Summer Fellowship Report

On

Implementation of ATtiny Family of Micro-controllers

Submitted by

Sumanto Kar

Vadisa Yamini

Shubangi Mahajan

Under the guidance of

Prof.Kannan M. Moudgalya
Chemical Engineering Department
IIT Bombay

June 17, 2020

Acknowledgment

We would like to express our gratefulness to the FOSSEE team for giving us a wonderful experience. When the world had broken down during the COVID-19 lock-down, the FOSSEE team bore us a wonderful opportunity to serve our country. It also gave us a fruitful experience of "work from home" during the time of lock-down which we might not have got from anywhere else.

We would like to express our gratefulness to Prof. Kannan M. Moudgalya for his valuable and constructive suggestions during the planning and development of this research work. His willingness to give his time so generously and being enthusiastic all the time has been very much appreciated. His farsightnedness made us feel many impossible things possible which increased our enthusiasm to a great extent.

We would also like to express our gratitude towards Prof. Madhav Desai for showing us a clear path to the solutions where we were stuck and found ourselves helpless. This not only increased our spirit but also helped us to regain our confidence. We would like to thank the eSim team for helping us and providing us all the resources required to work with and guiding us all throughout the project. We would like to thank our mentors Mr. Saurabh Bansode, Mr. Rahul Paknikar, Mrs. Gloria Nandihal, Mrs. Usha Vishwanathan and whole team for imparting immense of knowledge and for enhancing various technical skills in us which would definitely help us in near future. Their presence and their hard work for us day and night had made our fellowship experience blissful. A special thanks to Mr. Ashutosh Jha for sharing his knowledge with us and helping us whenever we were in need. His helping nature is really to be appreciated.

We would utilize everything we got from here for our career growth as well as for the betterment of our society. The wonderful experiences which we took from here while the world had been passing through a crisis would be memorable throughout our lives.

Contents

Chapter 1

Introduction

FOSSEE (Free/Libre and Open Source Software for Education) project promotes the use of FLOSS tools to improve the quality of education in our country. It aims to reduce dependency on proprietary software in educational institutions. It encourages the use of FLOSS tools through various activities to ensure commercial software is replaced by equivalent FLOSS tools. It also develops new FLOSS tools and upgrade existing tools to meet requirements in academia and research.[1]

The FOSSEE project is part of the National Mission on Education through Information and Communication Technology (ICT), Ministry of Human Resource Development (MHRD), Government of India.

Microcontroller contains on chip central processing unit (CPU), Read only memory (ROM), Random access memory (RAM), input/output unit, interrupts controller etc. Therefore a microcontroller is used for high speed signal processing operation inside an embedded system. It acts as major component used in designing of an embedded system.[2]

AVR is a family of microcontrollers developed since 1996 by Atmel, acquired by Microchip Technology in 2016. These are modified Harvard architecture 8-bit RISC 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.[2]

Some mostly used AVR microcontrollers are:-

- ATtiny and family of microcontrollers
- ATmega8 microcontroller
- ATmega16 microcontroller
- ATmega32 microcontroller
- ATmega328 microcontroller

Our project deals with the implementation of ATtiny microcontrollers in eSim. The ATtiny25/45/85 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATtiny25/45/85 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.[4]

eSim is a free/libre and open source EDA tool for circuit design, simula- tion, analysis and PCB design developed by FOSSEE, IIT Bombay. 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.[5]

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.

In this project, the ATtiny x5 family of microcontrollers is implemented using NGHDL (mixed mode simulation feature of eSim). With NGHDL one can define new digital models by writing a vhdl code and do mixed mode simulation using ngspice.Thus this feauture of eSim is used to efficiently describe the behavior of the Attiny microcontrollers.

Chapter 2

Problem Statement

Implementing the ATtiny x5 family of microcontrollers in eSim using NGHDL module present already in eSim so that the user can easily simulate microcontroller based projects by uploading a HEX Code of the C file written by compilers like AVR-GCC, Arduino IDE, etc.

2.1 Framework Developed

The framework which was already developed for the micro-controller implementation in eSim is to design the the layout and peripherals of the micro-controller (pin configurations) in VHDL language and simulating its internal processing in C language and co-simulating them (by linking them together) during run time. This method is selected to be most optimal because we already have NGHDL module in eSim by which we create new models by writing its functionality in a VHDL code and utilizes the best parts of the two programming languages C (ease and flexibility of writing)and VHDL(ease of defining a digital component).

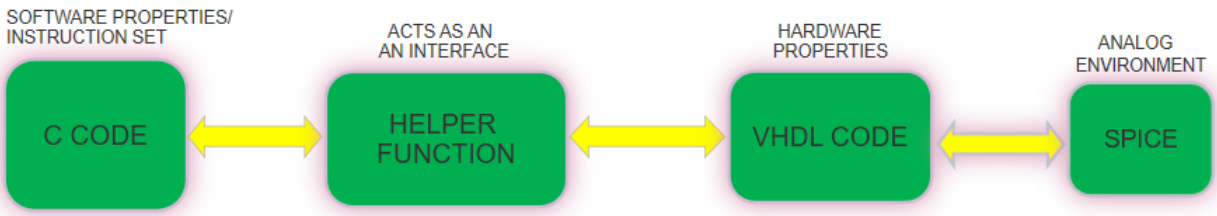

Figure 2.1: Framework Developed

2.2 Approach

As the framework to implement the micro-controller is already designed, the next step was to write the internal processing of the micro-controllers in C code. Thus 66 instructions of the ATtiny family micro-controllers were implemented by us in the C code(while some were implemented earlier) by the following methodology:

- Each instruction is identified by its opcode by using a if else ladder and the functionality of the instruction is defined in that code.
- At the end of each instruction the PC value is incremented or jumped according to the instruction working.
- Each instruction is tested after implementation in eSim by giving its opcode to the hex file and edited if changes required

Modules like timer 0 and timer 1 were also implemented by following method:

- Define the functionality in C file as a function.
- Test the implemented modes by verifying the timer waveform and the output frequency with the standard frequency formula defined for each mode .

Chapter 3

Software Requirements

The above architecture depends upon some software. These software provide a platform for our proposed implementation. The software used are listed below:

3.1 NGHDL

Ngspice supports mixed mode simulation. It can simulate both digital and analog components.

Ngspice has something called code-model which defines the behavior of your component and can be used in the netlist. For example you can create a full-adder's code-model in Ngspice and use it in any circuit netlist of Ngspice.[8]

Now the question is if we already have digital model creation in Ngspice, then why this interfacing?

Well, in Ngspice, it is difficult to write your own digital code-models. Though, many people are familiar with GHDL and can easily write the VHDL code. So the idea of interfacing is just to write VHDL code for a model and use it as a dummy model in Ngspice. Thus, whenever Ngspice looks for that model, it will actually call GHDL to get the results. GHDL's foreign language interface is used for this inter-process communication.

This framework is tried and tested, and gives accurate simulation results. Also NGHDL is only available for Ubuntu v16.04 as of 30th May,2020.[8]

3.2 Linux Commands

As of now NGHDL is based on the Ubuntu operating system. Therefore, in order to work with the NGHDL, one must be very much familiar with the Linux commands. Here in the proposed implementation as well the linux commands were used exhaustively. As of now NGHDL is compatible with Ubuntu 16.04-18.04.

3.3 AVR GCC

AVR GCC 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.

3.4 eSim

As discussed earier, eSim (previously known as Oscad / FreeEDA) is a free/libre and open source EDA tool for circuit design, simulation, analysis and PCB design. The proposed implementation is carried out using eSim in order to build the circuit using the blocks generated by NGHDL and the components already present in the eSim. The circuit to be tested and simulated is build in the EEschema. The circuit is then converted to Ngspice using KiCAD to Ngspice Converter. The simulation is carried out using the NGHDL feature of eSim.

Chapter 4

ATtiny x5 Family of Microcontrollers

The ATtiny25/45/85 is a low-power CMOS 8-bit microcontroller based on the AVR enhanced RISC architecture. By executing powerful instructions in a single clock cycle, the ATtiny25/45/85 achieves throughputs approaching 1 MIPS per MHz allowing the system designer to optimize power consumption versus processing speed.[4]

4.1 Pin Configuration

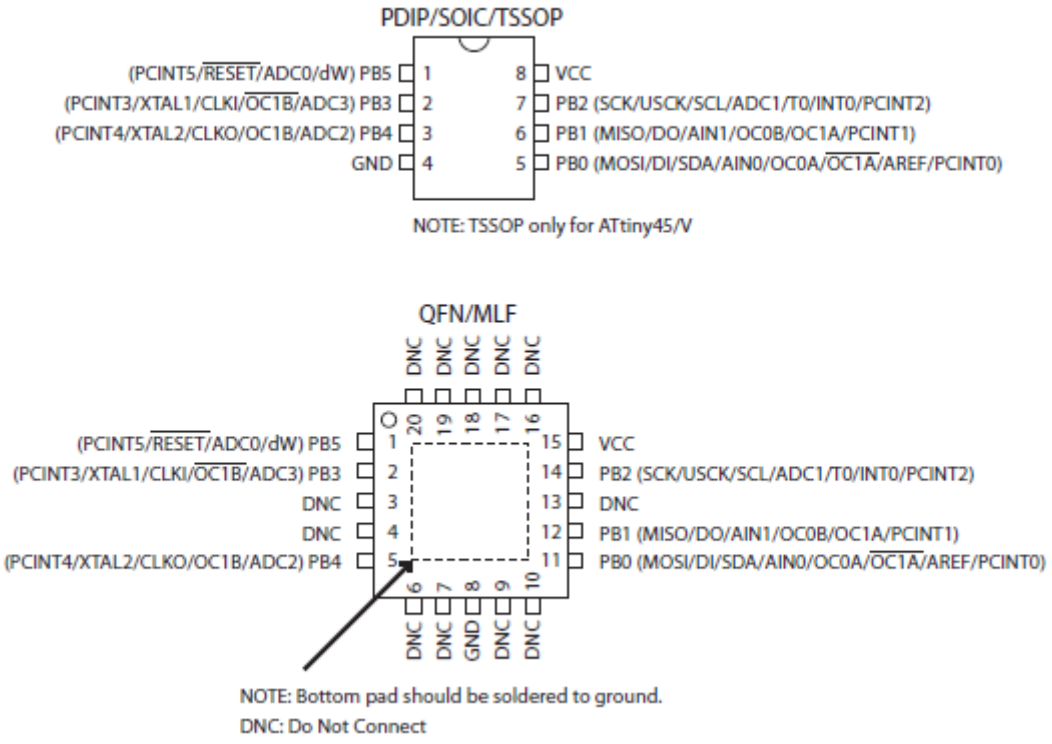

Figure 4.1: Pin Configuration

4.2 Block Diagram

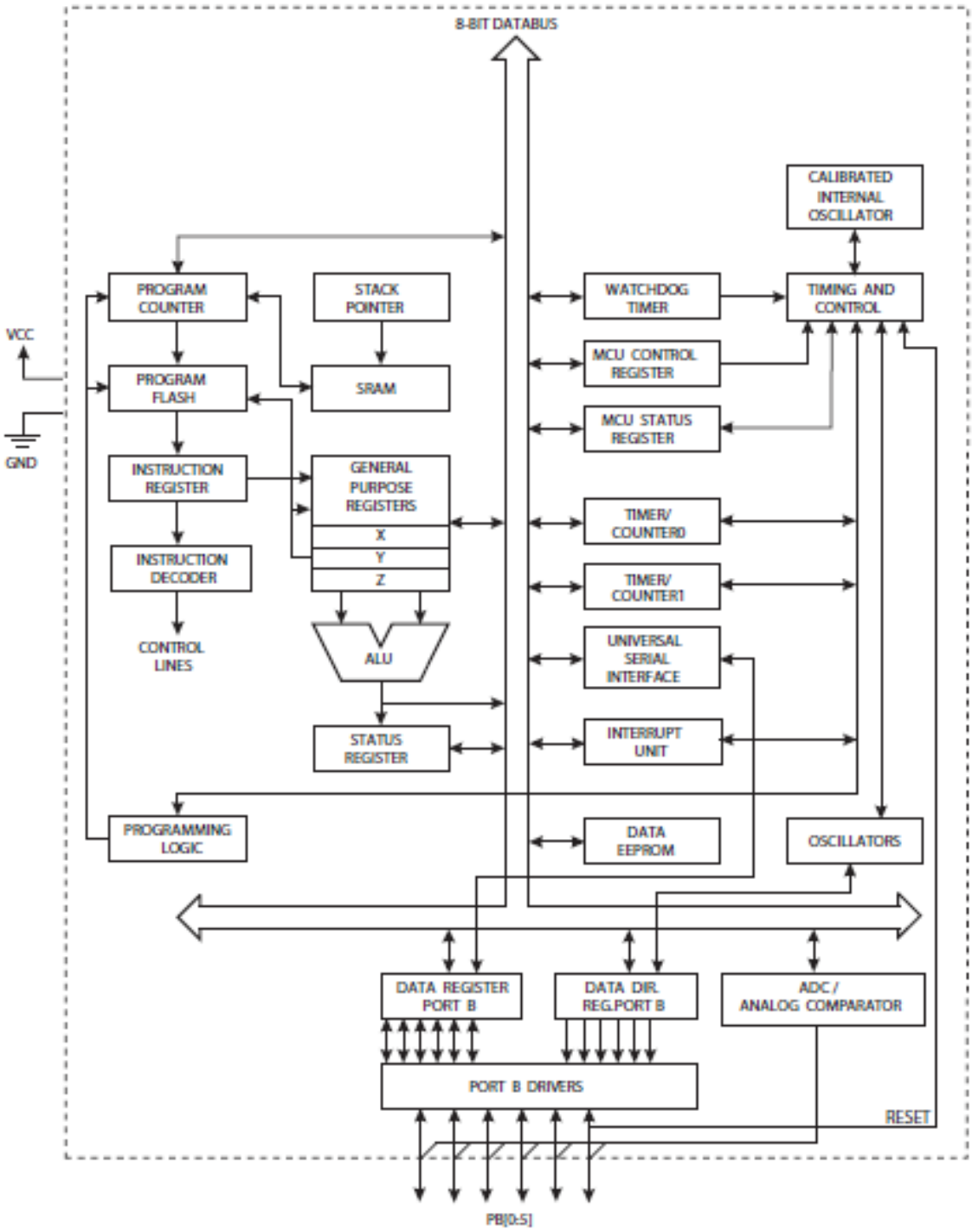

Figure 4.2: Block Diagram

The AVR core combines a rich instruction set with 32 general purpose working registers. All 32 registers are directly connected to the Arithmetic Logic Unit (ALU), allowing two independent registers to be accessed in one single instruction executed in one clock cycle. The resulting architecture is more code efficient while achieving throughputs up to ten times faster than conventional CISC microcontrollers.

4.3 ATtiny Memories

4.3.1 In-System Re-programmable Flash Program Memory

The ATtiny25/45/85 contains 2/4/8K bytes On-chip In-System Reprogrammable Flash memory for program storage. Since all AVR instructions are 16 or 32 bits wide, the Flash is organized as 1024/2048/4096 x 16.

4.3.2 SRAM Data Memory

The lower 224/352/607 Data memory locations address both the Register File, the I/O memory and the internal data SRAM. The first 32 locations address the Register File, the next 64 locations the standard I/O memory, and the last 128/256/512 locations address the internal data SRAM.

The five different addressing modes for the Data memory cover: Direct, Indirect with Displacement, Indirect, Indirect with Pre-decrement, and Indirect with Post-increment. In the Register File, registers R26 to R31 feature the indirect addressing pointer registers. The direct addressing reaches the entire data space. The Indirect with Displacement mode reaches 63 address locations from the base address given by the Y- or Zregister.

4.3.3 EEPROM Data Memory

The ATtiny25/45/85 contains 128/256/512 bytes of data EEPROM memory. It is organized as a separate data space, in which single bytes can be read and written. The EEPROM has an endurance of at least 100,000 write/erase cycles. The access between the EEPROM and the CPU is described in the following, specifying the EEPROM Address Registers, the EEPROM Data Register, and the EEPROM Control Register.

4.3.4 I/O Memory

All ATtiny25/45/85 I/Os and peripherals are placed in the I/O space. All I/O locations may be accessed by the LD/LDS/LDD and ST/STS/STD instructions, transferring data between the 32 general purpose working registers and the I/O space. I/O Registers within the address range 0x00 - 0x1F are directly bit-accessible using the SBI and CBI instructions. In these registers, the value of single bits can be checked by using the SBIS and SBIC instructions. Refer to the instruction set

section for more details. When using the I/O specific commands IN and OUT, the I/O addresses 0x00 - 0x3F must be used. When addressing I/O Registers as data space using LD and ST instructions, 0x20 must be added to these addresses.

Chapter 5

Implementation of Instruction Set

5.1 C Code

The following has been used and defined before executing the instructions:

```
int debugMode=1;
int PB0,PB1,PB2,PB3,PB4,PB5,wait_Clocks=0;
int PC = 0;
int SP = 512;
```

Debug Mode – It will show the execution of the instructions steps and the updated SREG (Status Register).

PB0-PB5 – The INOUT ports.

PC- Program counter. A program counter is a register that contains the addr (location) of the instruction being executed at the current time. As each instruction gets fetched, the program counter increases its stored value by 2.

SP- Stack Pointer. The Stack is mainly used for storing temporary data, like local variables and return addresses after interrupts and subroutine calls.The Stack Pointer Register always points to the top of the Stack. The Stack is filled from higher memory locations to lower memory locations that is from bottom to top therefore, SP is initialised as 512 which is the bottom most location. This implies that a Stack PUSH command will decrease the SP and PULL will increase the SP.

```
struct memory//Structure to store RAM and other registers
{
        unsigned char data;
}prog_mem[size],GPR[32],SREG[8],IOREG[64],SRAM[512];

struct BinArrays
{
```

```
    int arr[8];
}bin[3];
```

Struct Memory–It is the structure defined for storing the Program Memory (prog_mem), General Purpose Registers (GPR), Status Registers (SREG), I/O Registers (IOREG), Internal RAM (SRAM). The size of all the registers is 8 bit as they are defined by unsigned char.

Struct BinArray–It is the structure to define Binary arrays, there are three BinsArray of size 8 bit. These are defined for temporary use and for ease of writing code.

Few more functions are defined before the instructions are performed. They are:

- clearbins(to clear bin arrays),
- Hex2Bin(to convert hex to binary, this makes use of BinArrays),
- Maptoram (to map to external hex file),
- Void input
- Timer
- Setpins(to set pins PB0-PB5 for displaying output)
- Timer1
- Void compute(to perform main computations under which all instructions are defined)

The 16 bit opcode of the instructions is represented by b1,b2,b3,b4 (4 bits each) which are used in the below code . For example if the opcode is -1001 1100 0110 0001 then b1 is 1001, b2 is 1100, b3 is 0110 and b4 is 0001. Instructions are decoded by the opcodes and executed.

5.1.1 CP – Compare

```
else if(b1==0x1 && b2>=4 && b2<=7)
        {
                if(debugMode==1)
            printf("\nCP instruction decoded\n");
            //Comparing Rd and Rr (Reg data doesn't have to be modified)
            unsigned char r=((b2>>1)&1)*16+b4,
            ↪  d=(b2&1)*16+b3;
        if(debugMode == 1)
                printf("\n%X - %X = %d\n",GPR[d].data,GPR[r].data,GPR[d].data
                ↪  -GPR[r].data);
```

```
    //UPDATE FLAGS
    if(GPR[d].data < (GPR[r].data))
            SREG[0].data = 1;// carry flag
    else
            SREG[0].data = 0;

    if(GPR[d].data - GPR[r].data == 0)
            SREG[1].data = 1;//zero flag
    else
        SREG[1].data=0;

    unsigned char dl=GPR[d].data & 0x80;
    unsigned char rl=GPR[r].data & 0x80;
unsigned char fl=(GPR[d].data - GPR[r].data) & 0x80;
    if (fl==0x80)// negative flag
        SREG[2].data=1;
    else
        SREG[2].data=0;

    unsigned char f=(GPR[d].data & 0xf);//half carry flag
    unsigned char g=(GPR[r].data & 0xf);
    printf("%X,%X",f,g);
    if (f<g)
        SREG[5].data=1;
    else
        SREG[5].data=0;//overflow flag
    if(dl==0 && rl==0x80 && fl==0x80)
        SREG[3].data=1;
    else if(dl==0x80 && rl==0 && fl==0)
        SREG[3].data=1;
    else
        SREG[3].data=0;

    SREG[4].data=SREG[3].data ^ SREG[2].data;//signed flag

    PC += 0x2;
}
```

This instruction performs a compare between two registers Rd and Rr. None of the registers are changed. All conditional branches can be used after this instruction.[11] In this code, Rr and Rd have been defined as per the opcode and then compared. After that, the status register flags are updated. At the end PC is incremented by two to execute the instruction at next location.

5.1.2 CPC – Compare with Carry

```
else if(b1==0x0 && b2>=4)
        {
            if(debugMode==1)
            printf("\nCPC instruction decoded\n");
        unsigned char r=((b2>>1)&1)*16+b4;//setting r and d as per opcode
            unsigned char d=(b2&1)*16+b3;
            unsigned char c=SREG[0].data;

            if(debugMode == 1)
            printf("\n%X - %X -%X = %d\n",GPR[d].data,GPR[r].data,c,GPR[d].data
            ↪  -GPR[r].data- c);

            //UPDATE FLAGS
            if(GPR[d].data < (GPR[r].data + c))// carry flag
                    SREG[0].data = 1;
            else
                    SREG[0].data = 0;

            if(GPR[d].data - GPR[r].data - c == 0)//unchanged zero flag
                    SREG[1].data = SREG[1].data;
            else
                SREG[1].data=0;

            unsigned char dl=GPR[d].data & 0x80;
            unsigned char rl=GPR[r].data & 0x80;
            unsigned char fl=(GPR[d].data - GPR[r].data - c) & 0x80;
            if (fl== 0x80)// negative flag
                SREG[2].data=1;
            else
                SREG[2].data=0;

            unsigned char f=GPR[d].data & 0xf;//half carry flag
            unsigned char g=GPR[r].data & 0xf;
            if (f<(g+c))
                SREG[5].data=1;
            else
                SREG[5].data=0;//overflow flag
            if(dl==0 && rl==0x80 && fl==0x80)
                SREG[3].data=1;
            else if(dl==0x80 && rl==0 && fl==0)
                SREG[3].data=1;
            else
                SREG[3].data=0;
```

```
            SREG[4].data=SREG[3].data ^ SREG[2].data;//signed flag

            PC += 0x2;
        }
```

This instruction performs a compare between two registers Rd and Rr and also takes into account the previous carry. None of the registers are changed. All conditional branches can be used after this instruction.[11] In this code, Rr , Rd and carry flag have been defined as per the opcode and then compared. After that, the status register flags are updated. Then PC is incremented by 2 to execute next instruction.

5.1.3 CPI – Compare with Immediate

```
else if(b1==0x03)
{
                    if(debugMode==1)
            printf("\n CPI instruction decoded\n");
                    unsigned char k = b2*16 + b4;//setting k reg as per opcode
                    if(debugMode == 1)
                    printf("\n%X - %X = %d\n",GPR[b3].data,k,GPR[b3].data - k);

                //UPDATE FLAGS

                    if(GPR[b3].data == k)// zero flag
                         SREG[1].data = 1;
                   else if(GPR[b3].data != k)
                            SREG[1].data = 0;

            if(GPR[b3].data < k)// carry flag
                     SREG[0].data = 1;
        else
                     SREG[0].data = 0;

            unsigned char rl=GPR[b3].data & 0x80;
            unsigned char kl=k & 0x80;
            unsigned char fl=GPR[b3].data-k & 0x80;
            if (fl==0x80)// negative flag
                        SREG[2].data=1;
            else
                        SREG[2].data=0;
            unsigned char f=GPR[b3].data & 0xf;//half carry flag
        k= k & 0xf;
            if (f<k)
                        SREG[5].data=1;
```

```
            else
                        SREG[5].data=0;//overflow flag
            if(rl==0 && kl==0x80 && fl==0x80)
                        SREG[3].data=1;
            else if(rl==0x80 && kl==0 && fl==0)
                        SREG[3].data=1;
            else
                        SREG[3].data=0;

            SREG[4].data=SREG[3].data ^ SREG[2].data;//signed flag

            PC += 0x2;
}
```

This instruction performs a compare between register Rd and a constant. The register is not changed. All conditional branches can be used after this instruction. In the code we perform compare and then set the status register flags after it.Then PC is incremented by 2 to execute next instruction.

5.1.4 ICALL – Indirect Call to Subroutines

```
else if(b1==0x9 && b2==0x5 && b3==0 && b4==0x9)
        {
        if(debugMode==1)
                    printf("\nICALL instruction decoded\n");

                int k=PC+2;
                SRAM[SP-1].data=k;//lower 8 bits
                SRAM[SP].data=k>>8;//upper 8 bits

                SP -= 2;

                PC = GPR[30].data + GPR[31].data*256;// Z pointer register
        }
```

Calls to a subroutine within the Program memory. The return address (to the instruction after the CALL) will be stored onto the Stack. The Stack Pointer uses a post-decrement scheme during CALL. In the code, pc+2 (pc of the next instruction) is given to k. The upper and lower bits of k are given to the stack as pointed by the stack pointer , SP. Later the SP is decremented by 2 because we filled stack by two addresses (Stack is filled from higher memory locations to lower memory locations). Now it points to the top of the stack. Then we call the subroutine as given by the z pointer register.

5.1.5 IJMP – Indirect Jump

```
         else if(b1==0x9 && b2==0x4 && b3==0 && b4==0x9)
        {
        if(debugMode==1)
                    printf("\nIJMP instruction decoded\n");

                PC = GPR[30].data + GPR[31].data*256;// Z pointer register
}
```

Indirect jump to the address pointed to by the Z (16 bits) Pointer Register in the Register File. The Z- pointer Register is 16 bits wide (Register 30 and 31 in register file).Then PC is incremented by 2 to execute next instruction.

5.1.6 SER – Set all Bits in Register

```
else if(b1==0xD && b2==0xF && b4==0xF)
    {
        if(debugMode==1){
                    printf("SER instruction decoded\n");
            printf("\nBefore execution: Reg[%d] = %X\n",b3,GPR[b3].data);
            }
                GPR[b3].data = 0xFF;// loading in Rd
                if(debugMode==1)
                printf("\nAfter execution: Reg[%d] = %X\n",b3,GPR[b3].data);
                PC += 0x2;
    }
```

Loads 0xFF directly to register Rd. In the code , Rd is given by GPR[b3]. Then PC is incremented by 2 to execute next instruction.

5.1.7 SBI – Set Bit in I/O Register

```
else if(b1==0x9 && b2==0xA)
        {
                char b,bits;
            if (b4>7)
                        b=b4-8;
            else
                        b=b4;
            char A= b3*2 + (b4>>3);
```

```
            if(debugMode==1){
                printf("\nSBI instruction decoded\n");
                    printf("\nBefore execution: Reg[%d] =
                    ↪  %X\n",A,IOREG[A].data);
                    }
        if(b==0)
                    bits=1;
        else if(b==1)
                    bits=2;
        else if(b==2)
                    bits=4;
        else if(b==3)
                    bits=8;
        else if(b==4)
                    bits=16;
        else if(b==5)
                    bits=32;
        else if(b==6)
                    bits=64;
        else if(b==7)
                    bits=128;

        IOREG[A].data = IOREG[A].data | bits;

        if (debugMode==1)
                    printf("\nAfter execution: Reg[%d] =
                    ↪  %X\n",A,IOREG[A].data);
        PC += 0x2;
    }
```

Sets a specified bit in an I/O Register. This instruction operates on the lower 32 I/O Registers – addresses 0-31. In the code, A refers to the I/O Register number and b refers to the bit number. After that, OR is performed between IOREG and the bits , to set the specific bit. Then PC is incremented by 2 to execute next instruction.

5.1.8 CBI – Clear Bit in I/O Register

```
else if(b1==0x9 && b2==8)
        {
            char b,bits;
        if (b4>7)
                    b=b4-8;
        else
                    b=b4;
```

```
            char A= b3*2 + (b4>>3);
                if(debugMode==1){
                    printf("\nCBI instruction decoded\n");
                        printf("\nBefore execution: Reg[%d] =
                        ↪  %X\n",A,IOREG[A].data);
                        }

            if(b==0)
                        bits=0xfe;
            else if(b==1)
                        bits=0xfd;
            else if(b==2)
                        bits=0xfb;
            else if(b==3)
                        bits=0xf7;
            else if(b==4)
                        bits=0xef;
            else if(b==5)
                        bits=0xdf;
            else if(b==6)
                        bits=0xbf;
            else if(b==7)
                        bits=0x7f;

            IOREG[A].data = IOREG[A].data & bits;

            if (debugMode==1)
                        printf("\nAfter execution: Reg[%d] =
                        ↪  %X\n",A,IOREG[A].data);
            PC += 0x2;
        }
```

Clears a specified bit in an I/O register. This instruction operates on the lower 32 I/O registers – addresses 0-31. In the code, A refers to the I/O Register number and b refers to the bit number. After that, AND is performed between IOREG and the bits , to clear the specific bit. Then PC is incremented by 2 to execute next instruction.

5.1.9 AND – Logical AND

```
else if(b1==0x2 && b2>=0 && b2<=3)
        {
            unsigned char r=((b2>>1)&1)*16+b4,d=(b2&1)*16+b3;//setting r and d as
            ↪  per opcode
```

```
            if(r == d && debugMode==1)
                    printf("\nTST instruction decoded\n");//TST = AND Rd,Rd
            else if (r != d && debugMode==1)
                    printf("\nAND instruction decoded\n");

        if(debugMode == 1)
            printf("\n%X AND %X = %X\n",GPR[d].data,GPR[r].data,GPR[d].data &
            ↪  GPR[r].data);

                GPR[d].data = GPR[d].data & GPR[r].data;

            //UPDATE FLAGS
            if(GPR[d].data == 0X0)
                    SREG[1].data = 1;//zero flag
            else
                SREG[1].data=0;

            unsigned char dl=GPR[d].data & 0x80;
            if (dl==0x80)// negative flag
                SREG[2].data=1;
            else
                SREG[2].data=0;

            SREG[3].data=0;//overflow flag

            SREG[4].data=SREG[3].data ^ SREG[2].data;//signed flag


            PC += 0x2;
        }
```

Performs the logical AND between the contents of register Rd and register Rr, and places the result in the destination register Rd. In the code , Rd and Rr are identified as per opcode then and is performed. If Rd and Rr are equal, TST instruction is executed which performs AND between a register and itself. TST – Test for Zero or Minus. Then status register is updated and PC is incremented by 2 to execute next instruction.

5.1.10 EOR – Exclusive OR

```
else if(b1==0x2 && b2>=4 && b2<=7)
        {
            unsigned char r=((b2>>1)&1)*16+b4, d=(b2&1)*16+b3;//setting r and d
            ↪  as per opcode
```

```
            if(r==d && debugMode==1)
                    printf("\nCLR instruction decoded\n");//CLR = EOR Rd,Rd
            else if(r !=d && debugMode==1)
                    printf("\nEOR instruction decoded\n");

        if(debugMode == 1)
            printf("\n%X XOR %X = %X\n",GPR[d].data,GPR[r].data,GPR[d].data ^
            ↪  GPR[r].data);

            GPR[d].data = GPR[d].data ^ GPR[r].data;
            //UPDATE FLAGS
            if(GPR[d].data == 0)
                    SREG[1].data = 1;//zero flag
            else
                SREG[1].data=0;

            unsigned char dl=GPR[d].data & 0x80;
            if (dl==0x80)// negative flag
                SREG[2].data=1;
            else
                SREG[2].data=0;

            SREG[3].data=0;         //overflow flag

            SREG[4].data=SREG[3].data ^ SREG[2].data;//signed flag

            PC += 0x2;
        }
```

Performs the logical EOR between the contents of register Rd and register Rr and places the result in the destination register Rd. In the code, Rd and Rd are identified and exclusive OR is performed between them. If Rd is equal to Rr , then CLR instruction is performed. CLR – Clear Register. Later the status register is updated and PC is incremented by 2 to go to the next instruction.

5.1.11 SBC- Subtract with carry

```
else if(b1==0x00 && b2 >= 0x08 && b2 <= 0x0B)
{
                if(debugMode==1)
                printf("\nSBC instruction decoded\n");
                int dbits[5],rbits[5],Rd=0,Rr=0;
                //For finding Rd and Rr
                Hex2Bin(0,b2);
```

```
Hex2Bin(1,b3);
Hex2Bin(2,b4);
rbits[4] = bin[0].arr[1];
dbits[4] = bin[0].arr[0];
for(i=0;i<4;i++)
{
        dbits[i] = bin[1].arr[i];
        rbits[i] = bin[2].arr[i];
}
for(i=0;i<5;i++)
        {
                Rd += dbits[i]*pow(2,i);
                Rr += rbits[i]*pow(2,i);
        }
ClearBins(0); ClearBins(1); ClearBins(2);
if(debugMode == 1)
printf("\nBefore execution\nReg[%d] = %X\nReg[%d] =
↪  %X\n",Rd,GPR[Rd].data,Rr,GPR[Rr].data);
//For finding difference
Hex2Bin(0,GPR[Rd].data);
Hex2Bin(1,GPR[Rr].data);
//Rd = Rd - Rr-C
GPR[Rd].data =GPR[Rd].data - GPR[Rr].data - SREG[0].data;
Hex2Bin(2,GPR[Rd].data);

//For setting SREG
//For setting Carry flag bit
SREG[0].data = (!bin[0].arr[7]&bin[1].arr[7]) |
↪  (bin[1].arr[7]&bin[2].arr[7]) |
                                (bin[2].arr[7]&!bin[0].arr[7]);
//For setting Zero flag bit
SREG[1].data = !bin[2].arr[0] & !bin[2].arr[1] & !bin[2].arr[2] &
↪  !bin[2].arr[3] &
                                !bin[2].arr[4] & !bin[2].arr[5] &
                                ↪  !bin[2].arr[6] &
                                ↪  !bin[2].arr[7];
//For setting Negative flag bit
SREG[2].data = bin[2].arr[7];
//For setting Overflow flag bit
SREG[3].data = (bin[0].arr[7]&!bin[1].arr[7]&!bin[2].arr[7]) |
                                (!bin[0].arr[7]&bin[1].arr[7]&bin[2].arr[7]);
//For setting Signed bit
SREG[4].data = SREG[2].data ^ SREG[3].data;
//For setting Half Carry flag bit
SREG[5].data = (!bin[0].arr[3]&bin[1].arr[3]) |
↪  (bin[1].arr[3]&bin[2].arr[3]) |
```

```
                                                    (bin[2].arr[3]&!bin[0].arr[3]);

                if(debugMode==1)
printf("\nAfter execution\nReg[%d] = %X\nReg[%d] =
↪  %X\n",Rd,GPR[Rd].data,Rr,GPR[Rr].data);

                ClearBins(0); ClearBins(1); ClearBins(2);

                PC += 0x2;
        }
```

SBC instruction subtracts 2 registers(Rd and Rr) along with carry and saves the result in the first register. Thus from the opcode of SBC Rd and Rr are identified as per opcode by bin arrays and then subtraction along with carry option takes place. All status register bits are changed accordingly and at the end PC value is incremented.

5.1.12 SBR-Set Bits in Register

```
else if(b1==0x06)
        {
       unsigned char k = b2*16 + b4;//constant value
       if(debugMode == 1)
                  {
                printf("\nSBR instruction decoded\n");
                printf("\nReg[%d](%X) or %X =
                ↪  %X\n",b3,GPR[b3].data,k,GPR[b3].data | k);
                  }
                //Rd=Rd or k(to set specific bit in Rd)
        GPR[b3].data = GPR[b3].data | k;
                //overflow is always zero
        SREG[3].data=0;
        //setting negative flag
        if(GPR[b3].data>=80)
                   SREG[2].data = 1;
        else
                   SREG[2].data = 0;
        //sign flag(as v=0,signflag=negflag)
        SREG[4].data = SREG[2].data;
        //checking zero flag
        if(GPR[b3].data == 0x0)
                        SREG[1].data = 1;
        else
            SREG[1].data = 0;
```

```
        PC += 0x02;
        }
```

SBR instruction sets a particular bit in the register(Rd).Thus first register Rd and constant(K) value are identified and then perform OR operation between k and register.Thus turning kth bit of Rd. status bits are changed accordingly.Overflow flag is always zero as overflow doesn't takes place,thus sign flag is equated to negative flag.

5.1.13 SBRC- Skip if Bit in Register is Cleared

```
else if(b1==0x0F && (b2==0x0C || b2==0x0D) && (b4>=0x00 && b4<=0x07))
        {

                if(debugMode == 1)
                        {
                                printf("\nSBRC instruction decoded\n");
                        }
        unsigned char r= (b2 & 1)*16 + b3;//Rr value
        unsigned char d=0;
        //finding b value
        bin[0].arr[b4]=1;
        for(i=0;i<8;i++)
                  d += bin[0].arr[i]*pow(2,i);//converting decimal
        //checking if Rr(b) = 0
        unsigned char k= GPR[r].data & d;
        if (k == 0)
                  PC= PC+0x04;
        else
                   PC=PC+0x02;
        ClearBins(0);
        }
```

SBRC checks if the kth bit of a general purpose register is 1 or zero and then jumps or changes PC value accordingly .So,first register(r) and constant(d) values are found .Later d value is convered into binary(by use of bin array) such that dth bit of that binary is 1and then converted into decimal for bitwise AND operation.Now bitwise AND between r and d operation decides if the bit is 0 and 1.If zero it skips next instruction and pc increments by 4 or else pc as usually increments by 2.

5.1.14 SBRS- Skip if Bit in Register is Set

```
else if(b1==0x0F && (b2==0x0E || b2==0x0F) && (b4>=0x00 && b4<=0x07))
        {

                if(debugMode == 1)
                                printf("\nSBRS instruction decoded\n");
                unsigned char r= (b2 & 1)*16 + b3;
        unsigned char d=0;
        //finding b value
        bin[0].arr[b4]=1;
        for(i=0;i<8;i++)
                  d += bin[0].arr[i]*pow(2,i);//converting decimal
        //checking if Rr(b) = 1
        unsigned char k= GPR[r].data & d;
        if (k == d)
                  PC= PC+0x04;
        else
                   PC=PC+0x02;
        ClearBins(0);
        }
```

SBRC checks if the kth bit of a general purpose register is 1 or zero and then jumps or changes PC value accordingly .So,first register(r) and constant(d) values are found .Later d value is convered into binary(by use of bin array) such that dth bit of that binary is 1and then converted into decimal for bitwise AND operation.Now bitwise AND between r and d operation decides if the bit is 0 and 1.If one it skips next instruction and pc increments by 4 or else pc as usually increments by 2.

5.1.15 SBIC- Skip is Bit in I/O register is cleared

```
else if(b1==0x09 && b2==0x09)
        {

                if(debugMode == 1)
                                printf("\nSBIC instruction decoded\n");
                unsigned char r= b3*2 + (b4 & 8)%8;//   finding IO register  A
        unsigned char d=0;
        //finding b value
        bin[0].arr[b4 & 7]=1;
        for(i=0;i<8;i++)
                 d += bin[0].arr[i]*pow(2,i);
        //checking id A(b)=0
        unsigned char k= IOREG[r].data & d;
        if (k == 0)
```

```
                PC= PC+0x04;
        else
                 PC=PC+0x02;
        ClearBins(0);
        }
```

SBIC checks if the kth bit of a Inout register is 1 or zero and then jumps or changes PC value accordingly .So,first register(r) and constant(d) values are found .Later d value is convered into binary(by use of bin array) such that dth bit of that binary is 1and then converted into decimal for bitwise AND operation.Now bitwise AND between r and d operation decides if the bit is 0 and 1.If zero it skips next instruction and pc increments by 4 or else pc as usually increments by 2

5.1.16 PUSH

```
else if(b1==0x09 && (b2==0x02 || b2== 0x03) && b4==0x0F)
        {
                if(debugMode==1)
                        printf("\nPUSH instruction decoded\n");
        unsigned char k =(b2 & 1)*16 + b3;//Rd value
        SRAM[SP].data=GPR[k].data;//pushing into stack
        printf("\n SRAM[SP]: %X \n",SRAM[SP].data);//for testing
        SP=SP-1;
        PC=PC+0x02;
        }
```

PUSH operation pushes the selected General Purpose Register value into the stack, Decrement SP value by 1 and goes to next instruction. First,Register number is identified(Rd) and its values and pushed into stack.As registers contain only 8 bit value,its stack pointer is decremented only by 1 bit.Thus PC increments and goes to next instruction.

5.1.17 POP

```
else if(b1==0x09 && (b2==0x00 || b2== 0x01) && b4==0x0F)
        {
                if(debugMode==1)
                        printf("\nPOP instruction decoded\n");
        unsigned char k=(b2 & 1)*16 + b3;//finding Rd
        GPR[k].data=SRAM[SP+1].data;//getting values from stack
        printf("\n after execution GPR[k]: %X",GPR[k].data);//for testing
        SP=SP+1;
        PC=PC+0x02;
        }
```

In POP operation, the value from stack are given back to General Purpose register (Rd)and increments SP value.Thus first Rd value is found and the given value from stack.As General Purpose register takes only 8 bit value,stack pointer only increments by 1 value.The PC value is incremented to go to next instruction.

5.1.18 RCALL-Relative Call to Subroutine

```
else if(b1==0x0D)
{
    if(debugMode==1)
        printf(&quot;\nRCALL instruction decoded\n&quot;);

    int k=b2*256+b3*16+b4;//k value
    SRAM[SP-1].data=PC+2;//storing next instruction PC(lsb 4 bits)
    SRAM[SP].data=(PC+2)/256;//msb 4 bits of pc
    if(debugMode==1)
    printf(&quot;\nSRAM value: %X %X&quot;,SRAM[SP].data,SRAM[SP-1].data);
    SP - = 2;
    PC += k*2 + 2;
}
```

It relatively jumps to particular address by saving the PC value f next intruction into stack , so that it can reach to back to next instruction after completing the subroutine.So, first the k value to where the PC needs to updated is calculated.Later the PC value of next instruction is pushed into stack and thus stackpointer gets decremented by 2(as PC is 12 bits and Stack each memory cell stores 8 bit).Later PC is moved to PC+2K+2(subroutine).

5.1.19 RET-Return from Subroutine

```
else if(b1==0x09 && b2==0x05 && b3== 0x00 && b4== 0x08)
        {
                if(debugMode==1)
                        printf("\nRET instruction decoded\n");

        PC = SRAM[SP+1].data + SRAM[SP+2].data*256;//returning pc values from
        ↪  stack
        SP += 2;
    }
```

Returns from subroutine. The return address is loaded from the STACK. The Stack Pointer uses a pre- increment scheme during RET.AT the end of subroutine RET

instruction is used to coe back to pragram.Thus,C value stored in stack is pushed back to PC and SP is incremented by 2(as PC took 2 memory locations in stack).

5.1.20 NEG-Two's complement

```
else if(b1==0x09 && (b2==0x05 || b2==0x04) && b4==0x01)
        {
                unsigned char k=(b2 & 1)*16 + b3;//Rd value
        if(debugMode == 1)
            {
                printf("\nNEG instruction decoded\n");
                printf("\nBefore execution: Reg[%d]: %X",k,GPR[k].data);
            }
        GPR[k].data = 0x00-GPR[k].data;//2's complement
                //setting carry and zero flag
        if(GPR[k].data == 0x0)
                        {
            SREG[1].data = 1;
                        SREG[0].data = 0;
                        }
        else
                        {
            SREG[1].data = 0;
                        SREG[0].data = 1;
                        }
                if(GPR[k].data>=80)
                        SREG[2].data = 1;//neg flag
        else
            SREG[2].data = 0;
                if(GPR[k].data==80)
                        SREG[3].data = 1;
        else
            SREG[3].data = 0;

                if(SREG[3].data == SREG[2].data)
                        SREG[4].data = 0;//sign flag
        else
            SREG[4].data = 1;
                int s=GPR[k].data & 8;
                if(s == 8)
                        SREG[5].data = 1;//halfcarry
                else
            SREG[5].data = 0;
        if(debugMode == 1)
            printf("\nAfter execution: Reg[%d]: %X",k,GPR[k].data);
```

```
        PC += 0x02;
    }
```

Replaces the contents of register Rd with its two's complement; the value 0x80 is left unchanged.For finding 2's complement the logic used here is 2 subtract the number from 00 as for negative numbers are usually also stored in 2's complement.

Flags:negative flag is equal to MSB of the results.Overflow flag is will occur if and only if the contents of the Register after operation (Result) is 0x80. C flag Set if there is a borrow in the implied subtraction from zero; cleared otherwise. The C Flag will be set in all cases except when the contents of Register after operation is 0x00. Zero flag Set if the result is 0x00.h flag Set if there was a borrow from bit 3 and sign flag at end is xor of overflow and negative flag.

5.1.21 LSR- Logical Shift Right

```
else if(b1==0x09 && (b2==0x05 || b2==0x04) && b4==0x06)
    {
        unsigned char k=(b2&1)*16+b3;//Rd value
                if(debugMode == 1)
                {
                        printf("\nLSR instruction decoded\n");
                        printf("\nBefore execution: Reg[%d]: %X",k,GPR[k].data);
                }
                //giving Rd(0) to carry flag
        if(GPR[k].data & 0x01 == 0x01)
            SREG[0].data = 1;
                else
                        SREG[0].data = 0;
        //right shiting
                GPR[k].data = GPR[k].data/0x02;
        if(GPR[k].data == 0x0)
                        SREG[1].data = 1;//setting zero flag
                else
                        SREG[1].data = 0;

        SREG[2].data = 0;//neg flag
        SREG[3].data = SREG[0].data;//overflow flag
        SREG[4].data = SREG[3].data;//sign flag
        if(debugMode == 1)
                        printf("\nAfter execution: Reg[%d]: %X",k,GPR[k].data);

                PC += 0x02;
        }
```

Shifts all bits in Rd one place to the right. Bit 7 is cleared. Bit 0 is loaded into the C Flag of the SREG.This operation effectively divides an unsigned value by two. The C Flag can be used to round the result.Thus first the register to be left shifted is calculated.Then the right shifting is done by dividing number by 2 and all the flags are set accordingly.Flags: As MSB is always zero the negative flag is zero.Carry flag is LSB bit and overflow is xor of negative and carry thus equal to carry flag .So, overflow is equal negative flag.

5.1.22 BSET- Bit set in SREG

```
else if(b1==0x09 && b2==0x04 && b4==0x08 && b3>=0 && b3<=7)
    {
                if(debugMode == 1)
                {
                        if(b3 == 0x0)
                                printf("\nSEC instruction decoded\n");
                        else if(b3 == 0x01)
                                printf("\nSEZ instruction decoded\n");
                        else if(b3 == 0x02)
                                printf("\nSEN instruction decoded\n");
                        else if(b3 == 0x03)
                                printf("\nSEV instruction decoded\n");
                        else if(b3 == 0x04)
                                printf("\nSES instruction decoded\n");
                        else if(b3 == 0x05)
                                printf("\nSEH instruction decoded\n");
                        else if(b3 == 0x06)
                                printf("\nSET instruction decoded\n");
                        else if(b3 == 0x07)
                                printf("\nSEI instruction decoded\n");
                }
        SREG[b3].data = 0x01;
        PC += 0x2;
        }
```

Sets a single Flag or bit in SREG. The flag to be changed in SREG is value of b3 in opcode .Thus particular set instructions are identified according to b3 value and the flag is set using b3 value.

5.1.23 OR- logical OR

```
/  OR by VY            12-05-2020
    else if(b1==0x02 && b2>=0X08 && b2<=0X0B)
    {
        unsigned char d= (b2 & 1)*16+b3;//Rd
        unsigned char r= (b2 & 2)*8+b4;//Rr
        if(debugMode == 1)
        {
            printf("\nOR instruction decoded\n");
            printf("\nbefore execution: Reg[%d]: %X",d,GPR[d].data);
        }

        GPR[d].data = GPR[d].data | GPR[r].data;//Rd OR Rr

        if(debugMode == 1)
            printf("\nAfter execution with Reg[%d]: %X = Reg[%d]:
            ↪  %X",r,GPR[r].data,d,GPR[d].data);

        SREG[3].data=0;//overflow flag

        if(GPR[d].data>=80)
            SREG[2].data = 1;//neg flag
        else
            SREG[2].data = 0;

        SREG[4].data = SREG[2].data;//sign flag

        if(GPR[d].data == 0x0)//changing carry falg
            SREG[1].data = 1;
        else
            SREG[1].data = 0;

        PC += 0x2;
    }
```

Performs the logical OR between the contents of register Rd and register Rr, and places the result in the destination register Rd.First the Rd and Rr values (values of registers for which OR operation takes place)are found and logical OR is found and stored back in Rd.Flags: Here the overflow flag is always zero thus sign flag is equal to negative flag.Negative flag is equal to MSB and zero flag tests for zero result and sets.

5.1.24 BCLR-Bit Clear in SREG

```
else if(b1==0x09 && b2==0x04 && b4==0x08 && b3>=0X08 && b3<=0X0F)
    {
                if(debugMode == 1)
                {
                        if(b3 == 0x0)
                                printf("\nCEC instruction decoded\n");
                        else if(b3 == 0x01)
                                printf("\nCEZ instruction decoded\n");
                        else if(b3 == 0x02)
                                printf("\nCEN instruction decoded\n");
                        else if(b3 == 0x03)
                                printf("\nCEV instruction decoded\n");
                        else if(b3 == 0x04)
                                printf("\nCES instruction decoded\n");
                        else if(b3 == 0x05)
                                printf("\nCEH instruction decoded\n");
                        else if(b3 == 0x06)
                                printf("\nCET instruction decoded\n");
                        else if(b3 == 0x07)
                                printf("\nCEI instruction decoded\n");
                }
        unsigned char k = b3 & 7;
        SREG[k].data = 0x00;

        PC += 0x2;
        }
```

Clears a single Flag or bit in SREG.The flag to be changed in SREG is value of b3 in opcode .Thus particular set instructions are identified according to b3 value and the flag is cleared using b3 value.

5.1.25 RETI- Return from Interrupt

```
else if(b1==0x9 && b2==0x5 && b3==0x1 && b4==0x8)
        {
                if(debugMode==1)
                        printf("\nRETI instruction decoded\n");

                PC = (SRAM[SP-1].data << 4 | SRAM[SP].data);//storing the value
                ↪  of PC from the stack
        }
```

Returns from interrupt. The return address is loaded from the STACK and the Global Interrupt Flag is set.Note that the Status Register is not automatically stored when entering an interrupt routine, and it is not restored when returning from an interrupt routine. This must be handled by the application program. The Stack Pointer uses a pre-increment scheme during RETI.Thus the 12 bit PC value is stored back to PC by tis instruction.

5.1.26 MOV- Copy Register

```
else if(b1==0x2 && b2>=12 && b2<=15)
        {        unsigned char b4=((b2>>1)&1)*16+b4;//setting r and d as per
        ↪  opcode
            unsigned char b3=(b2&1)*16+b3;
                int a=GPR[b3].data,b=GPR[b4].data;
                if(debugMode==1)
                {
                        printf("\nMOV instruction decoded\n");
                        printf("\nBefore execution: Reg[%d] = %X, Reg[%d] =
                        ↪  %X\n",b3,GPR[b3].data,b4+,GPR[b4].data);
                }

                GPR[b3].data = GPR[b4].data;//moving the data to destination
                ↪  location

                if(debugMode==1)
                        printf("\nAfter execution Reg[%d] = %X, Reg[%d] =
                        ↪  %X\n",b3,GPR[b3].data,b4,GPR[b4].data);

                PC += 0x2;//Incrementing Program Counter
        }
```

This instruction makes a copy of one register into another. The source register Rr is left unchanged, while the destination register Rd is loaded with a copy of Rr.At start the 2 registers are identified from opcodes and value of 1 register is given to another.

5.1.27 ROR- Rotate Right through Carry

```
else if(b1==0x9 && (b2==0x4 || b2==0x5) &&
↪  b4==0x07)
↪
        {        int t;GPR[b3+16].data=0x55;
                if(debugMode==1)
```

```
        {
                printf("\nROR instruction decoded\n");
                printf("\nBefore execution: Reg[%d] =
                ↪  %X\n",b3+16,GPR[b3+16].data);
        }
        t=0x01 & GPR[b3+16].data;
        GPR[b3+16].data = GPR[b3+16].data>>1;                //Rotating
        ↪  Right
        if(SREG[0].data==1)
        GPR[b3+16].data = 0x80 | GPR[b3+16].data;
        SREG[0].data=t;                                       //updating
        ↪  C flag

        if(GPR[b3+16].data == 0x0)                           //updating Z
        ↪  flag
                SREG[1].data = 1;
        else
                SREG[1].data = 0;

        if(GPR[b3+16].data>=0x08)                           //updating N
        ↪  flag
                SREG[2].data = 1;
        else
                SREG[2].data = 0;

        SREG[3].data = SREG[0].data^SREG[2].data;           //updating V
        ↪  flag

        SREG[4].data = SREG[3].data^SREG[2].data;           //updating S
        ↪  flag

        if(debugMode==1)
        {
                printf("\nAfter execution Reg[%d] =
                ↪  %X\n",b3+16,GPR[b3+16].data);
        }
        PC += 0x2;
}
```

Shifts all bits in Rd one place to the right. The C Flag is shifted into bit 7 of Rd. Bit 0 is shifted into the C Flag.The Carry Flag can be used to round the result.Thus in code,the t variable stores the LSB bit of register and then register value is right shited .Later,the Cflag vlue is given to MSB and LSB value before rotation which is stored in t is given to Carry flag.Later zero and negative flags are set by checking MSB and if result is zero. Sign and overflow flag are set by using XOR operation.

5.1.28 BLD – Bit Load from the T Flag in SREG to a Bit in Register

```
else if(b1==0xf && b2>=8 && b2<=9 && b4<=7)
        {        unsigned char b4=((b2>>1)&1)*16+b4;//setting r and d as per
        ↪  opcode
            unsigned char b3=(b2&1)*16+b3;

                int a=GPR[b3].data,mask=0;
                if(debugMode==1)
                {
                        printf("\nBLD instruction decoded\n");
                        printf("\nBefore execution: Reg[%d] =
                        ↪  %X\n",b3,GPR[b3].data);
                }

                mask=1<<b4;//masking the bits

                GPR[b3].data =(GPR[b3].data & ~mask)|(SREG[6].data<<b4) ;//Bit
                ↪  Load                                  //from the T Flag in SREG
                ↪  to a Bit in Register
                if(debugMode==1)
                        printf("\nAfter execution Reg[%d] =
                        ↪  %X\n",b3,GPR[b3].data);

                PC += 0x2;
        }
```

Copies the T Flag in the SREG (Status Register) to bit b in register Rd.In code,first the Register is identified from opcode and and mask stores number with nth bit which has to be given T flag value as set to 1.Later that bit is given value of T flag by first making nth bit as zero by logical and with inverted mask and Logical OR with Staus register.

5.1.29 BST – Bit Store from Bit in Register to T Flag in SREG

```
else if(b1==0xf && b2>=0xA && b2<=0xB && b4<=7)
        {        unsigned char b4=((b2>>1)&1)*16+b4;//setting r and d as per
        ↪  opcode
            unsigned char b3=(b2&1)*16+b3;

                int a=GPR[b3].data;
```

```
                if(debugMode==1)
                {
                        printf("\nBST instruction decoded\n");
                        printf("\nBefore execution: Reg[%d] =
                        ↪  %X\n",b3,GPR[b3].data);
                }

                SREG[6].data = 1 & (GPR[b3].data>>(b4)) ;//Bit Store from Bit
                                        //in Register to T Flag in SREG
                PC += 0x2;
        }
```

Stores bit b from Rd to the T Flag in SREG (Status Register).In code the register is first identified and is shifted by n bits so that the bit to be given to S register comes to LSB and is given to T flag of Status register by logicakl and with 1.

5.1.30 SBIS – Skip if Bit in I/O Register is Set

```
else if(b1==0x9 &&
↪  b2==0x0B)
        {        unsigned char b4=((b2>>1)&1)*16+b4;//setting r and d as per
        ↪  opcode
            unsigned char b3=(b2&1)*16+b3;

                GPR[((0x7 & b3)<<1|(0x8 & b4))].data=0x55;
                int temp=0x01;
                if(debugMode==1)
                        printf("SBIS instruction decoded\n");
                temp=temp<<(0x7 & b4);

                if((GPR[((0x7 & b3)<<1|(0x8 & b4))].data & temp)!=0)//Skipping
                //the next instruction if a bit is set
                        PC += 0x4;
                else
                        PC+=0x02;
        }
```

This instruction tests a single bit in an I/O Register and skips the next instruction if the bit is set.

5.1.31 ASR – Arithmetic Shift Right

```
    else if(b1==0x9 && b2>=4 && b4==5)
    {
```

```
        unsigned char k=(b2&1)*16+b3;//set reg k according to opcode
        if(debugMode==1){
                    printf("\nASR instruction decoded\n");
            printf("\nBefore execution: Reg[%d] = %X\n",k,GPR[k].data);
            }

            SREG[0].data=GPR[k].data & 1;// set or reset carry flag
            char l=GPR[k].data & 10000000;
                GPR[k].data=((GPR[k].data) >> 1) | l;//right shifting by 1
            // update flags
                if (GPR[k].data==0)
                        SREG[1].data=1;// zero flag
                else
                        SREG[1].data=0;
                if (l==10000000)
                        SREG[2].data=1;//negative flag
                else
                        SREG[2].data=0;
                SREG[3].data=SREG[0].data ^ SREG[2].data;        //overflow flag
                SREG[4].data=SREG[3].data ^ SREG[2].data;        //signed flag

                if(debugMode==1)
                printf("\nAfter execution: Reg[%d] = %X\n",k,GPR[k].data);
        PC += 0x2;
    }
```

Shifts all bits in Rd one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C Flag of the SREG. This operation effectively divides a signed value by two without changing its sign. The Carry Flag can be used to round the result. In the code k denotes the Rd register. The zeroth bit is given to the carry flag. 'l' denotes the seventhbit and while right shift is performed it is ORed with l so that the sign of the value in register remains same. Then the SREG is updated and PC is incremented by 2 to go to the next instruction.

5.1.32 BRBC – Branch if Bit in SREG is Cleared

```
else if(b1==0xf && b2>=4 && b2<=7)
        {
                int kbits[7],jump=0,l=0;
                char temp=0x0;
                l=b4%0x8;
                if(debugMode == 1)
                {
                        if(l == 0x0)
                                printf("\nBRCC instruction decoded\n");
```

```
        else if(l == 0x01)
                printf("\nBRNE instruction decoded\n");
        else if(l == 0x02)
                printf("\nBRPL instruction decoded\n");
        else if(l == 0x03)
                printf("\nBRVC instruction decoded\n");
        else if(l == 0x04)
                printf("\nBRGE instruction decoded\n");
        else if(l == 0x05)
                printf("\nBRHC instruction decoded\n");
        else if(l == 0x06)
                printf("\nBRTC instruction decoded\n");
        else if(l == 0x07)
                printf("\nBRID instruction decoded\n");
}
if(SREG[l].data == 0)//Branch if a Bit in SREG is Cleared
{
        //For getting Kbits
        Hex2Bin(0,b2);
        Hex2Bin(1,b3);
        Hex2Bin(2,b4);
        kbits[6] = bin[0].arr[0];
        kbits[5] = bin[0].arr[1];
        for(i=0;i<4;i++)
                kbits[i+1] = bin[1].arr[i];
        kbits[0] = bin[2].arr[3];

        if(kbits[6] == 1)//Signed bit set (k is negative)
        {
                for(i=0;i<6;i++)
                        temp += kbits[i]*pow(2,i);
                temp -= 0x01;
                i=0;
                while(temp!=0 && i<=6)
                {
                        kbits[i] = temp % 2;
                        i++;
                        temp /= 2;
                }
                for(i=0;i<6;i++)
                        kbits[i] = !kbits[i];

                for(i=0;i<6;i++)
                        jump += kbits[i]*pow(2,i);
                jump *= -2;
        }
```

```
                else
                {
                        for(i=0;i<6;i++)
                                jump += kbits[i]*pow(2,i);
                        jump *= 2;
                }
                if(debugMode == 1)
                        printf("\nJumping from PC:%X to PC:
                        ↪  %X",PC,PC+jump+0x02);
                PC += jump + 0x02;

        }
        else
                PC += 0x2;

        ClearBins(0); ClearBins(1); ClearBins(2);

    }
```

Conditional relative branch. Tests a single bit in SREG and branches relatively to PC if the bit is cleared. This instruction branches relatively to PC in either direction (PC - 63 $\leq$ destination $\leq$ PC + 64). Parameter k is the offset from PC and is represented in two's complement form. In the code,the branch condition is checked and if any of the branch satisfies , the K value (value by which PC should relatively jump) is identified and given to kbits array. Then the sixth bit is identifies to check whether the value is negative or postive. If it is negative then two's compliment is performed and jump value is identied, otherwise if it is postive then jump value is simply the kbits in decimal form. Then PC value is incremented by jump + 2. If the branch conditon fails , then PC is simply incremented by 2, to go to the next instruction.

5.1.33 BRBS – Branch if Bit in SREG is Set

```
else if(b1==0xf && b2>=0 && b2<=3)
    {
            int kbits[7],jump=0,l=0;
            char temp=0x0;
            l=b4%0x8;                   //extracting the bit postion to be
            ↪  checked
            if(debugMode == 1)
            {
                    if(l == 0x0)
                            printf("\nBRCS instruction decoded\n");
                    else if(l == 0x01)
                            printf("\nBREQ instruction decoded\n");
```

```
        else if(l == 0x02)
                printf("\nBRMI instruction decoded\n");
        else if(l == 0x03)
                printf("\nBRVS instruction decoded\n");
        else if(l == 0x04)
                printf("\nBRLT instruction decoded\n");
        else if(l == 0x05)
                printf("\nBRHS instruction decoded\n");
        else if(l == 0x06)
                printf("\nBRTS instruction decoded\n");
        else if(l == 0x07)
                printf("\nBRIE instruction decoded\n");
}
if(SREG[l].data == 1)//Branch if Bit in SREG is Set
{
        //For getting Kbits
        Hex2Bin(0,b2);
        Hex2Bin(1,b3);
        Hex2Bin(2,b4);
        kbits[6] = bin[0].arr[0];
        kbits[5] = bin[0].arr[1];
        for(i=0;i<4;i++)
                kbits[i+1] = bin[1].arr[i];
        kbits[0] = bin[2].arr[3];

        if(kbits[6] == 1)//Signed bit set (k is negative)
        {
                for(i=0;i<6;i++)
                        temp += kbits[i]*pow(2,i);
                temp -= 0x01;
                i=0;
                while(temp!=0 && i<=6)
                {
                        kbits[i] = temp % 2;
                        i++;
                        temp /= 2;
                }
                for(i=0;i<6;i++)
                        kbits[i] = !kbits[i];

                for(i=0;i<6;i++)
                        jump += kbits[i]*pow(2,i);
                jump *= -2;
        }
        else
        {
```

```
                        for(i=0;i<6;i++)
                                jump += kbits[i]*pow(2,i);
                        jump *= 2;
                }
                if(debugMode == 1)
                        printf("\nJumping from PC:%X to PC:
                        ↪ %X",PC,PC+jump+0x02);
                PC += jump + 0x02;

        }
        else
                PC += 0x2;

}
```

Conditional relative branch. Tests a single bit in SREG and branches relatively to PC if the bit is set. This instruction branches relatively to PC in either direction (PC - 63 $\leq$ destination $\leq$ PC + 64). Parameter k is the offset from PC and is represented in two's complement form. In the code,the branch condition is checked and if any of the branch satisfies , the K value (value by which PC should relatively jump) is identified and given to kbits array. Then the sixth bit is identifies to check whether the value is negative or postive. If it is negative then two's compliment is performed and jump value is identied, otherwise if it is postive then jump value is simply the kbits in decimal form. Then PC value is incremented by jump + 2. If the branch conditon fails , then PC is simply incremented by 2, to go to the next instruction.

5.1.34 SWAP – Swap Nibbles

```
else if(b1==0x9 && (b2==0x4 || b2==0x5) && b4==0x2)
        {         unsigned char b3=(b2&1)*16+b3;        //setting d as per
        ↪ opcode

                int temp;
                if(debugMode == 1)
                        {
                                printf("\nSWAP instruction decoded\n");
                                printf("\nBefore execution: Reg[%d]:
                                ↪ %X",b3,GPR[b3].data);
                        }
                temp=GPR[b3].data;
                GPR[b3].data=((temp & 0x0F)<<4) | ((temp &0xF0)>>4);// Swapping
                ↪ the nibbles
                if(debugMode == 1)
                        printf("\nAfter execution: Reg[%d]: %X",b3,GPR[b3].data);
```

```
                PC += 0x02;
        }
```

Swaps high and low nibbles in a register. In the code Rd is identified and nibbles are swapped. Later the PC is incremented by 2 to go to the next instruction.

5.1.35 INC – Increment

```
else if(b1==0x09 && (b2==0x05 || b2==0x04) && b4==0x03)
        {//setting d as per opcode
            unsigned char b3=(b2&1)*16+b3;

                if(debugMode == 1)
                        {
                                printf("\nINC instruction decoded\n");
                                printf("\nBefore execution: Reg[%d]:
                                ↪  %X",b3,GPR[b3].data);
                        }

                if(GPR[b3].data==0x7F)//updating V flag
                        SREG[3].data = 1;
                else
                        SREG[3].data = 0;

                GPR[b3].data += 0x01;//incrementing data

                if(GPR[b3].data == 0x0)//updating Z flag
                        SREG[1].data = 1;
                else
                        SREG[1].data = 0;

                if(GPR[b3].data>=0x08)//updating N flag
                        SREG[2].data = 1;
                else
                        SREG[2].data = 0;

                SREG[4].data = SREG[3].data^SREG[2].data;//updating S flag

                if(debugMode == 1)
                        printf("\nAfter execution: Reg[%d]: %X",b3,GPR[b3].data);

                PC += 0x02;
        }
```

Adds one -1- to the contents of register Rd and places the result in the destination register Rd. The C Flag in SREG is not affected by the operation, thus allowing the INC instruction to be used on a loop counter in multiple-precision computations. In the code, the Rd register is identifed and is incremented by one. The flags are updated accordingly. PC is incremented by 2 to go to the next instruction.

5.1.36 LD-Load Indirect from Data Space to Register using Index X

```
else if(b1==0x9 && (b2==0x1 ||
↪  b2==0x0))
        {        unsigned char b4=((b2>>1)&1)*16+b4;//setting r and d as per
        ↪  opcode
            unsigned char b3=(b2&1)*16+b3;
                if(debugMode==1)
                {
                        printf("\nLD instruction decoded\n");
                        printf("\nBefore execution: Reg[%d] =
                        ↪  %X\n",b3,GPR[b3].data);
                }
                int x;//x is the value loaded in LDI instruction
                if(b4==0xC)
                        GPR[b3].data=x;//For normal LD
                else if(b4==0xD)
                {
                        x+=1;
                        GPR[b3].data=GPR[x].data;//For LD+ condition
                }
                else if(b4==0xE)
                {
                        x-=1;
                        GPR[b3].data=GPR[x].data;//For LD- condition
                }
                if(debugMode==1)
                        printf("\nAfter execution Reg[%d] =
                        ↪  %X\n",b3,GPR[b3].data);

                PC += 0x2;
        }
```

Loads one byte indirect from the data location pointed to by the X (16 bits) Pointer Register in the Register File to another register. The X-pointer Register is the Register loaded into a location by the LDI instruction. The X-pointer Register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented by specifying the value of b1 in the instruction as 0x0C, 0X0D and

0x0E respectively.

Example:

```
clr r27 ; Clear X high byte
ldi r26,$60 ; Set X low byte to £60
ld r1,X ; Load r1 with data space loc. £60
ldi r26,$63 ; Set X low byte to £63
ld r2,X ; Load r2 with data space loc. £63
```

5.1.37 LDS (16-bit) – Load Direct from Data Space

```
else if(b1==0xA && (b2>=0x0 &&
↪  b2<=0x7))
↪
{        unsigned char b4=(b2&7)*16+b4;//setting r as per opcode
        int x;
        if(debugMode==1)
                printf("\nLDS instruction decoded\n");
        GPR[b3].data = GPR[b4].data;//Loading the data to a particular
        ↪  location
        x=b4;
        PC += 0x2;
}
```

Loads one byte from the data space to a register. A 7-bit address must be supplied. The receiver address is set as per the opcode while the destination address remains same as b3.

5.1.38 ST – Store Indirect From Register to Data Space using Index X

```
else if(b1==0x9 && (b2==0x3 ||
↪  b2==0x2))
{       unsigned char b4=((b2>>1)&1)*16+b4;//setting r and d as per
↪  opcode
    unsigned char b3=(b2&1)*16+b3;

        int x;//x is the value loaded in LDI instruction
        if(debugMode==1)
        {
                printf("\nST instruction decoded\n");
```

```
                printf("\nBefore execution: Reg[%d] =
                ↪  %X\n",x,GPR[x].data);
        }
        if(b4==0xC)
                GPR[x].data=GPR[b3].data;//For normal ST
        else if(b4==0xD)
        {
                x+=1;
                GPR[x].data=GPR[b3].data;//For normal ST+
        }
        else if(b4==0xE)
        {
                x-=1;
                GPR[x].data=GPR[b3].data;;//For normal ST-
        }
        if(debugMode==1)
        {
                printf("\nAfter execution Reg[%d] = %X\n",x,GPR[x].data);
        }
        PC += 0x2;
}
```

Stores one byte indirect from a register to data location pointed to by the X (16 bits) Pointer Register in the Register File to another register. The X-pointer Register is the Register loaded into a location by the LDI instruction. The X-pointer Register can either be left unchanged by the operation, or it can be post-incremented or pre-decremented by specifying the value of b1 in the instruction as 0x0C, 0X0D and 0x0E respectively.

Example:

```
clr r27 ; Clear X high byte
ldi r26,$60 ; Set X low byte to £60
st X,r1 ; Store r1 in data space loc. £60
ldi r26,$63 ; Set X low byte to £63
st X,r2 ; Store r2 in data space loc. £63
```

5.1.39 STS (16-bit) – Store Direct to Data Space

```
else if(b1==0xA && (b2>=0x8 &&
↪  b2<=0x16))
↪
        {       unsigned char b4=((b2&7)*16+b4;//setting r and d as per opcode
```

```
        int x;
        if(debugMode==1)
        {
                printf("\nSTS instruction decoded\n");
                printf("\nBefore execution: Reg[%d] = %X\n",
                ↪  b4,GPR[b4].data);
        }
        GPR[b4].data=GPR[b3].data;//Storing the data to a particular
        ↪  location
        x=b4;
        if(debugMode==1)
                printf("\nAfter execution Reg[%d] =
                ↪  %X\n",b4,GPR[b4].data);
        PC += 0x2;
}
```

Stores one byte from the data space to a register. A 7-bit address must be supplied. The receiver address is set as per the opcode while the destination address remains same as b3.

5.1.40 Timer0 implementation

1. Normal Mode

```
unsigned char temp = 0x0;static unsigned char bit=0x00;
    if(IOREG[0x33].data >= 0x01)
    {
       if(debugMode == 1)
                              printf("\n**!!TIMER 0 operation detected!!**\n");
            Hex2Bin(3,IOREG[0x2A].data);
         Hex2Bin(2,IOREG[0x33].data);
         //for normal mode by vy
       if((bin[3].arr[0] == 0 && bin[3].arr[1]==0 && bin[2].arr[4]==0) ||
       ↪  (bin[3].arr[6] == 0 && bin[3].arr[7]==0))
       {
        printf("\n**normal mode!!**\n");
        temp = IOREG[0x38].data & 0x02; //TIFR overflow
                if(temp == 0x02)
                        {
                                printf("\n**!!TOV0 overflow!!**\n");
                                IOREG[0x32].data = 0; //counter made 0 coz of
                                ↪  overflow
                                IOREG[0x38].data = IOREG[0x38].data ^ 0x02;
                        }
                //incrementing
         if(IOREG[0x32].data < 255)
            {
                        IOREG[0x32].data += 0x01;
            IOREG[0x18].data = IOREG[0x18].data ^ 0x01;//for testing
                        printf("\nPORTB: %X\n",IOREG[0x18].data);
            }
                 else if(IOREG[0x32].data == 255)
                        {
                                IOREG[0x38].data = IOREG[0x38].data | 0x02;
                                printf("\nTOV0 set, TIFR:
                                ↪  %X\n",IOREG[0x38].data);//setting TIFR
                                ↪  overflow
                        }
         printf("\nTCCNT0 : %X ",IOREG[0x32].data);
                SetPins(1);
        }
```

The simplest mode of operation is the Normal mode (WGM0[2:0] = 0). In this mode the counting direction is always up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 8-bit value (TOP = 0xFF) and then restarts from the bottom (0x00). In the first TCCR0B

is check as if CSO[2:0] are 0 there is no clock source and timer doesnt work.Later the mode of operation is checked by TCCROA least significant 2 bits WGM00 and WGM01.If WGM0[2:0] = 0 normal mode is identified.Also if other modes are activated but if COM0A[1:0]=0 then also normal mode is executed.After checking mode it checks for timer overflow and clears TCNT0 to zero if flag is set.Later,if TCNT0 is less then 255 its value is incremented or else the overflow flag is set 1.This process keeps repeating for everyclock cycle.

2. Clear Timer on Compare Match (CTC) Mode

```
int CTCMode=0,COM0A_Con=0;
     if(bin[3].arr[1] == 1 && bin[3].arr[0]==0)
     {
            CTCMode = 1;
            printf("\n**!!CTC mode!!**\n");//testing
            temp = IOREG[0x38].data & 0x10;
        if(temp == 0x10)
            {
                IOREG[0x38].data = IOREG[0x38].data & 0xEF;
                printf("\nTIFR after resetting OCF0A: %X\n",IOREG[0x38].data);
            }
        if(bin[3].arr[7] == 0 && bin[3].arr[6] == 0)
            COM0A_Con = 0;
        else if(bin[3].arr[7] == 0 && bin[3].arr[6] == 1)
            COM0A_Con = 1;
        else if(bin[3].arr[7] == 1 && bin[3].arr[6] == 0)
            COM0A_Con = 2;
        else if(bin[3].arr[7] == 1 && bin[3].arr[6] == 1)
            COM0A_Con = 3;


        // Incrementing counter TCNT0
        if(CTCMode==1)
        {

        if(IOREG[0x32].data == IOREG[0x29].data)
            {    IOREG[0x32].data = 0x0;
                IOREG[0x38].data = IOREG[0x38].data | 0x10;
                printf("\n**!!TCNT0 = OCR0A!!**\n");
                printf("\nOC0A set, TIFR: %X\n",IOREG[0x38].data);
                bit=~bit;
            }
        else
            IOREG[0x32].data += 0x01;
        // Setting Timer0 overflow flag == 0
        if(IOREG[0x32].data == 256)
```

```
        {
            IOREG[0x38].data = IOREG[0x38].data  | 0x02;
            printf("\nTOV0 set, TIFR: %X\n",IOREG[0x38].data);

        }
    if(COM0A_Con == 1)
        {
            if(bit==0xFF)
            IOREG[0x18].data = IOREG[0x18].data | 0x01;
            else
            IOREG[0x18].data = IOREG[0x18].data & 0xFE;
            printf("\nPORTB: %X\n",IOREG[0x18].data);
        }
    else if(COM0A_Con == 2)
        {    if(bit==0xFF)
            IOREG[0x18].data = IOREG[0x18].data & 0xFE;
            printf("\nPORTB: %X\n",IOREG[0x18].data);
        }
    else if(COM0A_Con == 3)
        {    if(bit==0xFF)
            IOREG[0x18].data = IOREG[0x18].data | 0x01;
            printf("\nPORTB: %X\n",IOREG[0x18].data);
        }

    }
        }
    else
        CTCMode = 0;
}
```

In Clear Timer on Compare or CTC mode (WGM0[2:0] = 2), the OCR0A Register is used to manipulate the counter resolution. In CTC mode the counter is cleared to zero when the counter value (TCNT0) matches the OCR0A.[11] The OCR0A defines the top value for the counter, hence also its resolution. This mode allows greater control of the Compare Match output frequency. It also simplifies the operation of counting external events.

An interrupt can be generated each time the counter value reaches the OCR0A value by using the OCF0A Flag. If the interrupt is enabled, the interrupt handler routine can be used for updating the maximum value. However, changing OCR0A to a value close to 0x00 when the counter is running with none or a low prescaler value must be done with care since the CTC mode does not have the double buffering feature. If the new value written to OCR0A is lower than the current value of TCNT0, the counter will miss the Compare Match. The counter will then have to count to its maximum value (0xFF) and wrap around starting at 0x00 before the Compare Match can occur.

In the code, the mode of the timer is checked using the WGM bits. The compare match moes of the timer are decided by checking the COM bits. The OC0A is changed using the OR and AND operation of 0x18 register when the TCNT0A matches OCR0A. The TCNT0A register is incremented until it matches OCR0A.

The CTC frequency for the output can be calculated by the following equation:

$$f_{OCnx} = \frac{f_{clk\_I/O}}{2.N.(1 + OCRnx)}$$

The N variable represents the prescale factor (1, 8, 64, 256, or 1024). Here we have taken the clock frequency as 1MHz and N as 1. In the below output waveform, we can verify the frequency of the timer.

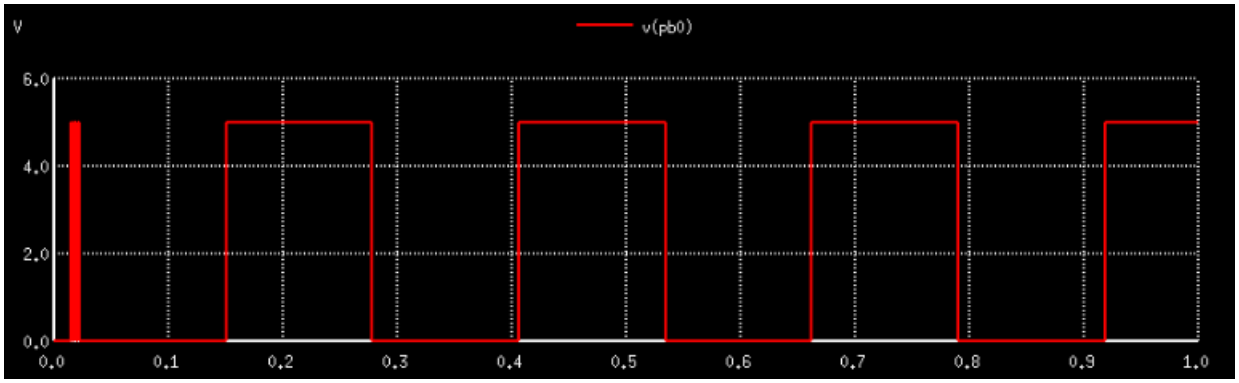

Figure 5.1: OUTPUT

Here, the value of OCR0A=40. So, the output will toggle whenever TCTNT0 reaches 40. The frequency of the clock is calculated using the above equation, as 1Mhz/(2 x 41) which comes out to be around 12.5kHz.

3. Phase correct PWM Mode

```
void timer(int flag)
{
// TIMER0 by SM
        unsigned char temp = 0x0;
        char k=IOREG[0x2A].data &0x01;
        if(k == 0x01)
        {        char top;
                if(debugMode == 1)
                        printf("\n**!!TIMER 0 operation detected!!**\n");

                int PhaseCorrectPWM = 0, OC0A_Con = 0;
                Hex2Bin(3,IOREG[0x2A].data);//TCCR0A
```

```
if(bin[3].arr[1] == 0 && bin[3].arr[0]==1){//WGM00 & WGM01
↪  checked
        PhaseCorrectPWM = 1;
        printf("PhaseCorrectPWM mode");
        }
else
        PhaseCorrectPWM = 0;

if(bin[3].arr[7] == 1 && bin[3].arr[6] == 0)//non inverting
        OC0A_Con = 1;
else if (bin[3].arr[7] == 1 && bin[3].arr[6] == 1)// inverting
        OC0A_Con = 0;
else
        printf("Not a Timer mode");

if (IOREG[0x33].data & 0x08 ==0x08)//TCCR0B, to check WGM02 Set
        top= IOREG[0x29].data;//OCR0A
else
        top=0xFF;

IOREG[0x18].data = IOREG[0x18].data | 0x01;//PORTB

//reset TOV0
if (IOREG[0X32].data == 0xFF)//TCNT0
        IOREG[0x38].data = IOREG[0x38].data &
        ↪  0xFD;//TIFR

// Decrmenting counter TCNT0
        k=IOREG[0x38].data & 0x02;
if(IOREG[0x32].data > 0 && k==0x00)//TOV is reset
        IOREG[0x32].data -= 0x01;
else if(IOREG[0x32].data == 0)
{
        IOREG[0x38].data = IOREG[0x38].data | 0x02;//TOV0 set
        printf("\nTOV0 set, TIFR: %X\n",IOREG[0x38].data);
        IOREG[0x38].data = IOREG[0x38].data & 0xEF;
        ↪           //OCF0 reset
        printf("\nTIFR after resetting OCF0A:
        ↪  %X\n",IOREG[0x38].data);
}

//If TOV0 is set, increment TCNT0
temp = IOREG[0x38].data & 0x02;
if(temp == 0x02 && IOREG[0X32].data < 0xFF)
        {
```

```
                                        IOREG[0x32].data += 0x01;
                        }

                    // IF TCNT0 == OCR0A
                    if(IOREG[0x32].data == IOREG[0x29].data)
                            {
                                    IOREG[0x38].data = IOREG[0x38].data | 0x10;//OCF0
                                    ↪ set
                                    printf("\n**!!TCNT0 = OCR0A!!**\n");
                                    printf("\nTIFR: %X\n",IOREG[0x38].data);
                            }

            k= IOREG[0X38].data & 0x02;
                    temp = IOREG[0x38].data & 0x10;//match condition
                    if(temp == 0x10 && k== 0x02)// TOV is set...incre
                    {
                    if (OC0A_Con == 1){
                            IOREG[0x18].data = IOREG[0x18].data & 0xFE;//reset OCF0
                            printf("\nPORTB: %X\n",IOREG[0x18].data);
                                    }
                    else{
                            IOREG[0x18].data = IOREG[0x18].data | 0x01 ;// set OCF0
                            printf("\nPORTB: %X\n",IOREG[0x18].data);
            }
                    }
                    else if(temp != 0x10 && k != 0x02){// Tov is reset...dec
                            if (OC0A_Con == 0){
                                    IOREG[0x18].data = IOREG[0x18].data | 0x01;//set
                                    ↪ OCF0
                                    }
                            else
                                    IOREG[0x18].data = IOREG[0x18].data &
                                    ↪ 0xFE;//reset OCF0
                            }

                    printf("\nTCCNT0 : %X ,OCR0A:
                    ↪ %X\n",IOREG[0x32].data,IOREG[0x29].data);
                    SetPins(1);

        }
}
```

In the code, timer operation is detected. Then top, PhaseCorrectPWM, OC0ACon is initialised. Through Hex2Bin function TCCR0A register is put into BinArray for later use. WGM00 and WGM01 is checked to find that it is Phase Correct PWM mode or not. Then the output mode (inverting or Non inverting)is found.WGM02

is then checked to define top. PORTB is set to 1 and TOV (Timer overflow flag) is reset before the timer operation. Timer operation starts with decrementing. If TCNT0 is greater than zero and TOV is reset , it will decrement till it becomes zero and the TOV is set. When it is zero, OCF0 will be reset to find match while incrementing. Incrementing will start if TOV is set and TCNT0 is less than top. If TCNT0 is equal to OCR0A, match happens. If TOV is set and TCNT0 is incrementing, for non inverting, output will be zero and inverting , it will be one. If TOV is reset and TCNT0 is decrementing, for non inverting, output will be one and inverting , it will be zero. Then the registers are printed and the pins are set to get the output.

The PWM frequency for the output when using phase correct PWM can be calculated by the following equation:

$$f_{OCnxPCPWM} = \frac{f_{clk\_I/O}}{N.510}$$

The N variable represents the prescale factor (1, 8, 64, 256, or 1024). In ATtiny, the clock frequency used is 1MHz and N is taken as 1. In the below output waveform, we can verify the frequency of the timer.

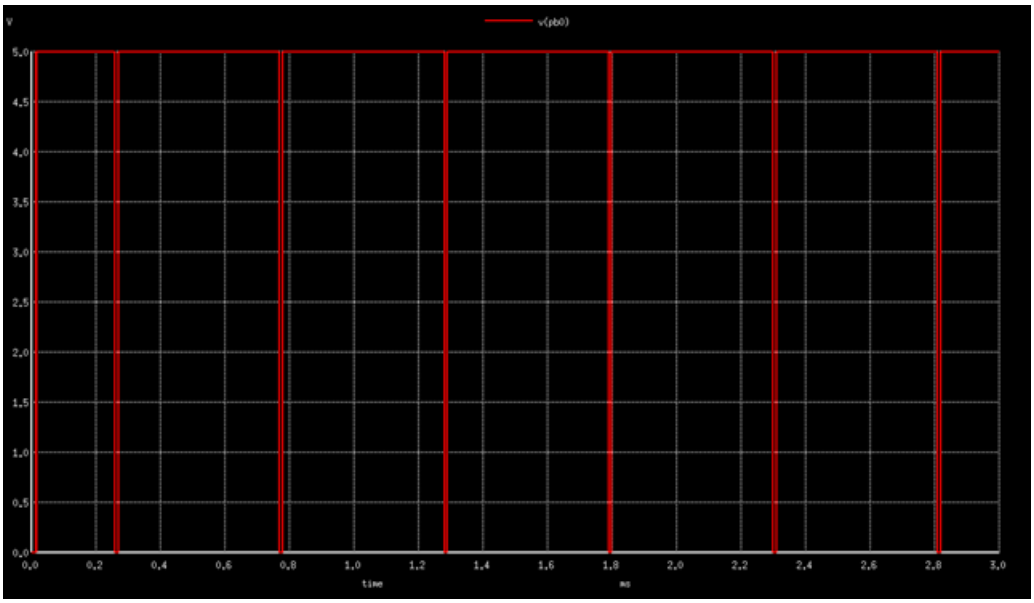

Figure 5.2: OUTPUT

Here , the OCR0A is 0xF7 . So , the output will be low for very less duration. The Time Period of the clock is calculated by taking inverse of the above equation as 510/1MHz which comes out to be around 0.51 ms.

5.1.41 Timer1 Implementation

Block Diagram:

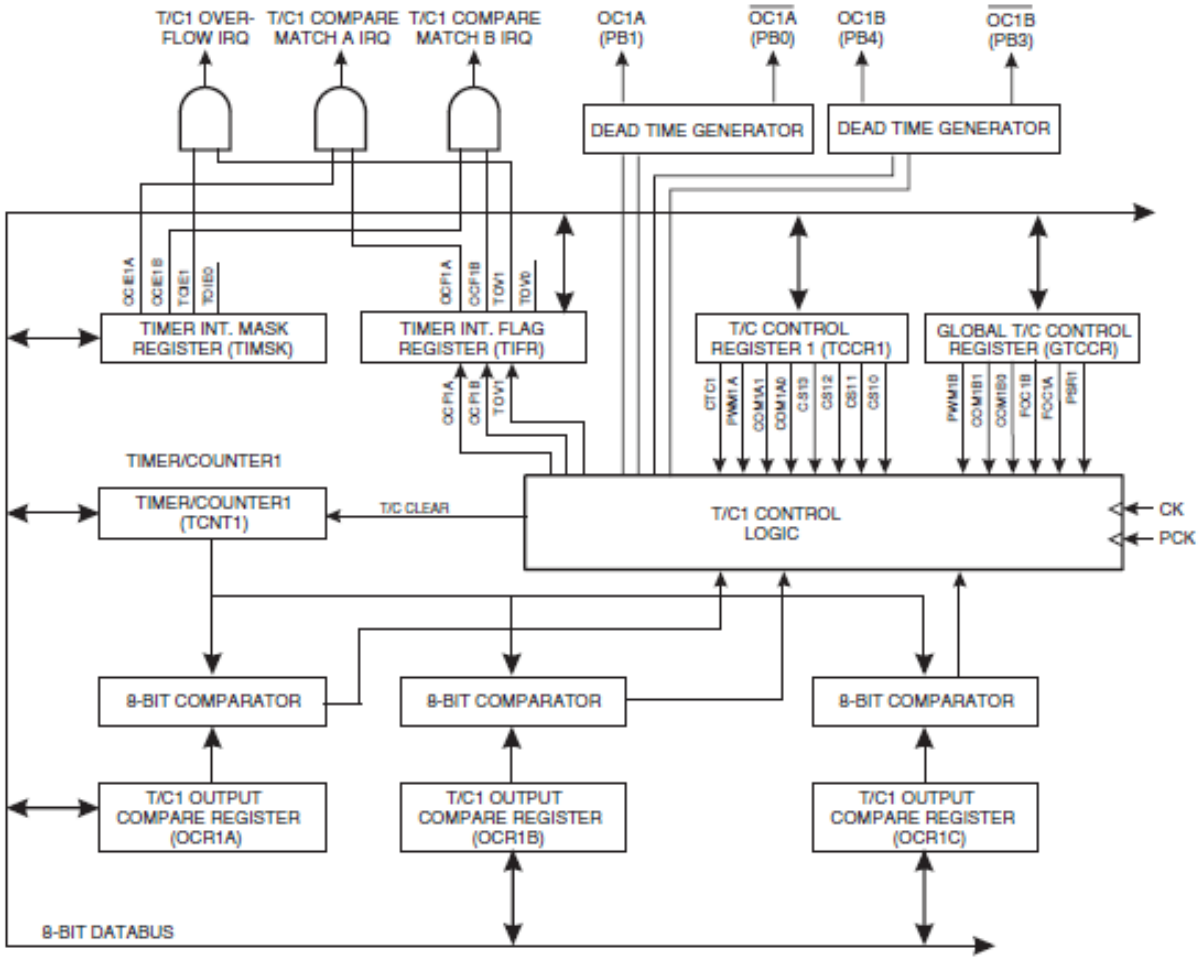

Figure 5.3: Timer 1 Block Diagram

C code:

```
void timer1(int flag)
{
//    TIMER 1 OPERATION BY SK        02/06/2020
    char temp = 0x0; static unsigned int count=0;
    if(IOREG[0x2E].data >= 0x01 )//0x2E=OCR1A
    {    int COM1A_Con=0,mode=0, CS_Con = 0;

        if(debugMode == 1)
            printf("\n**!!TIMER 1 operation detected!!**\n");

        Hex2Bin(3,IOREG[0x30].data);
        CS_Con=IOREG[0X30].data & 0x0F;//Masking the CS bits to get the clocking
        ↪ modes
```

```
mode=(int)pow(2,(int)CS_Con-1);//Refer Timer/Counter1 Prescale Select in
↪  datasheet
if(CS_Con!=0)
{    printf("\nCLOCKING MODE: %X\n",CS_Con);
    IOREG[0x27].data=IOREG[0x27].data|0x01;//Enabling the Plock bit in
    ↪  PLLCSR
    temp = IOREG[0x27].data & 0x07;//Checking whether Asynchronous mode
    ↪  is enabled or not
    if(temp==0x07)
        printf("Timer1 is in Asynchronous Mode");
    else
        printf("Timer1 is in Synchronous Mode");
    //If TOV1 is set, reset counter TCNT1
    temp = IOREG[0x38].data & 0x04;//Checking the timer overflow
    if(temp == 0x04)
    {
        printf("\n**!!TOV1 overflow!!**\n");
        IOREG[0x2F].data = 0;
        IOREG[0x38].data = IOREG[0x38].data ^ 0x04;
        printf("\nTIFR after toggling TOV1: %X\n",IOREG[0x38].data);
        IOREG[0x38].data = IOREG[0x38].data & 0xBF;
        printf("\nTIFR after resetting OCF1A: %X\n",IOREG[0x38].data);
    }


    if(bin[3].arr[5] == 0 && bin[3].arr[4] == 0)//Checking the compare
    ↪  match options
        COM1A_Con = 0;
    else if(bin[3].arr[5] == 0 && bin[3].arr[4] == 1)
        COM1A_Con = 1;
    else if(bin[3].arr[5] == 1 && bin[3].arr[4] == 0)
        COM1A_Con = 2;
    else if(bin[3].arr[5] == 1 && bin[3].arr[4] == 1)
        COM1A_Con = 3;

    temp = IOREG[0x38].data & 0xC0;// For non-inverting operation
    if(temp == 0xC0)
        IOREG[0x38].data = IOREG[0x38].data | 0x40;//Setting OC1A bit

    // Incrementing counter TCNT1
    if(count==mode)//counter to divide the frequency according to the
    ↪  mode
    {    count=0;
        temp = IOREG[0x39].data & 0x04;
        if(IOREG[0x2F].data < 255)
        IOREG[0x2F].data += 0x01;
```

```
        else if(IOREG[0x2F].data == 255 && temp==0x04)//IF TCNT1 == 255
        ↪  and TOIE1 is set
        {
            IOREG[0x38].data = IOREG[0x38].data | 0x04;
            printf("\nTOV1 set, TIFR: %X\n",IOREG[0x38].data);
        }
    }
    else
        count++;//incrementing the counter

    // Setting Timer1 overflow flag == 0
    temp=IOREG[0x39].data & 0x40;//Checking OCIE1A in TIMSK register

    if(IOREG[0x2F].data == IOREG[0x2E].data && temp==0x40)//IF TCNT1 ==
    ↪  OCR1A and OCIE1A is set
    {
        IOREG[0x38].data = IOREG[0x38].data | 0x40;
        printf("\n**!!TCNT1 = OCR1A!!**\n");
        printf("\nTIFR: %X\n",IOREG[0x38].data);
    }

    temp = IOREG[0x38].data & 0x40;

    if(COM1A_Con == 1 )//Checking the Compare match Modes
    {
        if(temp==0x40)
            IOREG[0x18].data = IOREG[0x18].data | 0x01;
        else if(temp!=0x40)
            IOREG[0x18].data = IOREG[0x18].data & 0xFE;
        printf("\nPORTB: %X\n",IOREG[0x18].data);
    }
    else if(COM1A_Con == 2 )
    {
        if(temp == 0x40)
            IOREG[0x18].data = IOREG[0x18].data & 0xFE;
    }
    else if(COM1A_Con == 3 )
    {
        if(temp == 0x40)
            IOREG[0x18].data = IOREG[0x18].data | 0x01;
    }

    printf("\nTCCNT1 : %X ,OCR1A:
    ↪  %X\n",IOREG[0x2F].data,IOREG[0x2E].data);
    SetPins(1);
}
```

```
        else
            printf("\nCLOCKING MODE: %X , TIMER STOPPED\n",CS_Con);    //Stopping
            ↪  timer in mode 1

    }
}
```

The Timer1 is a general purpose 8-bit Timer/Counter module that has a separate prescaling selection from the separate prescaler.[11]

The Timer1 general operation is described in the asynchronous mode and the operation in the synchronous mode is mentioned only if there are differences between these two modes. The Timer1 register values go through the internal synchronization registers, which cause the input synchronization delay, before affecting the counter operation.

The Timer1 features a high resolution and a high accuracy usage with the lower prescaling opportunities. It can also support two accurate, high speed, 8-bit Pulse Width Modulators using clock speeds up to 64 MHz (or 32 MHz in Low Speed Mode). In this mode, Timer1 and the output compare registers serve as dual standalone PWMs with non-overlapping non-inverted and inverted outputs. Refer to page 86 for a detailed description on this function. Similarly, the high prescaling opportunities make this unit useful for lower speed functions or exact timing functions with infrequent actions.

In order to implement the above functionality. The mode of the timer is stored in a variable "mode" using the value of CS bits. A counter variable "count" counts from 0 to the value of mode. If the value matches the value of mode, the TCNT1 is incremented and the "count" variabe is set to zero. Hence the output is prescaled.

When the TCNT1 matches OCR1A, the OC0A is toggled by setting/resetting the 0th bit of the 0x18 register as per the compare match mode of the timer1. In this way the functionality is implemented in the above code.

For calculations and examples based on Timer 1 refer page 64.

5.2 Implemented Examples

5.2.1 Example to test ATtiny25:Driving LEDs circuit:

In this example the ATtiny 25 is used to drive the LEDs. The LEDs are given different voltage levels using resistors and transistors. Therefore the LEDs show different fading and glowing effects. The schematic is shown below.

Schematic of Driving LEDs circuit:

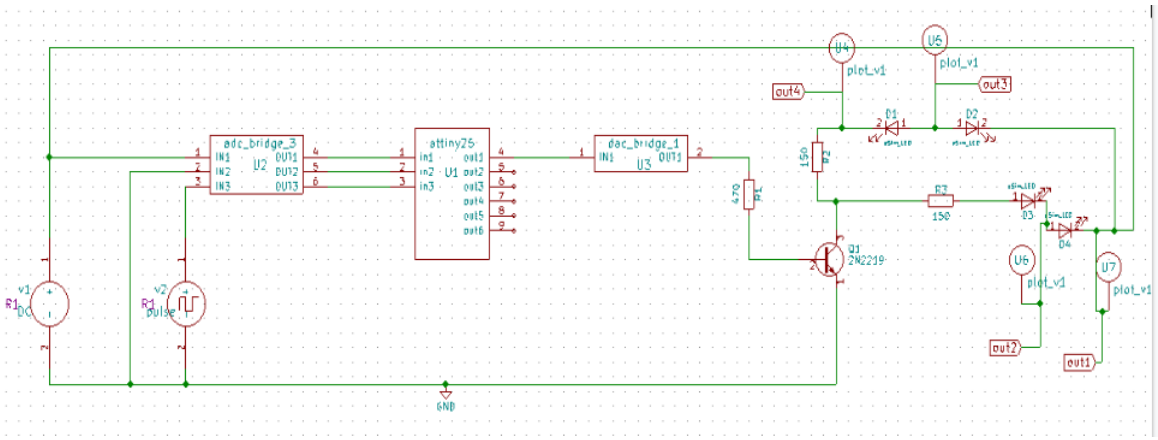

Figure 5.4: Schematic of Driving LEDs circuit

Output:

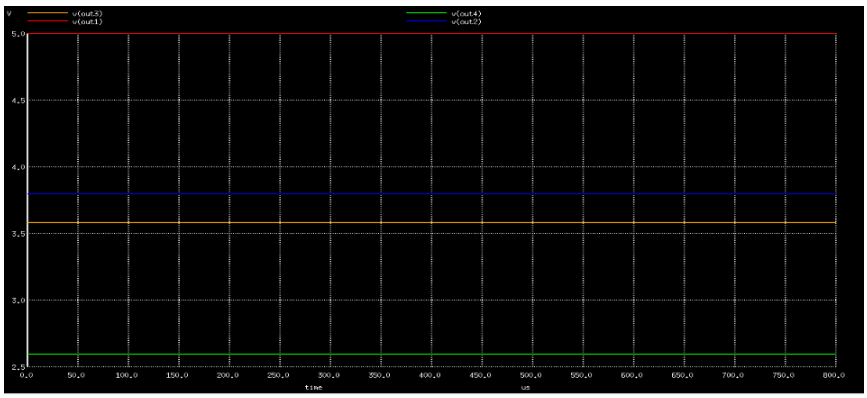

Figure 5.5: Output

5.2.2 Example to test ATtiny45:PPM Generator:

In PPM the amplitude and width of the pulses is kept constant but the position of each pulse is varied accordance with the amplitudes of the sampled value of the modulating signal.The Pulse Position Modulation (PPM) is a modulation technique designed to achieve the goals like simple transmitter and receiver circuitry, noise performance, constant bandwidth and the power efficiency and constant transmitter power. The Pulse Position Modulation (PPM) can be actually easily generated from a PWM waveform which has been modulated according to the input signal waveform. The technique is to generate a very small pulse of constant width at the end of the duty time of each and every PWM pulses by using a differentiator and mno stable multivibrator as follows:

Schematic of the PPM Generator:

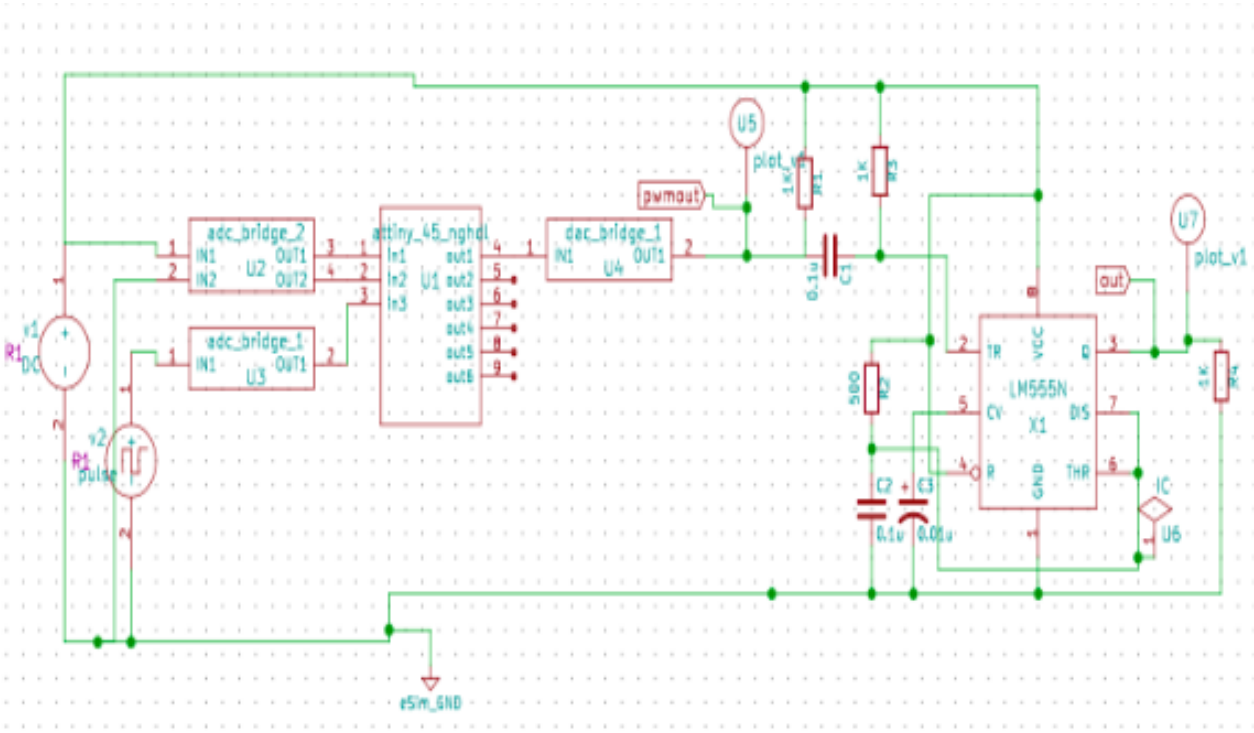

Figure 5.6: Schematic of PPM Generator

Output:

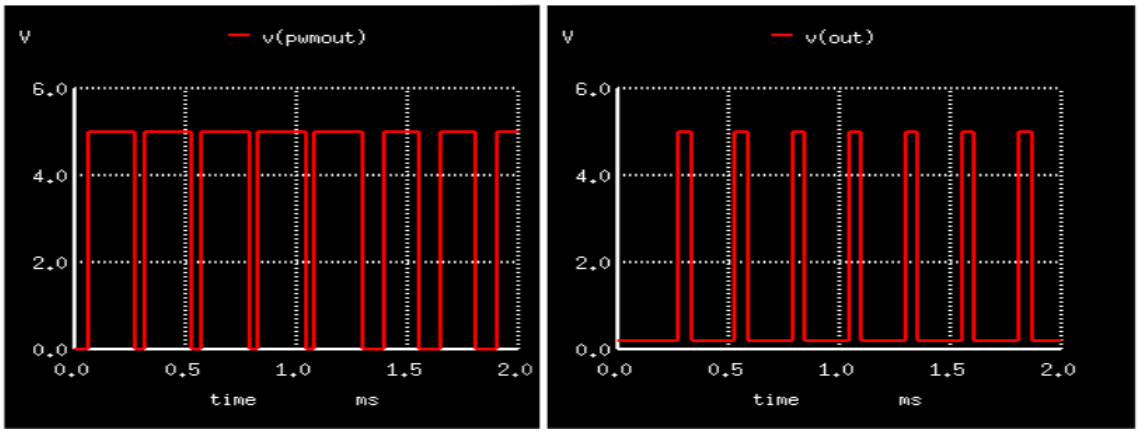

Figure 5.7: Output 1

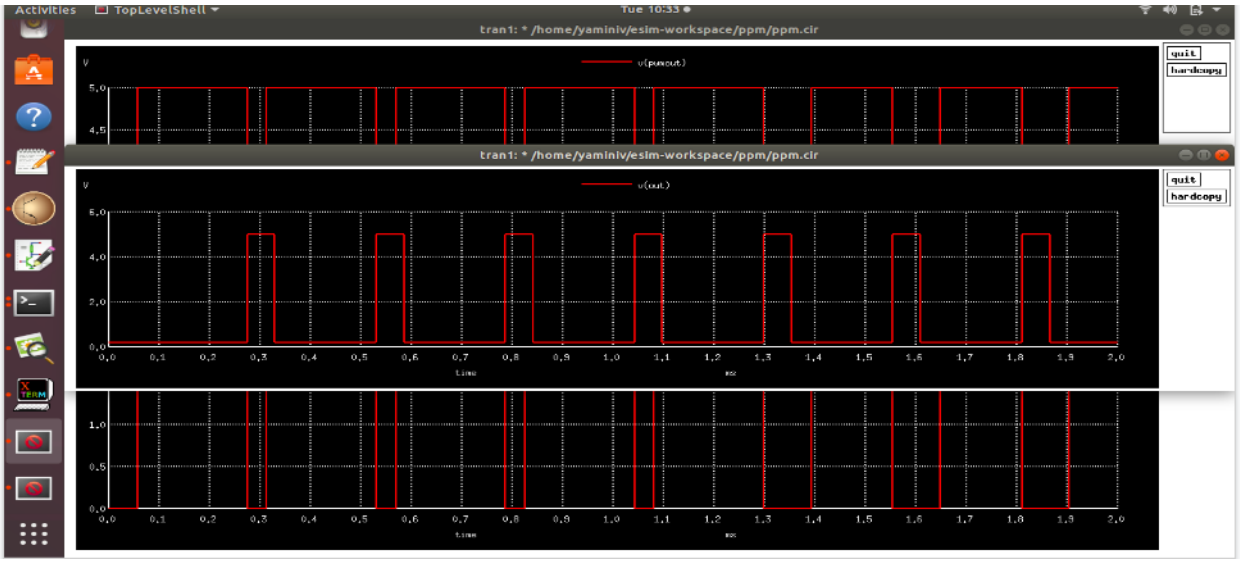

Figure 5.8: Output 2

5.2.3 Example to test ATtiny85:Square Wave Generator

Delay using micro-controller timer is the most accurate and surely the best method over the timer ICs like 555 Timer.

A timer can be generalized as a multi-bit counter which increments/decrements itself on receiving a clock signal and produces an interrupt signal up on roll over. When the counter is running on the processor's clock , it is called a "Timer", which counts a predefined number of processor clock pulses and generates a programmable delay.[13]

Here we have implemented Square Wave Generator with variable frequency and duty cycle using the timer1 of ATtiny microcontroller. The Timer1 is programmed according to the needs of the user. The user can vary the input frequency by giving inputs to pins PB2 and PB3. The user can also vary the duty cycle by giving inputs to pins PB4 and PB5. The output of the required square wave is taken at pin 0 of ATtiny microcontroller.

The frequency and duty cycle of the timer 1 is given by:

$$f_{PWM} = \frac{f_{TCK1}}{OCR1C + 1} \qquad D = \frac{OCR1A + 1}{OCR1C + 1}$$

The value of the f_{TCK1} is equal to the frequency of the internal clock divided by the mode of the timer.

The OCRC1 is taken as 255 in the implementation of timer 1. Hence the value of f_{PWM} becomes 3.912 kHz frequency for mode 1 and internal clock frequency 1MHz. The Duty Cycle becomes 50 percent for OCR1A=127

The mode of the timer is set by varying the value of the TCCR1 according to the user input and the frequency of the timer is set by varying the value of the OCR1A according to the user input as shown in the C code below.

C Code:

```
#include <avr/io.h>  //including header files
#include <avr/sleep.h>
#include <math.h>
int main(void)
{
      DDRB|=_BV(PB0);//setting PB0 as output pin
      TIMSK=0xFF;  //setting the timer 1 mask
```

```
    unsigned char a=PINB&0x0C;//masking the frequency inputs
    unsigned char b=PINB&0x30;//masking the duty cycle inputs for rise and fall
    ↪  time

    if(b==0x00)  //checking the duty cycle inputs
        TCCR1=0x11; //setting the control register of Timer 1
    else if(b==0x10)
        TCCR1=0x12;
    else if(b==0x20)
        TCCR1=0x13;
    else if(b==0x30)
        TCCR1=0x14;
    if(a==0x00) //checking the frequency inputs
        OCR1A=1;//setting the output compare register of Timer 1
     else if(a==0x04)
        OCR1A=63;
     else if(a==0x08)
        OCR1A=128;
     else if(a==0x0C)
        OCR1A=254;
    while(1)
    {
        };

}
```

Schematic of the Square Wave Generator:

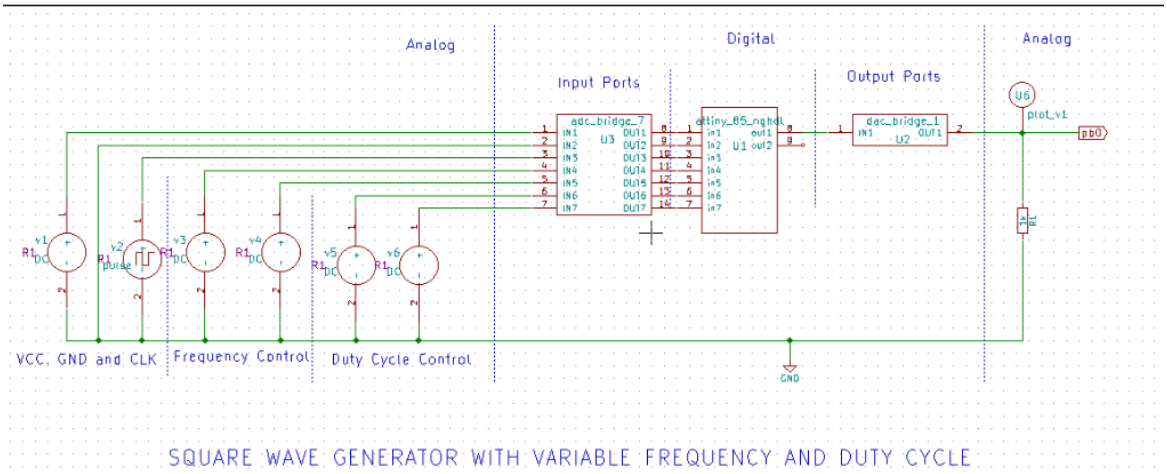

Figure 5.9: Schematic of Square Wave Generator

Output:

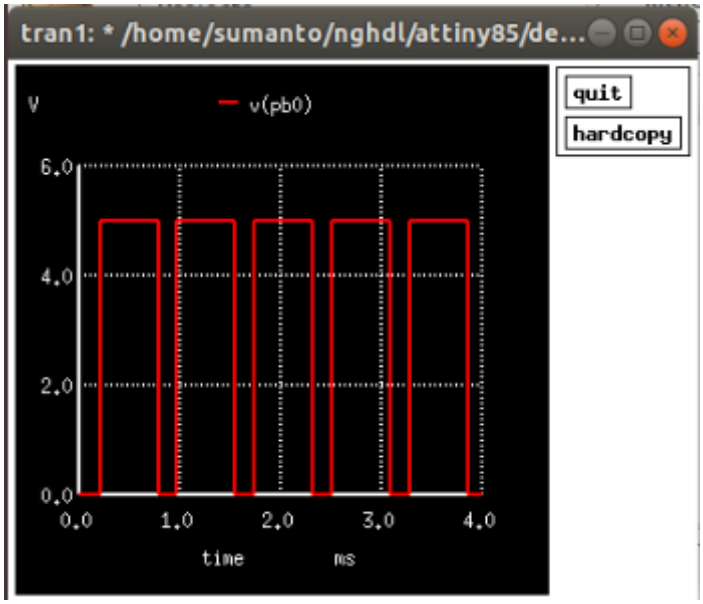

Figure 5.10: Output for Frequency input=01 and Duty Cycle input=01

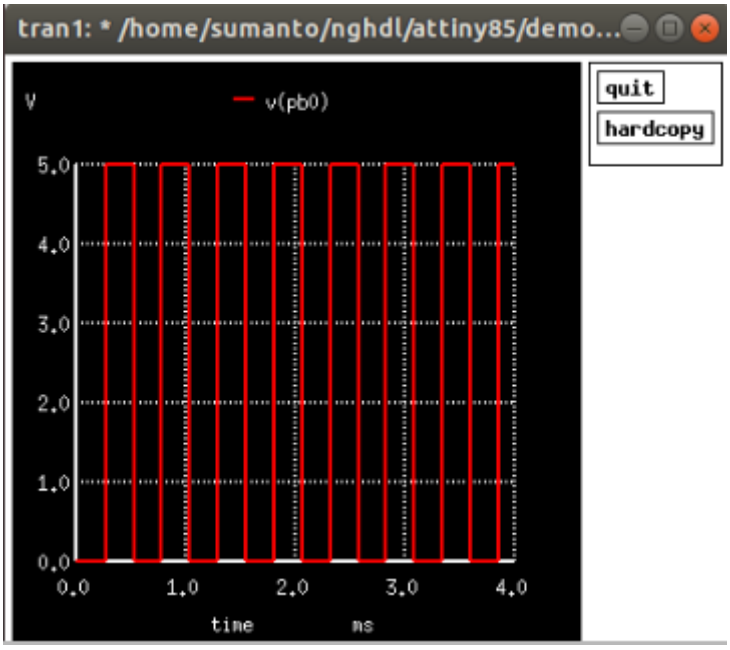

Figure 5.11: Output for Frequency input=00 and Duty Cycle input=10

Calculations for Frequency input=01 and Duty Cycle input=01

Assuming Clock frequency as 1MHz.

$$TimerMode = 2$$

$$OCR1A = 63$$

$$f_{TCK1} = \frac{1MHz}{2} = 500kHz$$

$$f_{PWM} = \frac{500kHz}{255+1} = 1.953kHz$$

$$D = \frac{63+1}{255+1} = 0.25$$

In Inverting mode,

$$D = 1 - 0.25 = 0.75$$

Calculations for Frequency input=00 and Duty Cycle input=10

Assuming Clock frequency as 1MHz.

$$TimerMode = 1$$

$$OCR1A = 127$$

$$f_{TCK1} = \frac{1MHz}{1} = 1MHz$$

$$f_{PWM} = \frac{1MHz}{255+1} = 3.906kHz$$

$$D = \frac{127+1}{255+1} = 0.5$$

In inverting Mode,

$$D = 1 - 0.5 = 0.5$$

Calculations for Frequency input=10 and Duty Cycle input=11

Assuming Clock frequency as 1MHz.

$$TimerMode = 3$$

$$OCR1A = 255$$

$$f_{TCK1} = \frac{1MHz}{3} = 333.33MHz$$

$$f_{PWM} = \frac{333.33MHz}{255+1} = 1.302kHz$$

$$D = \frac{254+1}{255+1} = 0.996$$

In inverting Mode,

$$D = 1 - 0.996 = 0.004$$

5.2.4 Example to test ATtiny85:Triangular Wave Generator

The triangular wave generator can be designed by adding an integrator to the square wave generator. Hence we have added a low pass RC filter to the above square wave generator to get the triangular wave. The rise time and the fall time of the triangular wave generator can be varied by varying the duty cycle of the square wave as shown in the above example. The frequency of the triangular wave can be varied by varying the frequency of the square wave. Hence we have implemented Triangular Wave Generator with variable rise and fall times and variable frequency.

C Code:

```
#include <avr/io.h> //including header files
#include <avr/sleep.h>
#include <math.h>
int main(void)
{
      DDRB|=_BV(PB0);//setting PB0 as output pin
      TIMSK=0xFF; //setting the timer 1 mask

    unsigned char a=PINB&0x0C;//masking the frequency inputs
    unsigned char b=PINB&0x30;//masking the duty cycle inputs for rise and fall
    ↪  time

    if(b==0x00)//checking the duty cycle inputs
        TCCR1=0x11; //setting the control register of Timer 1
    else if(b==0x10)
        TCCR1=0x12;
    else if(b==0x20)
        TCCR1=0x13;
    else if(b==0x30)
        TCCR1=0x14;
    if(a==0x00) //checking the frequency inputs
        OCR1A=1;//setting the output compare register of Timer 1
     else if(a==0x04)
        OCR1A=63;
     else if(a==0x08)
        OCR1A=128;
     else if(a==0x0C)
        OCR1A=254;
    while(1)
    {
        };

}
```

Schematic of the Triangular Wave Generator:

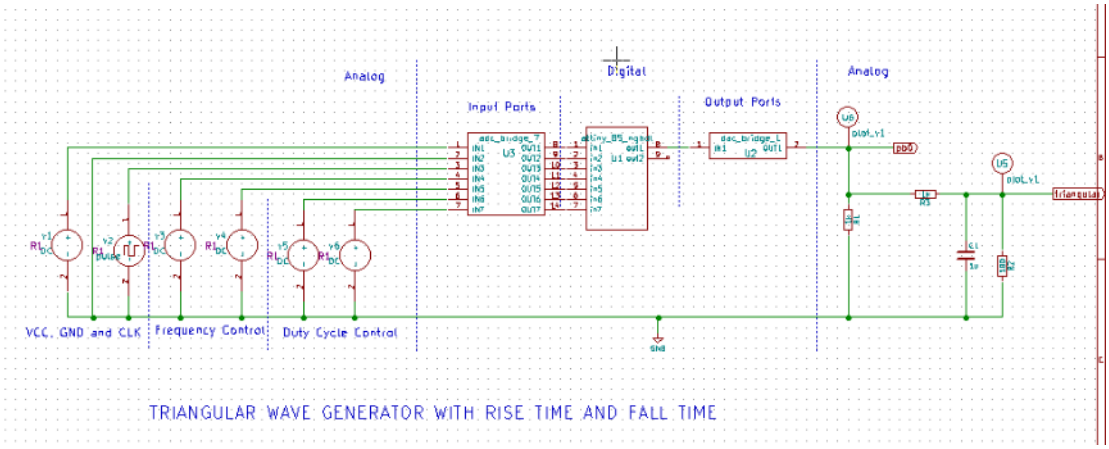

Figure 5.12: Schematic of Triangular Wave Generator

Output:

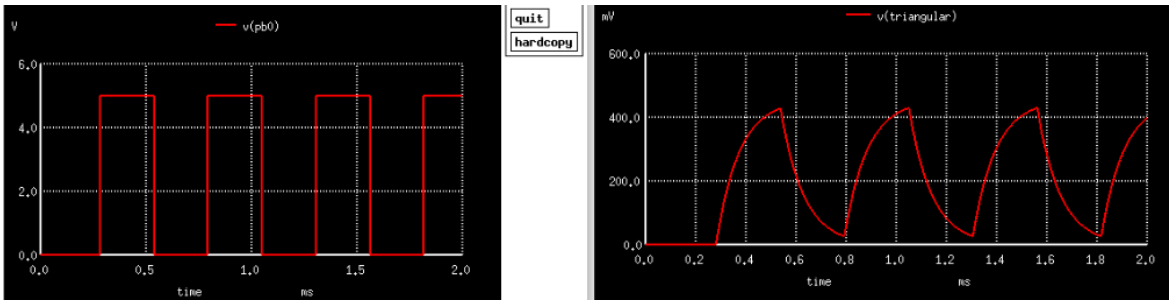

Figure 5.13: Output for PBIN=0x08

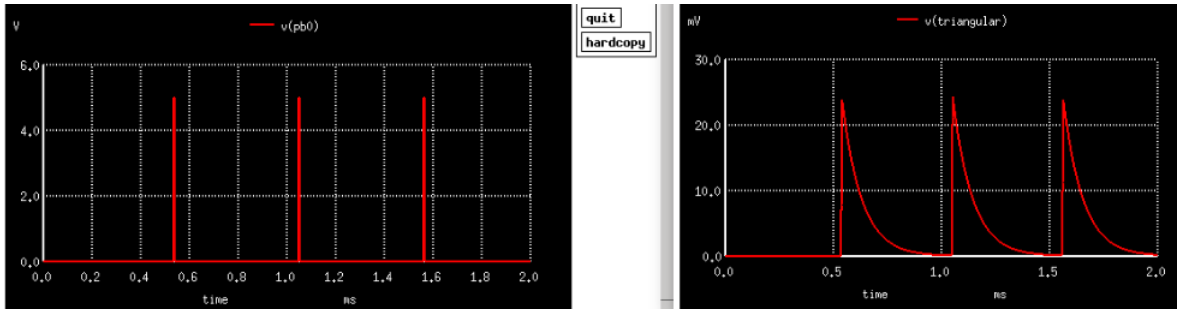

Figure 5.14: Output for PBIN=0x0C

Chapter 6

Conclusion and Future Scope

We were able to achieve the target of enhancing the instruction set emulator, for microcontroller simulations in eSim, using the already existing framework, designed by leveraging the NGHDL feature of eSim. Also, the Timer 0 and Timer 1 modules were implemented. The ATtiny25, ATtiny45, ATtiny85 microcontrollers were realised by using NGHDL for declaring its peripheral and C language for ISA emulation. In the ATtiny series, the interrupts and USI has to be completed further for simulating a fully functional microcontroller. Microcontrollers of other families such as Renesas or PIC could also be modelled using the same architecture.

Bibliography

[1] FOSSEE Official Website. 2020.
URL: https://fossee.in/about

[2] Java T Point Official Website. 2020.
URL:https://https://www.javatpoint.com/what-is-avr-microcontroller

[3] Wikipedia Official Website. 2020.
URL: https://en.wikipedia.org/wiki/KiCad/

[4] Microchip Official Website. 2020.
URL: https://www.microchip.com/downloads/en/DeviceDoc/Atmel/

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

[6] FOSSEE official webpage. 2020.
URL: https://fossee.in/fellowship/2019

[7] Geeks For Geeks official webpage. 2020.
URL: https://www.geeksforgeeks.org/bcd-to-7-segment-decoder

[8] Github FOSSEE NGHDL Repository. 2020. URL:https://github.com/FOSSEE/nghdl

[9] Ubuntu Official Wbpage. 2020.
URL: https://ubuntu.com/tutorials/command-line-for-beginners

[10] AVR GCC documentation. 2020.
URL: https://gcc.gnu.org/wiki/avr-gcc

[11] AVR Instruction Set Manual. 2020.
URL:http://ww1.microchip.com/downloads/en/devicedoc/
atmel-0856-avr-instruction-set-manual.pdf

[12] Circuitstoday Official Website. 2020.
URL: http://www.circuitstoday.com/delay-using-8051-timer

[13] Embedded Thoughts. Official Website. 2020.
URL:https://embeddedthoughts.com/2016/05/25/
getting-started-with-the-attiny85

[14] Embedded Thoughts. Official Website. 2020.
URL:https://embeddedthoughts.com/2016/05/25/
getting-started-with-the-attiny85

[15] Arduino Project Hub. 2020.
URL:https://create.arduino.cc/projecthub/Oniichan_is_ded/
learn-how-to-program-attiny85-and-attiny13a-167359