANALYSIS_IC 16384
```

If this flag is set, then `analysis("ic")` returns `1`. See the section 4.6.1 of Verilog-AMS language reference manual (2.4.0) for details. Additionally, while this flag is active the initial condition for `idt` and `idtmod` time integrals is used. Note that initial condition is always used if `CALC_REACT_JACOBIAN` is disabled.

6.5.15 ANALYSIS_STATIC

```
#define ANALYSIS_STATIC 32768
```

If this flag is set, then `analysis("static")` returns `1`. See the section 4.6.1 of Verilog-AMS language reference manual (2.4.0) for details.

6.5.16 ANALYSIS_NODESET

```
#define ANALYSIS_NODESET 65536
```

If this flag is set, then `analysis("nodeset")` returns `1`. See the section 4.6.1 of Verilog-AMS language reference manual (2.4.0) for details.

6.5.17 ANALYSIS_NOISE

```
#define ANALYSIS_NOISE 1024
```

If this flag is set, then `analysis("noise")` returns `1`. See the section 4.6.1 of Verilog-AMS language reference manual (2.4.0) for details.

6.6 `eval` return flags

```
#define EVAL_RET_FLAG_LIM 1
#define EVAL_RET_FLAG_FATAL 2
#define EVAL_RET_FLAG_FINISH 4
#define EVAL_RET_FLAG_STOP 8
```

Verilog-A allows behavioral code to control the simulation flow. This functionality is accommodated by setting bit-flags for the return value of the `eval` function.

6.6.1 EVAL_RET_FLAG_LIM

```
#define EVAL_RET_FLAG_LIM 1
```

This flag indicates that a `$limit` function (like `pnjlim`) has reduced the change of a potential. While this flag is set the simulator must not allow the current simulation to converge.

6.6.2 EVAL_RET_FLAG_FATAL

```
#define EVAL_RET_FLAG_FATAL 2
```

If this flag is set at a fatal error occurred and the simulator must **abort** the current simulation with an error.

6.6.3 EVAL_RET_FLAG_FINISH

```
#define EVAL_RET_FLAG_FINISH 4
```

The `EVAL_RET_FLAG_FINISH` flag indicates that `$finish` was called. If the current iteration has converged the simulator must **exit gracefully**. Otherwise, this flag should be ignored.

6.6.4 EVAL_RET_FLAG_STOP

```
#define EVAL_RET_FLAG_STOP 8
```

This flag indicates that `$stop` was called. If the current iteration has converged the simulator must **pause** the current simulation. Otherwise, this flag should be ignored.

6.7 Log Level

```
#define LOG_LVL_MASK 8
#define LOG_LVL_DEBUG 0
#define LOG_LVL_DISPLAY 1
#define LOG_LVL_INFO 2
#define LOG_LVL_WARN 3
#define LOG_LVL_ERR 4
#define LOG_LVL_FATAL 5
#define LOG_FMT_ERR 16
```

Log message handling is highly simulator specific yet not performance critical. Therefore, OSDI delegates this functionality to the `osdi_log` callback. This callback receives a bitflags that indicates how to handle this message.

6.7.1 LOG_LVL_MASK

```
#define LOG_LVL_MASK 8
```

The bits masked by this constant contains the log `lvl` flag. The logo `lvl` indicate what kind of function was used to generate this log message. This level is only intended to determine how, when and if to display the message. The simulation flow should not be changed (e.g. terminated) based upon this flow. The flags documented in section 6.6 are used for that purpose.

6.7.2 LOG_LVL_DEBUG

```
#define LOG_LVL_DEBUG 0
```

Indicates logging with the `$debug` functions. These messages should be printed immediately.

6.7.3 LOG_LVL_DISPLAY

```
#define LOG_LVL_DISPLAY 1
```

Indicates logging with the `$strobe` / `$display` / `$write` functions. These messages should be printed after convergence.

6.7.4 LOG_LVL_INFO

```
#define LOG_LVL_INFO 2
```

Indicates logging with the `$info` ‘function. These messages should be printed after convergence.

6.7.5 LOG_LVL_WARN

```
#define LOG_LVL_WARN 3
```

Indicates logging with the `$warning` function. These messages should be printed after convergence.

6.7.6 LOG_LVL_ERR

```
#define LOG_LVL_ERR 4
```

Indicates logging with the `$error` function. These messages should be printed after convergence.

6.7.7 LOG_LVL_FATAL

```
#define LOG_LVL_FATAL 5
```

Indicates logging with the `$fatal` function. These messages should be printed immediately as the simulation is terminated immediately afterwards.

6.7.8 LOG_FMT_ERR

```
#define LOG_FMT_ERR 16
```

This flag is set when formatting the raw format string failed. When this flag is set the unprocessed format literal is passed to the `msg` argument. This literal is a constant that must not be freeded.

6.8 `OsdiInitError` error-codes

```
#define INIT_ERR_OUT_OF_BOUNDS 1
```

The `setup_instance` and `setup_model` routines process use inputs (like parameters). These inputs may be invalid, and therefore these routines can produce `OsdiInitError`. The `code` field contains an error code that indicate the error that occurred. These errors are documented below.

6.8.1 INIT_ERR_OUT_OF_BOUNDS

```
#define INIT_ERR_OUT_OF_BOUNDS 1
```

This error occurs if one of the model or instance parameters violates the boundaries set in the Verilog-A code. In this case the `payload` field contains the `parameter_id` of the parameter with invalid values.

7 Verilog-A Standard Compliance

OSDI is predominantly aimed at compact modelling. In the compact modelling community some parts of the Verilog-A language are de-facto not used. For allowing fast and consistent results, some limitations of the Verilog-A language subset are necessary as defined below.

7.1 Hidden State

OSDI compliant compilers must assume that a compiled model does not have hidden states. This often allows more code to be moved into the `model_setup` and `instance_setup` functions, significantly improving performance.

If a variable has the `hidden_state` attribute in Verilog-A, the compiler can not make that assumption. The same attribute can be placed on a Verilog-A module to allow hidden states for all variables within the module.

7.2 `limexp`

The `limexp` function is commonly used in Verilog-A because exponential overflow is a major cause of convergence issues. The language standard defines that `limexp` should limit the change of its argument between iterations. This requires access to values from previous iterations and also a limiting algorithm that is applicable in the general case. As a result all implementations known to the author have opted to use a simple linearized exponential instead, for example:

```
if (x < EXP_LIM){
    return exp(x)
}else{
    return exp(EXP_LIM) *(x + 1 - EXP_LIM)
}
```

OSDI uses this linearized function as well, with `EXP_LIM = 80`. This may change in future versions if tangible improvements can be demonstrated with a different algorithm.

8 Files

8.1 osdi.h

```
/*
 * This file is automatically generated by header.lua DO NOT EDIT MANUALLY
 * Copyright© 2022 SemiMod UG. All rights reserved.
 */

#pragma once
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>


#define OSDI_VERSION_MAJOR_CURR 0
#define OSDI_VERSION_MINOR_CURR 3

#define PARA_TY_MASK 3
#define PARA_TY_REAL 0
#define PARA_TY_INT 1
#define PARA_TY_STR 2
#define PARA_KIND_MASK  (3 << 30)
#define PARA_KIND_MODEL (0 << 30)
#define PARA_KIND_INST  (1 << 30)
#define PARA_KIND_OPVAR (2 << 30)

#define ACCESS_FLAG_READ 0
#define ACCESS_FLAG_SET 1
#define ACCESS_FLAG_INSTANCE 4

#define JACOBIAN_ENTRY_RESIST_CONST 1
#define JACOBIAN_ENTRY_REACT_CONST 2
#define JACOBIAN_ENTRY_RESIST 4
#define JACOBIAN_ENTRY_REACT 8

#define CALC_RESIST_RESIDUAL 1
#define CALC_REACT_RESIDUAL 2
```

```
#define CALC_RESIST_JACOBIAN 4
#define CALC_REACT_JACOBIAN 8
#define CALC_NOISE 16
#define CALC_OP 32
#define CALC_RESIST_LIM_RHS 64
#define CALC_REACT_LIM_RHS 128
#define ENABLE_LIM 256
#define INIT_LIM 512
#define ANALYSIS_NOISE 1024
#define ANALYSIS_DC 2048
#define ANALYSIS_AC 4096
#define ANALYSIS_TRAN 8192
#define ANALYSIS_IC 16384
#define ANALYSIS_STATIC 32768
#define ANALYSIS_NODESET 65536

#define EVAL_RET_FLAG_LIM 1
#define EVAL_RET_FLAG_FATAL 2
#define EVAL_RET_FLAG_FINISH 4
#define EVAL_RET_FLAG_STOP 8


#define LOG_LVL_MASK 8
#define LOG_LVL_DEBUG 0
#define LOG_LVL_DISPLAY 1
#define LOG_LVL_INFO 2
#define LOG_LVL_WARN 3
#define LOG_LVL_ERR 4
#define LOG_LVL_FATAL 5
#define LOG_FMT_ERR 16

#define INIT_ERR_OUT_OF_BOUNDS 1



typedef struct OsdiLimFunction {
  char *name;
  uint32_t num_args;
  void *func_ptr;
}OsdiLimFunction;

typedef struct OsdiSimParas {
  char **names;
  double *vals;
```

```
  char **names_str;
  char **vals_str;
}OsdiSimParas;

typedef struct OsdiSimInfo {
    OsdiSimParas paras;
    double abstime;
    double *prev_solve;
    double *prev_state;
    double *next_state;
    uint32_t flags;
}OsdiSimInfo;

typedef union OsdiInitErrorPayload {
  uint32_t parameter_id;
}OsdiInitErrorPayload;

typedef struct OsdiInitError {
  uint32_t code;
  OsdiInitErrorPayload payload;
}OsdiInitError;

typedef struct OsdiInitInfo {
  uint32_t flags;
  uint32_t num_errors;
  OsdiInitError *errors;
}OsdiInitInfo;

typedef struct OsdiNodePair {
  uint32_t node_1;
  uint32_t node_2;
}OsdiNodePair;

typedef struct OsdiJacobianEntry {
  OsdiNodePair nodes;
  uint32_t react_ptr_off;
  uint32_t flags;
}OsdiJacobianEntry;

typedef struct OsdiNode {
  char *name;
  char *units;
  char *residual_units;
  uint32_t resist_residual_off;
```

```
  uint32_t react_residual_off;
  uint32_t resist_limit_rhs_off;
  uint32_t react_limit_rhs_off;
  bool is_flow;
}OsdiNode;

typedef struct OsdiParamOpvar {
  char **name;
  uint32_t num_alias;
  char *description;
  char *units;
  uint32_t flags;
  uint32_t len;
}OsdiParamOpvar;

typedef struct OsdiNoiseSource {
  char *name;
  OsdiNodePair nodes;
}OsdiNoiseSource;

typedef struct OsdiDescriptor {
  char *name;

  uint32_t num_nodes;
  uint32_t num_terminals;
  OsdiNode *nodes;

  uint32_t num_jacobian_entries;
  OsdiJacobianEntry *jacobian_entries;

  uint32_t num_collapsible;
  OsdiNodePair *collapsible;
  uint32_t collapsed_offset;

  OsdiNoiseSource *noise_sources;
  uint32_t num_noise_src;

  uint32_t num_params;
  uint32_t num_instance_params;
  uint32_t num_opvars;
  OsdiParamOpvar *param_opvar;

  uint32_t node_mapping_offset;
  uint32_t jacobian_ptr_resist_offset;
```

```
  uint32_t num_states;
  uint32_t state_idx_off;

  uint32_t bound_step_offset;

  uint32_t instance_size;
  uint32_t model_size;

  void *(*access)(void *inst, void *model, uint32_t id, uint32_t flags);

  void (*setup_model)(void *handle, void *model, OsdiSimParas *sim_params,
                                    OsdiInitInfo *res);
  void (*setup_instance)(void *handle, void *inst, void *model,
                                    double temperature, uint32_t num_terminals,
                                    OsdiSimParas *sim_params, OsdiInitInfo *res);

  uint32_t (*eval)(void *handle, void *inst, void *model, OsdiSimInfo *info);
  void (*load_noise)(void *inst, void *model, double freq, double *noise_dens,
                  double *ln_noise_dens);
  void (*load_residual_resist)(void *inst, void* model, double *dst);
  void (*load_residual_react)(void *inst, void* model, double *dst);
  void (*load_limit_rhs_resist)(void *inst, void* model, double *dst);
  void (*load_limit_rhs_react)(void *inst, void* model, double *dst);
  void (*load_spice_rhs_dc)(void *inst, void* model, double *dst,
                  double* prev_solve);
  void (*load_spice_rhs_tran)(void *inst, void* model, double *dst,
                  double* prev_solve, double alpha);
  void (*load_jacobian_resist)(void *inst, void* model);
  void (*load_jacobian_react)(void *inst, void* model, double alpha);
  void (*load_jacobian_tran)(void *inst, void* model, double alpha);
}OsdiDescriptor;
```

8.2 diode.va

```
`include "constants.vams"
`include "disciplines.vams"

module diode_va(A,C,dT);
    // simple diode with self-heating network
    inout A, C, dT;
    electrical A,C,CI,dT;

    branch (A,CI) br_a_ci;
    branch (CI,C) br_ci_c;
    branch (dT )  br_sht; //self-heating

    (*desc= "Saturation current", units = "A"*) parameter real Is = 1e-14 from [0:inf];

    (*desc= "Ohmic res", units = "Ohm" *) parameter real Rs = 0.0 from [0:inf];

    (*desc= "Temperature coefficient of ohmic res"*) parameter real zetars = 0.0 from [-10:10];

    (*desc= "Emission coefficient"*) parameter real N = 1.0 from [0:inf];

    (*desc= "Junction capacitance", units = "F"*) parameter real Cj0 = 0.0 from [0:inf];

    (*desc= "Junction potential", units = "V"*) parameter real Vj = 1.0 from [0.2:2];

    (*desc= "Grading coefficient"*) parameter real M = 0.5 from [0:inf];

    (*desc= "Thermal resistance", units = "K/W"*) parameter real Rth = 0 from [0:inf];

    (*desc= "Temperature coefficient of thermal res"*) parameter real zetarth = 0.0 from [-10:10];

    (*desc= "Temperature coefficient of Is"*) parameter real zetais = 0.0 from [-10:10];

    (*desc= "Reference temperature", units = "K"*) parameter real Tnom = 300 from [0:inf];


    real Vd, Vr, Qd;

    (*retrieve*) real Id;
    real Cd;

    real VT,x,y,vf,Tdev,pterm,rs_t, rth_t;
```

```
    analog begin
        Tdev = $temperature+V(br_sht);
        VT = `P_K*Tdev/`P_Q;
        Vd = V(br_a_ci);
        Vr = V(br_ci_c);
        Id = Is * pow(Tdev/Tnom,zetais) * (limexp(Vd / (N * VT)) - 1);
        rs_t = Rs*pow(Tdev/Tnom,zetars);
        rth_t = Rth*pow(Tdev/Tnom,zetarth);
        // dissipated power
        pterm = Id*Vd + pow(Vr,2.0)/rs_t;
        //junction capacitance
        //smoothing of voltage over cap
        vf   = Vj*(1 - pow(3.0, -1/M));
        x    = (vf-Vd)/VT;
        y    = sqrt(x*x + 1.92);
        Vd   = vf-VT*(x + y)/(2);
        Qd   = Cj0*Vj * (1-pow(1-Vd/Vj, 1-M))/(1-M);
        Cd   = ddx(Qd,V(A));
        I(br_a_ci) <+ Id + ddt(Qd) + $simparam("gmin")*V(br_a_ci);
        I(br_sht)  <+ V(br_sht)/rth_t-pterm;
        I(br_ci_c) <+ Vr / rs_t;
    end
endmodule
```

8.3 diode.c

```
/*
 * Copyright© 2022 SemiMod UG. All rights reserved.
 *
 * This is an exemplary implementation of the OSDI interface for the Verilog-A
 * model specified in diode.va. In the future, the OpenVAF compiler shall
 * generate an comparable object file. Primary purpose of this is example to
 * have a concrete example for the OSDI interface, OpenVAF will generate a more
 * optimized implementation.
 *
 */

#include "osdi.h"
#include "string.h"
#include <math.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>

// public interface
extern uint32_t OSDI_VERSION_MAJOR;
extern uint32_t OSDI_VERSION_MINOR;
extern uint32_t OSDI_NUM_DESCRIPTORS;
extern OsdiDescriptor OSDI_DESCRIPTORS[1];
extern OsdiLimFunction OSDI_LIM_TABLE[1];
extern uint32_t OSDI_LIM_TABLE_LEN;

#define sqrt2 1.41421356237309504880168872420969

#define IGNORE(x) (void)x

// number of nodes and definitions of node ids for nicer syntax in this file
// note: order should be same as "nodes" list defined later
#define NUM_NODES 4
#define A 0
#define C 1
#define TNODE 2
#define CI 3

#define NUM_COLLAPSIBLE 2
```

```
// number of matrix entries and definitions for Jacobian entries for nicer
// syntax in this file
#define NUM_MATRIX 14
#define CI_CI 0
#define CI_C 1
#define C_CI 2
#define C_C 3
#define A_A 4
#define A_CI 5
#define CI_A 6
#define A_TNODE 7
#define C_TNODE 8
#define CI_TNODE 9
#define TNODE_TNODE 10
#define TNODE_A 11
#define TNODE_C 12
#define TNODE_CI 13

// The model structure for the diode
typedef struct DiodeModel {
  double Rs;
  bool Rs_given;
  double Is;
  bool Is_given;
  double zetars;
  bool zetars_given;
  double N;
  bool N_given;
  double Cj0;
  bool Cj0_given;
  double Vj;
  bool Vj_given;
  double M;
  bool M_given;
  double Rth;
  bool Rth_given;
  double zetarth;
  bool zetarth_given;
  double zetais;
  bool zetais_given;
  double Tnom;
  bool Tnom_given;
  double mfactor; // multiplication factor for parallel devices
  bool mfactor_given;
```

```
  // InitError errors[MAX_ERROR_NUM],
} DiodeModel;

// The instace structure for the diode
typedef struct DiodeInstace {
  double mfactor; // multiplication factor for parallel devices
  bool mfactor_given;
  double temperature;
  double residual_resist[NUM_NODES];
  double lim_rhs_resist_A;
  double lim_rhs_resist_CI;
  double lim_rhs_react_A;
  double lim_rhs_react_CI;
  double residual_react_A;
  double residual_react_CI;
  double jacobian_resist[NUM_MATRIX];
  double jacobian_react[NUM_MATRIX];
  bool collapsed[NUM_COLLAPSIBLE];
  double *jacobian_ptr_resist[NUM_MATRIX];
  double *jacobian_ptr_react[NUM_MATRIX];
  uint32_t node_off[NUM_NODES];
  uint32_t state_idx;
} DiodeInstace;

#define EXP_LIM 80.0

static double limexp(double x) {
  if (x < EXP_LIM) {
    return exp(x);
  } else {
    return exp(EXP_LIM) * (x + 1 - EXP_LIM);
  }
}

static double dlimexp(double x) {
  if (x < EXP_LIM) {
    return exp(x);
  } else {
    return exp(EXP_LIM);
  }
}

// implementation of the @access@access@ function as defined by the OSDI spec
static void *osdi_access(void *inst_, void *model_, uint32_t id,
```

```
                        uint32_t flags) {
DiodeModel *model = (DiodeModel *)model_;
DiodeInstace *inst = (DiodeInstace *)inst_;

bool *given;
void *value;

switch (id) // id of params defined in param_opvar array
{
case 0:
  if (flags & ACCESS_FLAG_INSTANCE) {
    value = (void *)&inst->mfactor;
    given = &inst->mfactor_given;
  } else {
    value = (void *)&model->mfactor;
    given = &model->mfactor_given;
  }
  break;
case 1:
  value = (void *)&model->Rs;
  given = &model->Rs_given;
  break;
case 2:
  value = (void *)&model->Is;
  given = &model->Is_given;
  break;
case 3:
  value = (void *)&model->zetars;
  given = &model->zetars_given;
  break;
case 4:
  value = (void *)&model->N;
  given = &model->N_given;
  break;
case 5:
  value = (void *)&model->Cj0;
  given = &model->Cj0_given;
  break;
case 6:
  value = (void *)&model->Vj;
  given = &model->Vj_given;
  break;
case 7:
  value = (void *)&model->M;
```

```
    given = &model->M_given;
    break;
  case 8:
    value = &model->Rth;
    given = &model->Rth_given;
    break;
  case 9:
    value = (void *)&model->zetarth;
    given = &model->zetarth_given;
    break;
  case 10:
    value = (void *)&model->zetais;
    given = &model->zetais_given;
    break;
  case 11:
    value = (void *)&model->Tnom;
    given = &model->Tnom_given;
    break;
  default:
    return NULL;
  }

  if (flags & ACCESS_FLAG_SET) {
    *given = true;
  }

  return value;
}

// implementation of the @setupmodel@setup_model@ function as defined in the OSDI spec
static void setup_model(void *handle, void *model_, OsdiSimParas *sim_params,
                        OsdiInitInfo *res) {
  DiodeModel *model = (DiodeModel *)model_;

  IGNORE(handle);
  IGNORE(sim_params);

  // set parameters and check bounds
  if (!model->mfactor_given) {
    model->mfactor = 1.0;
  }
  if (!model->Rs_given) {
    model->Rs = 1e-9;
  }
```

```
  if (!model->Is_given) {
    model->Is = 1e-14;
  }
  if (!model->zetars_given) {
    model->zetars = 0;
  }
  if (!model->N_given) {
    model->N = 1;
  }
  if (!model->Cj0_given) {
    model->Cj0 = 0;
  }
  if (!model->Vj_given) {
    model->Vj = 1.0;
  }
  if (!model->M_given) {
    model->M = 0.5;
  }
  if (!model->Rth_given) {
    model->Rth = 0;
  }
  if (!model->zetarth_given) {
    model->zetarth = 0;
  }
  if (!model->zetais_given) {
    model->zetais = 0;
  }
  if (!model->Tnom_given) {
    model->Tnom = 300;
  }

  *res = (OsdiInitInfo){.flags = 0, .num_errors = 0, .errors = NULL};
}

// implementation of the setup_instace function as defined in the OSDI spec
static void setup_instance(void *handle, void *inst_, void *model_,
                           double temperature, uint32_t num_terminals,
                           OsdiSimParas *sim_params, OsdiInitInfo *res) {

  IGNORE(handle);
  IGNORE(num_terminals);
  IGNORE(sim_params);

  DiodeInstace *inst = (DiodeInstace *)inst_;
```

```
  DiodeModel *model = (DiodeModel *)model_;

  // Here the logic for node collapsing ist implemented. The indices in this
  // list must adhere to the "collapsible" List of node pairs.
  if (model->Rs < 1e-9) { // Rs between Ci C
    inst->collapsed[0] = true;
  }
  if (model->Rth < 1e-9) { // Rs between Ci C
    inst->collapsed[1] = true;
  }

  if (!inst->mfactor_given) {
    if (model->mfactor_given) {
      inst->mfactor = model->mfactor;
    } else {
      inst->mfactor = 1;
    }
  }

  inst->temperature = temperature;
  *res = (OsdiInitInfo){.flags = 0, .num_errors = 0, .errors = NULL};
}

#define CONSTsqrt2 1.41421356237309504880168872420969
typedef double (*pnjlim_t)(bool, bool *, double, double, double, double);

// implementation of the @eval@eval@ function as defined in the OSDI spec
static uint32_t eval(void *handle, void *inst_, void *model_,
                     OsdiSimInfo *info) {
  IGNORE(handle);
  DiodeModel *model = (DiodeModel *)model_;
  DiodeInstace *inst = (DiodeInstace *)inst_;

  // get voltages
  double *prev_solve = info->prev_solve;
  double va = prev_solve[inst->node_off[A]];
  double vc = prev_solve[inst->node_off[C]];
  double vci = prev_solve[inst->node_off[CI]];
  double vdtj = prev_solve[inst->node_off[TNODE]];

  double vcic = vci - vc;
  double vaci = va - vci;

  double gmin = 1e-12;
```

```
for (int i = 0; info->paras.names[i] != NULL; i++) {
  if (strcmp(info->paras.names[i], "gmin") == 0) {
    gmin = info->paras.vals[i];
  }
}

uint32_t ret_flags = 0;
////////////////////////////////
// evaluate model equations
////////////////////////////////

// temperature update
double pk = 1.3806503e-23;
double pq = 1.602176462e-19;
double t_dev = inst->temperature + vdtj;
double tdev_tnom = t_dev / model->Tnom;
double rs_t = model->Rs * pow(tdev_tnom, model->zetars);
double rth_t = model->Rth * pow(tdev_tnom, model->zetarth);
double is_t = model->Is * pow(tdev_tnom, model->zetais);
double vt = t_dev * pk / pq;

double delvaci = 0.0;
if (info->flags & ENABLE_LIM && OSDI_LIM_TABLE[0].func_ptr) {
  double vte = inst->temperature * pk / pq;
  bool icheck = false;
  double vaci_old = info->prev_state[inst->state_idx];
  pnjlim_t pnjlim = OSDI_LIM_TABLE[0].func_ptr;
  double vaci_new = pnjlim(info->flags & INIT_LIM, &icheck, vaci, vaci_old,
                           vte, vte * log(vte / (sqrt2 * model->Is)));
  printf("%g %g\n", vaci, vaci_new);
  delvaci = vaci_new - vaci;
  vaci = vaci_new;
  info->prev_state[inst->state_idx] = vaci;
} else {
  printf("ok?");
}

// derivatives w.r.t. temperature
double rs_dt = model->zetars * model->Rs *
               pow(tdev_tnom, model->zetars - 1.0) / model->Tnom;
double rth_dt = model->zetarth * model->Rth *
                pow(tdev_tnom, model->zetarth - 1.0) / model->Tnom;
double is_dt = model->zetais * model->Is *
               pow(tdev_tnom, model->zetais - 1.0) / model->Tnom;
```

```
double vt_tj = pk / pq;

// evaluate model equations and calculate all derivatives
// diode current
double id = is_t * (limexp(vaci / (model->N * vt)) - 1.0);
double gd = is_t / vt * dlimexp(vaci / (model->N * vt));
double gdt = -is_t * dlimexp(vaci / (model->N * vt)) * vaci / model->N / vt /
                 vt * vt_tj +
             1.0 * exp((vaci / (model->N * vt)) - 1.0) * is_dt;

// resistor
double irs = 0;
double g = 0;
double grt = 0;
if (!inst->collapsed[0]) {
  irs = vcic / rs_t;
  g = 1.0 / rs_t;
  grt = -irs / rs_t * rs_dt;
}

// thermal resistance
double irth = 0;
double gt = 0;
if (!inst->collapsed[1]) {
  irth = vdtj / rth_t;
  gt = 1.0 / rth_t - irth / rth_t * rth_dt;
}

// charge
double vf = model->Vj * (1.0 - pow(3.04, -1.0 / model->M));
double x = (vf - vaci) / vt;
double x_vt = -x / vt;
double x_dtj = x_vt * vt_tj;
double x_vaci = -1.0 / vt;
double y = sqrt(x * x + 1.92);
double y_x = 0.5 / y * 2.0 * x;
double y_vaci = y_x * x_vaci;
double y_dtj = y_x * x_dtj;
double vd = vf - vt * (x + y) / (2.0);
double vd_x = -vt / 2.0;
double vd_y = -vt / 2.0;
double vd_vt = -(x + y) / (2.0);
double vd_dtj = vd_x * x_dtj + vd_y * y_dtj + vd_vt * vt_tj;
double vd_vaci = vd_x * x_vaci + vd_y * y_vaci;
```

```
double qd = model->Cj0 * vaci * model->Vj *
            (1.0 - pow(1.0 - vd / model->Vj, 1.0 - model->M)) /
            (1.0 - model->M);
double qd_vd = model->Cj0 * model->Vj / (1.0 - model->M) * (1.0 - model->M) *
               pow(1.0 - vd / model->Vj, 1.0 - model->M - 1.0) / model->Vj;
double qd_dtj = qd_vd * vd_dtj;
double qd_vaci = qd_vd * vd_vaci;

// thermal power source = current source
double ith = id * vaci;
double ith_vtj = gdt * vaci;
double ith_vcic = 0;
double ith_vaci = gd * vaci + id;
if (!inst->collapsed[0]) {
  ith_vcic = 2.0 * vcic / rs_t;
  ith += pow(vcic, 2.0) / rs_t;
  ith_vtj -= -pow(vcic, 2.0) / rs_t / rs_t * rs_dt;
}

id += gmin * vaci;
gd += gmin;

double mfactor = inst->mfactor;

////////////////
// write rhs
////////////////

if (info->flags & CALC_RESIST_RESIDUAL) {
  // write resist rhs
  inst->residual_resist[A] = id * mfactor;
  inst->residual_resist[CI] = -id * mfactor + irs * mfactor;
  inst->residual_resist[C] = -irs * mfactor;
  inst->residual_resist[TNODE] = -ith * mfactor + irth * mfactor;
}

if (info->flags & CALC_RESIST_LIM_RHS) {
  // write resist rhs
  inst->lim_rhs_resist_A = gd * mfactor * delvaci;
  inst->lim_rhs_resist_CI = -gd * mfactor * delvaci;
}

if (info->flags & CALC_REACT_RESIDUAL) {
  // write react rhs
```

```
  inst->residual_react_A = qd * mfactor;
  inst->residual_react_CI = -qd * mfactor;
}

if (info->flags & CALC_REACT_LIM_RHS) {
  // write resist rhs
  inst->lim_rhs_react_A = qd_vaci * mfactor * delvaci;
  inst->lim_rhs_react_CI = -qd_vaci * mfactor * delvaci;
}

//////////////////
// write Jacobian
//////////////////

if (info->flags & CALC_RESIST_JACOBIAN) {
  // stamp diode (current flowing from Ci into A)
  inst->jacobian_resist[A_A] = gd * mfactor;
  inst->jacobian_resist[A_CI] = -gd * mfactor;
  inst->jacobian_resist[CI_A] = -gd * mfactor;
  inst->jacobian_resist[CI_CI] = gd * mfactor;
  // diode thermal
  inst->jacobian_resist[A_TNODE] = gdt * mfactor;
  inst->jacobian_resist[CI_TNODE] = -gdt * mfactor;

  // stamp resistor (current flowing from C into CI)
  inst->jacobian_resist[CI_CI] += g * mfactor;
  inst->jacobian_resist[CI_C] = -g * mfactor;
  inst->jacobian_resist[C_CI] = -g * mfactor;
  inst->jacobian_resist[C_C] = g * mfactor;
  // resistor thermal
  inst->jacobian_resist[CI_TNODE] = grt * mfactor;
  inst->jacobian_resist[C_TNODE] = -grt * mfactor;

  // stamp rth flowing into node dTj
  inst->jacobian_resist[TNODE_TNODE] = gt * mfactor;

  // stamp ith flowing out of T node
  inst->jacobian_resist[TNODE_TNODE] -= ith_vtj * mfactor;
  inst->jacobian_resist[TNODE_CI] = (ith_vcic - ith_vaci) * mfactor;
  inst->jacobian_resist[TNODE_C] = -ith_vcic * mfactor;
  inst->jacobian_resist[TNODE_A] = ith_vaci * mfactor;
}

if (info->flags & CALC_REACT_JACOBIAN) {
```

```
    // write react matrix
    // stamp Qd between nodes A and Ci depending also on dT
    inst->jacobian_react[A_A] = qd_vaci * mfactor;
    inst->jacobian_react[A_CI] = -qd_vaci * mfactor;
    inst->jacobian_react[CI_A] = -qd_vaci * mfactor;
    inst->jacobian_react[CI_CI] = qd_vaci * mfactor;

    inst->jacobian_react[A_TNODE] = qd_dtj * mfactor;
    inst->jacobian_react[CI_TNODE] = -qd_dtj * mfactor;
  }

  return ret_flags;
}

// TODO implementation of the @loadnoise@load_noise@ function as defined in the OSDI spec
static void load_noise(void *inst, void *model, double freq, double *noise_dens,
                       double *ln_noise_dens) {
  IGNORE(inst);
  IGNORE(model);
  IGNORE(freq);
  IGNORE(noise_dens);
  IGNORE(ln_noise_dens);
  // TODO add noise to example
}

#define LOAD_RESIDUAL_RESIST(name)                                             \
  dst[inst->node_off[name]] += inst->residual_resist[name];

// implementation of the load_rhs_resist function as defined in the OSDI spec
static void load_residual_resist(void *inst_, void *model, double *dst) {
  DiodeInstace *inst = (DiodeInstace *)inst_;

  IGNORE(model);
  LOAD_RESIDUAL_RESIST(A)
  LOAD_RESIDUAL_RESIST(CI)
  LOAD_RESIDUAL_RESIST(C)
  LOAD_RESIDUAL_RESIST(TNODE)
}

// implementation of the load_rhs_react function as defined in the OSDI spec
static void load_residual_react(void *inst_, void *model, double *dst) {
  IGNORE(model);
  DiodeInstace *inst = (DiodeInstace *)inst_;
```

```
  dst[inst->node_off[A]] += inst->residual_react_A;
  dst[inst->node_off[CI]] += inst->residual_react_CI;
}

// implementation of the load_lim_rhs_resist function as defined in the OSDI
// spec
static void load_lim_rhs_resist(void *inst_, void *model, double *dst) {
  DiodeInstace *inst = (DiodeInstace *)inst_;

  IGNORE(model);
  dst[inst->node_off[A]] += inst->lim_rhs_resist_A;
  dst[inst->node_off[CI]] += inst->lim_rhs_resist_CI;
}

// implementation of the load_lim_rhs_react function as defined in the OSDI spec
static void load_lim_rhs_react(void *inst_, void *model, double *dst) {
  DiodeInstace *inst = (DiodeInstace *)inst_;

  IGNORE(model);
  dst[inst->node_off[A]] += inst->lim_rhs_react_A;
  dst[inst->node_off[CI]] += inst->lim_rhs_react_CI;
}

#define LOAD_MATRIX_RESIST(name)                                               \
  *inst->jacobian_ptr_resist[name] += inst->jacobian_resist[name];

// implementation of the load_matrix_resist function as defined in the OSDI spec
static void load_jacobian_resist(void *inst_, void *model) {
  IGNORE(model);
  DiodeInstace *inst = (DiodeInstace *)inst_;
  LOAD_MATRIX_RESIST(A_A)
  LOAD_MATRIX_RESIST(A_CI)
  LOAD_MATRIX_RESIST(A_TNODE)

  LOAD_MATRIX_RESIST(CI_A)
  LOAD_MATRIX_RESIST(CI_CI)
  LOAD_MATRIX_RESIST(CI_C)
  LOAD_MATRIX_RESIST(CI_TNODE)

  LOAD_MATRIX_RESIST(C_CI)
  LOAD_MATRIX_RESIST(C_C)
  LOAD_MATRIX_RESIST(C_TNODE)

  LOAD_MATRIX_RESIST(TNODE_TNODE)
```

```
  LOAD_MATRIX_RESIST(TNODE_A)
  LOAD_MATRIX_RESIST(TNODE_C)
  LOAD_MATRIX_RESIST(TNODE_CI)
}

#define LOAD_MATRIX_REACT(name)                                                \
  *inst->jacobian_ptr_react[name] += inst->jacobian_react[name] * alpha;

// implementation of the load_matrix_react function as defined in the OSDI spec
static void load_jacobian_react(void *inst_, void *model, double alpha) {
  IGNORE(model);
  DiodeInstace *inst = (DiodeInstace *)inst_;
  LOAD_MATRIX_REACT(A_A)
  LOAD_MATRIX_REACT(A_CI)
  LOAD_MATRIX_REACT(CI_A)
  LOAD_MATRIX_REACT(CI_CI)

  LOAD_MATRIX_REACT(A_TNODE)
  LOAD_MATRIX_REACT(CI_TNODE)
}

#define LOAD_MATRIX_TRAN(name)                                                 \
  *inst->jacobian_ptr_resist[name] += inst->jacobian_react[name] * alpha;

// implementation of the load_matrix_tran function as defined in the OSDI spec
static void load_jacobian_tran(void *inst_, void *model, double alpha) {
  DiodeInstace *inst = (DiodeInstace *)inst_;

  // set dc stamps
  load_jacobian_resist(inst_, model);

  // add reactive contributions
  LOAD_MATRIX_TRAN(A_A)
  LOAD_MATRIX_TRAN(A_CI)
  LOAD_MATRIX_TRAN(CI_A)
  LOAD_MATRIX_TRAN(CI_CI)

  LOAD_MATRIX_TRAN(A_TNODE)
  LOAD_MATRIX_TRAN(CI_TNODE)
}

// implementation of the @loadspicerhsdc@load_spice_rhs_dc@ function as defined in the OSDI spec
static void load_spice_rhs_dc(void *inst_, void *model, double *dst,
                              double *prev_solve) {
```

```
  IGNORE(model);
  DiodeInstace *inst = (DiodeInstace *)inst_;
  double va = prev_solve[inst->node_off[A]];
  double vci = prev_solve[inst->node_off[CI]];
  double vc = prev_solve[inst->node_off[C]];
  double vdtj = prev_solve[inst->node_off[TNODE]];

  dst[inst->node_off[A]] += inst->jacobian_resist[A_A] * va +
                            inst->jacobian_resist[A_TNODE] * vdtj +
                            inst->jacobian_resist[A_CI] * vci +
                            inst->lim_rhs_resist_A - inst->residual_resist[A];

  dst[inst->node_off[CI]] += inst->jacobian_resist[CI_A] * va +
                             inst->jacobian_resist[CI_TNODE] * vdtj +
                             inst->jacobian_resist[CI_CI] * vci +
                             inst->lim_rhs_resist_CI -
                             inst->residual_resist[CI];

  dst[inst->node_off[C]] +=
      inst->jacobian_resist[C_C] * vc + inst->jacobian_resist[C_CI] * vci +
      inst->jacobian_resist[C_TNODE] * vdtj - inst->residual_resist[C];

  dst[inst->node_off[TNODE]] += inst->jacobian_resist[TNODE_A] * va +
                                inst->jacobian_resist[TNODE_C] * vc +
                                inst->jacobian_resist[TNODE_CI] * vci +
                                inst->jacobian_resist[TNODE_TNODE] * vdtj -
                                inst->residual_resist[TNODE];
}

// implementation of the @loadspicerhstran@load_spice_rhs_tran@ function as defined in the OSDI
// spec
static void load_spice_rhs_tran(void *inst_, void *model, double *dst,
                                double *prev_solve, double alpha) {

  DiodeInstace *inst = (DiodeInstace *)inst_;
  double va = prev_solve[inst->node_off[A]];
  double vci = prev_solve[inst->node_off[CI]];
  double vdtj = prev_solve[inst->node_off[TNODE]];

  // set DC rhs
  load_spice_rhs_dc(inst_, model, dst, prev_solve);

  // add contributions due to reactive elements
  dst[inst->node_off[A]] +=
```

```
      alpha *
      (inst->jacobian_react[A_A] * va + inst->jacobian_react[A_CI] * vci +
       inst->jacobian_react[A_TNODE] * vdtj + inst->lim_rhs_react_A);

  dst[inst->node_off[CI]] +=
      alpha *
      (inst->jacobian_react[CI_CI] * vci + inst->jacobian_react[CI_A] * va +
       inst->jacobian_react[CI_TNODE] * vdtj + inst->lim_rhs_react_CI);
}

#define RESIST_RESIDUAL_OFF(NODE)                                              \
  (offsetof(DiodeInstace, residual_resist) + sizeof(uint32_t) * NODE)

// structure that provides information of all nodes of the model
const OsdiNode nodes[NUM_NODES] = {
    {
        .name = "A",
        .units = "V",
        .residual_units = "A",
        .resist_residual_off = RESIST_RESIDUAL_OFF(A),
        .react_residual_off = offsetof(DiodeInstace, residual_react_A),
    },
    {
        .name = "C",
        .units = "V",
        .residual_units = "A",
        .resist_residual_off = RESIST_RESIDUAL_OFF(C),
        .react_residual_off = UINT32_MAX, // no reactive residual

    },
    {
        .name = "dT",
        .units = "K",
        .residual_units = "W",
        .resist_residual_off = RESIST_RESIDUAL_OFF(TNODE),
        .react_residual_off = UINT32_MAX, // no reactive residual
    },
    {
        .name = "CI",
        .units = "V",
        .residual_units = "A",
        .resist_residual_off = RESIST_RESIDUAL_OFF(TNODE),
        .react_residual_off = offsetof(DiodeInstace, residual_react_CI),
```

```
    },
};
#define JACOBI_ENTRY(N1, N2)                                                \
  {                                                                         \
    .nodes = {N1, N2}, .flags = JACOBIAN_ENTRY_RESIST | JACOBIAN_ENTRY_REACT,  \
    .react_ptr_off = offsetof(DiodeInstace, jacobian_ptr_react) +           \
                     sizeof(double *) * N1##_##N2                           \
  }

#define RESIST_JACOBI_ENTRY(N1, N2)                                         \
  {                                                                         \
    .nodes = {N1, N2}, .flags = JACOBIAN_ENTRY_RESIST,                      \
    .react_ptr_off = UINT32_MAX                                             \
  }

// these node pairs specify which entries in the Jacobian must be accounted for
OsdiJacobianEntry jacobian_entries[NUM_MATRIX] = {
    JACOBI_ENTRY(CI, CI),
    RESIST_JACOBI_ENTRY(CI, C),
    RESIST_JACOBI_ENTRY(C, CI),
    RESIST_JACOBI_ENTRY(C, C),
    JACOBI_ENTRY(A, A),
    JACOBI_ENTRY(A, CI),
    JACOBI_ENTRY(CI, A),
    JACOBI_ENTRY(A, TNODE),
    RESIST_JACOBI_ENTRY(C, TNODE),
    JACOBI_ENTRY(CI, TNODE),
    RESIST_JACOBI_ENTRY(TNODE, TNODE),
    RESIST_JACOBI_ENTRY(TNODE, A),
    RESIST_JACOBI_ENTRY(TNODE, C),
    RESIST_JACOBI_ENTRY(TNODE, CI),
};
OsdiNodePair collapsible[NUM_COLLAPSIBLE] = {
    {CI, C},
    {TNODE, NUM_NODES},
};

#define NUM_PARAMS 12
// the model parameters as defined in Verilog-A, bounds and default values are
// stored elsewhere as they may depend on model parameters etc.
OsdiParamOpvar params[NUM_PARAMS] = {
    {
        .name = (char *[]){"$mfactor"},
        .num_alias = 0,
```

```
    .description = "Verilog-A multiplication factor for parallel devices",
    .units = "",
    .flags = PARA_TY_REAL | PARA_KIND_INST,
    .len = 0,
},
{
    .name = (char *[]){"Rs"},
    .num_alias = 0,
    .description = "Ohmic res",
    .units = "Ohm",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
},
{
    .name = (char *[]){"Is"},
    .num_alias = 0,
    .description = "Saturation current",
    .units = "A",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
},
{
    .name = (char *[]){"zetars"},
    .num_alias = 0,
    .description = "Temperature coefficient of ohmic res",
    .units = "",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
},
{
    .name = (char *[]){"N"},
    .num_alias = 0,
    .description = "Emission coefficient",
    .units = "",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
},
{
    .name = (char *[]){"Cj0"},
    .num_alias = 0,
    .description = "Junction capacitance",
    .units = "F",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
```

```
},
{
    .name = (char *[]){"Vj"},
    .num_alias = 0,
    .description = "Junction potential",
    .units = "V",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
},
{
    .name = (char *[]){"M"},
    .num_alias = 0,
    .description = "Grading coefficient",
    .units = "",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
},
{
    .name = (char *[]){"Rth"},
    .num_alias = 0,
    .description = "Thermal resistance",
    .units = "K/W",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
},
{
    .name = (char *[]){"zetarth"},
    .num_alias = 0,
    .description = "Temperature coefficient of thermal res",
    .units = "",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
},
{
    .name = (char *[]){"zetais"},
    .num_alias = 0,
    .description = "Temperature coefficient of Is",
    .units = "",
    .flags = PARA_TY_REAL | PARA_KIND_MODEL,
    .len = 0,
},
{
    .name = (char *[]){"Tnom"},
    .num_alias = 0,
```

```
        .description = "Reference temperature",
        .units = "",
        .flags = PARA_TY_REAL | PARA_KIND_MODEL,
        .len = 0,
    },
};

// fill exported data
uint32_t OSDI_VERSION_MAJOR = OSDI_VERSION_MAJOR_CURR;
uint32_t OSDI_VERSION_MINOR = OSDI_VERSION_MINOR_CURR;
uint32_t OSDI_NUM_DESCRIPTORS = 1;
// this is the main structure used by simulators, it gives @access@access@ to all
// information in a model
OsdiDescriptor OSDI_DESCRIPTORS[1] = {{
    // metadata
    .name = "diode_va",

    // nodes
    .num_nodes = NUM_NODES,
    .num_terminals = 3,
    .nodes = (OsdiNode *)&nodes,
    .node_mapping_offset = offsetof(DiodeInstace, node_off),

    // matrix entries
    .num_jacobian_entries = NUM_MATRIX,
    .jacobian_entries = (OsdiJacobianEntry *)&jacobian_entries,
    .jacobian_ptr_resist_offset = offsetof(DiodeInstace, jacobian_ptr_resist),

    // node collapsing
    .num_collapsible = NUM_COLLAPSIBLE,
    .collapsible = collapsible,
    .collapsed_offset = offsetof(DiodeInstace, collapsed),

    // noise
    .noise_sources = NULL,
    .num_noise_src = 0,

    // parameters and op vars
    .num_params = NUM_PARAMS,
    .num_instance_params = 1,
    .num_opvars = 0,
    .param_opvar = (OsdiParamOpvar *)&params,

    // step size bound
```

```
    .bound_step_offset = UINT32_MAX,

    .num_states = 1,
    .state_idx_off = offsetof(DiodeInstace, state_idx),

    // memory
    .instance_size = sizeof(DiodeInstace),
    .model_size = sizeof(DiodeModel),

    // setup
    .access = osdi_access,
    .setup_model = setup_model,
    .setup_instance = setup_instance,
    .eval = eval,
    .load_noise = load_noise,
    .load_residual_resist = load_residual_resist,
    .load_residual_react = load_residual_react,
    .load_spice_rhs_dc = load_spice_rhs_dc,
    .load_spice_rhs_tran = load_spice_rhs_tran,
    .load_jacobian_resist = load_jacobian_resist,
    .load_jacobian_react = load_jacobian_react,
    .load_jacobian_tran = load_jacobian_tran,
    .load_limit_rhs_react = load_lim_rhs_react,
    .load_limit_rhs_resist = load_lim_rhs_resist,
}};

OsdiLimFunction OSDI_LIM_TABLE[1] = {{.name = "pnjlim", .num_args = 2}};

uint32_t OSDI_LIM_TABLE_LEN = 1;
```

Glossary

device descriptor An instance of the `OsdiDescriptor` struct. Corresponds to a single Verlog-A module and therefore usually a single compact model like `BSIMCMG` or `HICUM/L2`. 8, 21, 22

instance A single physical device within a netlist. 8, 9, 10, 11, 12, 13, 14, 15, 16, 22, 23, 24, 25, 31, 32, 33, 36, 41, 42, 49, 79

jacobian entry An entry within the jacobian matrix of the system of equations solved by the circuit simulator. As a matrix entry an *jacobian entry* is characterised by a row and a column. The row and the column refer to a residual (row) and the unkown (column) its derived by. Both the residual entries and unkowns are represented by nodes . 9, 10, 13, 15, 16, 22, 24, 32, 33, 42, 43, 44

model A set of parameters that is shared between multiple physical devices (instances). 8, 9, 11, 12, 13, 25, 41, 42, 49

node A *node* refers to an unkown within the system of equations solved by the circuit simulator. In most cases a *node* refers to the potential of a circuit-node (hence the name). However other unkowns (like currents) are also possible. Each unkown has an associated equation whose residual (usually the currents flowing into a circuit-node) is also refered to by the node . 9, 21, 22, 24, 28, 30, 31, 32, 33, 36, 79

residual Circuit simulators solve the equation $\vec{F}(\vec{x}) = \vec{0}$ with a newton iteration $J(\vec{x}_{k+1} - \vec{x}_k) = -\vec{F}(\vec{x}_k)$. The residual refers to the value $\vec{F}(\vec{x}_k)$. For many simulator the residual is equal to the RHS. . 10, 13, 14, 15, 16, 31, 43, 44, 79

RHS Circuit simulators solve the equation $\vec{F}(\vec{x}) = \vec{0}$ with a newton iteration $J(\vec{x}_{k+1} - \vec{x}_k) = -\vec{F}(\vec{x}_k)$. To that end the equations $A\vec{y} = \vec{b}$ is solved for $\vec{y}$. A is the jacobian J and $\vec{b}$ is the RHS. The exact definition of the RHS $\vec{b}$ depends on the formulation of the newton algorithm used by the simulator. Usually the RHS is either equal to the residual $\vec{b} = \vec{F}(\vec{x}_k)$ or also involves the jacobian $\vec{b} = J\vec{x}_k - \vec{F}(\vec{x}_k)$. 10, 13, 14, 15, 79

terminal Terminals are those nodes of an instance that can be connected to other instances. They Correspond to Verlog-A ports . 21