GEZEL
v2 Code Generator
User Manual

(Version January 1, 2006)

Patrick Schaumont (pschaumont@gmail.com)
Doris Ching (doris.kc@gmail.com)

1.0 Introduction

The GEZEL code generation tool converts cycle-based hardware modules captured in the GEZEL language into synthesizable VHDL code. This manual briefly reviews installation of the tool, and describes how to use the code generator and the resulting VHDL code.

Support questions for the tool can be directed to Patrick Schaumont, Virginia Tech ECE Department, 302 Whittemore Hall (0111), Blacksburg, VA 24060. Email: schaum@vt.edu. Phone: +1 540 231 3553.

Installation

The GEZEL code generation tool, fdlvhd.exe, is distributed as an executable for Cygwin. Cygwin can be downloaded from www.cygwin.com and runs on Windows Machines. After installing cygwin, simply unpack the distribution to your home directory.

```
> $ tar zxfv gcgeval.tgz
gezelcg-2.0.1-eval/
gezelcg-2.0.1-eval/bin/
gezelcg-2.0.1-eval/bin/fdlvhd.exe
gezelcg-2.0.1-eval/test/
gezelcg-2.0.1-eval/test/euclid.fdl
gezelcg-2.0.1-eval/test/Makefile
```

Next, test the installation. The example in the test/ directory is the EUCLID example from the standalone GEZEL simulator suite.

```
> cd gezelcg-2.0.1-eval/test/

> make
../bin/fdlvhd.exe euclid.fdl
Pre-processing System ...
Output VHDL source ...
---------------------------
Generate file: euclid.vhd
Generate file: test_euclid.vhd
Generate file: euclid_sys.vhd
Generate file: system.vhd
Generate file: std_logic_arithext.vhd
```

In the next section, the use of fdlvhd.exe is explained, including the implementation path to VHDL simulation (ModelSim).

2.0 Using fdlvhd

Using the fdlvhd tool, GEZEL designs can be converted automatically into synthesizable VHDL code. The code generation feature shows one of the benefits of working with the restricted semantics of GEZEL: All GEZEL code that you can write and that simulates correctly can be converted into synthesizable VHDL. Apart from the simulation directives, there are no 'non-synthesizable' constructs. The library modules (ipblock) are converted into black boxes.

We discuss the use of the code generator tool, fdlvhd, as well as the use of the generated code in external tools. In addition, GEZEL can also record test vector stimuli for use in VHDL simulation.

2.1 The fdlvhd tool

The command line of fdlvhd is as follows

```
> fdlvhd [-d] [-i] [-s] [-c clock reset] [<filename>]
```

Parameters in between square brackets are optional.

- The `-d` flag enables debug mode for the VHDL code generator, which creates a statistics report for the generated code.
- The `-i` flag creates an active-low reset for the generated VHDL code. The default is an active-high reset.
- The `-s` flag creates a synchronous reset for the generated VHDL code. The default is an asynchronous reset.
- The `-c` flag allows to specify the names for the clock and reset nets in the generated VHDL code. When specifying for example -c myclock myreset then the generated code will use the name myclock for the clock net and myreset for the reset net.
- `<filename>` is the filename of the GEZEL code. This is an optional parameter. When this parameter is not provided, fdlvhd will try to read a GEZEL description from standard input. This can be done with redirection or with shell scripting, as discussed in the GEZEL User Manual.

In the following, the VHDL conversion of the euclid design is discussed.

```
dp euclid(in  m_in, n_in : ns(16);
          out gcd        : ns(16)) {
  reg m, n               : ns(16);
  reg done               : ns(1);
  reg factor             : ns(4);

  sfg init    { m = m_in; n = n_in; factor = 0; done = 0; gcd = 0;
                $display("cycle=",$cycle," m=",m_in," n=",n_in); }
  sfg shiftm  { m = m >> 1; }
  sfg shiftn  { n = n >> 1; }
  sfg reduce  { m = (m >= n) ? m - n : m;
                n = (n >  m) ? n - m : n; }
```

```
  sfg shiftf  { factor = factor + 1; }
  sfg outidle { gcd = 0; done = ((m == 0) | (n == 0)); }
  sfg complete{ gcd = ((m > n) ? m : n) << factor;
                $display("cycle=", $cycle, " gcd=", gcd);}
}
fsm euclid_ctl(euclid) {
  initial s0;
  state s1, s2;
  @s0 (init) -> s1;
  @s1 if (done)               then (complete)          -> s2;
      else if ( m[0] &  n[0]) then (reduce, outidle)   -> s1;
      else if ( m[0] & ~n[0]) then (shiftn, outidle)   -> s1;
      else if (~m[0] &  n[0]) then (shiftm, outidle)   -> s1;
                              else (shiftn, shiftm,
                                    shiftf, outidle)   -> s1;
  @s2 (outidle) -> s2;
}

dp test_euclid(out m, n : ns(16)) {
  always {
    m = 2322;
    n = 654;
  }
}

dp euclid_sys {
   sig m, n, gcd : ns(16);
   use euclid(m, n, gcd);
   use test_euclid(m, n);
}

system S {
  euclid_sys;
}
```

We start by converting the design from GEZEL to VHDL.

```
> ../bin/fdlvhd.exe euclid.fdl
Pre-processing System ...
Output VHDL source ...
----------------------------
Generate file: euclid.vhd
Generate file: test_euclid.vhd
Generate file: euclid_sys.vhd
Generate file: system.vhd
Generate file: std_logic_arithext.vhd
```

For each datapath module in the system, a separate VHDL file is created. Some features of the VHDL code generator, including the structure and contents of the generated files, are as follows. Only partial listings will be shown, an ellipsis (...) is used were code was skipped.

The generated VHDL uses word-level semantics, similar to the GEZEL code. For this purpose, it makes use of the standard IEEE libraries for arithmetic. It also makes use of one extra library, `std_logic_arithext`, which contains a few support functions required for the generated code. This library is generated during the conversion process

```
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
library work;
use work.std_logic_arithext.all;
```

For each datapath module, a separate entity is created. The ports on this entity are the same as for the GEZEL module, but also include a clock pin and a reset pin. The euclid topcell, for example, looks as follows.

```
--datapath entity
entity euclid is
   port(
      m_in:in std_logic_vector(15 downto 0);
      n_in:in std_logic_vector(15 downto 0);
      gcd:out std_logic_vector(15 downto 0);
      RST : in std_logic;
      CLK : in std_logic
   );
end euclid;
```

The generated code uses the same net names as in the GEZEL code. During conversion, a check is done to verify that these names are valid VHDL identifiers. For example, GEZEL identifiers could be a VHDL keywords and not acceptable as identifiers in the generated VHDL code.

A hardwired datapath is modeled using two VHDL processes, one to evaluate combinational logic, and one to evaluate sequential logic. The sequential logic can use either low- or high-asserted reset signals, with synchronous or asynchronous reset. The reset style can be chosen in the fdlvhd command line. The initial values of the registers are zero, the same as in the GEZEL simulation. For each register, two VHDL signals are created. If `r0` is a GEZEL register, then the VHDL signal `r0` indicates the output of the register, and the VHDL signal `r0_signal` indicates the input of the register. Any other intermediate signals that are created by the VHDL code generator are called `sig_xx` with `xx` a decimal number. In case no registers are present in a hardwired datapath (such as in `test_euclid` in euclid.fdl), only a single VHDL process is created.

```
--signal declaration
architecture RTL of test_euclid is
signal m_int:std_logic_vector(15 downto 0);
signal n_int:std_logic_vector(15 downto 0);
begin


   --combinational logics
   dpCMB: process (m_int,n_int)
      begin
```

```
            m_int <= (others=>'0');
            n_int <= (others=>'0');
            m <= (others=>'0');
            n <= (others=>'0');

            m <= m_int;
            n <= n_int;
            m_int <= conv_std_logic_vector(2322,16);
            n_int <= conv_std_logic_vector(654,16);

        end process dpCMB;

end RTL;
```

The generated VHDL code reflects structural hierarchy in the same way as it appears in the GEZEL code. That is, enclosed modules will become enclosed components in VHDL.

An FSMD module, such as `euclid` in euclid.fdl, is translated to four VHDL processes: a combinational and sequential process for the datapath, and a combinational and sequential process for the controller. In combinational processes, choice is modeled using `case` statements. For the datapath, the entries of this `case` correspond to the different instructions that are possible. For the FSM controller, the entries of the `case` correspond to the different states the FSM can assume.

State assignment for the controller is symbolic. The generated code uses the same symbolic state names as the GEZEL code. The code generator will also create a symbolicinstruction set that the FSM will use to control the datapath. This symbolic instruction set is determined by the combinations of `sfg` that must be executed. For example, suppose that a datapath in GEZEL contains three `sfg` called `sfg0`, `sfg1`, `sfg2`. Considering all possible state transitions in the FSM, the GEZEL code generator finds that either `(sfg0)` executes or else both `(sfg1, sfg2)`. The resulting datapath in VHDL will decode two symbolic instructions, the first called `sfg0` and the second called `sfg1_sfg2`. The following snippet illustrates the VHDL layout of `euclid`.

```
--signal declaration
architecture RTL of euclid is
signal m     :std_logic_vector(15 downto 0);
signal m_wire:std_logic_vector(15 downto 0);
...
type STATE_TYPE is (s0,s1,s2);
signal STATE:STATE_TYPE;
begin
   --register updates
   dpREG: process (CLK,RST)
      begin
         if (RST = '1') then
            m <= (others=>'0');
            ...
         elsif CLK' event and CLK = '1' then
           m <= m_wire;
           ...
         end if;
```

```
      end process dpREG;
   --combinational logics
dpCMB: process (...)
     begin
         m_wire <= m;
         ...
         case cmd is
            when init =>
               ...
            when complete =>
               ...
            when others=>
         end case;
       end process dpCMB;

   --controller
   fsmREG: process (CLK,RST)
      ...
   end process fsmREG;

   --controller cmb
   fsmCMB: process (...)
     begin
      ...
   end process fsmCMB;
end RTL;
```

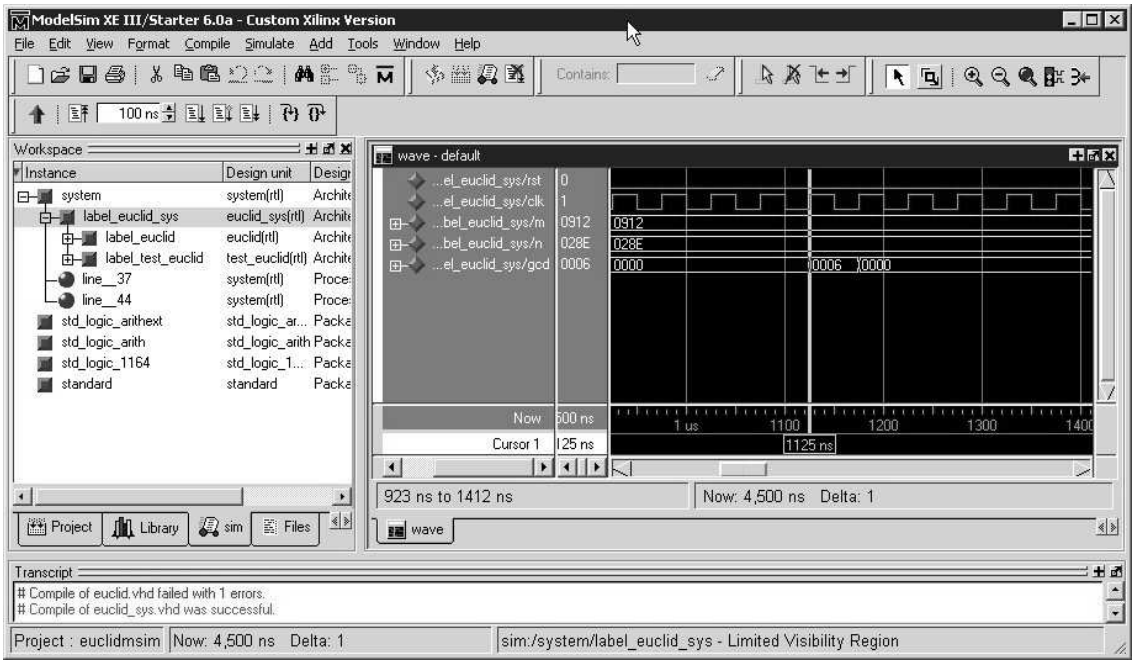
Finally, the system module (system.vhd) instantiates the toplevel module and connects a reset and a clock signal. If a GEZEL file simulates as a standalone module, then it can also simulate as a standalone VHDL file with the system module as top entity.

2.2 VHDL simulation in Modelsim

The VHDL code that is generated by GEZEL can be taken further to synthesis or simulation tools. One example is shown here: the use of the VHDL in conjunction with an the Modelsim VHDL simulator by ModelTech, Inc. The steps to follow in order to simulate your design in Modelsim are as follows.

1. Make sure your design works correctly in the GEZEL simulator (fdlsim) before you start using the code generator.
2. Generate VHDL code for the GEZEL design using fdlvhd.
3. Copy the generated files, as well as the support library `std_logic_arithext.vhd` to a separate directory, say mydirectory.
4. Start Modelsim, and create a new project that points to this mydirectory. (File - New - Project)
5. Assign all VHDL files, including the support library, to this project. (Project - Add File to Project)

6. Compile the VHDL files while respecting the structural hierarchy. (Design - Compile). This means that you have to compile lower-level entities first. Start with the support library, then the leaf modules, then the toplevel modules.
7. Load the design and select the toplevel module, system. (Design - Load Design).
8. Assign signals to the wave window and simulate.



2.3 Stimuli Directives

When developing a big design it is sometimes required simulate only part of the generated VHDL. For example, when developing a cryptographic coprocessor for an ARM, then we might want to simulate the VHDL for the coprocessor without including a VHDL model of an ARM processor, or even without running a cosimulation at the VHDL level. In such a case, you could create a VHDL testbench that only includes the block you are interested in. You would also need to have a set of testvectors or stimuli for the VHDL simulation of this block.

These stimuli can be created using stimuli directives. Stimuli directives allow you to record all I/O of a GEZEL module during GEZEL simulation, and store these into a file. Later, during VHDL simulation, you can replay these I/O activities on a stand-alone VHDL model of the same module. Thus you can run a system cosimulation of an ARM instruction-set simulator and a GEZEL coprocessor, and later rerun the same simulation on a VHDL model of the GEZEL coprocessor without including the ARM.

The $trace stimuli directive can record the value of any signal in the simulation into a file. Each clock cycle, the value of that signal is recorded and stored into a file. We will give an example of the use of $trace, by creating a set of testvectors for the euclid module.

```
dp euclid(in  m_in, n_in : ns(16);
          out gcd        : ns(16)) {
  reg m, n               : ns(16);
  reg done               : ns(1);
  reg factor             : ns(4);

  $trace(m, "m.txt");
  $trace(n, "n.txt");
  ...
}
```

When the simulation executes, two extra files will be generated. One will contain the value of m (register output) at each clock cycle, while the other will contain the value of n. For example, the first few lines of m.txt are:

```
0000000000000000
0000100100010010
0000010010001001
0000001101000010
0000000110100001
0000000001011010
0000000000101101
0000000000101101
0000000000101101
```

These stimuli files can be used in a VHDL testbench. The example process below uses the text IO functions in VHDL to read the files m.txt and n.txt, one line at each clock cycle. Such a process can be used in a customized version of the system testbench.

Note that, while the stimuli directive helps in collecting test vectors from your GEZEL simulation, the design of the VHDL testbench code that uses these files is not yet automated.

```
-- Here is a process that reads out data from m_in.txt
-- and n_in.txt, and drives these onto signals
-- To use the text IO functions in VHDL
-- use IEEE.std_logic_textio.all;
-- use std.textio.all;

process
  file f_m_in : text is in "m_in.txt";
  file f_n_in : text is in "n_in.txt";
  variable l_m_in : line;
  variable l_n_in : line;
  variable v_m_in : std_logic_vector(16 downto 0);
  variable v_n_in : std_logic_vector(16 downto 0);
  begin
  wait until RST'event and RST = '1';
  loop
    if (not(endfile(f_d_in))) then
      readline(f_m_in, l_m_in);
      read(l_m_in, v_m_in);
```

```
    else
      assert false
        report "End of input on din.txt"
        severity warning;
    end if;
    if (not(endfile(f_n_in))) then
      readline(f_n_in, l_n_in);
      read(l_n_in, v_n_in);
    else
      assert false
        report "End of input on n_in.txt"
        severity warning;
    end if;
    sig_0 <= v_m_in;
    sig_2 <= v_n_in;
    wait until CLK'event and CLK = '1';
  end loop;
end process;
```