COLUMBIA UNIVERSITY
SCHOOL OF ENGINEERING AND APPLIED SCIENCE

EHDL

Easy Hardware Description Language

COMS W4115 Programming Languages and Translators Project Proposal

Paolo Mantovani (pm2613)
Mashooq Muhaimen (mm3858)
Neil Deshpande (nad2135)
Kaushik Kaul (kk2746)

September 28, 2011

1. Motivation

Fueled by advances in integrated circuits technology, hardware systems have grown increasingly complex over the past several decades. This rise in complexity makes the hardware designer's job difficult. Our work aims to alleviate some of the difficulty by introducing an HDL (Hardware Description Language) that is easy to learn and code in. We feel that traditional HDLs, such as VHDL, are unnecessarily verbose and more low-level than they need to be. By developing a language that is succinct and at the same time defers more implementation details to the compiler, we hope to increase the productivity of hardware designers.

2. Language Overview

We propose a language with a C-like syntax. The user of our language is assumed to have rudimentary knowledge of digital logic. Experience with an imperative programming language such as C will also be beneficial. We call our language EHDL (Easy Hardware Description Language).The EHDL compiler will translate the EHDL code to VHDL. Each EHDL statement will be translated into a combinational block. Functions will be mapped to components (entity and architecture, signals list, component declaration and instantiation). The compiler will automatically add to the logic component the inputs clock and reset along with an output valid flag. The output valid flag will be used to indicate if the output values are valid/stable. The flag will always equal '1' when the function describes a combinational block (the synthesizer will remove the clock, reset and the valid flag in this case). In the case of a multi-cycle unit, the user can read the flag to handle the handshake between logic components (if <function_name.valid> …). EHDL functions will be able to return more than one value.

A novel feature of our language will be a special keyword (POS) that gives the user the ability to place positive edge triggered registers that break the combinational path. This can be used to pipeline/speed up the design. Re-timing will require only moving the POS statement within the code, without having to rewrite any of the functional blocks.

Our language will handle loops. If the same variable is modified in every loop cycle, a register will be automatically inferred to allow feedback in the circuit. Due to the fact that a synthesizable register must have a constant reset value, when the arguments of a function are used as loop variables (see the gcd example is Section 4), the input sensing of the associated logic block will be turned off until the loop finishes its execution. Once the loop is done executing (i.e. the loop condition ceases to hold), the output valid flag will be used to indicate that the output ports now contain valid values. At the next clock cycle, the loop will restart and load the new input values.

Our language will also support VHDL like “for generate” loops.

3. Keywords

EHDL will include the following keywords (this list is not exhaustive):

- CLOCK($period) identifier is the key word to create a clock signal.

- RESET_SYNC($clock, HI/LO) identifier or RESET_ASYNC(HI/LO) identifier are the key words to create a reset signal.

- POS($clock; $reset; $enable) generates a register for each temporary variable introduced after the previous keyword POS or from the beginning of a logic block (function). After applying POS, the identifiers for variables will not change, however the "old" variables (inputs of the registers) will not be accessible. An exception can be introduced to manage the situation of an asynchronous control signal (such as clock enable). In this case the variable must be declared as ASYNC.

- INPUT identifier = {testbench input pattern} defines an input for the top entity and can be used to generate the testbench.

- OUTPUT identifier = {testbench input pattern} defines an output for the top entity and can be used to generate the testbench.

- TOP identifier = {blocks list} defines the top entity of the design. The block list can contain simple expressions (combinational logic), variables declarations (internal signals), POS (registers), function call (components instantiation).

- Common logic and arithmetic operators

4. Example

This example illustrates how a classic algorithm for computing the greatest common divider (GCD) will be implemented in our language. Figure 1 shows the algorithm written in the high level programming language C, so that the reader may become familiar with the algorithm. Figure 2 shows the EHDL code for the GCD algorithm. Appendix A shows the same algorithm implemented in VHDL, this is the version we intend to produce with our compiler when it gets the EHDL code as input. A cursory inspection of Figure 2 and Appendix A should convince the reader of the ease and conciseness of our proposed language.

```
int gcd(int a, int b)
{
        while (a != b) {
                if (a > b) a -= b;
                else b -= a;
        }
        return a;
}
```

Figure 1: An implementation of the GCD algorithm in C [1]

```
CLOCK(100 ns) clk;
RESET_ASYNC rst;

INPUT int a = 37, int b= 55;
OUTPUT int c;

int gcd (int a, int b) {
   while ( a!=b ){
            if (a > b) a = a – b;
            else  b = b – a;
   }
return b;
}

TOP gcd_wrap { //inputs and outputs are already declared.
   c = gcd(a,b);
   POS(clk, rst, gcd.valid);
   return;   //assignment of the main output. c is seen at the output of a register!
}
```

Figure 2: An implementation of the GCD algorithm in EHDL.

5. References

[1] Edwards, Stephen. COMS W4115 Programming Languages and Translators Lecture. Retrieved on September 27, 2011 from
http://www.cs.columbia.edu/~sedwards/classes/2011/w4115-fall/ocaml.pdf

6. Appendix A

An implementation of the GCD algorithm in VHDL:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity gcd is

  port (
    clk     : in  std_logic;
    rst     : in  std_logic;
    gcd_a   : in  std_logic_vector(31 downto 0);
    gcd_b   : in  std_logic_vector(31 downto 0);
    gcd_out : out std_logic_vector(31 downto 0);
    gcd_valid : out std_logic);

end gcd;

architecture beh of gcd is

  --Combinational logic output signals
  signal while_l7, if_l8 : std_logic;
  signal sub_l8, sub_l9 : std_logic_vector(31 downto 0);

  --Automatically generated registers
  signal a_reg, b_reg : std_logic_vector(31 downto 0);

  --Automatically generated internal input signals
  signal a : std_logic_vector(31 downto 0);
  signal b : std_logic_vector(31 downto 0);

  --Automatically generated internal output signals
  signal gcd_out_int : std_logic_vector(31 downto 0);
  signal gcd_valid_int : std_logic;

begin  -- beh

  --input signals assignment
  a <= gcd_a;
  b <= gcd_b;

  --Automatically generated register with input sensig when the loop completes
  process (clk, rst)
  begin  -- process
    if rst = '0' then                   -- asynchronous reset (active low)
      a_reg <= (others => '0');
      b_reg <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
```

```
      if gcd_valid_int = '1' then        -- input sensing
                                         -- must happen after reset!
        a_reg <= a;
        b_reg <= b;
      elsif if_l8 = '1' then
        a_reg <= sub_l8;
      else
        b_reg <= sub_l9;
      end if;
    end if;
  end process;

  --Combinational logic
  if_l8 <= '1' when a_reg > b_reg else '0';
  while_l7 <= '0' when a_reg = b_reg else '1';
  sub_l8 <= a_reg - b_reg;
  sub_l9 <= b_reg - a_reg;

  --Output and Valid flag assignment
  gcd_valid_int <= not while_l7;
  gcd_out_int <= b_reg;

  gcd_valid <= gcd_valid_int;
  gcd_out <= gcd_out_int;

end beh;


library ieee;
use ieee.std_logic_1164.all;

entity gcd_wrap is

  port (
    clk     : in  std_logic;
    rst     : in  std_logic;
    a_in    : in  std_logic_vector(31 downto 0);
    b_in    : in  std_logic_vector(31 downto 0);
    c_out   : out std_logic_vector(31 downto 0));

end gcd_wrap;

architecture beh of gcd_wrap is

  signal a         : std_logic_vector(31 downto 0);
  signal b         : std_logic_vector(31 downto 0);
  signal c         : std_logic_vector(31 downto 0);

  signal gcd_a     : std_logic_vector(31 downto 0);
  signal gcd_b     : std_logic_vector(31 downto 0);
  signal gcd_out   : std_logic_vector(31 downto 0);
  signal gcd_valid : std_logic;

  component gcd
    port (
      clk       : in  std_logic;
      rst       : in  std_logic;
      gcd_a     : in  std_logic_vector(31 downto 0);
      gcd_b     : in  std_logic_vector(31 downto 0);
      gcd_out   : out std_logic_vector(31 downto 0);
      gcd_valid : out std_logic);
  end component;

  signal c_reg : std_logic_vector(31 downto 0);

begin  -- beh

  --Explicit main input assignment
  a <= a_in;
  b <= b_in;

  --Explicit component signals assignment
  gcd_a <= a;
  gcd_b <= b;
  c <= gcd_out;

  --Component intance
  gcd_1: gcd
    port map (
      clk       => clk,
      rst       => rst,
      gcd_a     => gcd_a,
      gcd_b     => gcd_b,
      gcd_out   => gcd_out,
      gcd_valid => gcd_valid);


  --POS statement
  process (clk, rst)
  begin  -- process
    if rst = '0' then                   -- asynchronous reset (active low)
      c_reg <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if gcd_valid = '1' then
        c_reg <= c;
      end if;
    end if;
  end process;

  c_out <= c_reg;

end beh;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.std_logic_arith.all;

entity gcd_wrap_tb is

end gcd_wrap_tb;

architecture tb of gcd_wrap_tb is


  signal a_in,b_in,c_out : std_logic_vector(31 downto 0);
  signal clk, rst : std_logic := '0';
  constant h_period : time := 50 ns;

  component gcd_wrap
    port (
      clk   : in  std_logic;
      rst   : in  std_logic;
      a_in  : in  std_logic_vector(31 downto 0);
      b_in  : in  std_logic_vector(31 downto 0);
      c_out : out std_logic_vector(31 downto 0));
  end component;

begin  -- tb

  --CLOCK and RESET
  clk <= not clk after h_period;
  rst <= '1' after (4*h_period);

  --INPUTS
  a_in <= conv_std_logic_vector(37,32);
  b_in <= conv_std_logic_vector(55,32);

  gcd_wrap_1: gcd_wrap
    port map (
      clk   => clk,
      rst   => rst,
      a_in  => a_in,
      b_in  => b_in,
      c_out => c_out);


end tb;
```