Microcontroller in eSim - Documentation

1. Overview –

Currently, ATTINY 85 has been targeted to be implemented in eSim. The component outline including in / out ports (a completed skeleton of the microcontroller) is developed in a main VHDL file – “attiny85_nghdl.vhdl” and the microcontroller is logically simulated by a C file – “tiny85_c.c”.
Please note that this isn’t a cycle accurate simulation but a logically accurate one.

2. Process –

In order to simulate a microcontroller, we need to simulate a C file and a VHDL file together during runtime. And to facilitate communication of variables and functions between the two files, we first need to make object files of them individually and link them. This step is done before simulating the project. The following steps need to be followed before simulation.

Please note that these steps have to be performed only once when simulating a project.

2.1 Before simulation –

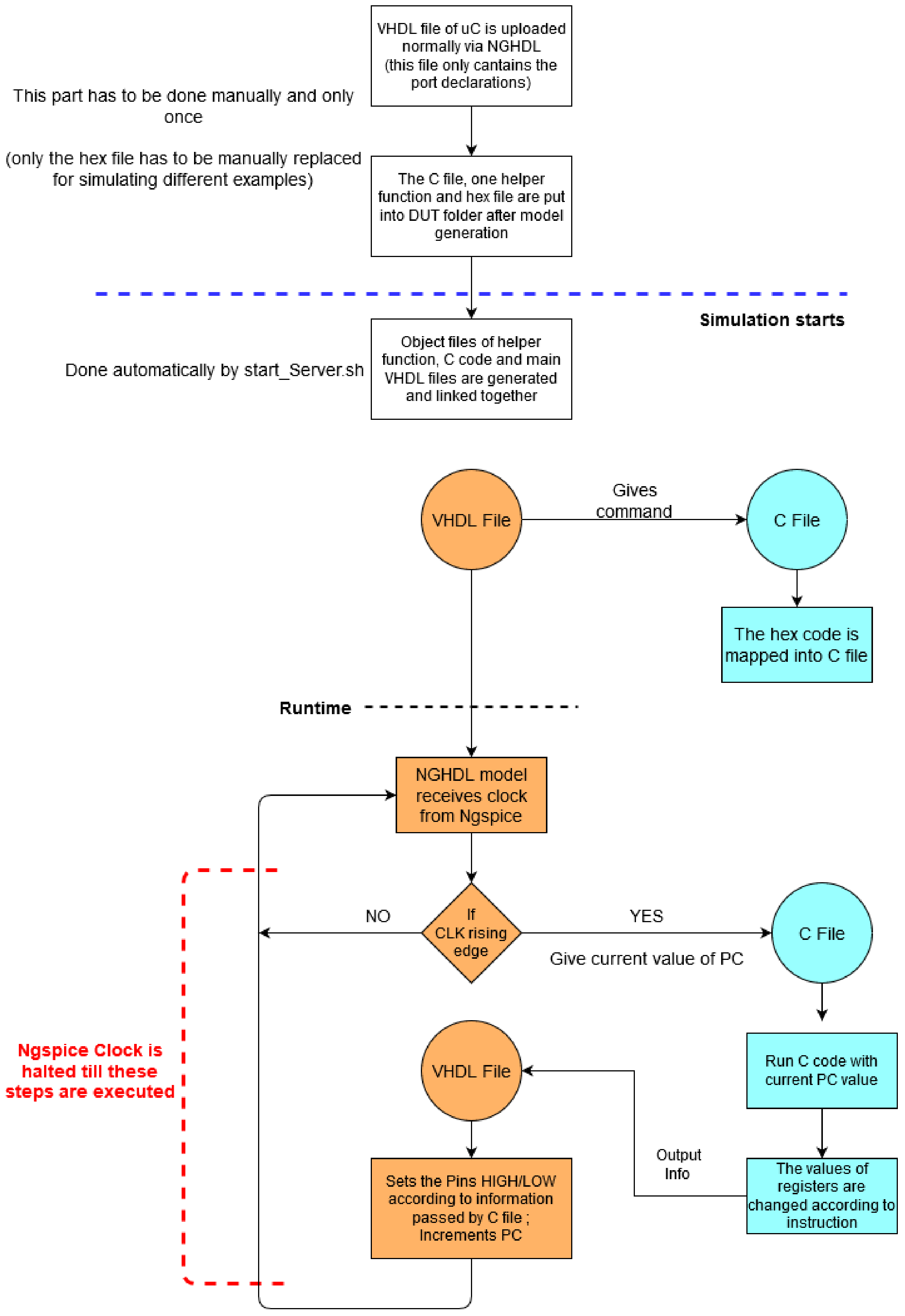
i) The microcontroller model needs to be created in NGHDL via the VHDL file for microcontroller (“attiny85_nghdl.vhdl” here).
ii) Then the helper ghdl function, modified start_server.sh file, C file for microcontroller and hex file for the example have to be moved into the DUTghdl folder of the microcontroller.

2.2 When simulation starts –

i) The object file for microcontroller C file is created.
ii) The object files for helper function and main VHDL file are created.
iii) The object files of C file and main VHDL file are linked.
iv) The hex file is mapped into a structure in C file.

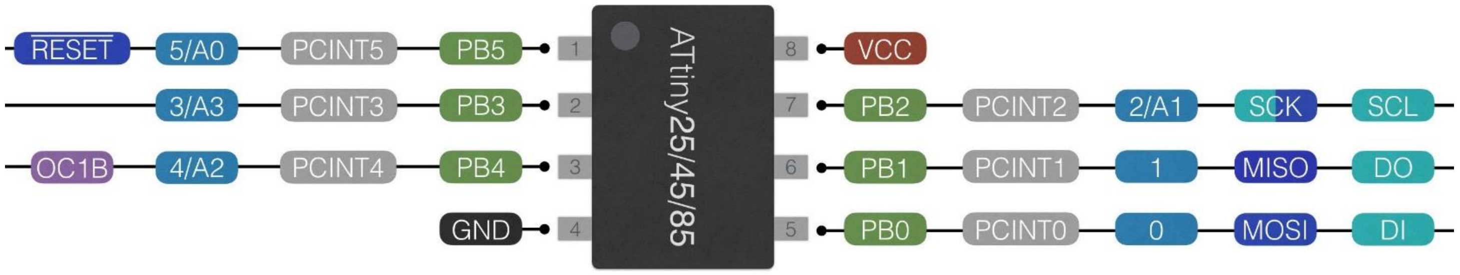
(Steps 1-4 happen only once after simulation starts)

v) Following steps are repeated till simulation ends –
vi) If CLK is rising edge, C file is given the current value of PC.
vii) C file runs once, setting required I/O pins HIGH or LOW depending on the instruction and changing the PC value as per the current instruction.


VHDL file of uC is uploaded normally via NGHDL (this file only cantains the port declarations)
This part has to be done manually and only once
(only the hex file has to be manually replaced for simulating different examples)
The C file, one helper function and hex file are put into DUT folder after model generation
Simulation starts
Done automatically by start_Server.sh
Object files of helper function, C code and main VHDL files are generated and linked together
VHDL File
Gives command
C File
The hex code is mapped into C file
Runtime
NGHDL model receives clock from Ngspice
NO
If CLK rising edge
YES
C File
Give current value of PC
Ngspice Clock is halted till these steps are executed
VHDL File
Run C code with current PC value
Output Info
Sets the Pins HIGH/LOW according to information passed by C file ; Increments PC
The values of registers are changed according to instruction

3. Detailed explanation –

3.1 ATTINY 85 pin configuration –



- ATTINY 85 has 1 PORT – PORTB which has 6 I/O pins – PB0 to PB5.
- Pins 2, 3 and 7 (PB3, PB4 and PB2) are Analog I/O pins.
- Pins 5 and 6 (PB0 and PB1) are digital I/O pins.
- Pins 8 and 4 are for ground and VCC respectively.
- Pin 1 is for resetting the microcontroller.
- ATTINY series of µC have in-built oscillator so no external clock has to be connected to the IC physically.

3.2 ATTINY 85 VHDL Code –

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

library work;
use work.ghdl_access.all;

entity attiny_85_nghdl is
port(VCC : in std_logic;
     GND : in std_logic;
     clk : in std_logic;
     PB0 : out std_logic_vector(0 downto 0);
     PB1 : out std_logic_vector(0 downto 0);
     PB2 : out std_logic_vector(0 downto 0);
     PB3 : out std_logic_vector(0 downto 0);
     PB4 : out std_logic_vector(0 downto 0);
     PB5 : out std_logic_vector(0 downto 0));
end attiny_85_nghdl;

architecture bhv of attiny_85_nghdl is

signal pc_vhdl: integer := 0;

begin
MapToRam(1);
      process(clk)
      begin
            if(rising_edge(clk) and VCC = '1' and GND = '0') then
                  output(pc_vhdl);
                  PB0 <= std_logic_vector(to_unsigned(var0.all, PB0'length));
                  PB1 <= std_logic_vector(to_unsigned(var1.all, PB1'length));
                  PB2 <= std_logic_vector(to_unsigned(var2.all, PB2'length));
                  PB3 <= std_logic_vector(to_unsigned(var3.all, PB3'length));
                  PB4 <= std_logic_vector(to_unsigned(var4.all, PB4'length));
                  PB5 <= std_logic_vector(to_unsigned(var5.all, PB5'length));
                  pc_vhdl <= pc_vhdl + var6.all;
            elsif(VCC = '0') then
                  PB0 <= "0";
                  PB1 <= "0";
                  PB2 <= "0";
                  PB3 <= "0";
                  PB4 <= "0";
                  PB5 <= "0";
            end if;
      end process;
end bhv;
```

- In the port declaration, there are 6 output ports from PB0 to PB1 and three input ports VCC, GND and clock. The Attiny85 microcontroller has in-built oscillator in real life but for simulation, we are going to provide external Clock through the "clock" port.
- The functions "***MapToRam***" and "***output***" are linked to the C file. The functions shared with C code are showed in **red**. Code for "***MapToRam***" is given below -

```
void MapToRam(int flag)
{
        int i=0,filesize;
        SetRam(0,size,0x0);
        if(flag==1)
        {
                FILE *fptr;
                unsigned char c;
                fptr = fopen("hex.txt", "r");

                fseek(fptr, 0L, SEEK_END);
             filesize = ftell(fptr);

                rewind(fptr);
                c = fgetc(fptr);
             while (c != EOF && i<filesize)
             {
                // to skip newline character in file
                if(c == '\n')
                        c = fgetc(fptr);
                // to skip ":" character in file
                else if(c == ':')
                        c = fgetc(fptr);
                else if(c >= 127)
                        c = 0;

                // the ascii eqivalent of char c is converted to hex
                if(c>=48 && c<=57)
                        c-=48;
                else if(c>=65 && c<=70)
                                c-=55;
                else if(c>=97 && c<=102)
                        c-=87;
                  ram[i].data = c;
                  i++;
                  c = fgetc(fptr);
             }
             fclose(fptr);
        }
        //Clearing Status Register at the beginning of simulation
        for(i=0;i<8;i++)
                SREG[i].data = 0;

        PrintRam(0,filesize+5);
}
```

- "***MapToRam***" is called only once at the beginning of the simulation. It copies the contents of hex file given by user into the C file for the entire duration of the simulation. The function also clears the Status Register SREG at the beginning of the simulation once.

```
struct memory              //Structure to store RAM and other registers
{
        unsigned char data;
}ram[size],reg[32],SREG[8],PORTB,DDRB;
```

- The above structure in C is used to store RAM and other register's data like SREG, PORTB, etc.
- "***output***" is called for each clock cycle (on the rising edge). It checks the instruction which is present at the location pointed by PC (which is given to C file by VHDL) and computes the values of output pins and registers accordingly.

```
void output(int pc_vhdl)
{
        printf("\nPC: %d\n",pc_vhdl);
        Compute(pc_vhdl);
        PrintSREG();
}
```

- The "***output***" function sends the value of PC given by VHDL file to a "Compute" function which decodes and does the computation. The "***output***" function also prints the value of Status Register after every instruction is executed for debugging purpose.
- After computation of the current instruction, C file shares the pins that have to be set HIGH or LOW to VHDL file. These shared variables are in **blue** in VHDL code for reference.
- After setting of pins, C file also shares the amount by which PC has to be incremented / decremented for fetching next instruction. The VHDL file makes the necessary changes in PC and returns it to C file in the next clock and this cycle goes on till the end of simulation.

3.3 C Code –

The entire C code is too big to be displayed here, so only the main "***Compute***" function will be displayed and explained.

```
void Compute(char pc)
{
      int i,j,t;
      unsigned char b1=ram[pc+0x2].data, b2=ram[pc+0x3].data,
      b3=ram[pc].data, b4=ram[pc+0x1].data;
      printf("instruction:%X%X%X%X\n",b1,b2,b3,b4);

      if(b1==0x0 && b2>=12 && b2<=15)                       //ADD
      {
            printf("ADD instruction decoded\n");

            // Do some computation

            chg_Pc = 4;
      }

/****************************************************************************/

      else if(b1==0x1 && b2>=12 && b2<=15)                  //ADC
      {
            printf("ADC instruction decoded\n");
            int a=reg[b3+16].data,b=reg[b4+16].data;

            // Do some computation

            chg_Pc = 4;
      }

/****************************************************************************/

      else if(b1==0x1 && b2 >= 8 && b2 <= 11)               //SUB
      {
            printf("SUB instruction decoded\n");

            // Do some computation

            chg_Pc = 4;
      }

// And so on...
```

- The “***Compute***” function first scans the current instruction present in RAM location given by the PC.
- It then compares the opcode of the current instruction in the code by if else ladder.
- Once the instruction is decoded, the function performs necessary computation and gives the value by which PC has to be incremented / decremented for next clock cycle to VHDL code (the chg_Pc variable is shared between C and VHDL files).

3.4 GHDL helper function –

In order to facilitate the sharing of functions and variables between C and VHDL file, a helper VHDL function “ghdl_access.vhdl” is developed and explained below –

```
package ghdl_access is
  -- Defines a pointer to an integer:
type int_access is access integer;


function get_ptr0 return int_access;
      attribute foreign of get_ptr0 :
            function is "VHPIDIRECT get_ptr0";

.........

function get_ptr6 return int_access;
      attribute foreign of get_ptr6 :
            function is "VHPIDIRECT get_ptr6";

  -- declaration of functions in C
procedure output(f : integer);
      attribute foreign of output :
            procedure is "VHPIDIRECT output";

procedure MapToRam(f : integer);
      attribute foreign of MapToRam :
            procedure is "VHPIDIRECT MapToRam";


  -- create variables aliased to the variable in C
      shared variable var0 : int_access := get_ptr0;
      shared variable var1 : int_access := get_ptr1;
      shared variable var2 : int_access := get_ptr2;
      shared variable var3 : int_access := get_ptr3;
      shared variable var4 : int_access := get_ptr4;
      shared variable var5 : int_access := get_ptr5;
      shared variable var6 : int_access := get_ptr6;

end ghdl_access;

package body ghdl_access is
      function get_ptr0 return int_access is
      begin
            assert false report "VHPI" severity failure;
      end get_ptr0;

      ...
            assert false report "VHPI" severity failure;
      end get_ptr6;

      procedure output(f : integer) is
      begin
            assert false report "VHPI" severity failure;
      end output;

      procedure MapToRam(f : integer) is
      begin
            assert false report "VHPI" severity failure;
      end MapToRam;
```

- The parts written in **red** are for sharing functions and ones written in **blue** are for sharing variables.
- The function given below retrieves the value of variable given by get_ptr0 function in C file. Since there are 6 shared variable, the function is repeated 6 times on the helper file.

```
function get_ptr0 return int_access;
      attribute foreign of get_ptr0 :
            function is "VHPIDIRECT get_ptr0";
```

- The procedure given below shares the function specified in the procedure from C file to VHDL i.e. when the function specified in procedure is called in VHDL, it is sent to be processed in C file.

```
procedure MapToRam(f : integer);
      attribute foreign of MapToRam :
            procedure is "VHPIDIRECT MapToRam";
```

3.5 start_server file –

In order to run the simulation, object files of VHDL and C codes need to be generated and linked. The start_server.sh file takes care of that normally –

```
#!/bin/bash

###This server run ghdl testebench for infinite time till ngspice send END signal to
stop it

cd /home/fossee/ngspice-nghdl/src/xspice/icm/ghdl/attiny_85_nghdl/DUTghdl/
chmod 775 sock_pkg_create.sh &&
./sock_pkg_create.sh $1 $2 &&
ghdl -a sock_pkg.vhdl &&
ghdl -a attiny_85_nghdl_tb.vhdl  &&
ghdl -e -Wl,ghdlserver.o attiny_85_nghdl_tb &&
./attiny_85_nghdl_tb
```

But in order to co-simulate C and VHDL files, we need to do two things –

- Generate the object files of C, main VHDL and helper VHDL codes
- Link the object file of C and main VHDL codes

To do this, following changes were made in start_server.sh file (changes made in **red**) –

```
#!/bin/bash

###This server run ghdl testebench for infinite time till ngspice send END signal to
stop it

cd /home/fossee/ngspice-nghdl/src/xspice/icm/ghdl/attiny_85_nghdl/DUTghdl/
chmod 775 sock_pkg_create.sh &&
./sock_pkg_create.sh $1 $2 &&
ghdl -a sock_pkg.vhdl &&

### The following lines (till line 23) are added by Ashutosh Jha
### Date - 3/3/2020

gcc -c tiny85_c.c -o tiny85_c.o &&
# Compiles and generates object file of microcontroller C code

ghdl -a ghdl_access.vhdl attiny_85_nghdl.vhdl &&
# Compiles and generates object files of VHDL code of helper function and the main
model respectively

mv attiny_85_nghdl.o attiny_85_nghdl1.o &&
ld -r -o attiny_85_nghdl.o tiny85_c.o attiny_85_nghdl1.o &&
rm attiny_85_nghdl1.o &&
# The object files of main VHDL and microcontroller C code need to be linked.
# The above three commands do that

ghdl -a attiny_85_nghdl_tb.vhdl  &&
ghdl -e -Wl,ghdlserver.o attiny_85_nghdl_tb &&
./attiny_85_nghdl_tb
```