Documentation

Microcontroller Implementation in eSim

by

Ashutosh Jha

March 2020

Contents

List of Figures

1 INTRODUCTION

A microcontroller is a small and low-cost microcomputer, which is designed to perform specific tasks of embedded systems like displaying information on a LCD screen, receiving remote signals, etc. Nowadays, approximately all electronic circuits (especially embedded and closed loop ones) employ micrcocontroller(s) to perform logical operations in order to increase their overall safety, flexibility, accuracy, ease of design and implementation.

Due to wide spread usage of microcontrollers, there is also a big demand for simulation platforms which can simulate a microcontroller along with all the analog circuitry around it. This allows users to design, modify and test the system before practically implementing it. Currently, there are two broad categories of such platforms - one is an assembler which is used to understand in depth about what is going inside the microcontroller e.g. - Keil, IAR, etc. and the other are EDA software which simulates the output of a circuit when microcontroller interacts with an analog circuitry around it e.g. - Proteus Design Suite [12], PSPICE [13], etc. These are commercial software and acquiring their license is very expensive and are suited for big organizations.

The individual users such as hobbyists, students and even small scale industries and educational institutions often find it very difficult to acquire licenses and use the commercially available products. Apart from from these commercial solutions, there is no open source software which can provide such functionality (some solutions are available but are either in development stage, not accessible to common users by being too platform specific or not easy to operate). Therefore there is a big need to develop a platform/modify an existing one that can simulate microcontrollers and provide accurate results along with being easy to use.

eSim [5] is a free/libre and open source EDA tool for circuit design, simulation, analysis and PCB design. It is an integrated tool built using free/libre and open source software such as KiCad, Ngspice, NGHDL and GHDL. eSim is released under GPL. Because of this, it has the necessary packages and tools to integrate microcontroller into it.

eSim offers similar capabilities and ease of use as any equivalent proprietary software for schematic creation, simulation and PCB design, without having to pay a huge amount of money to procure licenses. Hence it can be an affordable alternative to educational institutions and SMEs. It can serve as an alternative to commercially available/licensed software tools like OrCAD, Xpedition and HSPICE.

2 LITERATURE SURVEY

There are few commercially available software (both assemblers and EDA tools) and a couple of open sources software which are discussed in brief.

2.1 COMMERCIALLY AVAILABLE SOLUTIONS

These software are highly accurate and of industry standards but the major drawback associated with them is that the cost procuring a license is very high and is feasible only for big organizations and institutions.

2.1.1 PROTEUS DESIGN SUITE

The Proteus Design Suite is a proprietary software tool suite used primarily for electronic design automation. The software is used mainly by electronic design engineers and technicians to create schematics and electronic prints for manufacturing printed circuit boards. It was developed in Yorkshire, England by Labcenter Electronics Ltd and is available in English, French, Spanish and Chinese languages.

Product Modules -

- **Schematic Capture -** The Proteus Design Suite is a Windows application for schematic capture, simulation, and PCB (Printed Circuit Board) layout design. It can be purchased in many configurations, depending on the size of designs being produced and the requirements for microcontroller simulation. All PCB Design products include an autorouter and basic mixed mode SPICE simulation capabilities.

- **Microcontroller Simulation -** The micro-controller simulation in Proteus works by applying either a hex file or a debug file to the microcontroller part on the schematic. It is then co-simulated along with any analog and digital electronics connected to it. This enables its use in a broad spectrum of project prototyping in areas such as motor control, temperature control and user interface design. It also finds use in the general hobbyist community and, since no hardware is required, is convenient to use as a training or teaching tool. Support is available for co-simulation of -

    1. Microchip Technologies PIC10, PIC12 PIC16,PIC18,PIC24,dsPIC33 Microcontrollers.
    2. Atmel AVR (and Arduino), 8051 and ARM Cortex-M3 Microcontrollers.
    3. NXP 8051, ARM7, ARM Cortex-M0 and ARM Cortex-M3 Microcontrollers.
    4. Texas Instruments MSP430, PICCOLO DSP and ARM Cortex-M3 Microcontrollers.
    5. Parallax Basic Stamp, Freescale HC11, 8086 Microcontrollers.

- **PCB Design -** The PCB Layout module is automatically given connectivity information in the form of a netlist from the schematic capture module. It applies this information, together with the user specified design rules and various design automation tools, to assist with error free board design. PCB's of up to 16 copper layers can be produced with design size limited by product configuration.

- **3D Verification -** The 3D Viewer module allows the board under development to be viewed in 3D together with a semi-transparent height plane that represents the boards enclosure. STEP output can then be used to transfer to mechanical CAD software such as Solidworks [14] or Autodesk [2] for accurate mounting and positioning of the board.

2.2 OPEN SOURCE SOLUTIONS

Currently there is no open source software providing complete functionality like the commercially available software. Some solutions are available but are either in development stage, not accessible to common users by being too platform specific or not easy to operate.

2.2.1 MCUSIM

By far, MCUSim [10] is the best solution available that can simulate microcontrollers. MCUSim is a digital simulator and NGSpice library with microcontrollers. It is created to assist in circuit simulation, firmware debugging, testing and signal tracing.At the 8-bit AVR (RISC) microcontrollers are aimed at the moment.

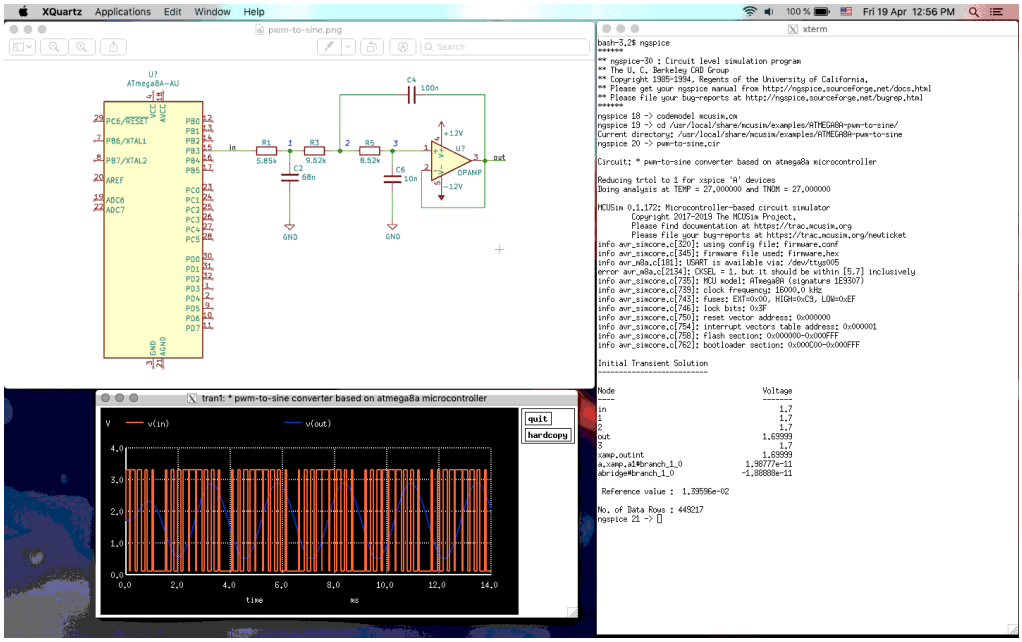


Figure 1: *MCUSim PWM to Sine simulation output*

3 PROBLEM STATEMENT

To design a framework in eSim utilizing its existing modules (NGHDL) and implementing any new modules if required to implement a commonly used microcontroller that a user can program either via uploading a C code or a HEX code.

The framework should -

- Be as user-friendly as possible
- Easy to scale up for implementing other microcontrollers
- Ideally not contain any dependencies that might cause problem while porting
- Be as accurate as possible, while not being very resource intensive

For implementing the above points, several approaches were identified and are discussed below -

3.1 APPROACH I

Designing all the internal components of micrcocontroller in VHDL and linking them together to make a highly accurate but complex and very resource intensive model of microcontroller.

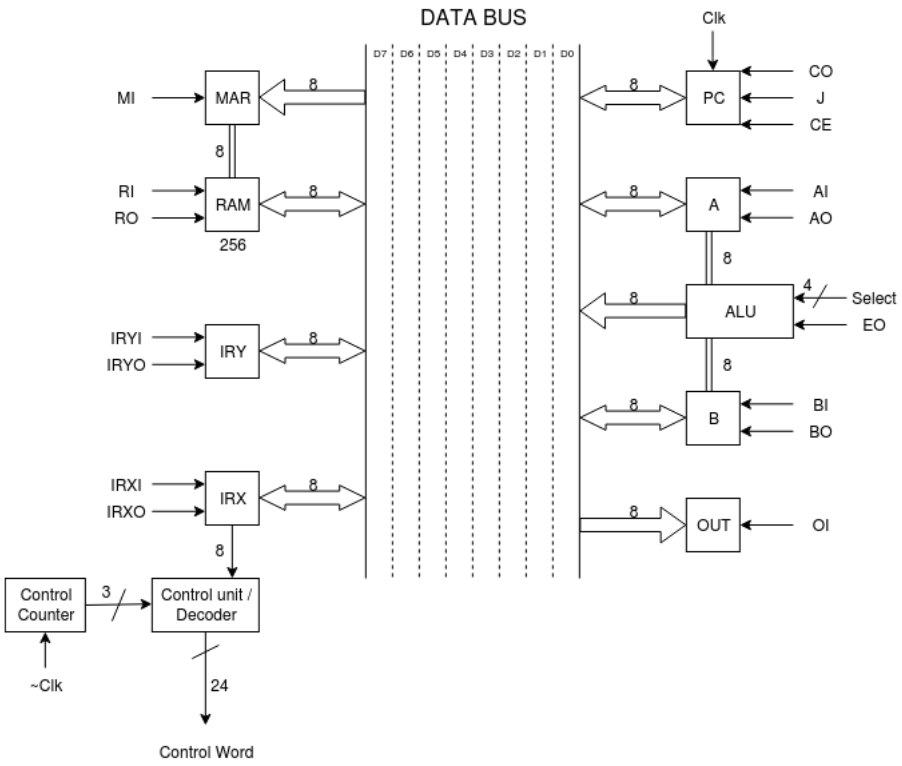


Figure 2: *Proposed architecture of mircontroller for Approach I*

The above architecture was designed for testing Approach I. Each of the block would have to written in VHDL and then linked together to create a basic 8-bit microcontroller.

PROS
- Highly accurate simulation - since each block mimics the internal registers of an actual microcontroller, the simulation results would identical to a physical microcontroller.

CONS
- Since each block has to be written separately in VHDL, complexity is very high.
- Since each block is simulated independently, it is very resource intensive.
- All the internal blocks of a micrcontroller are not shared by the manufacturer. This renders accurate depiction of a wide variety of microcontroller impossible.
- The proposed architecture does not represent any actual microcontroller which is in use by the masses - it is just a hypothetical model. To implement even the simplest real world microcontroller (6502, 8051, etc.) the number of internal blocks will increase approximately five fold (60 block). Therefore scaling it will be very difficult.

3.2 APPROACH II

Designing the layout of the microcontroller (pin configurations) in VHDL and simulating its internal processing in a C program and co-simulating them (by linking them together for exchange of variables and functions before simulation begins).

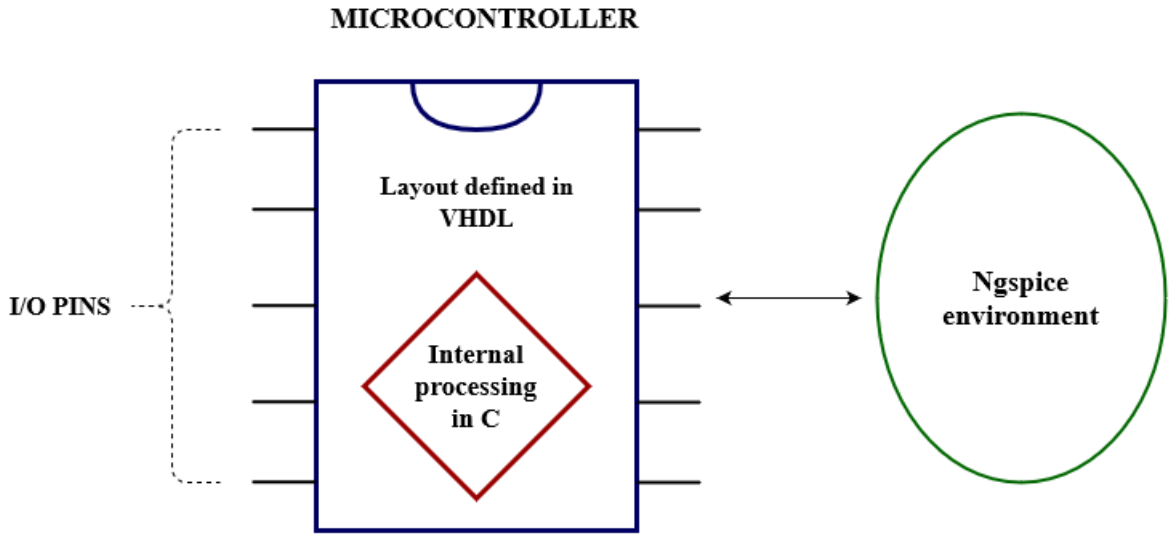


Figure 3: *Framework for Approach II*

PROS

- Since eSim already has NGHDL package implemented in it (which supports simulation of digital models in Ngspice environment), it is very easy to define the layout and basic functionality if a digital model in VHDL.
- Once the layout and pin configurations are defined for the digital microcontroller block, writing its internal working in C is quite easy.
- Due to above two points, it is easy to scale up, modify and maintain when new microcontrollers have to be implemented.
- Processing time of C code is quite low and simulation results are also very close to what we would actually get to observe in a practical microcontroller.

CONS

- Linking two different programming languages is difficult and it is further compounded by the fact that VHDL and C codes need to execute simultaneously while simulation.
- Lack of information and non availability of documentation for reference, since something like this has not been tried and tested in open source environment yet.

3.3 APPROACH III

Designing the layout of the microcontroller (pin configurations) and simulating its internal processing entirely in a single C program. This is made possible by Xspice, in which custom code models for simulating custom analog/digital components can be defined in C language (in a particular syntax).

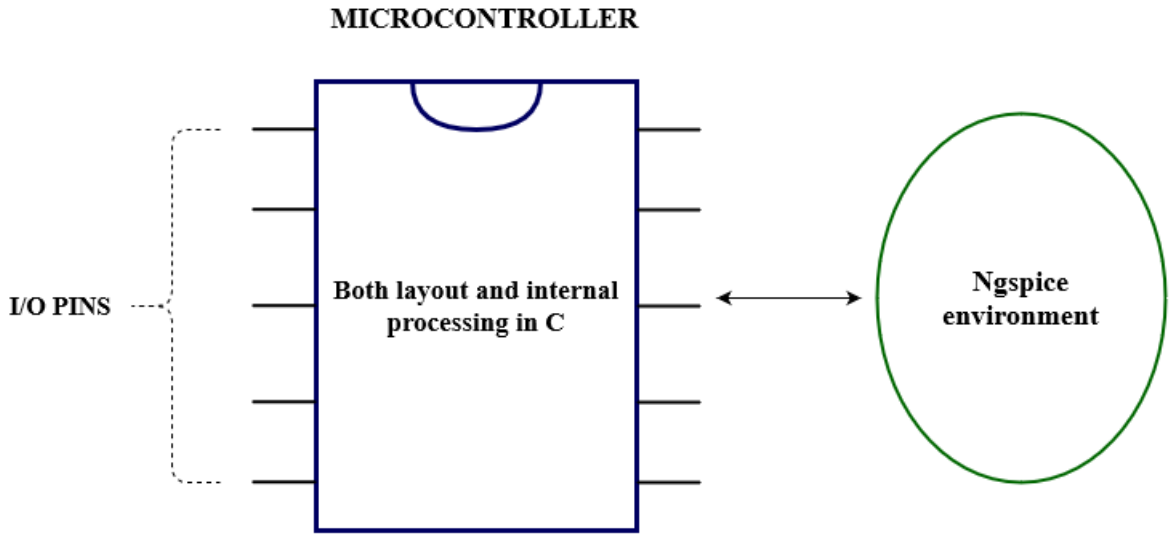


Figure 4: *Framework for Approach III*

PROS

- Faster simulation time than Approach II

- Same accuracy as Approach II
- Since everything will be written in C only, there will not be a problem of interfacing two different programming languages like in Approach II.

CONS

- The way C code has to be structured and how it is executed is fixed and cannot be modified easily.
- The syntax in which the custom code model for micrcontroller is to be written is very difficult to work with and unsuitable for creating a complex component.
- Development time would be high even for creating a prototype model, and it will be impossible to scale up.
- The entire C code will be executed once during simulation of one clock cycle. This means there will be problems when instructions like jump which require multiple clock cycles to execute are encountered.
- The information in the variables defined in C code only stays in them in a single cycle. Once a cycle is complete, C code is executed again and all the information contained in the C code during previous cycle is lost.

3.4 APPROACH TAKEN

From the Approaches discussed above, ***Approach II*** was chosen as the best way to implement an user programmable microcontroller in eSim. Though two different programming languages are to be linked and simulated together with proper timing, its benefits outweigh this drawback -

- Once the initial technical hurdle is overcome, there are no challenges to be faced in implementing the framework.
- This method utilizes the best parts of the two programming languages ***C*** and ***VHDL*** - ease of defining a component in VHDL with the ease and flexibility of writing logical statements in C.
- Users need not understand the framework completely to make changes to the individual codes if they wish to implement new microcontrollers.

4 SOFTWARE ARCHITECTURE

4.1 OVERVIEW

Design of the selected Approach focuses mainly on three blocks - ***The VHDL code (where microcontroller component is defined), the C code (where microcontroller logic is defined) and the helper function (which links the two codes together by facilitating communication between VHDL and C codes).***

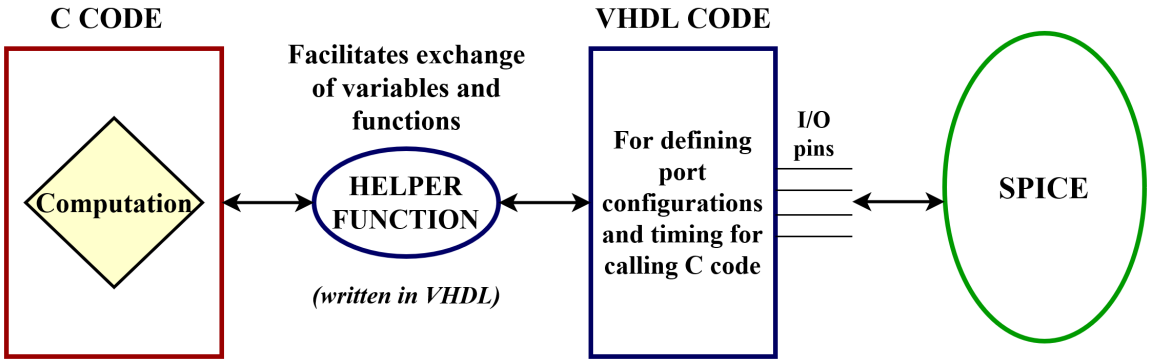


Figure 5: *Software architecture of the chosen approach*

- **The VHDL code** is written for defining the number and types and functionality of ports. These ports act as I/O pins of the microcontroller. This code defines when to get input from the Spice environment and when to give output to it, when to call what function in the C code and what variables to share with the C code. So basically, this acts as a master and C acts as a slave - C code will execute itself whenever the VHDL code gives the command. After completion of command, C code shares the value of I/O pins with VHDL code so that it can output them to Spice environment.

- **The C code** is written to simulate the internal working of the microcontroller. It stores the values of all registers and RAM as variables. When given the command to execute an instruction by the VHDL code, it fetches the instruction according to the value of Program Counter register and decodes which instruction it is, performs the computation according to the instruction (adding, subtracting registers, moving data from register to register, etc). After processing, it shares the value of I/O pins with VHDL so that it can output them.

- **The helper function** is written to link the C and VHDL codes together. The VHDL and C codes share variables and functions. This means a variable or function defined in C code can be accessed in VHDL code and vice versa. To facilitate this, a separate VHDL code (known as helper function) has to be written which basically typecasts variables and functions defined in C to VHDL.

4.2 FRAMEWORK

Technicalities of the approach finalized are explained in depth. Note that final codes for ATtiny85 is provided and explained in section 6 from page 35 to page 51 .

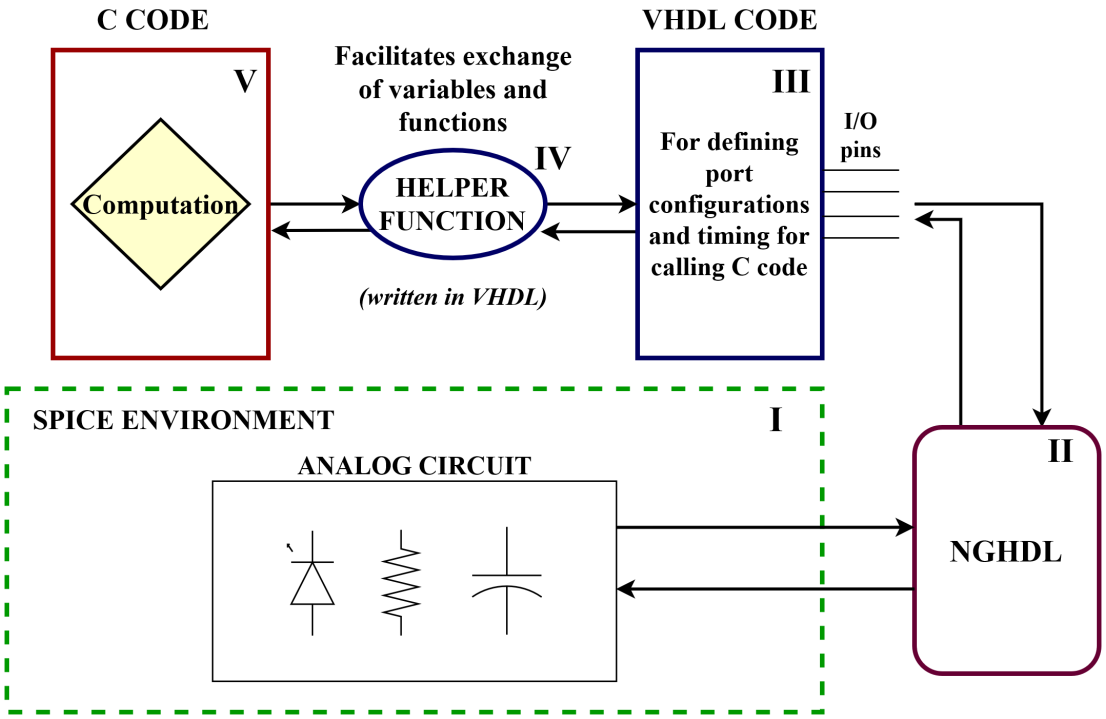


Figure 6: *Detailed flow of software architecture*

1. Clock pulse or a voltage signal from the analog circuit ***(Ngspice)*** reaches the microcontroller ***(VHDL code)*** via NGHDL. NGHDL is an in built package that comes with eSim which is used to interface Ngspice with digital models created in VHDL.

2. The VHDL code then decides, based on the clock pulse and the input it gets, which functions to call in C code for computation and what variables need to be shared.

3. The helper function converts the variables and functions given by VHDL code to variables and functions understandable and useable by C code.

4. The C code, based on the input from VHDL code and the current instruction stored in the HEX code, does the internal processing and transfer of data between registers and share the output with VHDL code in the form of variables.

5. The helper function converts the variables given by C code to variables understandable and usable by VHDL code.

6. VHDL code sets the I/O pins of microcontroller HIGH/LOW based on the output given by C code.

7. These voltage levels are then fed back to analog circuit in Ngspice via NGHDL.

4.3 TESTING THE FRAMEWORK

In order to test the feasibility and find out drawbacks of the implemented framework, we need to design a small example and simulate it. For this, Parallel In Serial Out (PISO) register example is chosen. Its working is described below -

4.3.1 PISO REGISTER

PISO register takes a set of parallel inputs and outputs serial data based on the inputs and clock pulse given to it. For testing the framework, a 4-bit PISO register was designed. In a normal digital circuit, the inputs to a digital component (written

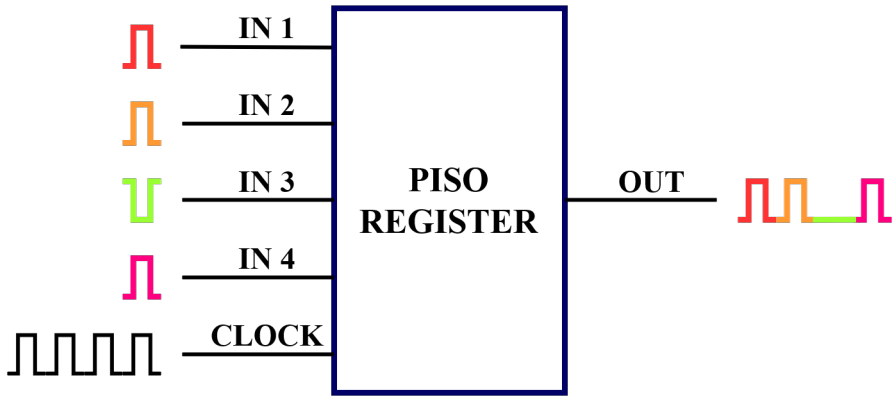


Figure 7: *4-bit PISO block diagram*

in VHDL) is given by Ngspice and then the digital components provides an output to Ngspice based on the inputs.

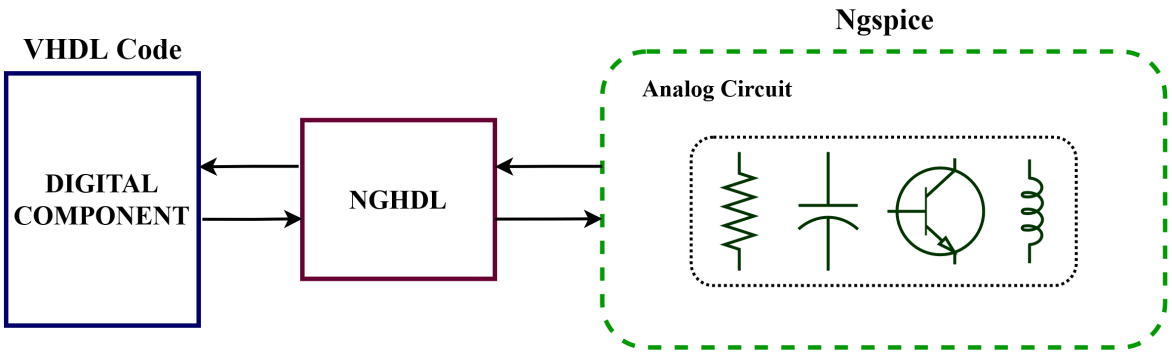


Figure 8: *Normal operation flow*

This PISO example demonstrates two things -

1. Inputs can be generated via a C code and fed to a digital component (written in VHDL) and the component would then generate output and give it to Ngspice like in a normal operation.

2. Exchange of variables and functions between C and VHDL code by helper function.

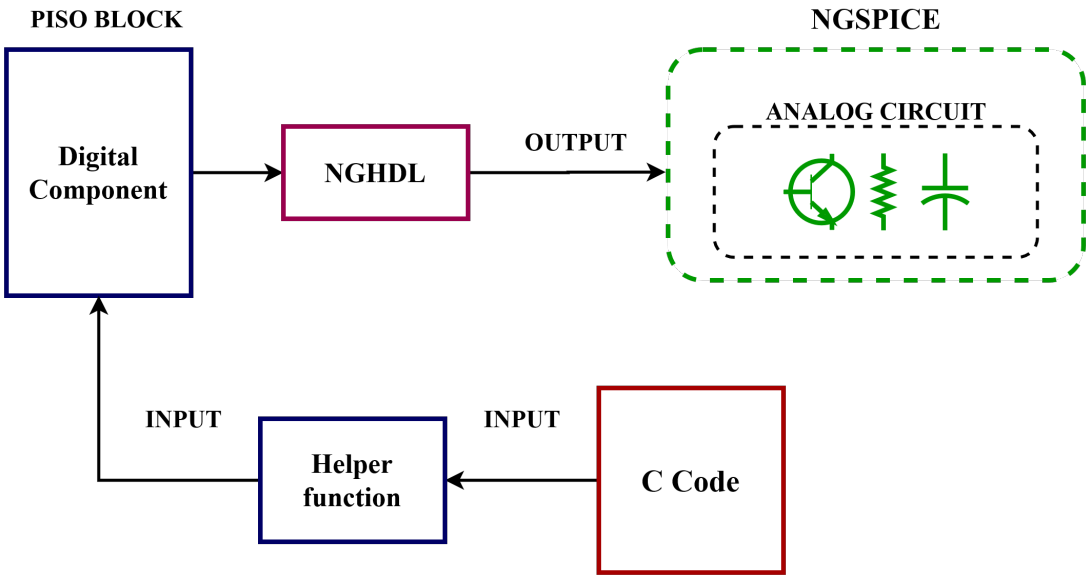


Figure 9: *Operation flow of proposed framework*

Schematic created -

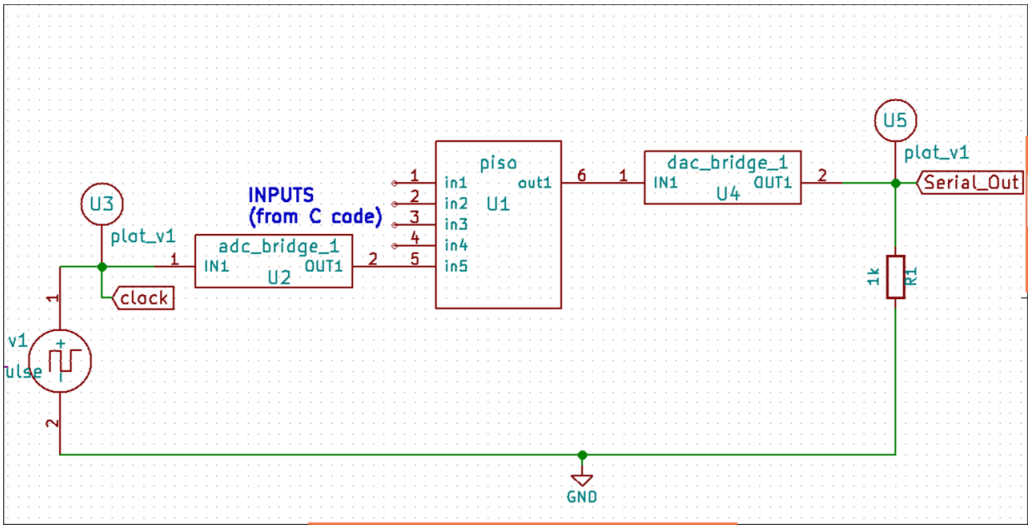


Figure 10: *Schematic for PISO example*

In the above schematic, only clock input is given to PISO block (which is written in VHDL) by Ngspice and the rest of its 4 inputs are given by C code. The steps for implementation and simulation are provided in section 6.6.1 on page 53, but it is advisable to go through the codes given below first and then follow the steps.

VHDL Code -

IEEE standard logic libraries are defined in lines 2 - 6. The ***ieee.std_logic_1164*** library defines ***std_logic*** and ***std_logic_vector*** datatypes and some other functions and ***ieee.numeric_std*** defines the ***signed*** and ***unsigned*** datatypes.
The helper function is named ***ghdl_access*** and it has to be included in the header since the shared variables and functions are declared in ***ghdl_access***. These discussed in depth later on.

```
2 library ieee;
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5 library work;
6 use work.ghdl_access.all;
```

The port configuration are defined in line 9 - 17. Ports ***a,b,c and d*** are input ports of datatype ***std_logic*** for getting parallel data. Port ***clk*** is for getting clock signal from Ngspice. Port ***o*** is output port which outputs serial data. Its datatype is ***std_logic_vector*** instead of ***std_logic*** because the although the conversion of ***integer*** to ***std_logic*** should happen seamlessly, it was not working in practice. The inputs a,b,c and d are given by C code.

```
9  entity piso is
10 port(a : in std_logic;
11      b : in std_logic;
12      c : in std_logic;
13      d : in std_logic;
14      clk : in std_logic;
15      o : out std_logic_vector(0 downto 0)
16      );
17 end piso;
```

Behaviour of PISO block is described in lines 19 - 44.

1. A variable count is defined which is a counter that keeps track of clock signals.
2. When clock signal is HIGH, count is incremented by one and function "output()" is executed by passing the value 2. This function is linked to C code and defined there.
3. var1 to var4 are variables which are shared between C and VHDL codes. Once the "output" function is called, C code gives values (in the form of variables) of 4 input data to VHDL code, and are generated based on the value of integer that is passed to "output" function by VHDL.
4. Based on the value of count, VHDL code puts these values on output port "o" sequentially. (Since shared variables are defined as ***integer***, it needs to be converted to ***std_logic_vector***.
5. Once the count reaches 3 (corresponding to 4 clock cycles), the count is reset and entire process is repeated again.

```
19 architecture bhv of piso is
20     signal count: integer := 0;
21 begin
22 process (clk)
23   begin
24     if (clk = '1') then
25           count <= count + 1;
26           output(2);
27           if(count = 0) then
28             o <= std_logic_vector(to_unsigned(var1.all, o'length));
29
30           elsif(count = 1) then
31             o <= std_logic_vector(to_unsigned(var2.all, o'length));
32
```

```
32
33          elsif(count = 2) then
34            o <= std_logic_vector(to_unsigned(var3.all, o'length));
35
36          elsif(count = 3) then
37            o <= std_logic_vector(to_unsigned(var4.all, o'length));
38            count <= 0;
39          end if;
40    elsif(clk = '0') then
41        o <= "0";
42    end if;
43 end process;
44 end bhv;
```

C Code -

The variables pin1 to pin4 are shared with VHDL code. In VHDL these variables are defined as var1 to var4 but in C these are defined as pin1 to pin4. Getters are defined in lines 3 - 17. These ***give*** the value of pin1 to pin4 to VHDL.

```
1  int pin1,pin2,pin3,pin4;
2
3  int * get_ptr() {
4    return &pin1;
5  }
6
7  int * get_ptr1() {
8    return &pin2;
9  }
10
11 int * get_ptr2() {
12   return &pin3;
13 }
14
15 int * get_ptr3() {
16   return &pin4;
17 }
```

The function “output” is defined in line 20 - 50. This function takes an ***integer*** value as parameter from VHDL code and based on that, it sets the values of variables pin1 to pin4. In this example, VHDL code passes the value “ 2 ” to “output”.

```
void output(int flag)
{
    if(flag==0)
        {
            pin1=0;
            pin2=1;
            pin3=0;
            pin4=1;
        }
    else if(flag==1)
        {
            pin1=1;
            pin2=0;
            pin3=1;
            pin4=0;
        }
    else if(flag==2)
        {
            pin1=0;
            pin2=1;
            pin3=1;
            pin4=0;
        }
    else if(flag==3)
        {
            pin1=1;
            pin2=1;
            pin3=0;
            pin4=1;
        }
}
```

Output -

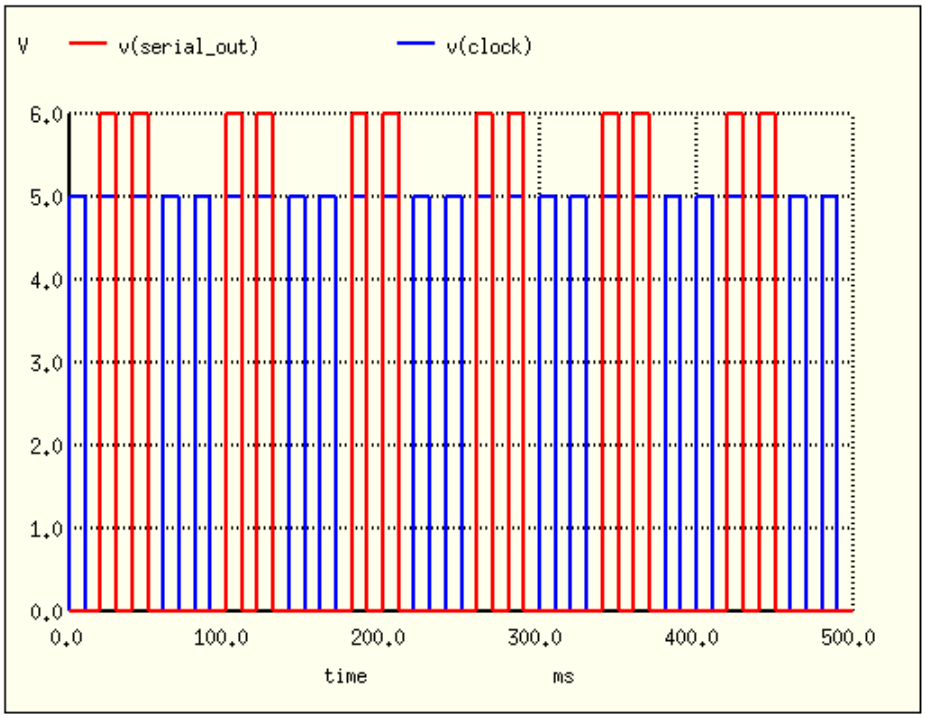


Figure 11: *Output from PISO example*

Helper Function -

Helper function takes variables from VHDL and converts them into a format understandable by C and vice versa. It also links functions defined in one language so that they can be used in another language just by calling it. Lines 8 - 22 are used to declare getter functions defined in C. These are used for sharing variables.

```
3  package ghdl_access is
4    -- Defines a pointer to an integer:
5    type int_access is access integer;
6
7
8    function get_ptr return int_access;
9      attribute foreign of get_ptr :
10       function is "VHPIDIRECT get_ptr";
11
12   function get_ptr1 return int_access;
13     attribute foreign of get_ptr1 :
14       function is "VHPIDIRECT get_ptr1";
15
```

```
15
16    function get_ptr2 return int_access;
17      attribute foreign of get_ptr2 :
18        function is "VHPIDIRECT get_ptr2";
19
20    function get_ptr3 return int_access;
21      attribute foreign of get_ptr3 :
22        function is "VHPIDIRECT get_ptr3";
```

The function "output" is declared in lines 24 - 27. Note that in line 25, the type of parameter that the function takes is declared.

```
24    -- declaration of functions in C
25      procedure output(f : integer);
26        attribute foreign of output :
27          procedure is "VHPIDIRECT output";
```

The getters get_ptr to get_ptr3 were declared in lines 3 - 22 but these only return a pointer to an integer. To store these values, variables are declared in lines 30 - 35. Therefore var1 to var4 equal to pin1 to pin4 variables defined in C code.

```
30    -- create variables aliased to the variable in C
31    shared variable var1 : int_access := get_ptr;
32    shared variable var2 : int_access := get_ptr1;
33    shared variable var3 : int_access := get_ptr2;
34    shared variable var4 : int_access := get_ptr3;
35  end ghdl_access;
```

The body of this helper function is defined in lines 37 - 63.

```
37  package body ghdl_access is
38    function get_ptr return int_access is
39    begin
40      assert false report "VHPI" severity failure;
41    end get_ptr;
42
43    function get_ptr1 return int_access is
44    begin
45      assert false report "VHPI" severity failure;
46    end get_ptr1;
47
```

```
47
48    function get_ptr2 return int_access is
49    begin
50      assert false report "VHPI" severity failure;
51    end get_ptr2;
52
53    function get_ptr3 return int_access is
54    begin
55      assert false report "VHPI" severity failure;
56    end get_ptr3;
57
58    procedure output(f : integer) is
59    begin
60      assert false report "VHPI" severity failure;
61    end output;
62
63  end ghdl_access;
```

start_server BASH FILE -

In order to simulate C and VHDL codes together, their object files needs to be created and linked together first. This is done by "start_server" file. This file is generated automatically by NGHDL when a VHDL code is uploaded but needs to be modified according to the proposed framework. Only the modified lines (10-14) are explained and explanation of rest of the code can be found in *NGHDL documentation*. In this example the main VHDL filename is "piso.vhdl", the C code filename is "piso_c.c" and the helper function filename is "ghdl_access.vhdl". The procedure to link C and VHDL code is -

1. The C code has to be compiled and its object file has to be generated. This is done by line 10.

2. The main VHDL code and helper function have to be compiled and their object file has the be generated. This is done by line 11. Note that the object file of helper function ***has*** to be generated before or simultaneously with the main VHDL code's. This is because main VHDL file calls and uses the helper function and if it is compiled first, GHDL will give an error.

3. The object files of C and VHDL code have to be linked. Note that the filename of the linked object file should be that of original VHDL file i.e. in this example, it should be "piso". This is done by -

   (a) Renaming object file of VHDL code from "piso.o" to "piso1.o". This is done by line 12.

   (b) Linking (combining) the object file of C code ("piso_c.o") with renamed object file of VHDL code "piso1.o" and naming their linked object file as "piso.o". This is done by line 13.

(c) "piso1.o" is not needed so is deleted by line 14.

```
5  cd /home/ash98/ngspice-nghdl/src/xspice/icm/ghdl/piso/DUTghdl/
6  chmod 775 sock_pkg_create.sh &&
7  ./sock_pkg_create.sh $1 $2 &&
8  ghdl -a sock_pkg.vhdl &&
9
10 gcc -c piso_c.c -o piso_c.o &&            #Added by Ashutosh Jha
11 ghdl -a ghdl_access.vhdl piso.vhdl &&    #Added by Ashutosh Jha
12 mv piso.o piso1.o                        #Added by Ashutosh Jha
13 ld -r -o piso.o piso_c.o piso1.o &&      #Added by Ashutosh Jha
14 rm piso1.o &&                            #Added by Ashutosh Jha
15
16 ghdl -a piso_tb.vhdl  &&
17 ghdl -e -Wl,ghdlserver.o piso_tb &&
18 ./piso_tb
```

4.4 SOFTWARE DEPENDENCIES

The above architecture depends upon some already implemented pieces of software and are mentioned in brief below. These pieces of software my be platform specific and may cause problems at the time of porting the entire framework from one platform to other. For further details regarding this, please refer section 7 on page 57.

1. **NGHDL -** NGHDL is used to interface the VHDL model with Ngspice. Whenever Ngspice calls the VHDL model, the inputs go to GHDL via NGHDL, and GHDL provides the output which is communicated back to Ngspice. This framework is tried and tested, and gives accurate simulation results with the only drawback being at the time that port type "inout" cannot be defined in NGHDL. Also NGHDL is only available for Ubuntu v16.04 as of 4th April,2020 .

2. **LINUX COMMANDS -** In the bash file, we are using many commands which may be platform specific. Thus, these commands will work flawlessly in Linux based distributions but probably not on other platforms like Windows. So if ported to other platforms, the bash commands will have to be modified accordingly.

3. **AVR GCC** - AVR GCC [3] is an open source compiler which is used to convert user made C code to hex code. This is not integrated in the framework at the moment as the user is intended to directly upload hex file (compiling C to hex himself/herself). But may be integrated inside eSim in the future - which will allow user to directly upload the C code.

5 AVR MICROCONTROLLERS

AVR is a family of microcontrollers developed since 1996 by Atmel, acquired by Microchip Technology [11] in 2016. These are modified Harvard architecture [9] 8-bit RISC [4] single-chip microcontrollers. AVR was one of the first microcontroller families to use on-chip flash memory for program storage, as opposed to one-time programmable ROM, EPROM, or EEPROM used by other microcontrollers at the time.

AVR microcontrollers find many applications as embedded systems. They are especially common in hobbyist and educational embedded applications, popularized by their inclusion in many of the Arduino [1] line of open hardware development boards.

5.1 OVERVIEW

Flash, EEPROM, and SRAM are all integrated onto a single chip, removing the need for external memory in most applications. Some devices have a parallel external bus option to allow adding additional data memory or memory-mapped devices. Almost all devices (except the smallest TinyAVR chips) have serial interfaces, which can be used to connect larger serial EEPROMs or flash chips.

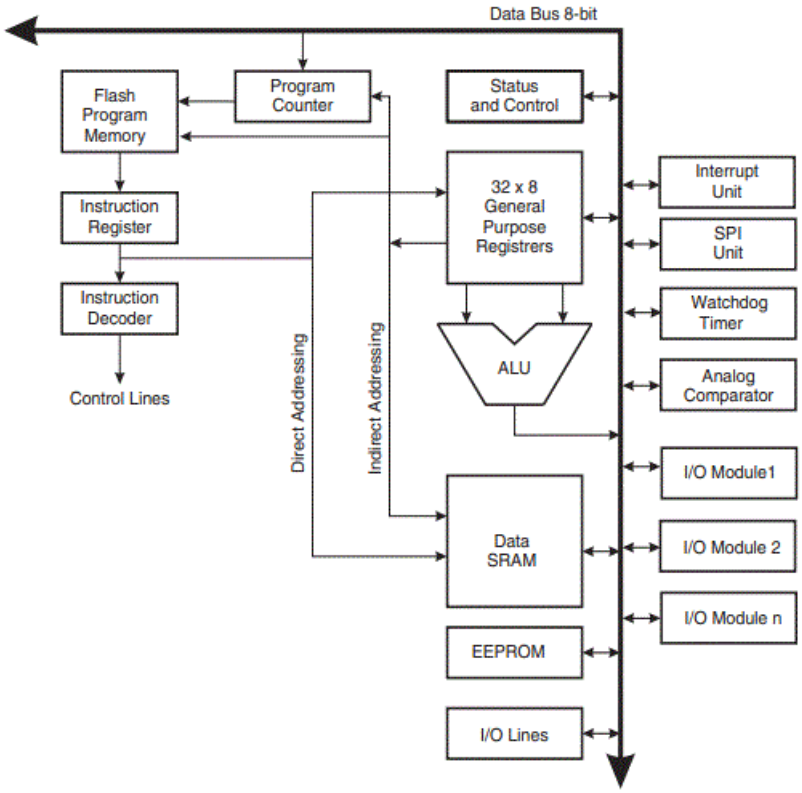


Figure 12: *AVR Architecture*

5.1.1 PROGRAM MEMORY

Program instructions are stored in non-volatile flash memory. Although the MCUs are 8-bit, each instruction takes one or two 16-bit words.

5.1.2 INTERNAL DATA MEMORY

The data address space consists of the register file, I/O registers, and SRAM. Some small models also map the program ROM into the data address space, but larger models do not.

5.1.3 INTERNAL REGISTERS

The AVRs have 32 single-byte registers and are classified as 8-bit RISC devices. In the tinyAVR and megaAVR variants of the AVR architecture, the working registers are mapped in as the first 32 memory addresses.

5.1.4 GPIO PORTS

A general-purpose input/output (GPIO) is an uncommitted digital signal pin on an integrated circuit or electronic circuit board whose behavior can be defined by the user.

Each GPIO port on a tiny or mega AVR drives up to eight pins and is controlled by three 8-bit registers - DDRx, PORTx and PINx, where x is the port identifier.

- **DDRx -** Data Direction Register, configures the pins as either inputs or outputs.
- **PORTx -** Output port register. Sets the output value on pins configured as outputs. Enables or disables the pull-up resistor on pins configured as inputs.
- **PINx -** Input register, used to read an input signal. On some devices, this register can be used for pin toggling: writing a logic one to a PINx bit toggles the corresponding bit in PORTx, irrespective of the setting of the DDRx bit.

5.1.5 PROGRAM EXECUTION

Atmel's AVRs have a two-stage, single-level pipeline design. This means the next machine instruction is fetched as the current one is executing. Most instructions take just one or two clock cycles, making AVRs relatively fast among 8-bit microcontrollers.

5.1.6 INSTRUCTION SET

The Atmel AVR instruction set is the machine language for the Atmel AVR, a modified Harvard architecture 8-bit RISC single chip microcontroller which was developed by Atmel in 1996.

5.1.7 INSTRUCTION TIMING

Arithmetic operations work on registers R0–R31 but not directly on RAM and take one clock cycle, except for multiplication and word-wide addition (ADIW and SBIW) which take two cycles.

RAM and I/O space can be accessed only by copying to or from registers. Indirect access (including optional post increment, pre decrement or constant displacement) is possible through registers X, Y, and Z. All accesses to RAM takes two clock cycles. Moving between registers and I/O is one cycle. Moving eight or sixteen bit data between registers or constant to register is also one cycle. Reading program memory (LPM) takes three cycles.

5.1.8 MCU SPEED

The AVR line can normally support clock speeds from 0 to 20 MHz, with some devices reaching 32 MHz. Lower-powered operation usually requires a reduced clock speed. All recent (Tiny, Mega, and Xmega, but not 90S) AVRs feature an on-chip oscillator, removing the need for external clocks or resonator circuitry. Some AVRs also have a system clock prescaler that can divide down the system clock by up to 1024. This prescaler can be reconfigured by software during run-time, allowing the clock speed to be optimized.

Since all operations (excluding multiplication and 16-bit add/subtract) on registers R0–R31 are single-cycle, the AVR can achieve up to 1 MIPS per MHz, i.e. an 8 MHz processor can achieve up to 8 MIPS. Loads and stores to/from memory take two cycles, branching takes two cycles. Branches in the latest "3-byte PC" parts such as ATmega2560 are one cycle slower than on previous devices.

5.2 ATTINY85

The ATtiny85 is a 8-bit microcontroller in a similar vein to the ATMEGA328 used in Arduino, but with much less I/O pins, smaller memory and a smaller form factor. Which translates to almost same computing power, but in a much more compact board as demonstrated in the image below.

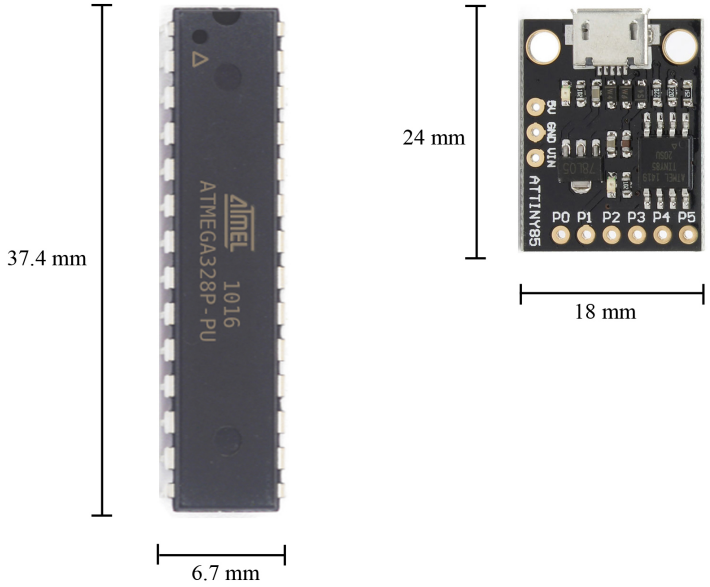


Figure 13: *Size comparison between ATMEGA328 and ATtiny85 development board*

Many times, most users do not require the 23 GPIO pins present in the Arduino board. Thus for smaller projects, ATtiny85 proves to be a better option. Important specifications -

- Advanced RISC architecture
  - Supports 120 instructions from AVR instruction set. *Detailed description on AVR instructions*
  - 32×8 General purpose working instructions
  - Fully static operation
- 8K bytes of in-system programmable flash memory
- 10-bit ADC
- 6 programmable I/O pins
- Operating speed - 0 to 20 MHz

The ATtiny85 follows AVR architecture discussed in section 5.1 on page 23. Detailed information and specification can be found in it is *datasheet*.

5.3 PRACTICAL WORKING

Microcontrollers are used for simulating logical operations and nowadays, these operations are generally written in a higher level language (mostly in some variation of C). This allows greater ease with which users can control the microcontroller and implement logical operation, but increases the steps involved to program it.

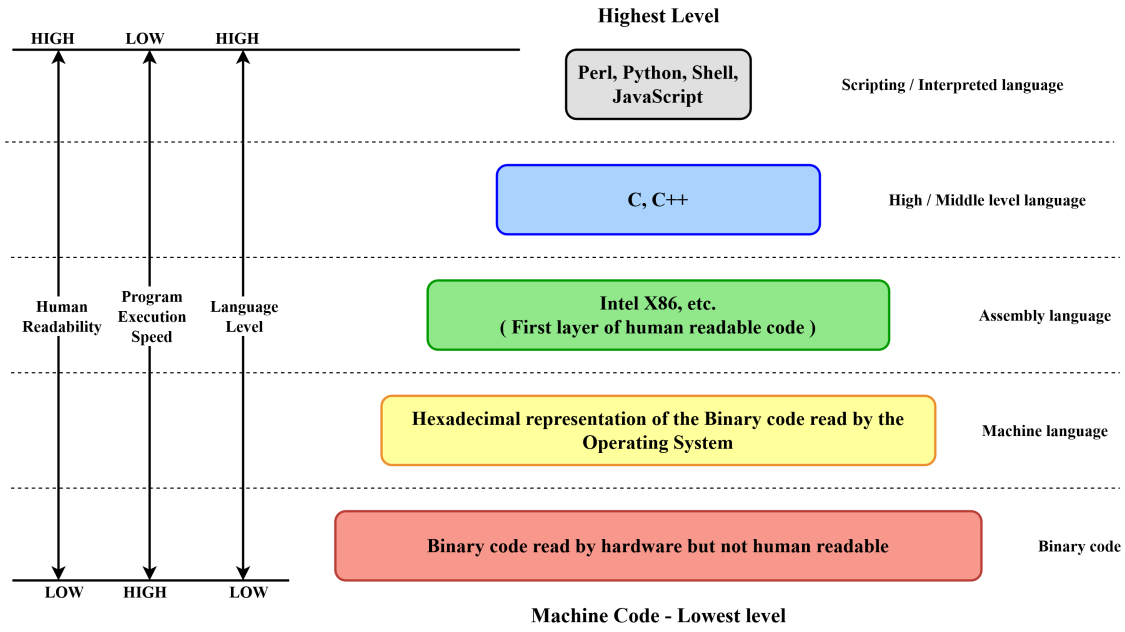


Figure 14: *Hierarchical representation of programming languages*

1. Any high level language is understood to users but not to machine. Therefore, in order to somehow store the high level code in microcontroller, it is first converter to assembly level language.

2. The machine then converts the assembly equivalent code into 1s and 0s and copies it inside the EEPROM of microcontroller via an external hardware interface.

3. These 1s and 0s are in hexadecimal form and known better as the ***HEX CODE***.

The process is explained by the help of a flowchart below -

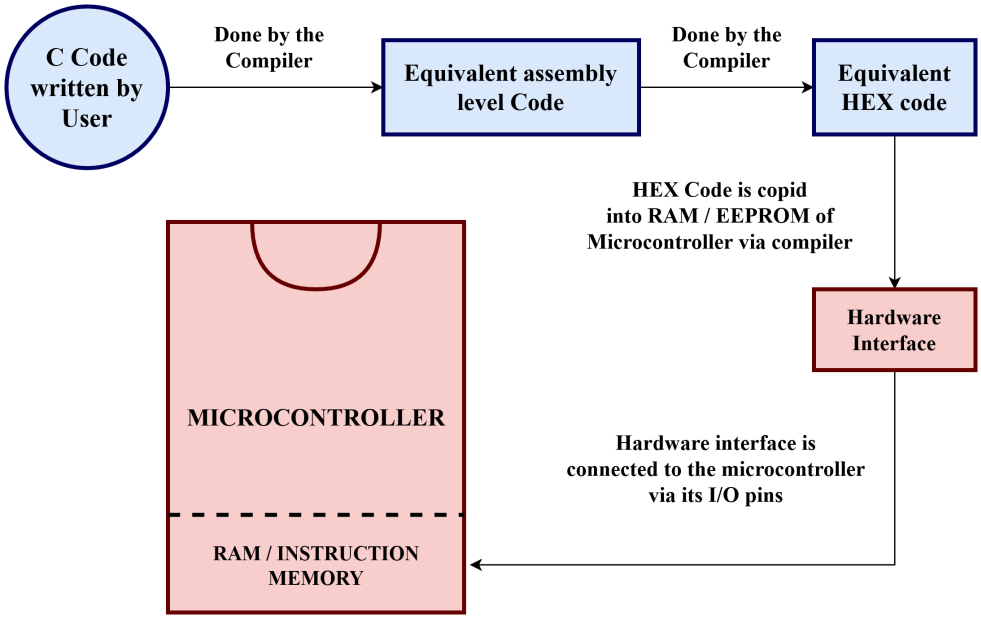


Figure 15: *Process depicting how a user generated C code is stored inside microcontroller*

5.3.1 UNDERSTANDING THE HEX CODE

The hex code is the machine level equivalent of the user generated High level (C) code. The program written by user is converted to hex code and stored inside the microcontroller. The compiler generates hex code and gives it to the hardware interface [8] [7] which burns it into the microcontroller. Thus, it is important to understand the contents of hex code since it is required for simulation and user is expected to upload a hex code in order to simulate the microcontroller logic that it intends to. Consider the following C code as example -

```
6  int main (void)
7  {
8      // set PB0 to be output
9      DDRB = 0x01;
10     while (1) {
11     // set PB0 high
12     PORTB = 0x01;
13     _delay_ms(2);
14     // set PB0 low
15     PORTB = 0x00;
16     _delay_ms(3);
17   }
18 }
```

If we compile the above C code with **AVR GCC** and generate its HEX code, we get the following output -

```
:100000000EC015C014C013C012C011C010C00FC064
:100010000EC00DC00CC00BC00AC009C008C011241E
:100020001FBECFE5D2E0DEBFCDBF02D012C0E8CF09
:1000300081E087BB88BBE7EEF3E03197F1F700C0C2
:10004000000018BAEBEDF5E03197F1F700C00000C1
:06005000F1CFF894FFCF90
:00000001FF
```

This is the hex equivalent of the C code. The compiler converts the C code into assembly level code. Each command in assembly code has a particular OPCODE. Essentially, the hex code is a collection of OPCODES of all the assembly level commands. Typically, all these steps are performed in background by the compiler and the individual steps are not shown to the user for ease of usage in almost all widely used compilers such as Arduino Ide [6]

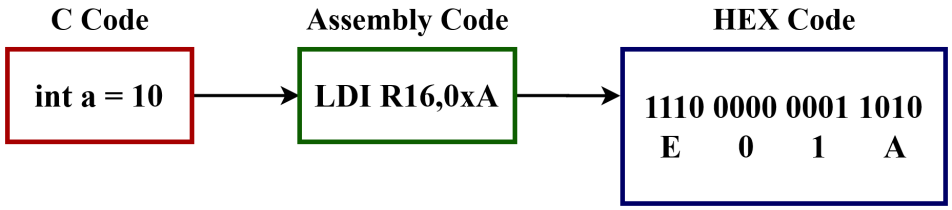


Figure 16: *Process of conversion from C to Hex code*

Essentially, the C code written by the user is converted line by line to assembly and then to Hex as illustrated above. Note that the OPCODE for "LDI" is "EXXX" where "X" repsresents the data contained in the "LDI" mnemonic (called the operand). List of assembly level mnemonics and their corresponding OPCODES can be found in Appendix A on page 61

NOTE -

The entire HEX code generated is not required for the simulation. The HEX code contains some padding at the beginning and ending of each line, shown above in green and blue colours respectively. The code also contains routines and subroutines other than the ones written in C code by user. These are used when the HEX code is being physically burned into the microcontroller and not required for simulation. The part necessary for simulation is in red colour. A quick crosscheck of the red part of the HEX code with code 5.3.1 tells us why only the red part is required.

Therefore, only the red part of HEX code has to be uploaded and utilized. Note that **in future** the user will upload the entire generated HEX file as it is, and the C code will automatically detect the important (red) portion and work on it. But as of now, we need to copy the important part i.e. the red portion only. The important content should be copied into a file named ***"hex.txt"*** and pasted in the DUTghdl

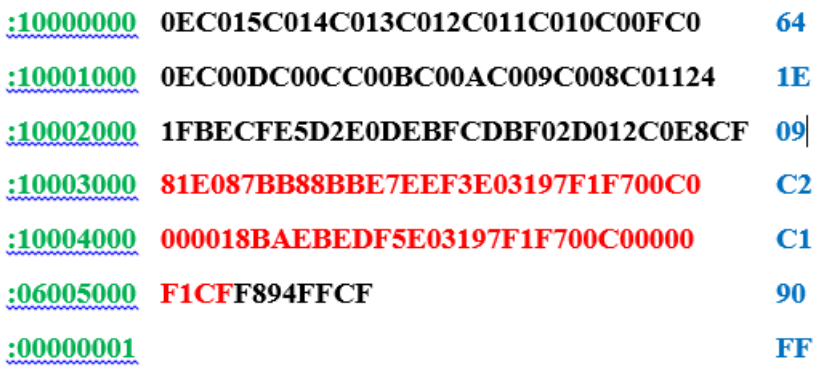


Figure 17: *Generated HEX code for the C code*

folder. Thus, for the above example, the final ***"hex.txt"*** file that has to be put in DUTghdl should have only the following content inside it -

```
1 81E087BB88BBE7EEF3E03197F1F700C0000018BAEBEDF5E03197F1F700C00000F1CF
```

6 IMPLEMENTED FRAMEWORK

The framework used to implement ATtiny85 is explained below in depth. Its framework is identical to PISO example explained in section 4.3.1 on page 13. The only changes made are in C and main VHDL code to implement microcontroller logic.

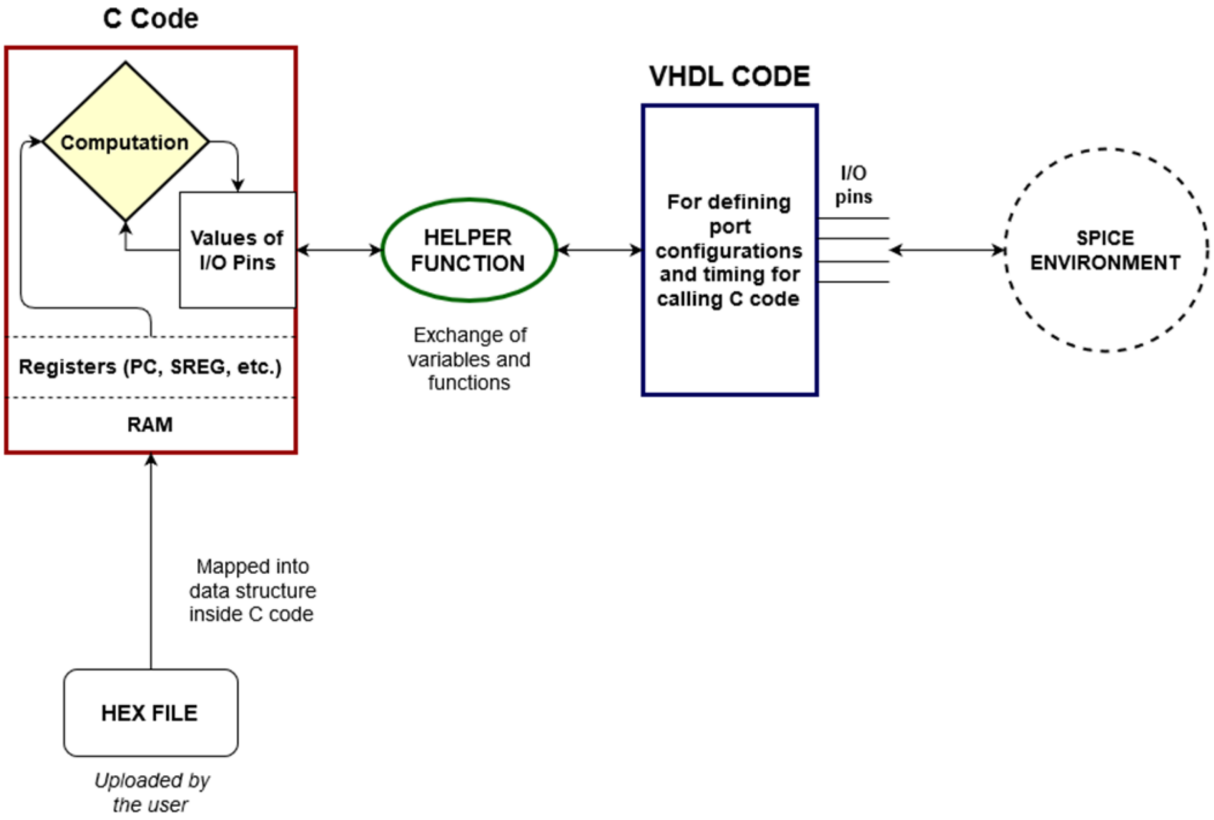


Figure 18: *Framework used to implement ATtiny85*

1. Before The simulation begins, the hex file uploaded by the user is copied and stored in a data structure inside C code. This is done by "MapToRam" function.

2. When simulation starts, the clock pulses and other inputs are given to the VHDL block (microcontroller component) by Ngspice.

3. Based on whether clock, VCC and GND are HIGH, HIGH and LOW respectively, the VHDL code calls "output" function.

4. The "output" function fetches the current instruction based on PC value, executes it, changes the value of I/O pins (by changing values of variables PB0 to PB5) and changes the value of PC according to current instruction for next instruction.

5. The VHDL code gets the value of output ports (PB0 to PB5 in C and var0 to var5 in VHDL) as integer variables, converts it to "std_logic_vector" form and puts it on output ports (I/O) pins of the microcontroller.

6. Ngspice gets the output values from VHDL block (microcontroller component) and uses those values as voltage levels to simulate the analog circuit around it.

7. Steps 2 - 5 are repeated until simulation stops.

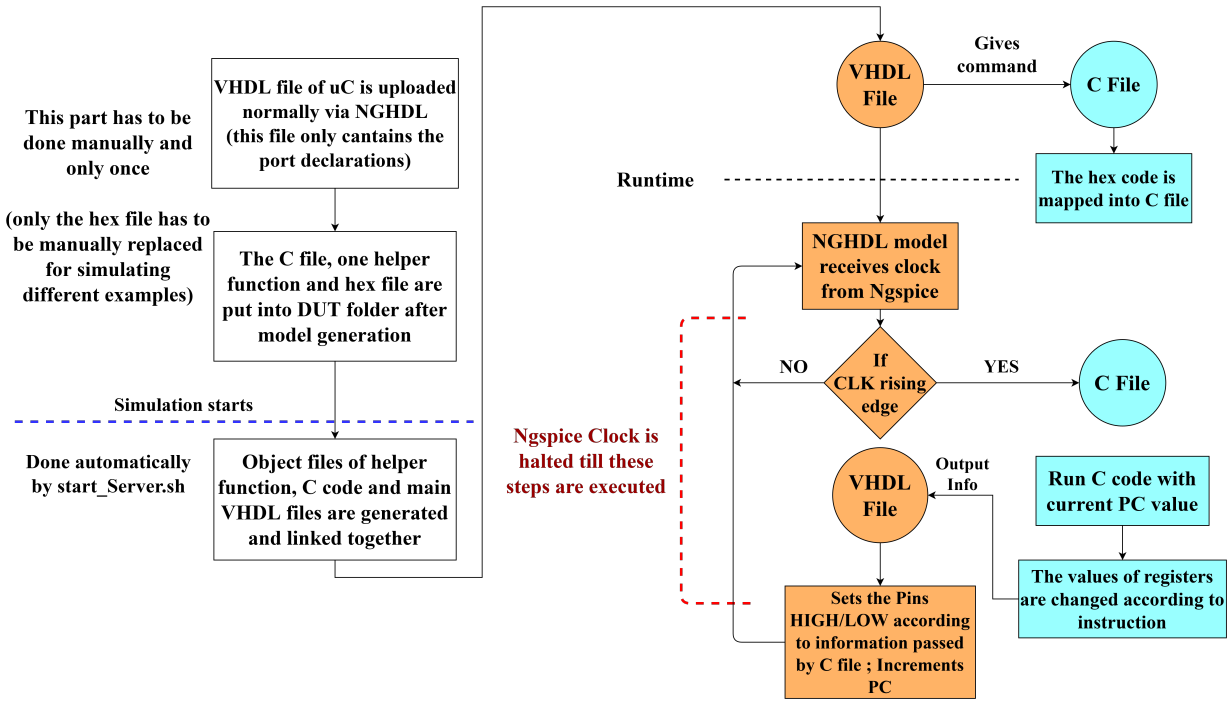


Figure 19: *Workflow of microcontroller simulation*

6.1 SAMPLE CODE AND OUTPUT

The steps for simulation are given in section 6.6.2 on page 53. For a demonstration, simple code for PWM is written. The code will set PB0 of ATtiny85 microcontroller HIGH for 2ms and LOW for 3ms.

```
2  #include <avr/io.h>
3  #define F_CPU 2.0E6
4  #include<util/delay.h>
5
6  int main (void)
7  {
8      // set PB0 to be output
9      DDRB = 0x01;
10     //    81E0        ldi r24,0x01        ; 1
11     //    87BB        out 0x17,r24        ; 23
12     while (1) {
13     // set PB0 high
14     PORTB = 0x01;
15     //    88BB        out 0x18,r24        ; 24
16     _delay_ms(2);
17     /*    E7EE        ldi    r30,0xE7       ; 231
18           F3E0        ldi    r31,0x03       ; 3
19           3197        sbiw r30,0x01       ; 1
20           F1F7        brne    .-4           ; 0x3a
21           00C0        rjmp    .+0           ; 0x40
22           0000        nop    */
23
24     // set PB0 low
25     PORTB = 0x00;
26     //    18BA        out    0x18,r31       ; 24
27     _delay_ms(3);
28     /*    EBED           ldi    r30,0xDB       ; 219
29           F5E0           ldi    r31,0x05       ; 5
30           3197           sbiw r30,0x01       ; 1
31           F1F7           brne    .-4           ; 0x48
32           00C0           rjmp    .+0           ; 0x4e
33           0000           nop
34           F1CF           rjmp    .-30          ; 0x34    */
35   }
36 }
```

The equivalent HEX code and assembly code for each line is also given for a better understanding. The C code shown above has to be converted to HEX code and the HEX code is uploaded into the microcontroller. The HEX code is generated by AVR GCC.

Schematic -

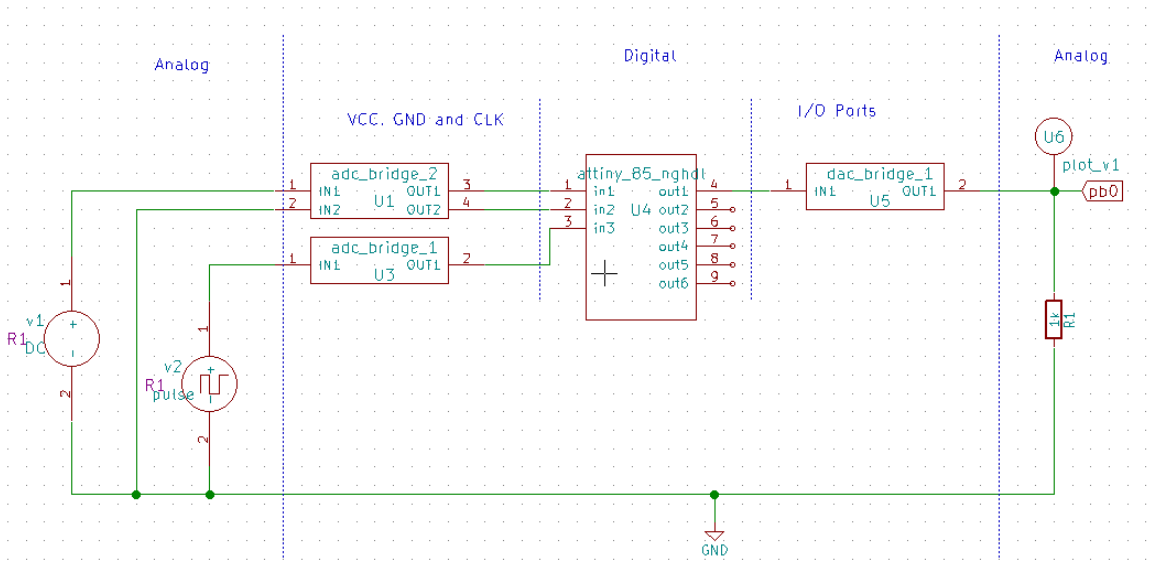


Figure 20: *Schematic for ATtiny85 demonstration*

Above schematic is used for demonstration.

Output -

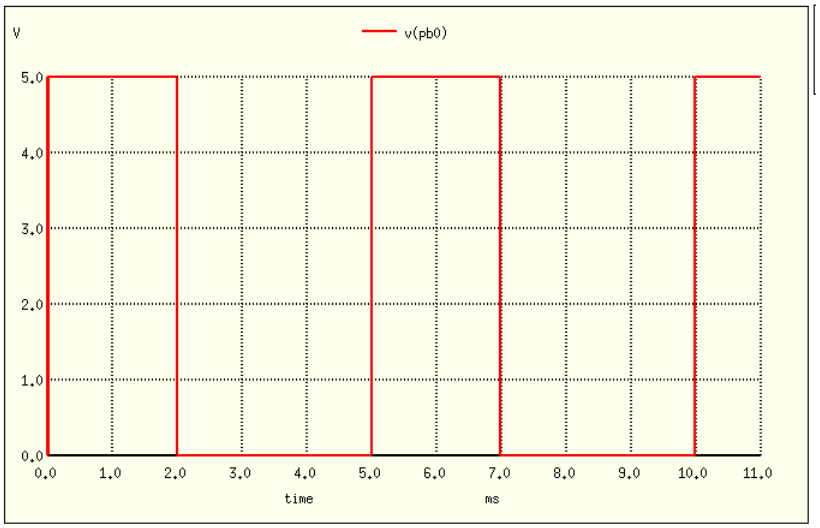


Figure 21: *Demo output*

The output showcases the accuracy of the framework. Note that the time for simulation was 11ms and step time was 0.1us. With these settings, it took around 120 seconds (2 minutes) for entire simulation. Device specifications - Intel i5 4th generation (4 cores - 2 physical and 2 logical), 8gb RAM and on UBUNTU v16.04LTS.

6.2 VHDL CODE FOR ATTINY85

The main VHDL code used to create microcontroller component is discussed below.

Header -
IEEE standard logic libraries are defined in lines 7 - 9. The ***ieee.std_logic_1164*** library defines ***std_logic*** and ***std_logic_vector*** datatypes and some other functions and ***ieee.numeric_std*** defines the ***signed*** and ***unsigned*** datatypes. The helper function is named ***"ghdl_access"*** and it has to be included in the header since the shared variables and functions are declared in ***"ghdl_access"***. These discussed in depth later on.

```
7  library ieee;
8  use ieee.std_logic_1164.all;
9  use ieee.numeric_std.all;
10
11 library work;
12 use work.ghdl_access.all;
```

Port Declarations -

The input ports ***VCC, GND*** and ***clk*** are for supply voltage, ground and clock respectively. The ports ***PB0, PB1 ... PB5*** are the 6 GPIO pins of microcontroller. These are configured as output ports as of now, but will be defined as ***inout*** later on.

```
14 entity attiny_85_nghdl is
15 port(VCC : in std_logic;
16      GND : in std_logic;
17      clk : in std_logic;
18      PB0 : out std_logic_vector(0 downto 0);
19      PB1 : out std_logic_vector(0 downto 0);
20      PB2 : out std_logic_vector(0 downto 0);
21      PB3 : out std_logic_vector(0 downto 0);
22      PB4 : out std_logic_vector(0 downto 0);
23      PB5 : out std_logic_vector(0 downto 0));
24 end attiny_85_nghdl;
```

Behaviour of ATtiny85 block -

1. Before the beginning of simulation function "MapToRam" is called. This function is linked to C code and defined there. It copies the HEX file uploaded by the user into a data structure inside the C code and stores it there for the entire duration of simulation. It is called once just before the simulation starts every time.

2. Line 31 checks for VCC and GND provided by user. If VCC and GND is provided and Clock pulse is on rising edge, the logic inside loop is executed.

3. The function "output" is called. It is linked to C code and defined there. The output function executes the instruction pointed by Program Counter (PC is defined in C code and increments after each instruction is executed). At the end of processing, C code shares the value of I/O pins in form of variables pin1 to pin6. These variables are defined as var0 to var5 in VHDL. Thus after calling "output" function, the output that is given by C code to VHDL code (var0 to var5) are converted to ***std_logic_vector*** and given to output ports PB0 to PB5.

4. Steps 2 and 3 are repeated continuously until simulation stops.

```
26 architecture bhv of attiny_85_nghdl is
27 begin
28 MapToRam(1);
29     process(clk)
30     begin
31         if(rising_edge(clk) and VCC = '1' and GND = '0') then
32             output(1);
33             PB0 <= std_logic_vector(to_unsigned(var0.all, PB0'length));
34             PB1 <= std_logic_vector(to_unsigned(var1.all, PB1'length));
35             PB2 <= std_logic_vector(to_unsigned(var2.all, PB2'length));
36             PB3 <= std_logic_vector(to_unsigned(var3.all, PB3'length));
37             PB4 <= std_logic_vector(to_unsigned(var4.all, PB4'length));
38             PB5 <= std_logic_vector(to_unsigned(var5.all, PB5'length));
39         elsif(VCC = '0') then
40             PB0 <= "0";
41             PB1 <= "0";
42             PB2 <= "0";
43             PB3 <= "0";
44             PB4 <= "0";
45             PB5 <= "0";
46         end if;
47     end process;
48 end bhv;
```

6.3 C CODE FOR ATTINY85

The C code contains data structures into which the RAM and registers of ATtiny85 are mapped and stored during simulation. It essentially decodes each instruction present in HEX file and based on the instruction and input given by VHDL, provides necessary output to VHDL code.

Header -

The C code utilizes only 2 header files - "stdio.h" and "math.h" to keep the code platform friendly. Attiny85 has 8kb of RAM as specified in its datasheet. Therefore a constant "size" is defined as 8192 (2^8) and it will be used later in the code.

```
9  #include<stdio.h>
10 #include<math.h>
11 #define size 8192        //8kb ram size for attiny 85
```

Global variables and data structures -

- Variable "debugMode" is used to see additional information during simulation in the console window for debugging the code. If it is "1", debugMode is ON and "0" means it is OFF. Please note it is meant only to be used by the developer and not the user.
- The variables PB0 to PB5 contain the value of 6 I/O pins and are shared with VHDL code. They are either "1" or "0" (HIGH/LOW)
- PC is defined as a unsigned short (since it can store HEX values directly and store 2 bytes of data) and thus can address memory locations 0x0000 to 0xFFFF.
- The structure "memory" is used to store RAM, general purpose register, status register and special purpose registers PORTB and DDRB. All the memory locations created in this structure are defined as unsigned char because in C, unsigned char datatype is of 1 byte and can store hex values from 0x00 to 0xFF which is exactly like how it is inside an actual ATtiny85.

```
13 int debugMode=1;
14 int PB0,PB1,PB2,PB3,PB4,PB5;
15 unsigned short PC=0;
16 struct memory            //Structure to store RAM and other registers
17 {
18     unsigned char data;
19 }ram[size],reg[32],SREG[8],PORTB,DDRB;
```

- Another structure “BinArrays” is used to create binary arrays of 8 bits each. These arrays are used to perform arithmetic operations.

```
28 struct BinArrays
29 {
30     int arr[8];
31 }bin[3];
```

Getters -

The getters are used to share variables (I/O) pin values with VHDL code. There are 6 I/O pins corresponding to 6 getters (line 33 - 55) returning 6 integers as pointers.

```
33 int * get_ptr0() {
34   return &PB0;
35 }
36
37 int * get_ptr1() {
38   return &PB1;
39 }
40
41 int * get_ptr2() {
42   return &PB2;
43 }
44
45 int * get_ptr3() {
46   return &PB3;
47 }
48
49 int * get_ptr4() {
50   return &PB4;
51 }
52
53 int * get_ptr5() {
54   return &PB5;
55 }
```

Auxiliary Functions -

The function “ClearBins” clears a specified binary array. The function takes an integer as parameter and clears the binary array specified by the integer.

```
void ClearBins(int binSel)
{
    int i;
        for(i=0;i<8;i++)
        {
            bin[binSel].arr[i]=0;
        }
}
```

The function "SetRam" stores the specified hex value in the RAM (defined in structure memory) from a lower address value to an upper address value. The function takes three parameters - integer upper and lower bounds and the hex value as an character.

```
void SetRam(int lb, int ub,char val)
{
    int i;
    for(i=lb;i<ub;i++)
        ram[i].data = val;
}
```

The function "PrintSREG" prints the value of status register.

```
void PrintSREG()
{
 printf("\nStatus Register:- \n");
 printf("I: %d ,T: %d ,H: %d ,S: %d ,V: %d ,N: %d ,Z: %d ,C: %d \n",
  ↪  SREG[7].data, SREG[6].data, SREG[5].data, SREG[4].data,
  ↪  SREG[3].data, SREG[2].data, SREG[1].data, SREG[0].data);

}
```

The function "PrintRam" prints the contents of RAM from a given lower to upper bound. The function accepts two integers as parameters for lower and upper bounds.

```
{
    int i=0;
    unsigned char b1,b2,b3,b4;
    printf("\n***********RAM*************\n");
    for(i=lb;i<ub;i+=4)
        {
            b1=ram[i+0x2].data, b2=ram[i+0x3].data ,b3=ram[i].data,
             ↪  b4=ram[i+0x1].data;
```

```
            printf("instruction %X: %X%X%X%X\n", i, b1, b2, b3, b4);
        }
    printf("\n************************\n");
}

```

The function "PrintReg" prints the data of general purpose registers from a lower to upper bound. The function takes two integers as parameters - one for lower and the other for upper bound.

```
    int i;
    printf("\n***********Register File*************\n");
    for(i=lb;i<ub;i++)
        printf("R[%d]: %X\n",i,reg[i].data);
    printf("\n************************\n");
}


```

The function "Hex2Bin" takes a hex value, converts it into 8-bit binary and stores it in a specified binary array. The function takes a hex value (as char) and the binary array (into which the binary equivalent of the hex values has to be stores) as an integer.

```
        int i=0,a,b,t=hex;

        while(hex!=0 && i<8)
        {
            bin[binSel].arr[i] = hex % 2;
            i++;
            hex /= 2;
        }
        if(t > 15)
        {
            int t0=bin[binSel].arr[0], t1=bin[binSel].arr[1],
             ↪  t2=bin[binSel].arr[2], t3=bin[binSel].arr[3];
            for(i=0;i<4;i++)
            {
                bin[binSel].arr[i]=bin[binSel].arr[4+i];
            }

            bin[binSel].arr[4]=t0;
            bin[binSel].arr[5]=t1;
```

```
            bin[binSel].arr[5]=t1;
            bin[binSel].arr[6]=t2;
            bin[binSel].arr[7]=t3;
        }
}

void TwosComp(int binSel)
```

The function "TwosComp" finds 2's complement of a given binary array and stores it in that array itself. The function accepts one integer as parameter which specifies the binary array whos 2's complement needs to be performed.

```

    int i,t,carry=0;
    for(i=0;i<8;i++)
    bin[binSel].arr[i] = !bin[binSel].arr[i];

    for(i=0;i<8;i++)
    {
        if(i==0)
            {
                t = carry + bin[binSel].arr[i] + 1;
                bin[binSel].arr[i] = bin[binSel].arr[i] ^ carry ^ 1;
            }
        else
            {
                t = carry + bin[binSel].arr[i];
                bin[binSel].arr[i] = bin[binSel].arr[i] ^ carry;
            }
        if(t<=1)
            carry = 0;
        else
            carry = 1;
    }

}

void Bin_Add(int a, int b, int c,int select,int withCarry)
```

The function "BinAdd" is used for performing binary addition of two numbers. It is used because when a arithmetic instruction is executed, the status flag bits Carry, Half Carry, Negative, Zero, Overflow and Signed needs to be set and reset

according to the operands and instruction. This is not possible by simply performing arithmetic operations two variables and we need to first convert the variables to binary equivalent and then perform arithmetic operations on them. For example if ADD instruction is executed, Reg31.data = Reg31.data + Reg30.data will only store the final sum but no status flag bits will be set.

The function takes 5 integers as parameters -

1. Parameter "a" specifies the first binary array (operand 1) for addition.
2. Parameter "b" specifies the first binary array (operand 2) for addition.
3. Parameter "c" specifies the binary array where the sum has to be stored.
4. Parameter "d" is to be set "1" only when one of the operands is in 2's complement form. (for subtraction) and "0" otherwise.
5. Parameter "e" is to be set "1" if addition is to be performed with carry and "0" if it has to be performed without carry.

```
158     int i,t;
159     for(i=0;i<8;i++)
160         {
161             if(i==0 && withCarry == 0)
162             {
163                 t = bin[b].arr[i] + bin[a].arr[i];
164                 bin[c].arr[i] = bin[a].arr[i] ^ bin[b].arr[i];
165             }
166             else
167             {
168                 t = bin[b].arr[i] + bin[a].arr[i] + SREG[0].data;
169                 bin[c].arr[i] = bin[a].arr[i] ^ bin[b].arr[i] ^
                     ↪  SREG[0].data;
170             }
171 
172             if(t<=1)
173             {
174                 SREG[0].data = 0;
175                 //printf("\nCarry bit is set\n");
176             }
177             else
178             {
179                 SREG[0].data = 1;
180                 //printf("\nCarry bit is set\n");
181                 if(i==3)
182                     {
183                         SREG[5].data = 1;
```

```
                        //printf("\nHalf Carry bit is set\n");
                    }
                else if(i==7)
                    {
                        SREG[2].data = 0;
                        //printf("\nNegative bit is reset\n");
                    }
            }
        }

    int t0=bin[c].arr[0], t1=bin[c].arr[1], t2=bin[c].arr[2],
    ↪  t3=bin[c].arr[3];
    for(i=0;i<4;i++)
    {
        bin[c].arr[i]=bin[c].arr[4+i];
    }

    bin[c].arr[4]=t0;
    bin[c].arr[5]=t1;
    bin[c].arr[6]=t2;
    bin[c].arr[7]=t3;

        if(select == 1)
        {
            SREG[5].data = !SREG[5].data;
            SREG[2].data = !SREG[0].data;
            SREG[0].data = !SREG[0].data;
        }
}


```

The function “SetPins” takes a hex value as parameter, converts it to 6-bit binary equivalent and sets the value of variables PB0 to PB5 (I/O pins of microcontroller) according to the binary equivalent. For example, if 0x5 is passed as parameter, its equivalent 6-bit binary is notation is 000101 and correspondingly, PB0=1, PB1=0,PB2=1,PB3=0,PB4=0 and PB5=0.

```
217     int bin[6]={0,0,0,0,0,0},i=0;
218     while(val!=0)
219     {
220         bin[i] = val % 2;
221         i++;
222         val /= 2;
223     }
224 
225     PB0 = bin[0];
226     PB1 = bin[1];
227     PB2 = bin[2];
228     PB3 = bin[3];
229     PB4 = bin[4];
230     PB5 = bin[5];
231 
232 }
233 
234 void MapToRam(int flag)
```

This function essentially copies the HEX file uploaded by use and stores it into "memory" structure inside C code which acts as RAM of the microcontroller. The function scans the user uploaded hex file character by character, converts it to hex (the hex file's contents are in hexadecimal notation but when C scans it, it scans it as character. For instance a 0xF in hex file is read as 70 instead of 15 by C code), then stores it in a 1 byte location in "memory" structure.

```
236     int i=0,filesize;
237     SetRam(0,size,0x0);
238     if(flag==1)
239     {
240         FILE *fptr;
241         unsigned char c;
242         fptr = fopen("hex.txt", "r");
243 
244         fseek(fptr, 0L, SEEK_END);
245         filesize = ftell(fptr);
246 
247         rewind(fptr);
248         c = fgetc(fptr);
249         while (c != EOF && i<filesize)
250         {
251             // to skip newline character in file
252             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);
    }
    for(i=0;i<8;i++)
        SREG[i].data = 0;

    if(debugMode==1)
        PrintRam(0,filesize+5);
}

void UpdateSreg()
```

The function “UpdateSreg” updates the value if Zero and Signed flag when called. It does not take any parameters.

```
    int i,t=0;
    for(i=0;i<8;i++)
    {
        if(bin[2].arr[i]==0)
                t++;
    }
    if(t == 8)
    {
        SREG[1].data = 1;
        //printf("\nZero bit is set\n");
```

```
292     }
293     else
294     {
295         SREG[1].data = 0;
296         //printf("\nZero bit is set\n");
297     }
298 
299     SREG[4].data = SREG[2].data ^ SREG[3].data;
300     //Signed bit is exor of Negative and Overflow bits
301     if(SREG[4].data == 1)
302     {
303         //printf("\nSigned bit is set\n");
304     }
305     else
306     {
307         //printf("\nSigned bit is reset\n");
308     }
309 
310 }
311 
312 void Compute()
```

The function "Compute" when called, gets the instruction to which the PC points, decodes the opcode via IF..ELSE ladder, performs the operations according to instruction and increments/decrements the PC for next clock cycle. This function performs the main computation.

```
312 void Compute()
313 {
314     int i,j,t;
315     unsigned b1=ram[PC+0x2].data, b2=ram[PC+0x3].data,
      ↪  b3=ram[PC].data, b4=ram[PC+0x1].data;
316     if (debugMode==1)
317     printf("instruction:%X%X%X%X\n",b1,b2,b3,b4);
318 
319     //ADD instruction added by Ashutosh Jha 6/3/2020
320     if(b1==0x0 && b2>=12 && b2<=15)
321     {
322         int a=reg[b3+16].data,b=reg[b4+16].data;
323         if(debugMode==1)
324         {
325             PrintReg(15,32);
326             printf("ADD instruction decoded\n");
327         }
```

```
329         ClearBins(0);
330         Hex2Bin(0,a);
331         ClearBins(1);
332         Hex2Bin(1,b);
333         ClearBins(2);
334 
335         Bin_Add(0,1,2,0,0);
336 
337         UpdateSreg();
338 
339         reg[b3+16].data += reg[b4+16].data;
340 
341         if(debugMode==1)
342         {
343             printf("\nAfter Operation - \n");
344             PrintReg(15,32);
345         }
346         PC += 0x4;
347     }
```

.
.
.

```
605     else if(b1==0xB && b2>8)
606     {
607         if(debugMode==1)
608             printf("OUT instruction decoded\n");
609         if(b4==0x8)                                    //Setting
            ↪  PORTB out pins
610             SetPins(reg[b3+16].data);                 //Setting DDRB
611         else if(b4==0x7)
612         {}
613         PC += 0x4;
614     }
```

.
.
.

```
    else if(b1==0x0 && b2==0x0)
    {
        printf("NOP instruction decoded");
        PC += 0x4;
    }
```

The function “output” is linked to VHDL and is called by it. Once it is called, it calls “output” function discussed above and that performs the main execution.

```
void output(int flag)
{
    if(flag == 1)
    {
        printf("\nPC: %X\n",PC);
        Compute();
        if(debugMode==1)
            PrintSREG();
    }
}
```

6.4 HELPER FUNCTION

Helper function takes variables from VHDL and converts them into a format understandable by C and vice versa. It also links functions defined in one language so that they can be used in another language just by calling it. Lines 4 - 33 are used to declare getter functions defined in C. These are used for sharing variables.

```
4  package ghdl_access is
5
6
7    -- Defines a pointer to an integer:
8  type int_access is access integer;
9
10
11 function get_ptr0 return int_access;
12     attribute foreign of get_ptr0 :
13         function is "VHPIDIRECT get_ptr0";
14
15 function get_ptr1 return int_access;
16     attribute foreign of get_ptr1 :
17         function is "VHPIDIRECT get_ptr1";
18
19 function get_ptr2 return int_access;
20     attribute foreign of get_ptr2 :
21         function is "VHPIDIRECT get_ptr2";
22
23 function get_ptr3 return int_access;
24     attribute foreign of get_ptr3 :
25         function is "VHPIDIRECT get_ptr3";
26
27 function get_ptr4 return int_access;
28     attribute foreign of get_ptr4 :
29         function is "VHPIDIRECT get_ptr4";
30
31 function get_ptr5 return int_access;
32     attribute foreign of get_ptr5 :
33         function is "VHPIDIRECT get_ptr5";
```

The function “output” and “MapToRam” is declared in lines 36 - 43. Note that in line 37 and 41, the type of parameter that each function takes is declared.

```
36    -- declaration of functions in C
37  procedure output(f : integer);
38      attribute foreign of output :
39              procedure is "VHPIDIRECT output";
40
41  procedure MapToRam(f : integer);
42      attribute foreign of MapToRam :
43              procedure is "VHPIDIRECT MapToRam";
```

The getters getptr0 to getptr5 were declared in lines 11 - 39 but these only return apointer to an integer. To store these values, variables are declared in lines 30 - 35.Therefore var0 to var5 equal to PB0 to PB5 variables defined in C code.

```
46    -- create variables aliased to the variable in C
47      shared variable var0 : int_access := get_ptr0;
48      shared variable var1 : int_access := get_ptr1;
49      shared variable var2 : int_access := get_ptr2;
50      shared variable var3 : int_access := get_ptr3;
51      shared variable var4 : int_access := get_ptr4;
52      shared variable var5 : int_access := get_ptr5;
53
54  end ghdl_access;
```

The body of this helper function is defined in lines 56 - 97.

```
56  package body ghdl_access is
57      function get_ptr0 return int_access is
58      begin
59          assert false report "VHPI" severity failure;
60      end get_ptr0;
61
62      function get_ptr1 return int_access is
63      begin
64          assert false report "VHPI" severity failure;
65      end get_ptr1;
66
67      function get_ptr2 return int_access is
68      begin
69          assert false report "VHPI" severity failure;
70      end get_ptr2;
71
72      function get_ptr3 return int_access is
```

```
73      begin
74          assert false report "VHPI" severity failure;
75      end get_ptr3;
76
77      function get_ptr4 return int_access is
78      begin
79          assert false report "VHPI" severity failure;
80      end get_ptr4;
81
82      function get_ptr5 return int_access is
83      begin
84          assert false report "VHPI" severity failure;
85      end get_ptr5;
86
87      procedure output(f : integer) is
88      begin
89          assert false report "VHPI" severity failure;
90      end output;
91
92      procedure MapToRam(f : integer) is
93      begin
94          assert false report "VHPI" severity failure;
95      end MapToRam;
96
97  end ghdl_access;
```

6.5 START_SERVER BASH FILE

In order to simulate C and VHDL codes together, their object files needs to be created and linked together first. This is done by "start_server" file. In this example the main VHDL filename is "Attiny85_.vhdl", the C code filename is "tiny85_c.c" and the helper function filename is "ghdl_access.vhdl". The procedure to link C and VHDL code is -

1. The C code has to be compiled and its object file has to be generated. This is done by line 13.

2. The main VHDL code and helper function have to be compiled and their object file has the be generated. This is done by line 16. Note that the object file of helper function ***has*** to be generated before or simultaneously with the main VHDL code's. This is because main VHDL file calls and uses the helper function and if it is compiled first, GHDL will give an error.

3. The object files of C and VHDL code have to be linked. Note that the filename of the linked object file should be that of original VHDL file i.e. in this

example, it should be “piso”. This is done by -

(a) Renaming object file of VHDL code from “piso.o” to “piso1.o”. This is done by line 19.

(b) Linking (combining) the object file of C code (“piso_c.o”) with renamed object file of VHDL code “piso1.o” and naming their linked object file as “piso.o”. This is done by line 20.

(c) “piso1.o” is not needed so is deleted by line 21.

```
5  cd
   ↪  /home/fossee/ngspice-nghdl/src/xspice/icm/ghdl/attiny_85_nghdl/DUTghdl/
6  chmod 775 sock_pkg_create.sh &&
7  ./sock_pkg_create.sh $1 $2 &&
8  ghdl -a sock_pkg.vhdl &&
9
10 ### The following lines (till line 23) are added by Ashutosh Jha
11 ### Date - 3/3/2020
12
13 gcc -c tiny85_c.c -o tiny85_c.o &&
14 # Compiles and generates object file of microcontroller C code
15
16 ghdl -a ghdl_access.vhdl attiny_85_nghdl.vhdl &&
17 # Compiles and generates object files of VHDL code of helper
   ↪  function and the main model respectively
18
19 mv attiny_85_nghdl.o attiny_85_nghdl1.o &&
20 ld -r -o attiny_85_nghdl.o tiny85_c.o attiny_85_nghdl1.o &&
21 rm attiny_85_nghdl1.o &&
22 # The object files of main VHDL and microcontroller C code need to
   ↪  be linked.
23 # The above three commands do that
24
25 ghdl -a attiny_85_nghdl_tb.vhdl  &&
26 ghdl -e -Wl,ghdlserver.o attiny_85_nghdl_tb &&
27 ./attiny_85_nghdl_tb
```

6.6 STEPS FOR IMPLEMENTATION

It is advisable to refer section 8 on page 58 once before proceeding for better understanding.

6.6.1 PISO

1. Upload the ***"piso.vhdl"*** via NGHDL. The PISO component should be added in NGHDL library and its DUTghdl folder should be created after a successful upload.
2. Create the PISO schematic as shown in 10 on page 15. Annotate it and generate netlist (No need for ERC check).
3. In Kicad to Ngspice conversion, add time parameters, source details and generate.
4. Copy the ***"piso_c.c"***, ***"ghdl_access.vhdl"*** and ***"start_server.sh"*** files into the DUTghdl folder.
5. Run simulation normally via eSim. Errors (if any) will be displayed in console window and can be rectified accordingly.

OR

Upload "piso.vhdl" via NGHDl, open the PISO project folder in eSim, perform Step 4 mentioned above and simulate normally. Refer section 9 on page 60 for the resources.

6.6.2 ATTINY85

1. Upload the ***"Attiny85_nghdl.vhdl"*** via NGHDL. The attiny85 component should be added in NGHDL library and its DUTghdl folder should be created after a successful upload.
2. Create the attiny85 schematic as shown in figure 20. Annotate it and generate netlist (No need for ERC check).
3. In Kicad to Ngspice conversion, add time parameters, source details and generate.
4. Copy the ***"tiny85_c.c"***, ***"ghdl_access.vhdl"***, ***"start_server.sh"*** and ***"hex.txt"*** files into the DUTghdl folder.
5. Run simulation normally via eSim. Errors (if any) will be displayed in console window and can be rectified accordingly.

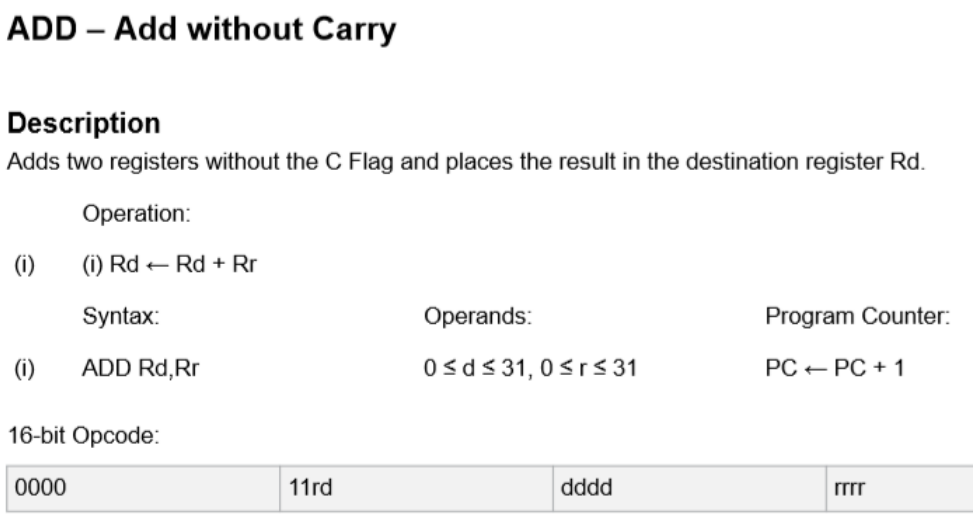
OR

Upload "Attiny85_nghdl.vhdl" via NGHDl, open the tiny85-test project folder in eSim, perform Step 4 mentioned above and simulate normally. Refer section 9 on page 60 for the resources.

6.6.3 INSTRUCTIONS

In order to implement a new instruction, we have to write its logic and append it in the IF..ELSE ladder of "Computation" function in C code discussed in section 6.3 on page 46. The SOP for implementing any new instruction is given below by taking "ADD" instruction as an example -

1. First of all, we need to understand the functionality and OPCODE of the instruction from *AVR instruction set.* The information given for ADD instruction is shown below for reference -



ADD – Add without Carry

Description

Adds two registers without the C Flag and places the result in the destination register Rd.

Operation:

(i) (i) Rd ← Rd + Rr

	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

0000	11rd	dddd	rrrr

Figure 22: *ADD instruction*

2. Now we need to find its OPCODE for recognizing it in the IF...ELSE ladder. From figure 22 we can see its specified 16-bit OPCODE. Out of these 16-bits (separated into 4 nibbles - b1,b2,b3,b4) we have to compare only those bits in the IF...ELSE ladder which remain constant no matter what operands are present in the instruction. For example in the ADD instruction b1 always stays "0000" = 0x0 so b1 nibble can be used.

   Also, the first two bits (MSB) of b2 are always "11" and the last two bits (LSB) can take any value from "00" to "11" according the operand present in the instruction. Thus b2 can take values from "1100" to "1111" or 0xC to 0xF. Thus we can use b2 >= 12 and b2 <= 15 too while checking for ADD instruction in the IF...ELSE ladder. We add this in the "Compute" function as follows -

```
319     //ADD instruction added by Ashutosh Jha 6/3/2020
320     if(b1==0x0 && b2>=12 && b2<=15)
```

3. Next, we need to understand what the instruction does functionally, which register it affects, what operations it performs ,etc. For ADD instruction we

see that it contains two operands - location of registers Rd and Rr. It adds the contents of these two registers and stores them in register Rd. Rd is selected by b3 and Rr by b4. Note that since b3 and b4 are 4-bit values, Rd and Rr can range from 0 to 16. Attiny85 has 32 general purpose registers (from R0 to R31).

Note that ADD instruction can address registers R16 to R31 only. Which means when b3 or b4 = "0000" or 0x0, it does not mean Rd or Rr is R0. Instead, it means that Rd or Rr = R16. Essentially, the zero value of Rd and Rr starts from R16 and not R0. Thus, when b3 = "0001" or 0x1, it means Rd = R17 and not R1. We add this in "Compute" function as follows -

```
321     {
322         int a=reg[b3+16].data,b=reg[b4+16].data;
323         if(debugMode==1)
324         {
325             PrintReg(15,32);
326             printf("ADD instruction decoded\n");
327         }
328 
329         ClearBins(0);
330         Hex2Bin(0,a);
331         ClearBins(1);
332         Hex2Bin(1,b);
333         ClearBins(2);
334 
335         Bin_Add(0,1,2,0,0);
336 
337         UpdateSreg();
338 
339         reg[b3+16].data += reg[b4+16].data;
340 
341         if(debugMode==1)
342         {
343             printf("\nAfter Operation - \n");
344             PrintReg(15,32);
345         }
```

- The data of Rd is stored in variable a and that of Rr is stored in variable b. ([b3+16] is done because the zero of registers starts with R16 and not R0).
- The hex values stored in Rd and Rr (which are in variables a and b) are converted to 8-bit binary values by lines 329 - 332. Then these binary values are added via binary addition in line 335 and Status register is updated. The reason for converting to decimal then adding instead of directly adding them is explained in section 6.3 on page 41.

- Now that the status flag bits are set/reset accordingly, we need to store the sum of data contents of Rd an Rr in location of Rd. This is done by line 339.

4. Final step is to increment/decrement the value of PC for next cycle according to the instruction. From figure 22 we see the value of PC increments by 1 instruction (4 nibbles because of 16 bit OPCODE) after execution, so we increment PC by 4 in the line -

```
346         PC += 0x4;
347     }
```

7 SCALING UP THE ARCHITECTURE

The implemented framework is tested and initially aimed at relatively simpler 8-bit microcontrollers, but almost all commonly used micrcocontrollers (including but not limited to ATtiny series, ATMEGA series, PIC series, 8051, 6502) are to be implemented using the framework in the future. This framework can also be used to implement higher level mircocontrollers, microprocessors and even SoCs without changing the framework (only the helper function and C codes need to be modified to facilitate the corresponding logic).

When implementing new microcontrollers, the SOP for adding new instruction will be same as that of ATtiny85 mentioned in section 6.6.3 on page 54.

Issues w.r.t. porting -

- The C, VHDL, helper function and bash file codes only require a simple text editor to be written and modified. So their creation and modification is possible on all platforms without any issues.
- The header files used in C code ("math.h" and "stdio.h") are platform independent and will not cause any problem while execution on any platform.
- AVR GCC [3] is an open source compiler which is used to convert user made C code to hex code. This is not integrated in eSim at the moment as the user is intended to directly upload hex file (compiling C to hex himself/herself). But may be integrated inside eSim in the future - which will allow user to directly upload the C code.
- The bash file and the commands would needed to be modified according to the platform. The PISO (section 4.3.1, page 21) and ATtiny85 (section 6.5, page 51) examples were implemented and tested on UBUNTU version 16.04, so the bash commands used are for linux distributions. If it has to be ported to Windows, linux commands are to be replaced with those which are supported in windows.
- The most crucial component of this framework which is essentially its bottleneck too is NGHDL. This framework will work flawlessly (with minor changes mentioned in the point above) in all the platforms that NGHDL will be able to function properly.

8 WORKFLOW

Steps to simulate an example using the proposed framework is given as follows -

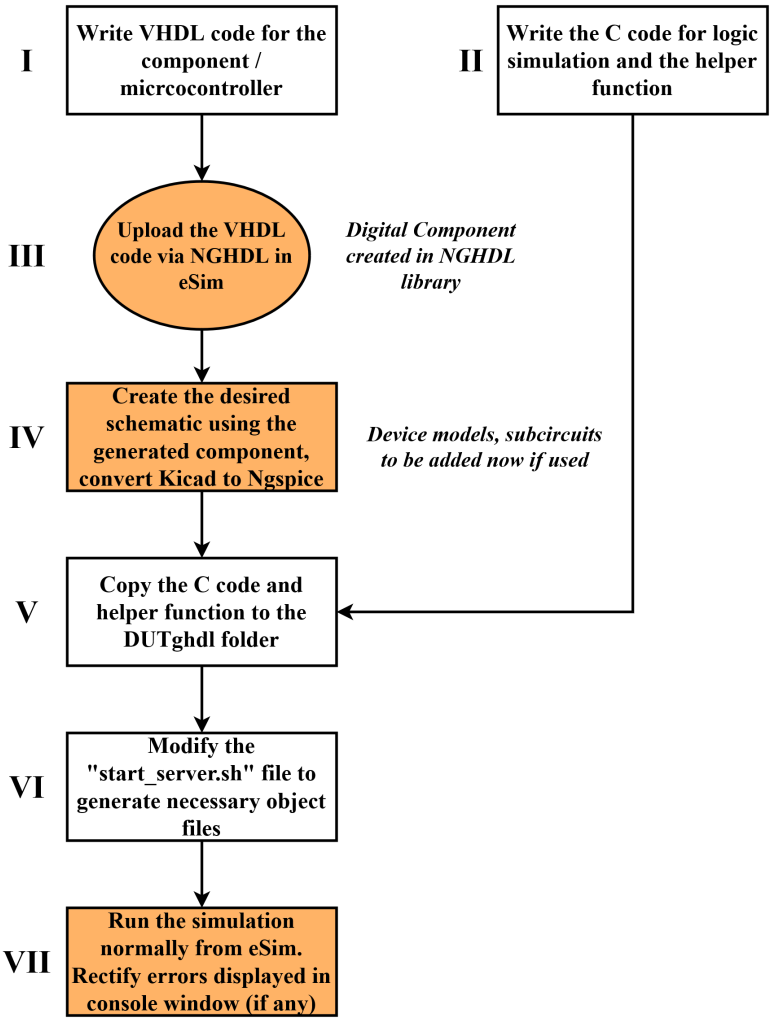


Figure 23: *Workflow*

1. The digital component or microcontroller that has to be implemented is written in VHDL. All the port configurations, when to call which functions from C is specified along with any additional processing (if required but not recommended).

2. The C code (for logic simulation) is written and the helper function is also written (in VHDL). Before copying them to DUTghdl folder in Step 5, it is recommended to compile them once for finding syntax errors.

3. The VHDL code describing the component is then uploaded into eSim via NGHDL. This creates a model of desired component in "NGHDL" library.

4. The schematic (circuit) has to be created in "eeschema" using eSim normally. It has to be annotated and netlist has to be created. Then time paramters,

source details, device models and subcircuits have to be uploaded like normal operation.

5. The C code and helper function created in Step 2 has to be copied into the DUTghdl folder. This step can be performed anytime before Step 6.

NOTE If microcontroller has to be simulated, the hex code should also be copied into the DUTghdl folder. The filename of hex code should be strictly kept as "hex.txt" and the instructions regarding hex code are provided in section 5.3.1 on page 28.

6. The "start_server.sh" file is automatically created via NGHDL when Step 3 is performed. We need to modify its contents so that it generates the object files for C code, helper function and link the VHDL and C codes together. Its details are given in section 4.3.1 on page 21 and section 6.5 on page 51.

7. Finally, simulation needs to be started normally via eSim. If any errors will be shown in console window and can be traced and rectified (if any) .

9 RESOURCES

The resources are uploaded on GITHUB in “nghdl” repository, “attiny-alpha” branch. The base repository is *FOSSEE/nghdl*. Users an download the PISO and ATtiny85 demo sketches which are explained in this documentation from the “Demo” folder.

Users can also access *this* repository for the latest resources and code version.

NOTE -

1. This is still in testing phase and is not released with official eSim releases nor will it be released anytime soon in the future, so there is no guarantee for accuracy.
2. eSim and NGHDL are open source - free for all software but prior permission is required from the necessary authorities before modification. Please refer the license policy on FOSSEE repository first.

A MNEMONICS AND OPCODES

References

[1] *Arduino Official website*. 2020. URL: https://www.arduino.cc/.

[2] *Autodesk Official website*. 2020. URL: https://www.autodesk.in/.

[3] *AVR GCC documentation*. 2020. URL: https://www.nongnu.org/avr-libc/.

[4] Greg Novick Crystal Chen and Kirk Shimano. *RISC architecture Breif information*. 2020. URL: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html.

[5] *eSim Official website*. 2020. URL: https://esim.fossee.in/.

[6] Thomas Fischl. *Arduino Ide Official Site*. 2020. URL: https://www.arduino.cc/.

[7] Thomas Fischl. *AVR ISP Mk2 Manual*. 2020. URL: https://www.olimex.com/Products/AVR/Programmers/AVR-ISP-MK2/resources/AVR-ISP-MK2.pdf.

[8] Thomas Fischl. *USB ASP official site*. 2020. URL: https://www.fischl.de/usbasp/.

[9] *Harvard Architecture Wikipedia link*. 2020. URL: https://en.wikipedia.org/wiki/Harvard_architecture.

[10] *MCUSim Official website*. 2020. URL: https://trac.mcusim.org/.

[11] *Microchip Official website*. 2020. URL: https://www.microchip.com/.

[12] *Proteus Official website*. 2020. URL: https://www.labcenter.com/.

[13] *PSPICE Official website*. 2020. URL: https://www.pspice.com/.

[14] *SOLIDWORKS Official website*. 2020. URL: https://www.solidworks.com/.