From vhd to thy

Zhé Hóu

September 19, 2016

The main task is to convert a VHDL file (.vhd) to an Isabelle file (.thy). Note that we do not use the complete VHDL syntax, but only a subset of it.

1 Basics of Isabelle files

A .thy file is wrapped in the following structure:

```
theory filename
imports Main vhdl_component vhdl_syntax_complex
begin
...
end
```

where filename must be identical to the name of the .thy file, and the ... part is the content of the file.

Optionally you can add a commented header (placed at the beginning of the file, before the above) for license and related issues, for example:

```
(*
 * Copyright 2016, NTU
 *
 * This software may be distributed and modified according to
 * the terms of the BSD 2-Clause license. Note that NO WARRANTY
 * is provided. See "LICENSE_BSD2.txt" for details.
 *
 * Author: Hongxu Chen and Nam.
```

*)

2 VHDL types, values, and expressions in Isabelle

2.1 VHDL types in Isabelle

The following are the VHDL types pre-defined in our Isabelle model, these include: `vhdl_boolean`, `vhdl_bit`, `vhdl_character`, `vhdl_integer`, `vhdl_positive`, `vhdl_natural`, `vhdl_real`, `vhdl_time`, `vhdl_string`, `vhdl_bitstr`, `vhdl_boolstr`, `vhdl_std_logic`,
`vhdl_std_ulogic`, `vhdl_std_logic_vector`, `vhdl_std_ulogic_vector`, `vhdl_signed`, `vhdl_unsigned`.

As far as I see in the iu.vhd, the LEON3 design only uses `vhdl_signed` and `vhdl_unsigned` in shifting operations, thus I just treat these two types in the same way as `vhdl_std_logic_vector`. Shifting on signed must be arithmetic shift, shifting on unsigned must be logical shift.

2.2 VHDL values in Isabelle

The values in VHDL are presented in Isabelle as below:

In VHDL	In Isabelle
If a value is an integer i	`val_i i`
If a value is a real r	`val_r r`
If a value is a character c	`val_c (CHR ''c'')`
If a value is a std_logic c	`val_c (CHR ''c'')`
If a value is a std_ulogic c	`val_c (CHR ''c'')`
If a value is a Boolean b	`val_b b`
If a value is a vector defined with `TO`	`val_list [...]`
If a value is a vector defined with `DOWNTO`	`val_rlist [...]`
Not used	`val_null`

2.3 VHDL expressions in Isabelle

The supported expressions in the Isabelle model is defined inductively:

```
datatype expression =
uexp uop expression
|bexpl expression lop expression
|bexpr expression rop expression
|bexps expression sop expression
|bexpa expression aop expression
|exp_sig "(name x type x signal_kind x expression)"
|exp_prt "(name x type x mode x connection x expression)"
|exp_var "(name x type x expression)"
|exp_con const
|exp_nth expression expression
|exp_sl expression expression expression
|exp_tl expression
|exp_trl expression
|exp_fc "name x (expression list) x type"
|exp_r rhsl
and rhsl =
rl_s "(name x type x signal_kind expression)"
|rl_p "(name x type x mode x connection x expression)"
|rl_v "(name x type x expression)"
|rnl "rhsl list"
```

`uexp` is for unary expressions. `uop` is one of `[abs]` (absolute value), `[not]`, `[-:]` (negative), `[+:]` (positive). For example:

In VHDL	In Isabelle
-e	uexp [-:] e

`bexpl` is for binary logical expressions. `rop` is one of `[and]`, `[or]`, `[nand]`, `[nor]`, `[xor]`, `[xnor]`. For example:

In VHDL	In Isabelle
e1 and e2	bexpl e1 [and] e2

bexpr is for binary relational expressions. `rop` is one of `[=]`, `['/=]` (not equal), `[<]`, `[<=]`, `[>]`, `[>=]`. For example:

In VHDL	In Isabelle
e1 /= e2	bexpr e1 ['/=] e2

bexps is for binary shifting expressions. `sop` is one of `[sll]`, `[srl]`, `[sla]`, `[sra]`, `[rol]`, `[ror]`. For example:

In VHDL	In Isabelle
e1 sll e2	bexps e1 [sll] e2

bexpa is for binary arithmetic expressions. `aop` is one of `[+]`, `[-]`, `[&]` (concatenate), `[*]`, `['/]` (division), `[mod]`, `rem` (reminder), `[**]` (exponential). For example:

In VHDL	In Isabelle
e1 / e2	bexpa e1 ['/] e2

exp_nth gets the nth member of a vector. In an instance `exp_nth e1 e2`, `e1` must be an expression of a vector type, and `e2` must be an expression of type `vhdl_natural`. This is useful when expressing member access of a vector type object in VHDL. For example:

In VHDL	In Isabelle
v(63)	exp_nth (exp_var v) (exp_con (vhdl_natural, val_i 63))

exp_sl gets a sub-vector of a vector. In an instance `exp_sl e1 e2 e3`, `e1` must be an expression of a vector type, `e1` and `e2` must be expressions of type `vhdl_natural`. This is useful when the right hand side of an assignment (**not the left hand side, which requires a different Isabelle construct**) is a range expression. For example:

In VHDL	In Isabelle
s1 <= s2(31 downto 0);	("":(lhs_s (sp_s s1)) <= (rhs_e (exp_sl (exp_sig s2) (exp_con (vhdl_natural, val_i 31)) (exp_con (vhdl_natural, val_i 0)))))

`exp_tl` converts an expression to a vector type defined with `TO`. This is used, e.g., when the VHDL code mixes the use of non-vector objects and vector objects. Isabelle is strongly typed, so we have to convert the type explicitly. For example:

In VHDL	In Isabelle
v & '1'	bexpa (exp_var v) [&] (exp_trl (exp_con (vhdl_std_logic, val_c (CHR "1"))))

That is, the VHDL code applies list concatenation to a list and a member of the list[1]. This is not allowed in Isabelle, so we need to convert the member to a singleton list first.

`exp_trl` converts an expression to a vector type defined with `DOWNTO`.

`exp_fc` is for function calls in expressions, **but** we never actually use this, see Section 6.10 for details.

`exp_r` is for the right hand side of the assignments of record types. For example, if the variables `v` and `rin` are of a record type in the VHDL code, we convert the record type assignment as below:

In VHDL	In Isabelle
rin <= v;	("": (clhs_spr rin) <= (rhs_e (exp_r (rhsl_of_vl v)))),

2.4 Useful abbreviations

The following as some useful abbreviations for expressions:

```
el b = exp_con (vhdl_std_logic,(val_c b))

eul b = exp_con (vhdl_std_ulogic,(val_c b))

en i = exp_con (vhdl_natural,(val_i i))

ell l = exp_con (vhdl_std_logic_vector,(val_list l))
```

[1]Or, you can say VHDL overloads the symbols for list concatenation and list appending.

```
eull l = exp_con (vhdl_std_ulogic_vector,(val_list l))

elrl l = exp_con (vhdl_std_logic_vector,(val_rlist l))

eulrl l = exp_con (vhdl_std_ulogic_vector,(val_rlist l))
```

3 Using signals/ports in Isabelle

In many occasions the Isabelle model use a type `sigprt` for either a signal or a port. A signal `s` in VHDL corresponds to `sp_s s` in the Isabelle code. A port `p` in VHDL corresponds to `sp_p p`. This is used in the definition of `env_sp`, and in the Isabelle code for the VHDL statements. For example, a signal assignment in VHDL

```
addsub <= vaddsub;
```

is translated to

```
('''': (clhs_sp (lhs_s (sp_s addsub))) <= (rhs_e (exp_var vaddsub)))
```

4 Declarations.

4.1 Library and import declarations

These two kinds of declarations in .vhd is ignored.

4.2 Entity declaration

Each .vhd file should have an entity declaration (usually only one). An entity declaration is of the form:

```
entity div32 is
generic (scantest  : integer := 0);
port (
    rst      : in  std_ulogic;
```

```
    clk      : in   std_ulogic;
    holdn    : in   std_ulogic;
    divi     : in   div32_in_type;
    divo     : out  div32_out_type;
    testen   : in   std_ulogic := '0';
    testrst  : in   std_ulogic := '1'
);
end;
```

A entity corresponds to a definition of type `vhdl_desc_complex` in Isabelle. This type is a triple of `environment`, `res_fn`, `conc_stmt_complex list`, and `subprogram_complex list`.

`environment` is a record with three fields: `env_sp` for the list of signals/ports; `env_v` for the list of variables/constants/generics; and `env_t` for the list of types defined in the .vhd file (record types, as said above, do not count). Usually `env_t` is blank, as my model covers the widely-used types.

`res_fn` is the resolution function for signals. This field is usually `\<lambda>x.(None)`, i.e., an empty function.

`conc_stmt_complex list` is the list of concurrent statements in this entity.

`subprogram_complex list` is the list of function bodies and procedure bodies in this entity.

The entity above is translated to the following definition in Isabelle:

```
definition div32:: "vhdl_desc_complex" where
"div32 \<equiv>
  let env = \<lparr>env_sp = [...] ,
             env_v = [...] ,
             env_t = [...]\ < rparr >;
      resfn = \<lambda>x.(None);
      cst_list = [...]
  in
  (env, resfn , cst_list , [])
"
```

where the ... parts are to be filled later.

4.3 Declaration of architecture

In the Isabelle code we do not explicitly define architectures. Instead, we associate each architecture to a state. For example, the architecture declaration

```
architecture rtl of div32 is
(* signal/variable/constant declarations *)
begin
(* concurrent statements *)
end
```

corresponds to the following state in Isabelle:

```
definition init_arch_state_rtl:: "vhdl_arch_state" where
"init_arch_state_rtl \<equiv> arch_state (''rtl'',
init_state,[])"
```

where the empty list `[]` contains the components used in this architecture. In case there are components in the architecture, each component will be a pair `(comp_port_map, vhdl_arch_state)` in the list. `comp_port_map` is simply a mapping from `sigprt` to `sigprt` (see Section 3). For instance, to define the port mappings

$$p1 \mapsto s1,\ p2 \mapsto p3,$$

use the following Isabelle definition:

```
definition my_port_map:: "comp_port_map" where "my_port_map
\<equiv> add_comp_port_map [(sp_p p1, sp_s s1),
  (sp_p p2, sp_p p3)] emp_comp_port_map"
```

Signal/variable/port/constant declarations are translated as below.

Concurrent statements are translated in the next section. The translated version goes to the list `cst_list` in definition for the entity.

4.4 Declaration of variables

Some entities have a **generic declaration**, which includes some declarations of variables. These are all translated to definitions of type `variables` in Isabelle. A **variable type in Isabelle** is defined as a triple of `name`, `type`, and `expression`.

`name` is a string in Isabelle, which is of the form `''...''` (enclosed by two single quotes, not a double quote) where ... is the content of the string.

`type` is the type of the variable.

`expression` is the initial expression of the variable.

For example, the generic variable "scantest" in the above entity declaration is translated to:

```
definition scantest:: "variable" where
"scantest \<equiv> (''scantest'', vhdl_integer,
(exp_con (vhdl_integer, (val_i 0))))"
```

N.B. constants in VHDL are defined as variables in Isabelle.

4.5 Declaration of ports

Ports in a VHDL design is usually declared right after the entity declaration. See the `div32` entity for an example. Sometimes there are no initial values for ports. But in Isabelle we need to make up some initial values, usually 0 for `integers`, and '0' for `std_logic` and `std_ulogic` and `char` and `bit`.

A port in my Isabelle model is defined as a tuple of `name`, `type`, `mode`, `connection`, `expression`. Some of them are explained before, the others stand for:

`mode` is the mode of the port, in Isabelle we have defined the following modes: `mode_in`, `mode_out`, `mode_inout`, `mode_buffer`, `mode_linkage`.

`connection` is either `connected` or `unconnected`. Usually use the former if not defined in VHDL.

`expression` is an expressions for the initial value of the port. Here we use a constant expression for the initial value. A constant is a pair of `type` and `val`. A constant expression is of the form `exp_con (type,val)`.

For example, the declaration of the port `rst` in `div32` is translated as below:

```
definition rst:: "port" where
"rst \<equiv> (''rst'', vhdl_std_ulogic, mode_in, connected,
  (exp_con (vhdl_std_ulogic, (val_c (CHR ''1'')))))"
```

4.6 Declaration of signals

As for ports, signal declarations in Isabelle requires an initial value. A signal is defined as a tuple of `name`, `type`, `signal_kind`, `expression`, the last of which is the initial value of the signal.

`signal_kind` is one of `register`, `bus`, and `discrete`. If it's not defined, use `register`.

Here's an example of signal declaration in VHDL:

```
signal arst   : std_ulogic;
```

This is translated to the following Isabelle definition:

```
definition arst:: "signal" where
"arst \<equiv> (''arst'', vhdl_std_ulogic, register,
(exp_con (vhdl_std_ulogic, (val_c (CHR ''1'')))))"
```

4.7 Declaration of signal/port/variable vectors

If a signal/port/variable is a vector (i.e., array/list), the value will be a `val_list` or `val_rlist`. For example, the following says `addin1` is a signal of type `std_logic_vector` and is defined with `downto`:

```
signal addin1: std_logic_vector(32 downto 0);
```

This is translated to

```
definition addin1:: "signal" where
"addin1 \<equiv> (''addin1'', vhdl_std_logic_vector, register,
(exp_con (vhdl_std_logic_vector,
(val_list (std_logic_vec_gen 33 (val_c (CHR ''0'')))))))"
```

Note that a vector defined with `downto` corresponds to values of `val_rlist` (reversed list); a vector defined with `to` corresponds to `val_list`.

We initialise the vector to all 0s. You can use an Isabelle function `std_logic_vec_gen x val` to generated a list of values `val` of length `x`. This functions can be used for both `downto` and `to`.

4.8 Declaration of signal/port/variable records

We do not explicitly define record types in Isabelle. Instead, a record in VHDL is defined as a list in Isabelle. For example, a signal record in VHDL corresponds to a signal list in Isabelle, where each field of the record corresponds to a signal in the list. Consider the following record type declaration and constant record declaration:

```
type div_regtype is record
  x      : std_logic_vector(64 downto 0);
  state  : std_logic_vector(2 downto 0);
  zero   : std_logic;
  zero2  : std_logic;
  qcorr  : std_logic;
  zcorr  : std_logic;
  qzero  : std_logic;
  qmsb   : std_logic;
  ovf    : std_logic;
  neg    : std_logic;
  cnt    : std_logic_vector(4 downto 0);
end record;

constant RRES : div_regtype := (
  x      => (others => '0'),
  state  => (others => '0'),
  zero   => '0',
  zero2  => '0',
  qcorr  => '0',
  zcorr  => '0',
```

```
qzero   => '0',
qmsb    => '0',
ovf     => '0',
neg     => '0',
cnt     => (others => '0'));
```

We define RRES as below in Isabelle:

```
definition rres:: "variable list" where
"rres \<equiv> vnl ('''',[
  vl_v (''rres_x'', vhdl_std_logic_vector, (exp_con (vhdl_std_logic_vector,
    (val_rlist (std_logic_vec_gen 65 (val_c (CHR ''0''))))))),
  vl_v (''rres_state'', vhdl_std_logic_vector, (exp_con (vhdl_std_logic_vector,
    (val_rlist (std_logic_vec_gen 3 (val_c (CHR ''0''))))))),
  vl_v (''rres_zero'', vhdl_std_logic, (exp_con (vhdl_std_logic, (val_c (CHR ''0''))))),
  vl_v (''rres_zero2'', vhdl_std_logic, (exp_con (vhdl_std_logic, (val_c (CHR ''0''))))),
  vl_v (''rres_qcorr'', vhdl_std_logic, (exp_con (vhdl_std_logic, (val_c (CHR ''0''))))),
  vl_v (''rres_zcorr'', vhdl_std_logic, (exp_con (vhdl_std_logic, (val_c (CHR ''0''))))),
  vl_v (''rres_qzero'', vhdl_std_logic, (exp_con (vhdl_std_logic, (val_c (CHR ''0''))))),
  vl_v (''rres_qmsb'', vhdl_std_logic, (exp_con (vhdl_std_logic, (val_c (CHR ''0''))))),
  vl_v (''rres_ovf'', vhdl_std_logic, (exp_con (vhdl_std_logic, (val_c (CHR ''0''))))),
  vl_v (''rres_neg'', vhdl_std_logic, (exp_con (vhdl_std_logic, (val_c (CHR ''0''))))),
  vl_v (''rres_cnt'', vhdl_std_logic_vector, (exp_con (vhdl_std_logic_vector,
    (val_rlist (std_logic_vec_gen 5 (val_c (CHR ''0'')))))))
])"
```

It is important that each member of the list has a name prefixed with the name of the list and an underscore. For example, all member names start with `rres_` if the name of the list is `rres`. Our Isabelle model uses the prefix to search for members of a list.

5 Concurrent statements

Our Isabelle model accepts three types of concurrent statements: *process statement*, *concurrent signal assignment*, and *generate statement*.

5.1 Process statement

A process statement in VHDL is of the form

```
name : process(s1,s2,p1,...)
(* variable declarations *)
begin
```

```
  seq_stmt1;
  seq_stmt2;
  ...
end process;
```

where seq_stmt1 and seq_stmt2 are sequential statements. In Isabelle we deal with variable declarations before the definition of the entity (of type vhdl_desc_complex). Thus we assume the variable declarations in a process statement have already been defined in Isabelle. The above is translated to

```
(''name'':PROCESS([sp_s s1, sp_s s2, sp_p p1,...])
  BEGIN [seq_stmt1', seq_stmt2',...]
  END PROCESS)
```

where seq_stmt1' and seq_stmt2' are the translated version of the corresponding sequential statements.

5.2 Concurrent signal assignment

Consider the following concurrent signal assignment in VHDL:

```
s <= s1 when exp1 else
     s2 when exp2 else
     p1 when exp3 else
     c
```

where s1, s2 are signals, p1 is a port, and c is a constant.

If s is not a record, the above is translated to

```
('''':(clhs_sp (lhs_s (sp_s s))) <= <[(rhs_e (exp_sig s1)) WHEN exp1' ELSE,
                                    (rhs_e (exp_sig s2)) WHEN exp2' ELSE,
                                    (rhs_e (exp_prt p1)) WHEN exp3' ELSE]>
                                   (rhs_e (exp_con c))
)
```

If s is a record, which means s1, s2, p1 and c are all records, the assignment is translated as follows:

```
('''':(clhs_r s) <= <[(rhs_e (exp_r (rhsl_of_spl s1))) WHEN exp1' ELSE,
                       (rhs_e (exp_r (rhsl_of_spl s2))) WHEN exp2' ELSE,
                       (rhs_e (exp_r (rhsl_of_spl p1))) WHEN exp3' ELSE]>
                      (rhs_e (exp_r (rhsl_of_vl c)))
)
```

Here's a concrete example:

```
arst <= testrst when (ASYNC_RESET and scantest/=0 and testen/='0') else
        rst when ASYNC_RESET else
        '1';
```

This is translated to:

```
('''': (clhs_sp (lhs_s (sp_s arst))) <= <[((rhs_e (exp_prt testrst)) WHEN
  (bexpl (exp_var async_reset) [and] (bexpl
  (bexpr (exp_var scantest) [/=] (exp_con (vhdl_integer, (val_i 0)))) [and]
  (bexpr (exp_prt testen) [/=] (eul (CHR ''0''))))) ELSE),
  ((rhs_e (exp_prt rst)) WHEN (exp_var async_reset) ELSE)]>
  (rhs_e (eul (CHR ''1'')))),
```

5.3 Generate statement

If generate statement is of the following form:

```
name : if exp generate begin
  conc_stmt1;
  conc_stmt2;
  ...
end generate
```

where `conc_stmt1` and `conc_stmt2` are concurrent statements. This is translated to

```
(''name'' : IF exp' GENERATE BEGIN
  conc_stmt1';
  conc_stmt2';
  ...
END GENERATE)
```

where `conc_stmt1'` and `conc_stmt2'` are translated concurrent statements.

For generate statement is of the following form:

```
name : for i in 0 to 10 generate begin
  conc_stmt1;
  conc_stmt2;
  ...
end generate
```

This is translated to

```
(''name'' : FOR (exp_var i) IN (en 0) TO (en 10) GENERATE BEGIN
  conc_stmt1 ';
  conc_stmt2 ';
  ...
END GENERATE)
```

6 Sequential statements

Our Isabelle model only covers a synthesisable subset of VHDL. The related sequential statements are translated as below.

6.1 Signal assignment

A signal assignment has the form

```
lhs <= rhs;
```

If the target signal/port is not a record , then `lhs` can be either a signal/-port (including a member of a signal/port record), or a signal/port vector with a range. These are translated as follows, where `s1` is a signal, `p1` is a port, and `s2.m` is a signal:

lhs In VHDL	lhs In Isabelle
s1	(clhs_sp (lhs_s (sp_s s1)))
p1	(clhs_sp (lhs_s (sp_p p1)))
s2.m	(clhs_sp (lhs_s (sp_s (s2 s.''s2_m''))))
s1(30 downto 1)	(clhs_sp (lhs_sa (sp_s s1) ((en 30) DOWNTO (en 1))))
s1(5 to 10)	(clhs_sp (lhs_sa (sp_s s1) ((en 5) TO (en 10))))

The right hand side of the assignment can be either an expression or an `other` expression. These are translated as below:

rhs In VHDL	rhs In Isabelle
e	(rhs_e e)
(others => '0')	(OTHERS => (el (CHR ''0'')))

Refer to Section 2.3 for how expressions are translated in Isabelle.

The assignment is translated to:

```
('''': lhs' <= rhs')
```

where `lhs'` and `rhs'` are the translated left hand side and right hand side.

If the target signal/port is a record , then `lhs` must be a signal record or a port record, which corresponds to a signal list or a port list in Isabelle. The `rhs` can be a signal record or a port record or a variable record. Assuming s1 is a signal vector, p1 is a port vector, and v1 is a variable vector, the translation of the left hand side is as follows:

lhs In VHDL	lhs In Isabelle
s1	(clhs_spr s1)
p1	(clhs_spr p1)

and the right hand side is translated as below:

rhs In VHDL	rhs In Isabelle
s1	(rhs_e (exp_r (rhsl_of_spl s1)))
p1	(rhs_e (exp_r (rhsl_of_spl p1)))
v1	(rhs_e (exp_r (rhsl_of_vl v1)))

In this case, the assignment is translated as below:

```
('''': lhs' <= rhs')
```

where `lhs'` and `rhs'` are the translated left hand side and right hand side.

6.2 Variable assignment

The treatment for variable assignments are very similar to that for signal assignments.

A variable assignment has the form

```
lhs := rhs;
```

If the target signal/port is not a record , then `lhs` can be either a variable (including a member of a variable record), or a variable vector with a range. These are translated as follows, where `v1` and `v2.m` are variables:

lhs In VHDL	lhs In Isabelle
v1	(clhs_v (lhs_v v1))
v2.m	(clhs_v (lhs_v (v2 v.''v2_m'')))
v1(30 downto 1)	(clhs_v (lhs_va v1 ((en 30) DOWNTO (en 1))))
v1(5 to 10)	(clhs_v (lhs_va v1 ((en 5) TO (en 10))))

The right hand side of a variable assignment is the same as the right hand side of a signal assignment. See Section 6.1.

The assignment is translated to:

```
('''': lhs' := rhs')
```

where `lhs'` and `rhs'` are the translated left hand side and right hand side.

If the target signal/port is a record , then `lhs` must be a variable record, which corresponds to a variable list in Isabelle. The `rhs` can be a signal record or a port record or a variable record. Assuming s1 is a signal vector, p1 is a port vector, and v1 is a variable vector, the translation of the left hand side is as follows:

lhs In VHDL	lhs In Isabelle
v1	(clhs_vr v1)

and the right hand side is translated as below:

rhs In VHDL	rhs In Isabelle
s1	(rhs_e (exp_r (rhsl_of_spl s1)))
p1	(rhs_e (exp_r (rhsl_of_spl p1)))
v1	(rhs_e (exp_r (rhsl_of_vl v1)))

In this case, the assignment is translated as below:

```
('''': lhs' := rhs')
```

where `lhs'` and `rhs'` are the translated left hand side and right hand side.

6.3 If statement

An `if` statement is of the form:

```
if exp1 then
  seq_stmt1;
  seq_stmt2;
  ...
elsif exp2 then
  ...
elsif exp3 then
  ...
else
  ...
end if;
```

where `exp1`, `exp2`, and `exp3` are Boolean expressions, `seq_stmt1` and `seq_stmt2` are sequential statements. This is translated to:

```
('''': IF exp' THEN
  [seq_stmt1',
  seq_stmt2',
  ...]
  [(ELSIF exp2' THEN [...]),
   (ELSIF exp3' THEN [...])]
  ELSE [...] END IF)
```

where `exp1'`, `exp2'`, and `exp3'` are translated Boolean expressions, `seq_stmt1'` and `seq_stmt2'` are translated sequential statements.

6.4 Case statement

A `case` statement has the form:

```
case exp is
when choices1 =>
  seq_stmt1;
  seq_stmt2;
```

```
  ...
when choices2 =>
  ...
when others =>
  ...
end case;
```

where `exp` is an expression, `choices1` and `choices2` are expressions separated by | (there may be only one expression), `seq_stmt1` and `seq_stmt2` are sequential statements. This is translated to:

```
(''''': CASE exp' IS
WHEN choices1' =>
  [seq_stmt1',
  seq_stmt2',
  ...]
WHEN choices2' =>
  [...]
WHEN OTHERS =>
  [...]
END CASE)
```

where `exp'` is the translated expression, `choices1'` and `choices2'` are lists of translated expressions, `seq_stmt1'` and `seq_stmt2'` are translated sequential statements.

6.5 While statement

A `while` statement is of the form

```
name: while exp loop
  seq_stmt1;
  seq_stmt2;
  ...
end loop;
```

where `exp` is an expression, `seq_stmt1` and `seq_stmt2` are sequential statements. This is translated to:

```
(''name'': WHILE exp' LOOP
  [seq_stmt1',
  seq_stmt2',
  ...]
END LOOP;
```

where `exp'` is the translated expression, `seq_stmt1'` and `seq_stmt2'` are translated sequential statements.

6.6 For statement

A `for` statement is of the form

```
name:for i in 1 to/downto 10 loop
  seq_stmt1;
  seq_stmt2;
  ...
end loop;
```

This is translated to:

```
(''name'': FOR (exp_var i) IN (1 TO/DOWNTO 10) LOOP
  [seq_stmt1',
  seq_stmt2',
  ...]
END LOOP)
```

6.7 Next statement

A `next` statement is of the form

```
next name when exp;
```

where `name` is the label of a while/for statement, `exp` is a Boolean expression. This is translated to:

```
('''': NEXT ''name'' WHEN exp')
```

where `exp'` is the translated expression.

6.8 Exit statement

An `exit` statement is of the form

```
exit name when exp;
```

where `name` is the label of a while/for statement, `exp` is a Boolean expression. This is translated to:

```
('''': EXIT ''name'' WHEN exp')
```

where `exp'` is the translated expression.

6.9 Null statement

A `null` statement is of the form

```
null;
```

This is translated to

```
NULL
```

6.10 Function Call

In VHDL function calls are expressions. In my Isabelle model, we support function calls as a specific type of variable assignment statements as below:

```
v := f(x);
```

Suppose `x` is a variable, and this function's return type is `vhdl_integer`, this is translated to the following syntax in Isabelle (if v is not a record):

```
ssc_fn '''' (clhs_v (lhs_v v)) (''f'', [(rhs_e (exp_var x))],
                vhdl_integer)
```

For each function call in VHDL of the following form:

```
v := f(x) + ...;
```

We create a fresh and distinct variable `v_tmp`, and create a variable assignment as below:

```
v_tmp := f(x);
v := v_tmp + ...;
```

If a variable is initialised by a function call, we **do not** use the function call as the variable's initial expression. Instead, initialise the variable as an arbitrary value, and assign the function call to the variable at the beginning of **every** process in the entity. This ensures that the variable is initialised correctly.

6.11 Procedure Call

Procedure calls in VHDL are statements of the form

```
p(x);
```

Suppose `x` is a variable, in Isabelle this is translated to:

```
ssc_pc '''' (''p'', [(rhs_e (exp_var x))], emptype)
```

Note that procedures don't return values, so the "return type" is always `emptype`.

7 Components

If a .vhd file uses components from other entities (defined in other .vhd files), all the related entities have to be included, for example:

```
definition vhdl_power_comp:: "vhdl_arch_desc_all" where
"vhdl_power_comp \<equiv> [(''MULT'', trans_vhdl_desc_complex
  vhdl_mult), (''POWER'', trans_vhdl_desc_complex vhdl_power)]"
```

Here, `vhdl_mult` and `vhdl_power` are two definitions of type `vhdl_desc_complex` (cf. Section 4.2). More info about components can be found in Section 4.3.

8 Code export

Finally, the last two lines in the .thy file should be the commands for Isabelle code export, for example:

```
export_code this_thy_file init_arch_state_power sim_arch in OCaml
module_name this_thy file output_file_name
```