CSEE 4280: Advanced Digital Design

Trent Jones

Abhi Boggavarapu

Assignment: Exercise 4

02 / 07 / 2025

Table of Contents

Roles and Responsibilities

Abhi	Software/Documentation
Trent	Software/Documentation

Github Code Link

The below is a link to the repository and folder within that includes the code used for this project.

https://github.com/abhibogga/CSEE4280Workspace/tree/main/4_Exercise

Process of Development

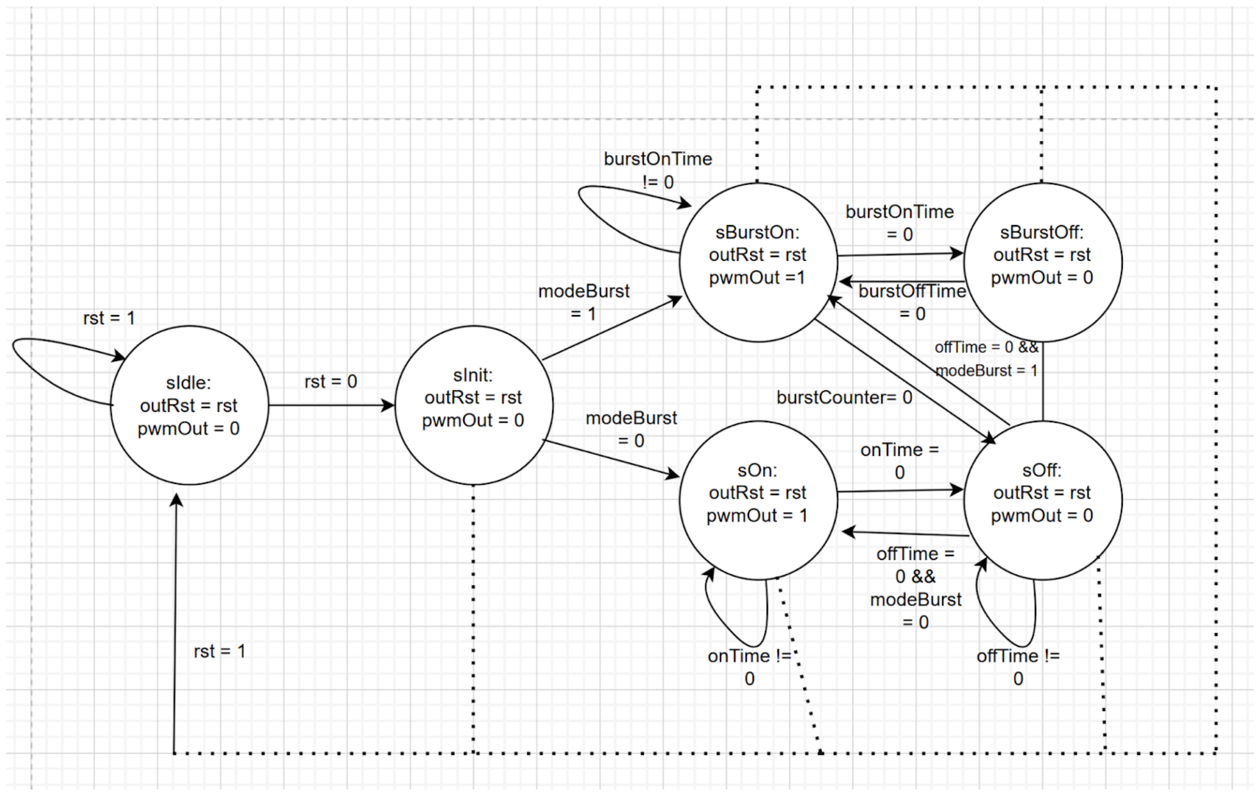
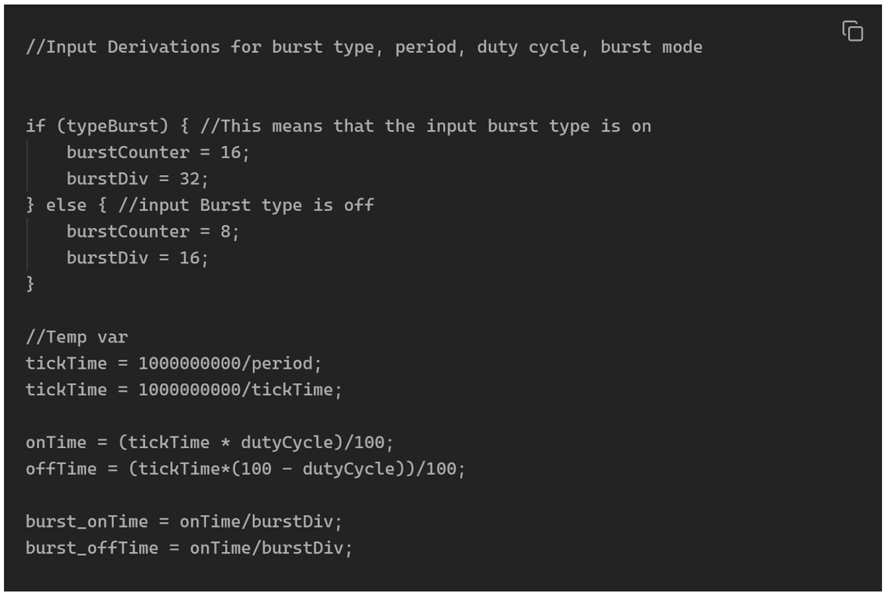


Figure 1.1: State Diagram of pwm.v

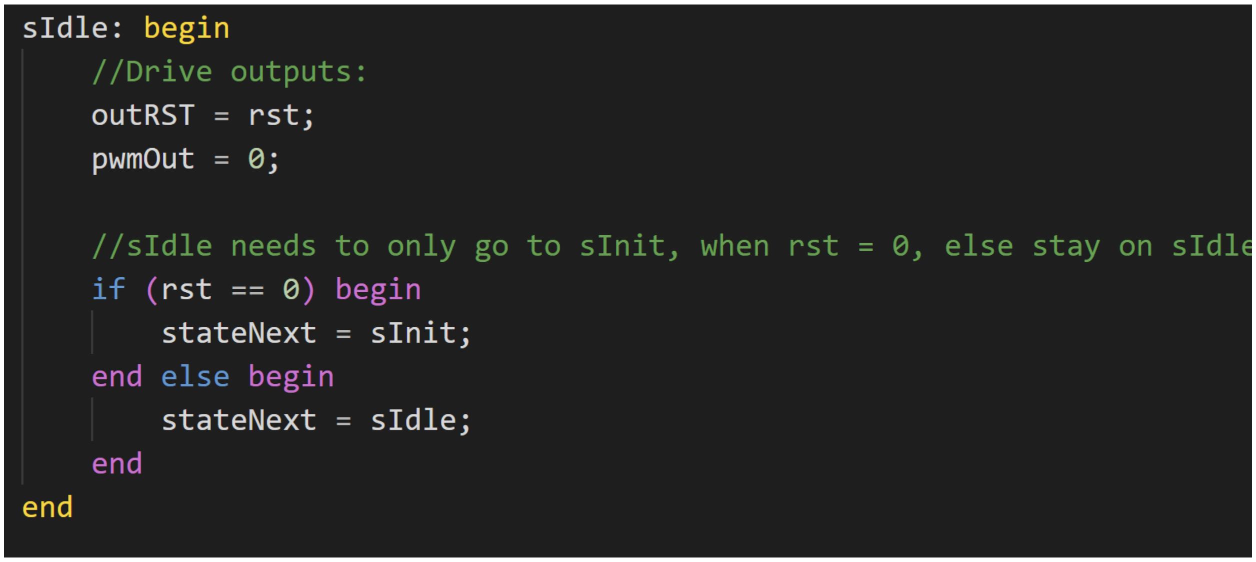
The following image above is the state diagram for the development process of the pwm.v module. As all state diagrams are the inputs drive the outputs but most of the inputs shown are not the actual inputs that are programmed in the module, but derivations of them and are shown in the figure below.



```
//Input Derivations for burst type, period, duty cycle, burst mode


if (typeBurst) { //This means that the input burst type is on
    burstCounter = 16;
    burstDiv = 32;
} else { //input Burst type is off
    burstCounter = 8;
    burstDiv = 16;
}

//Temp var
tickTime = 1000000000/period;
tickTime = 1000000000/tickTime;

onTime = (tickTime * dutyCycle)/100;
offTime = (tickTime*(100 - dutyCycle))/100;

burst_onTime = onTime/burstDiv;
burst_offTime = onTime/burstDiv;
```

Figure 1.2: All derivations of input values relating to state diagram

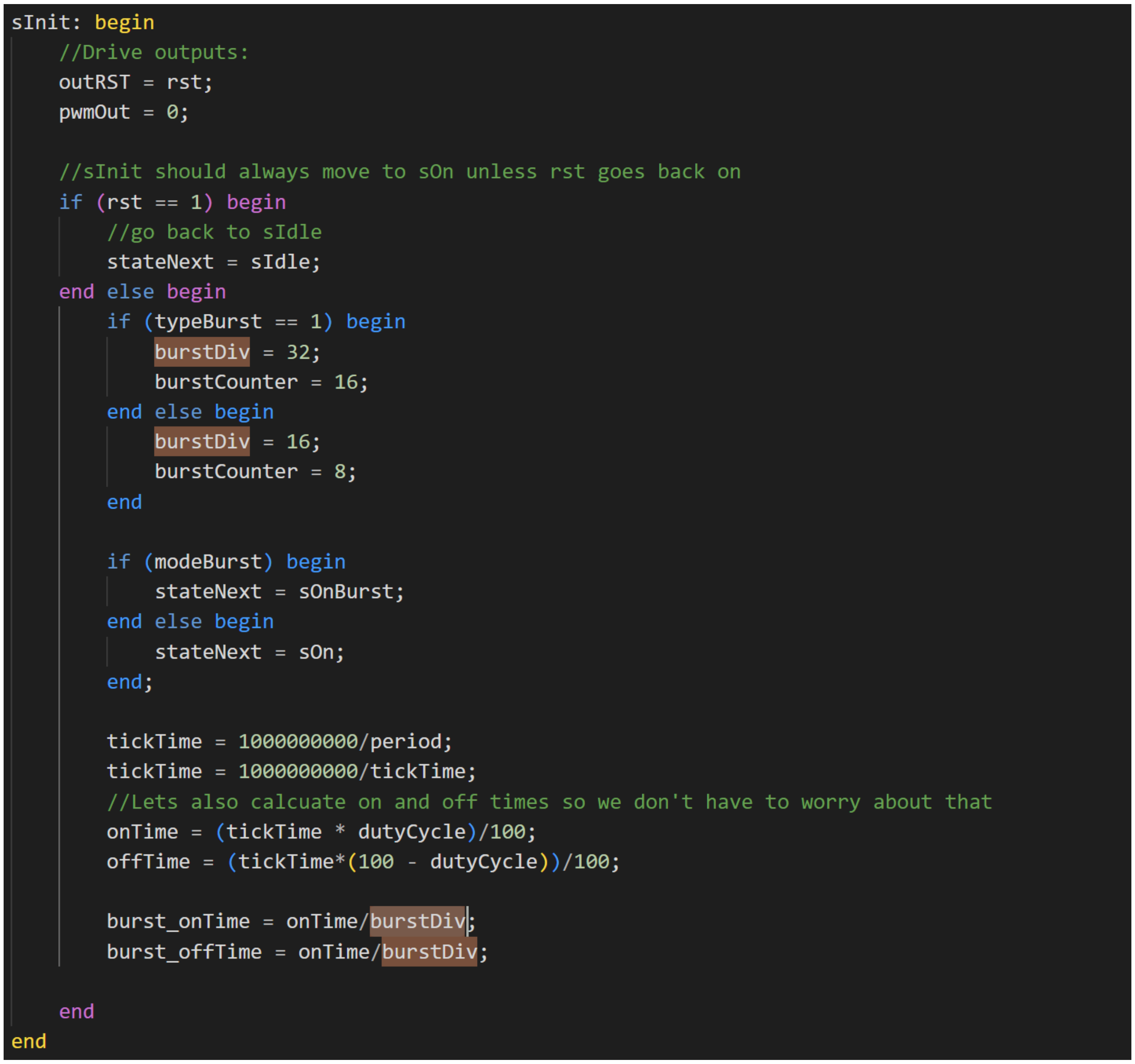
To break down the state machine further each state is part of the pwm signaling process. The first state sIdle is the idling state in which the program simply waits for reset to turn off. During this period the pwmOut is also off. Code shown below.



```
sIdle: begin
    //Drive outputs:
    outRST = rst;
    pwmOut = 0;

    //sIdle needs to only go to sInit, when rst = 0, else stay on sIdle
    if (rst == 0) begin
        stateNext = sInit;
    end else begin
        stateNext = sIdle;
    end
end
```

Figure 1.2: sIdle

Once rst is turned to a low, the pwm signaling process can begin. This is marked by the sInit stage which configures all the variables that rely on the inputs, such as onTime, offTime, burstOnTime, burstOffTime, and others. This is also simply a buffer state that only goes off one time. If the team were to do a revision of this project, this state could be taken out for a more efficient and simpler design. Code shown below.

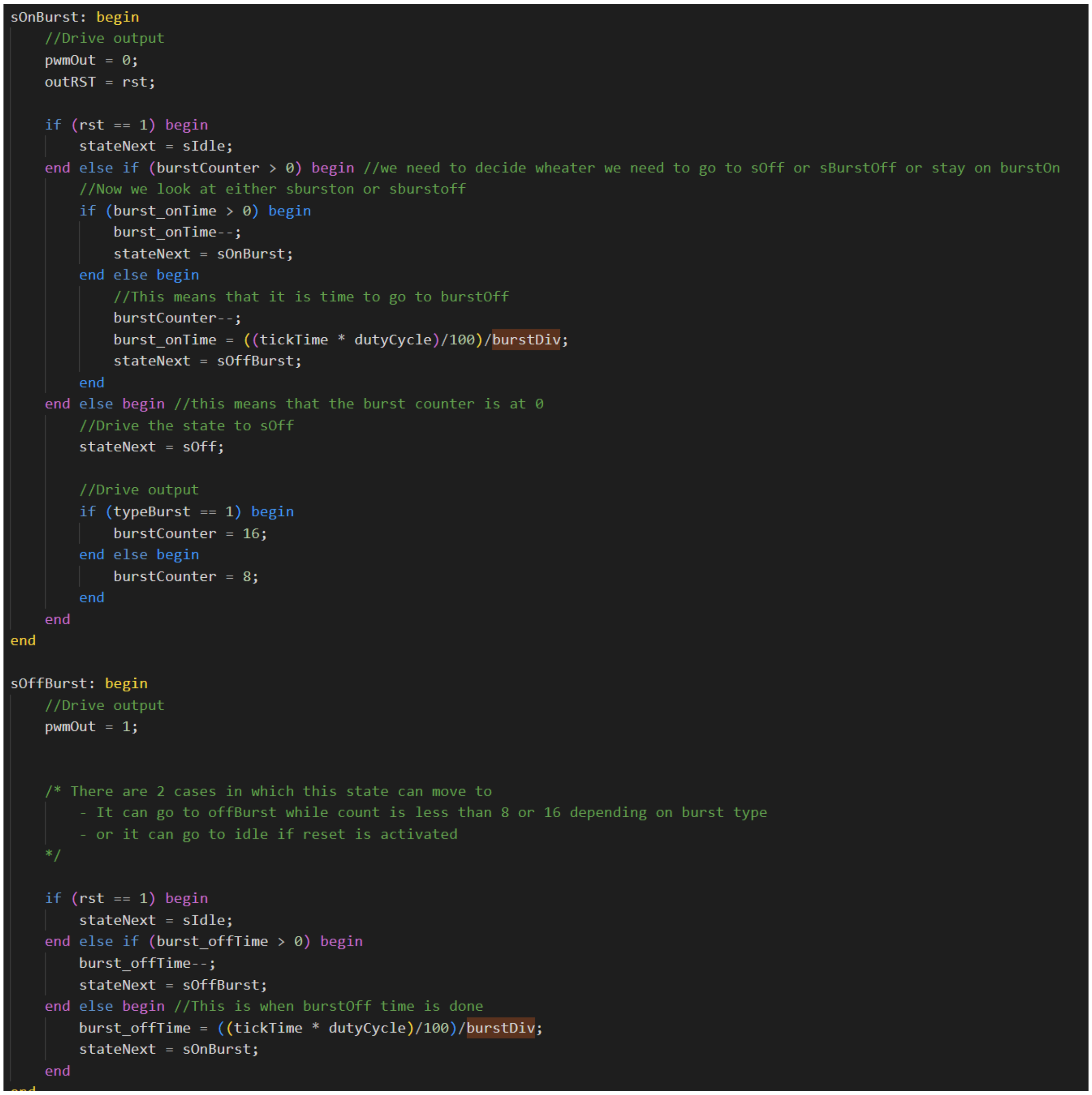


```
sInit: begin
    //Drive outputs:
    outRST = rst;
    pwmOut = 0;

    //sInit should always move to sOn unless rst goes back on
    if (rst == 1) begin
        //go back to sIdle
        stateNext = sIdle;
    end else begin
        if (typeBurst == 1) begin
            burstDiv = 32;
            burstCounter = 16;
        end else begin
            burstDiv = 16;
            burstCounter = 8;
        end

        if (modeBurst) begin
            stateNext = sOnBurst;
        end else begin
            stateNext = sOn;
        end;

        tickTime = 1000000000/period;
        tickTime = 1000000000/tickTime;
        //Lets also calcuate on and off times so we don't have to worry about that
        onTime = (tickTime * dutyCycle)/100;
        offTime = (tickTime*(100 - dutyCycle))/100;

        burst_onTime = onTime/burstDiv;
        burst_offTime = onTime/burstDiv;

    end
end
```

Figure 1.3: sInit

Once the sInit stage has been completed, there are 2 options that the state machine can take, it can either go to the burst section of the state machine or the regular pulse section of the state machine. This is all determined by the input burstMode. If it is turned high, then it will move to the burst mode section, if not it will continue to the regular pulse section.

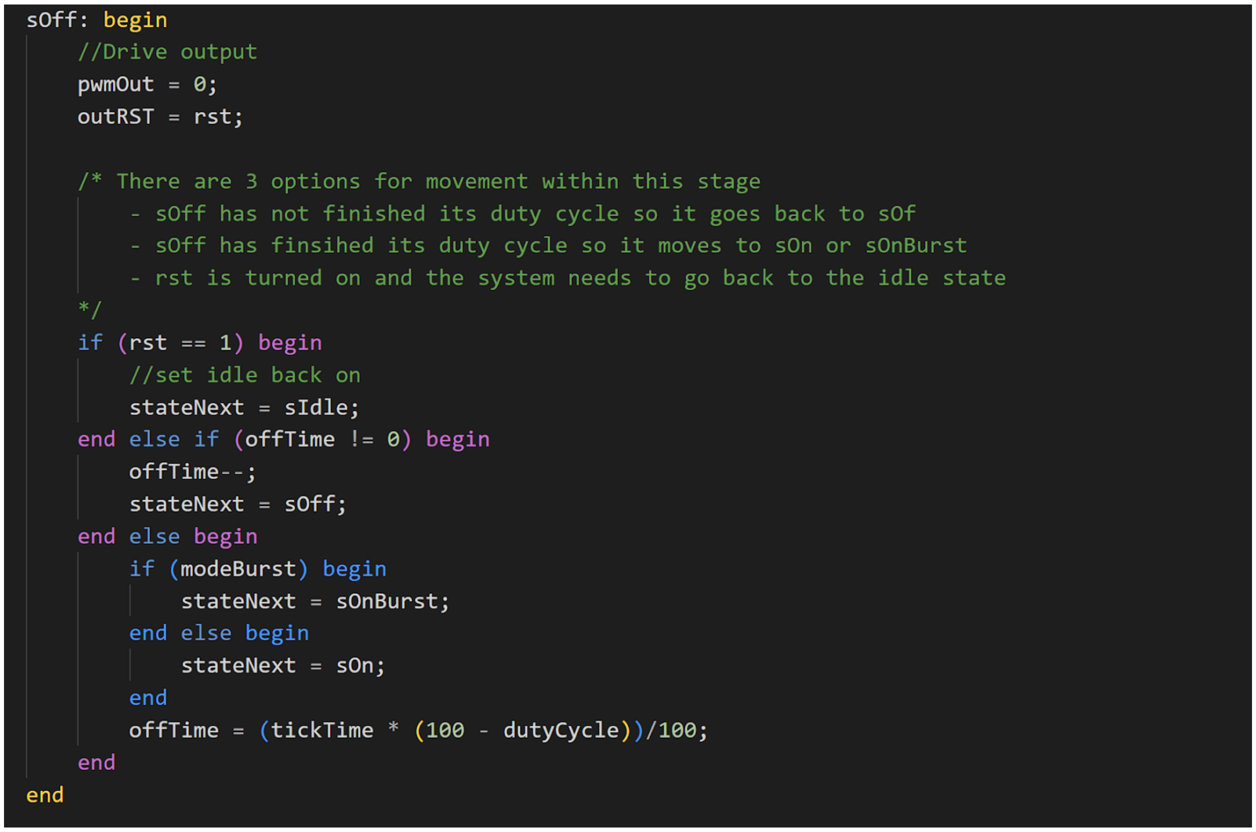
If modeBurst is turned high then it will move to the burst mode section of the state machine. Within the state: sBurstOn, there are a couple of things that happen. First of all when it is in this state, it will drive the pwmOut output high as this is a state when the pulse should be 1. The sBurst will loop for burstOnTime times. Once that variable reaches 0 the machine moves to the burstOffState which does the same thing: loop burstOffTime times. It is important to note that the sOffState drives the pwmOut to low. Once this is done it will move back to the sBurstOn state and repeat the process. The only way that this loop breaks is when the burstCounter reaches 0. Only then will the state machine move to the sOff state. Code shown below.



```
sOnBurst: begin
    //Drive output
    pwmOut = 0;
    outRST = rst;

    if (rst == 1) begin
        stateNext = sIdle;
    end else if (burstCounter > 0) begin //we need to decide wheater we need to go to sOff or sBurstOff or stay on burstOn
        //Now we look at either sburston or sburstoff
        if (burst_onTime > 0) begin
            burst_onTime--;
            stateNext = sOnBurst;
        end else begin
            //This means that it is time to go to burstOff
            burstCounter--;
            burst_onTime = ((tickTime * dutyCycle)/100)/burstDiv;
            stateNext = sOffBurst;
        end
    end else begin //this means that the burst counter is at 0
        //Drive the state to sOff
        stateNext = sOff;

        //Drive output
        if (typeBurst == 1) begin
            burstCounter = 16;
        end else begin
            burstCounter = 8;
        end
    end
end

sOffBurst: begin
    //Drive output
    pwmOut = 1;


    /* There are 2 cases in which this state can move to
        - It can go to offBurst while count is less than 8 or 16 depending on burst type
        - or it can go to idle if reset is activated
    */

    if (rst == 1) begin
        stateNext = sIdle;
    end else if (burst_offTime > 0) begin
        burst_offTime--;
        stateNext = sOffBurst;
    end else begin //This is when burstOff time is done
        burst_offTime = ((tickTime * dutyCycle)/100)/burstDiv;
        stateNext = sOnBurst;
    end
```

Figure 1.4: sOnBurst and sOffBurst

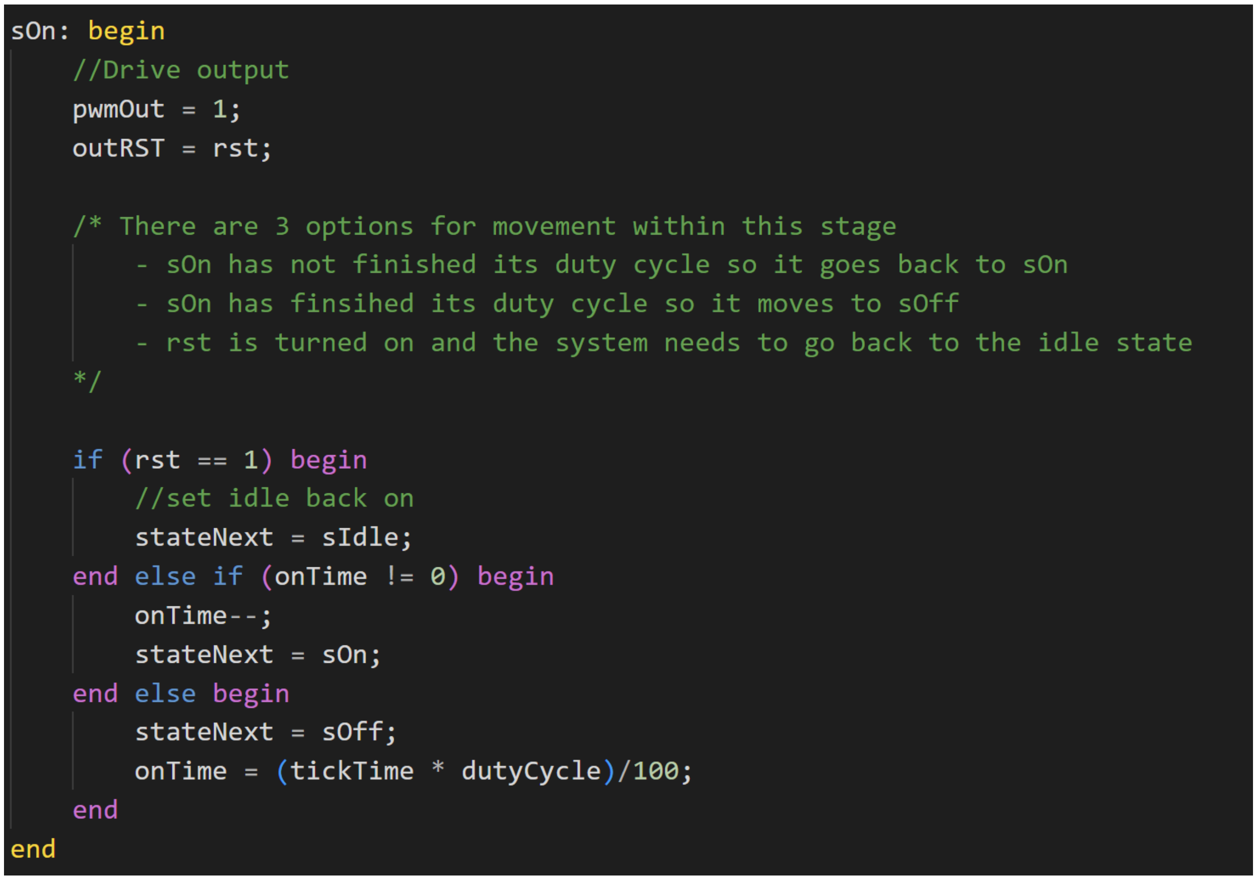
The sOff state is very similar to the sBurstOn and sBurstOff state, it has its own calculated loop time which will keep sending it back to its own state. The sOff state also drives the pwm signal to a low. Once this loop time, which is determined by the variable offTime reaches 0, it will either go back to sBurstOn or sOn depending on the modeBurst input. Code shown below



```
sOff: begin
    //Drive output
    pwmOut = 0;
    outRST = rst;

    /* There are 3 options for movement within this stage
        - sOff has not finished its duty cycle so it goes back to sOf
        - sOff has finsihed its duty cycle so it moves to sOn or sOnBurst
        - rst is turned on and the system needs to go back to the idle state
    */
    if (rst == 1) begin
        //set idle back on
        stateNext = sIdle;
    end else if (offTime != 0) begin
        offTime--;
        stateNext = sOff;
    end else begin
        if (modeBurst) begin
            stateNext = sOnBurst;
        end else begin
            stateNext = sOn;
        end
        offTime = (tickTime * (100 - dutyCycle))/100;
    end
end
```

Figure 1.5: sOff

Finally the last state to cover is the sOn state. This state is exactly the same as the sOff state but it cannot move to the burst section and it drives the pwmOut to a high. It also loops for a certain amount of time: calculated by the variable onTime, and will always move to the offState once that variable hits 0. Code shown below.

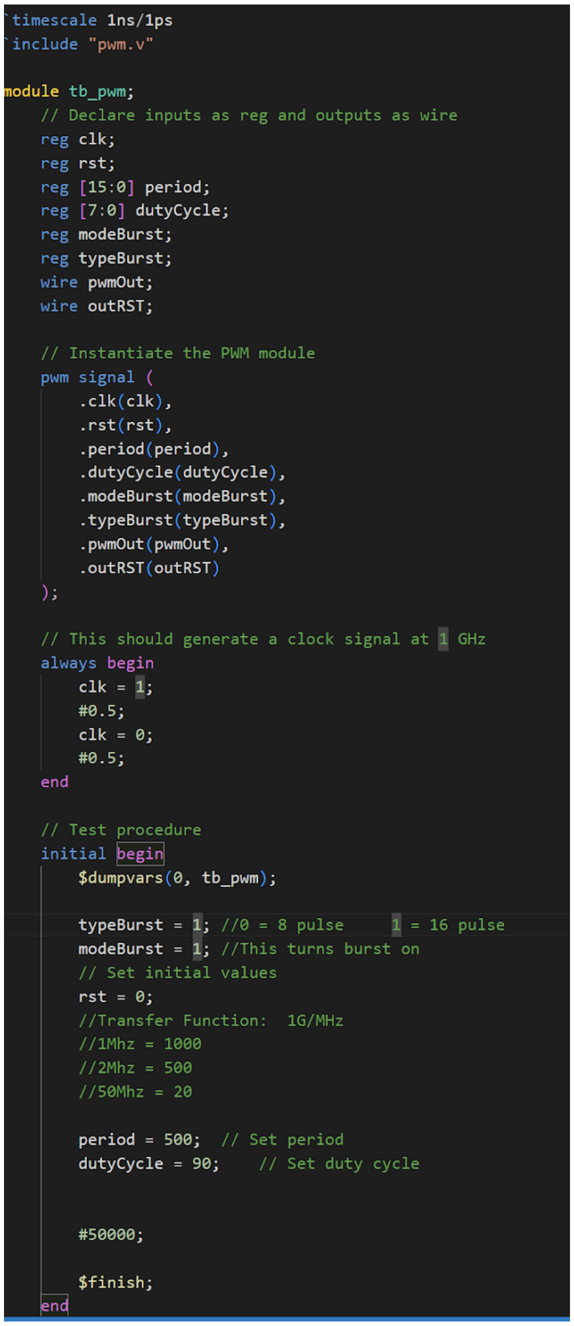


```
sOn: begin
    //Drive output
    pwmOut = 1;
    outRST = rst;

    /* There are 3 options for movement within this stage
        - sOn has not finished its duty cycle so it goes back to sOn
        - sOn has finsihed its duty cycle so it moves to sOff
        - rst is turned on and the system needs to go back to the idle state
    */

    if (rst == 1) begin
        //set idle back on
        stateNext = sIdle;
    end else if (onTime != 0) begin
        onTime--;
        stateNext = sOn;
    end else begin
        stateNext = sOff;
        onTime = (tickTime * dutyCycle)/100;
    end
end
```

Figure 1.6: sOn

It is important to note that if the input rst is ever turned on within any state other than sIdle, the state machine will immediately move to sIdle within the following clock cycle.

It is also important to mention the testbench program as well, programmed in tb_pwm.v, was also a trivial process. Since there was no need for a self checking test bench, the team simply set up the variables in the testbench program and ran one test case and kept switching it whenever need be. Code shown below



```
`timescale 1ns/1ps
`include "pwm.v"

module tb_pwm;
    // Declare inputs as reg and outputs as wire
    reg clk;
    reg rst;
    reg [15:0] period;
    reg [7:0] dutyCycle;
    reg modeBurst;
    reg typeBurst;
    wire pwmOut;
    wire outRST;

    // Instantiate the PWM module
    pwm signal (
        .clk(clk),
        .rst(rst),
        .period(period),
        .dutyCycle(dutyCycle),
        .modeBurst(modeBurst),
        .typeBurst(typeBurst),
        .pwmOut(pwmOut),
        .outRST(outRST)
    );

    // This should generate a clock signal at 1 GHz
    always begin
        clk = 1;
        #0.5;
        clk = 0;
        #0.5;
    end

    // Test procedure
    initial begin
        $dumpvars(0, tb_pwm);

        typeBurst = 1; //0 = 8 pulse      1 = 16 pulse
        modeBurst = 1; //This turns burst on
        // Set initial values
        rst = 0;
        //Transfer Function:  1G/MHz
        //1Mhz = 1000
        //2Mhz = 500
        //50Mhz = 20

        period = 500;  // Set period
        dutyCycle = 90;    // Set duty cycle


        #50000;

        $finish;
    end
```

Figure 1.7: Testbench: tb_pwm.v

Test Cases for Verilog

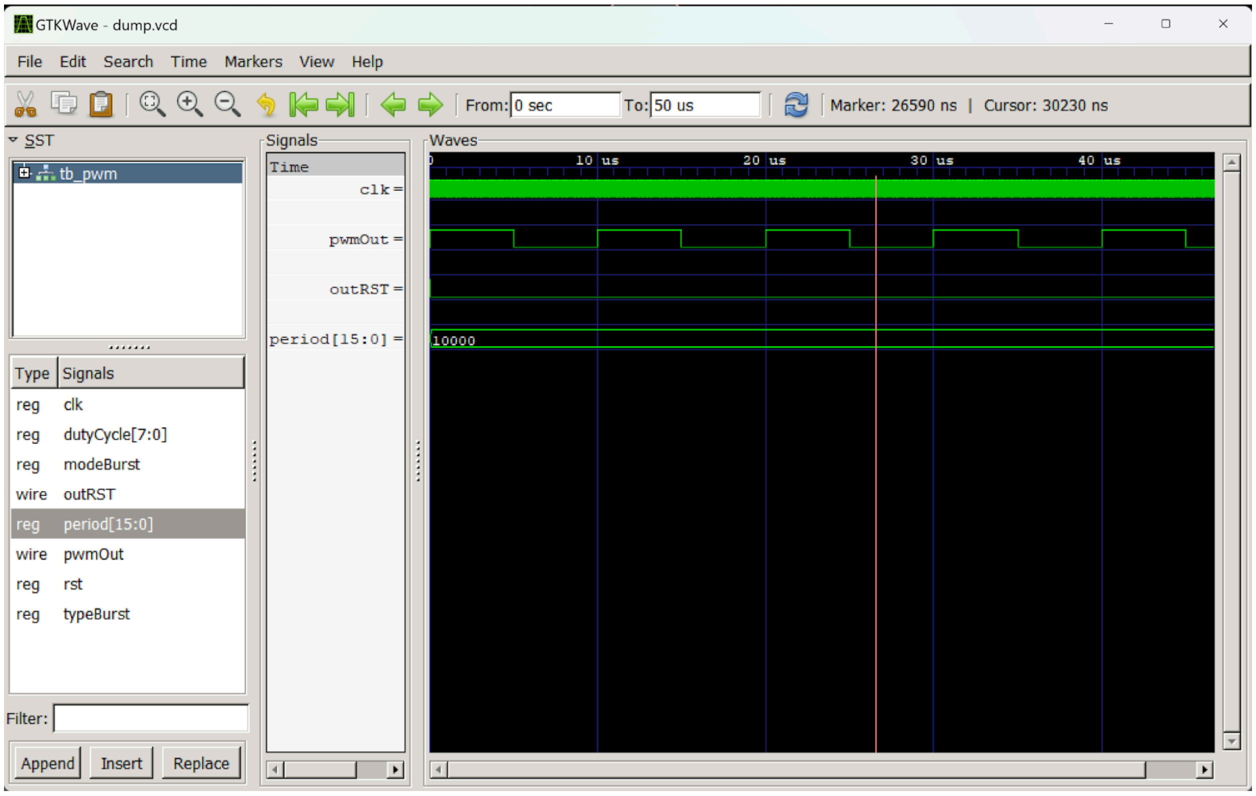


Figure 2.1: 100kHz with 50/50 duty cycle, no burst

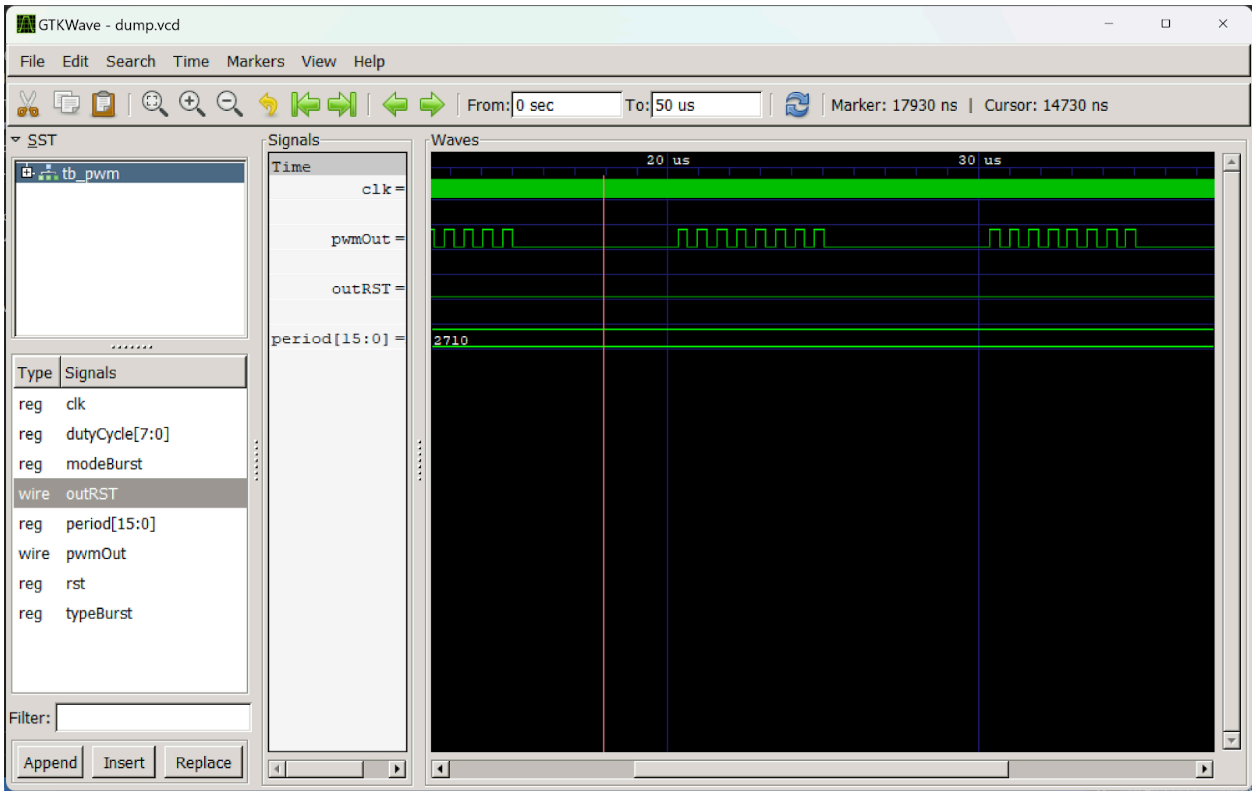
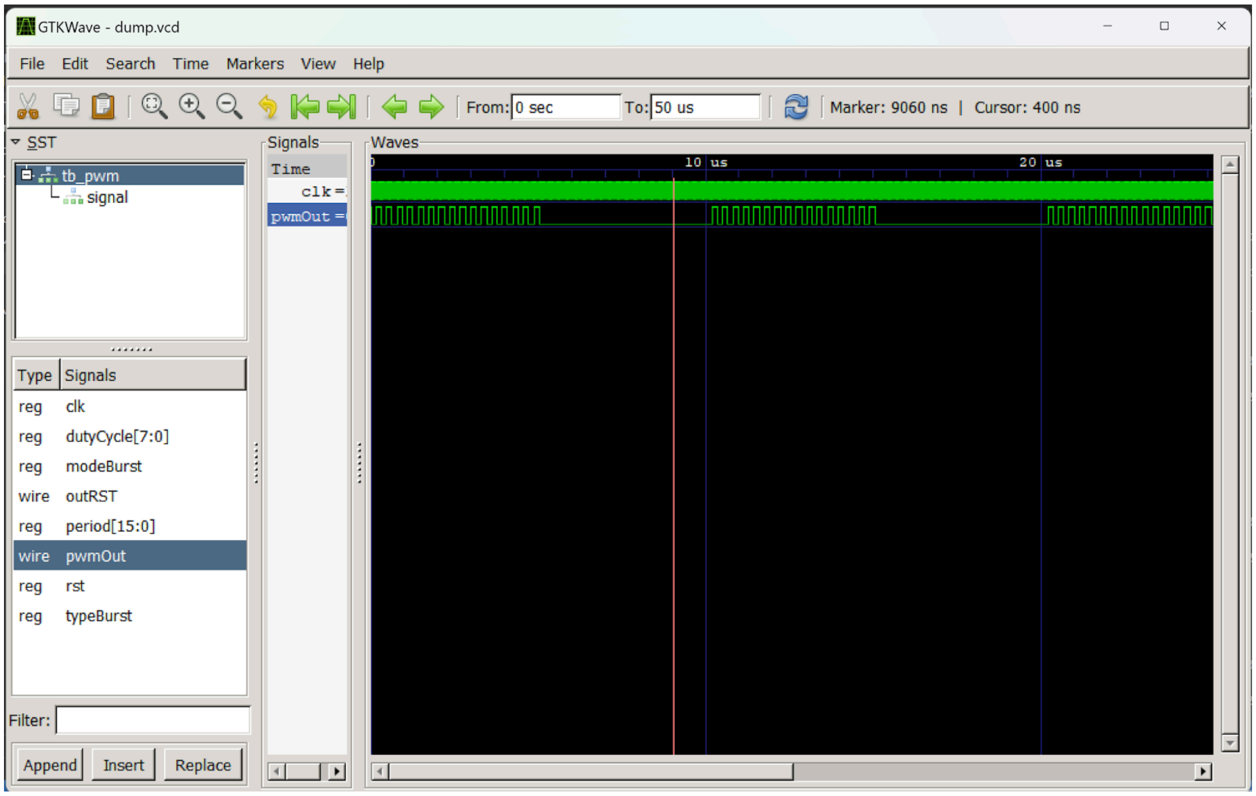
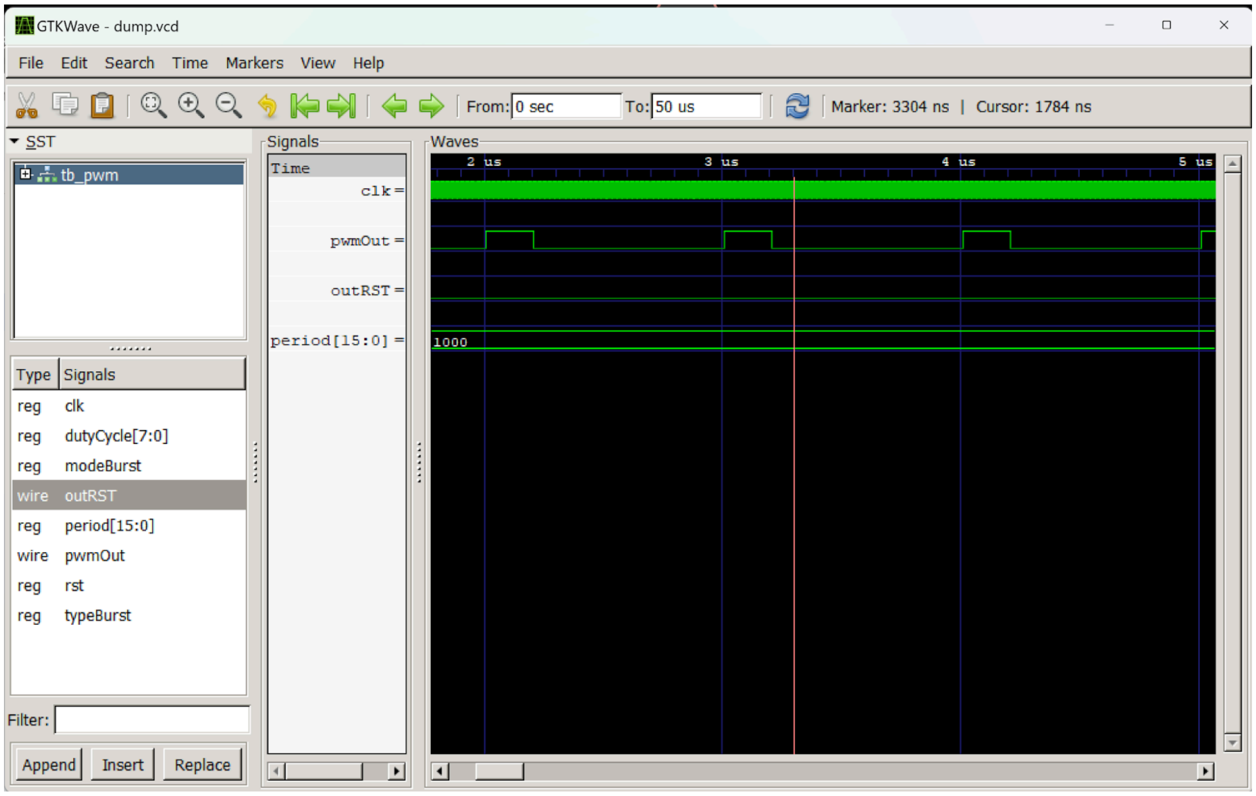


Figure 2.2: 100kHz with 50/50 duty cycle, burst on (8)



Type	Signals
reg	clk
reg	dutyCycle[7:0]
reg	modeBurst
wire	outRST
reg	period[15:0]
wire	pwmOut
reg	rst
reg	typeBurst

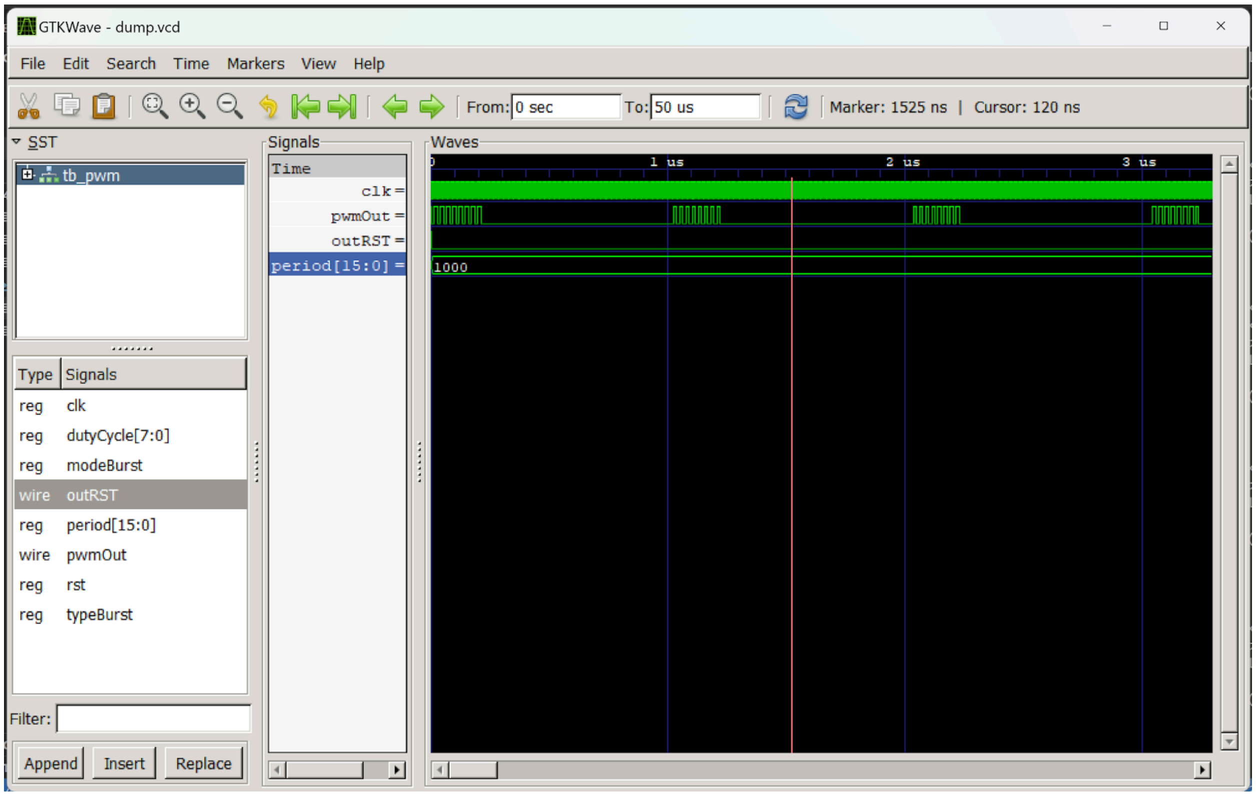
Figure 2.3: 100kHz with 50/50 duty cycle, burst on (16): Period Val- 10k



From: 0 sec To: 50 us | Marker: 3304 ns | Cursor: 1784 ns

Type	Signals
reg	clk
reg	dutyCycle[7:0]
reg	modeBurst
wire	outRST
reg	period[15:0]
wire	pwmOut
reg	rst
reg	typeBurst

Figure 2.4: 1MHz with 20/80 duty cycle, no burst



Type	Signals
reg	clk
reg	dutyCycle[7:0]
reg	modeBurst
wire	outRST
reg	period[15:0]
wire	pwmOut
reg	rst
reg	typeBurst

Figure 2.5: 1MHz with 20/80 duty cycle, burst on (8)

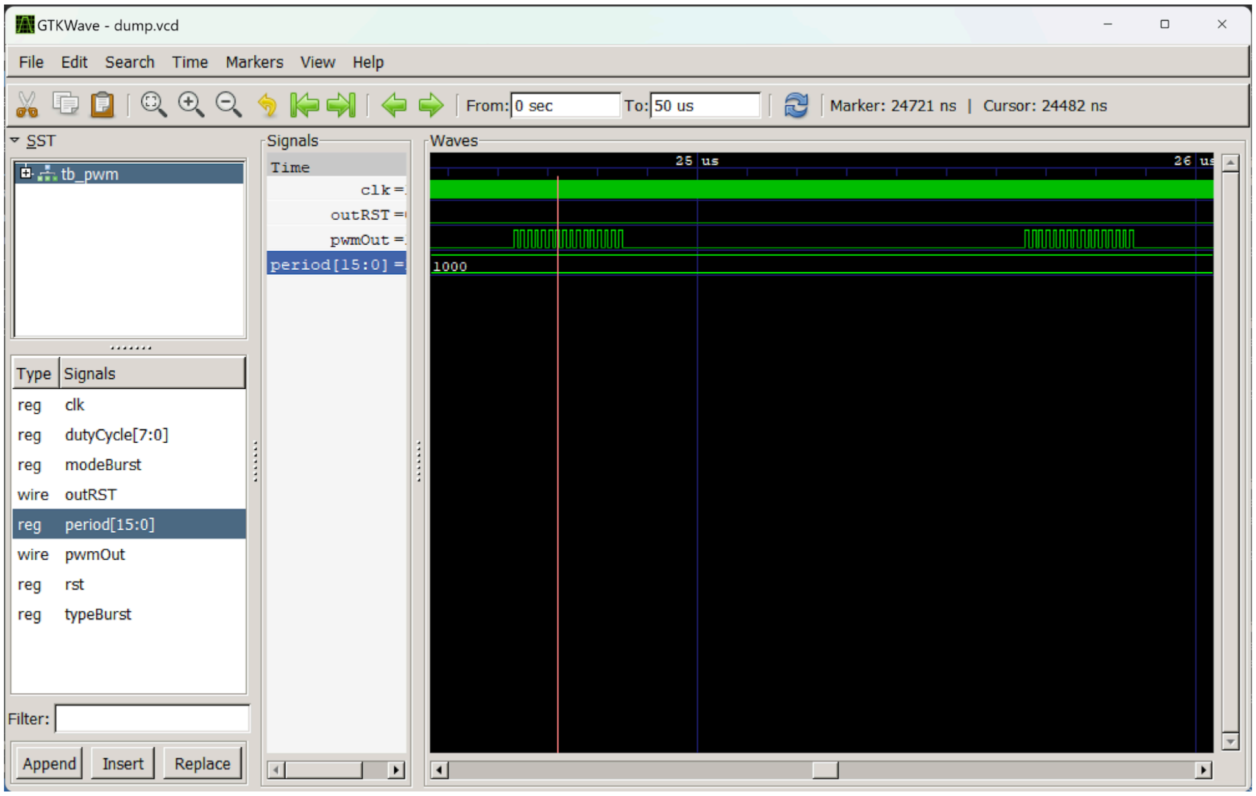
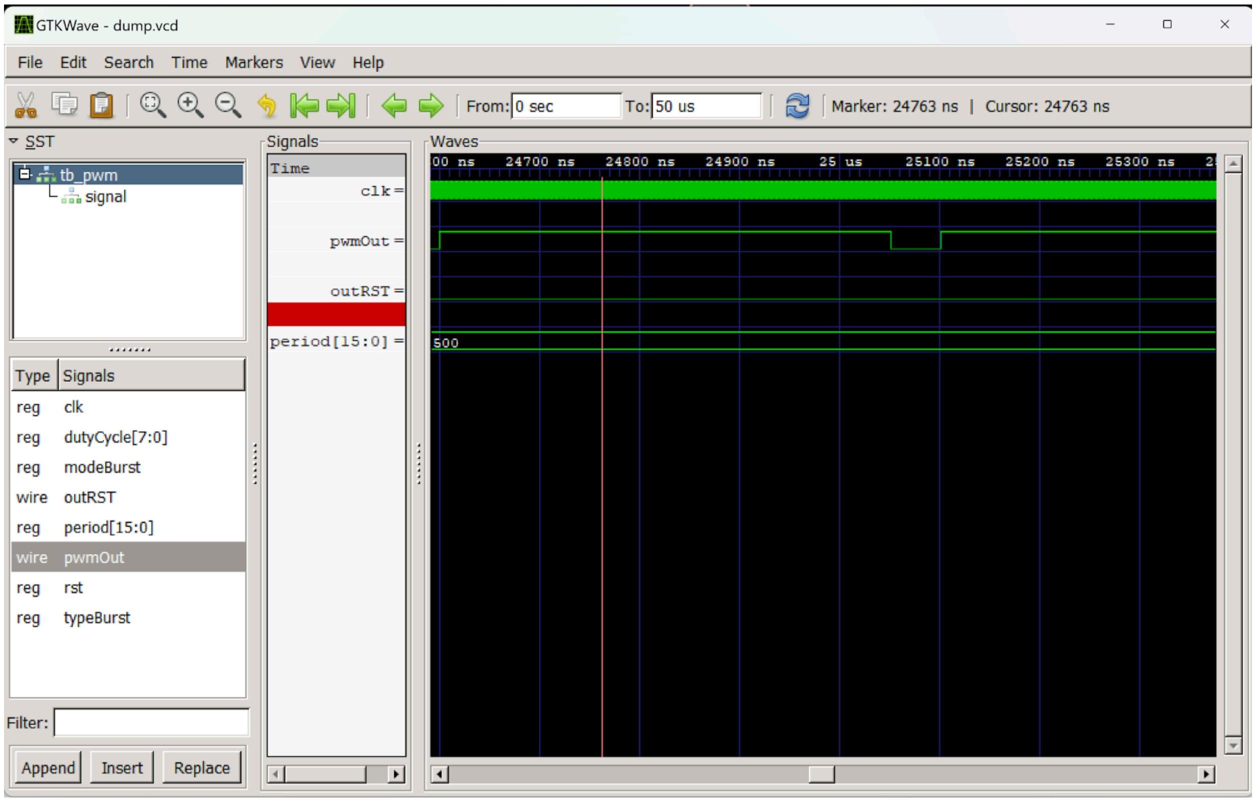
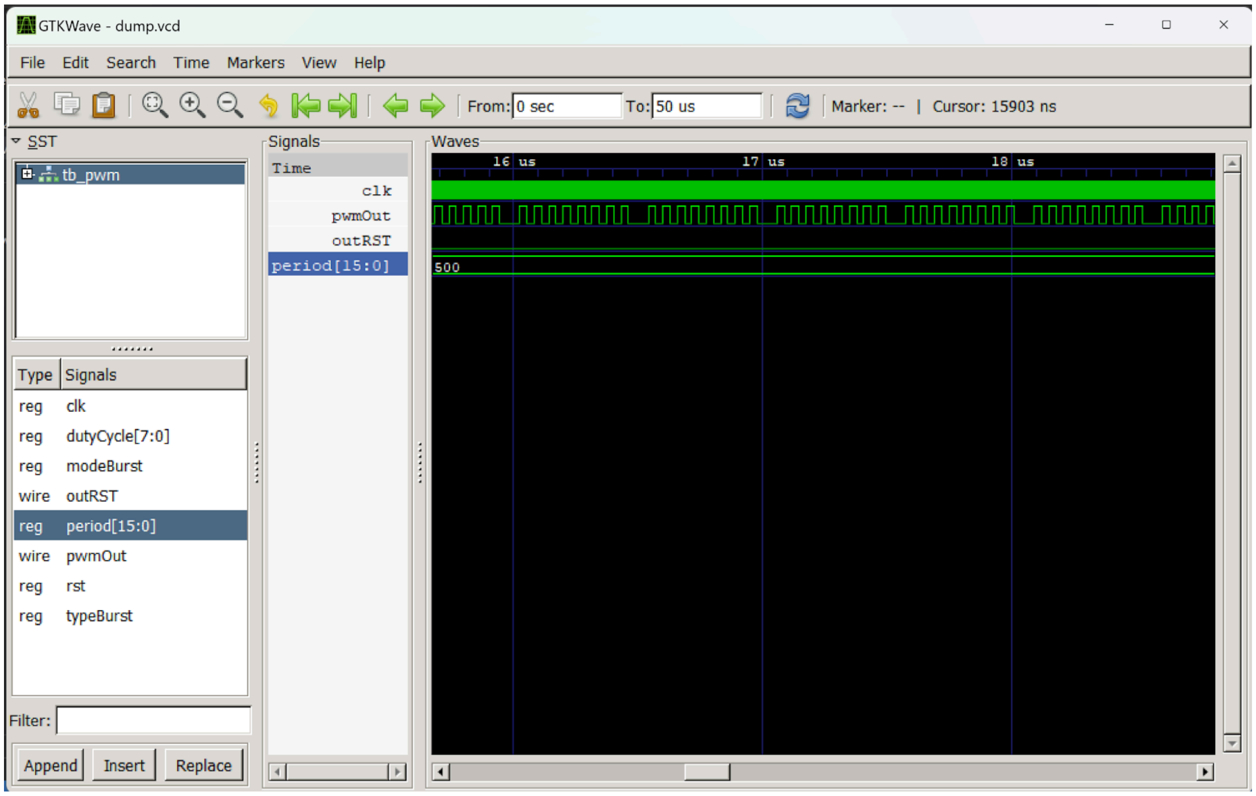


Figure 2.6: 1MHz with 20/80 duty cycle, burst on (16)



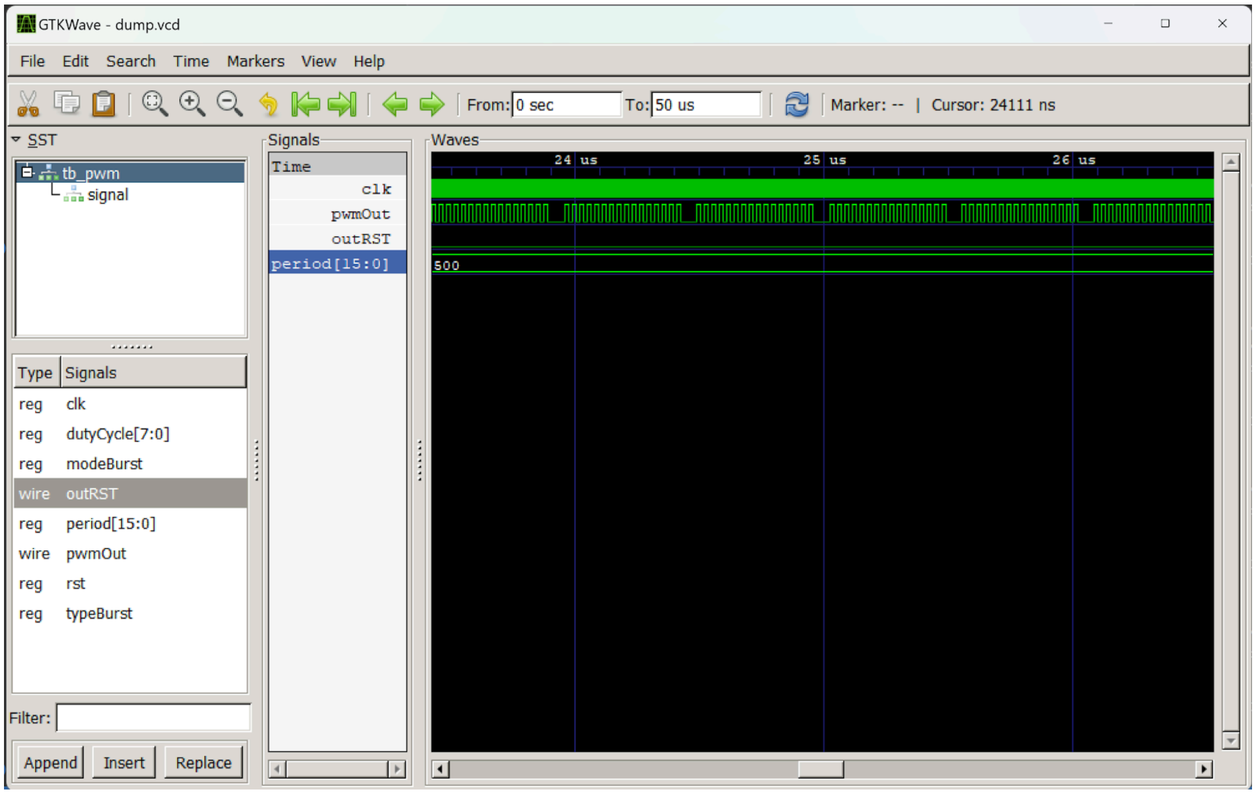
Type	Signals
reg	clk
reg	dutyCycle[7:0]
reg	modeBurst
wire	outRST
reg	period[15:0]
wire	pwmOut
reg	rst
reg	typeBurst

Figure 2.7: 2MHz with 90/10 duty cycle, no burst



Type	Signals
reg	clk
reg	dutyCycle[7:0]
reg	modeBurst
wire	outRST
reg	period[15:0]
wire	pwmOut
reg	rst
reg	typeBurst

Figure 2.8: 2MHz with 90/10 duty cycle, burst on (8)



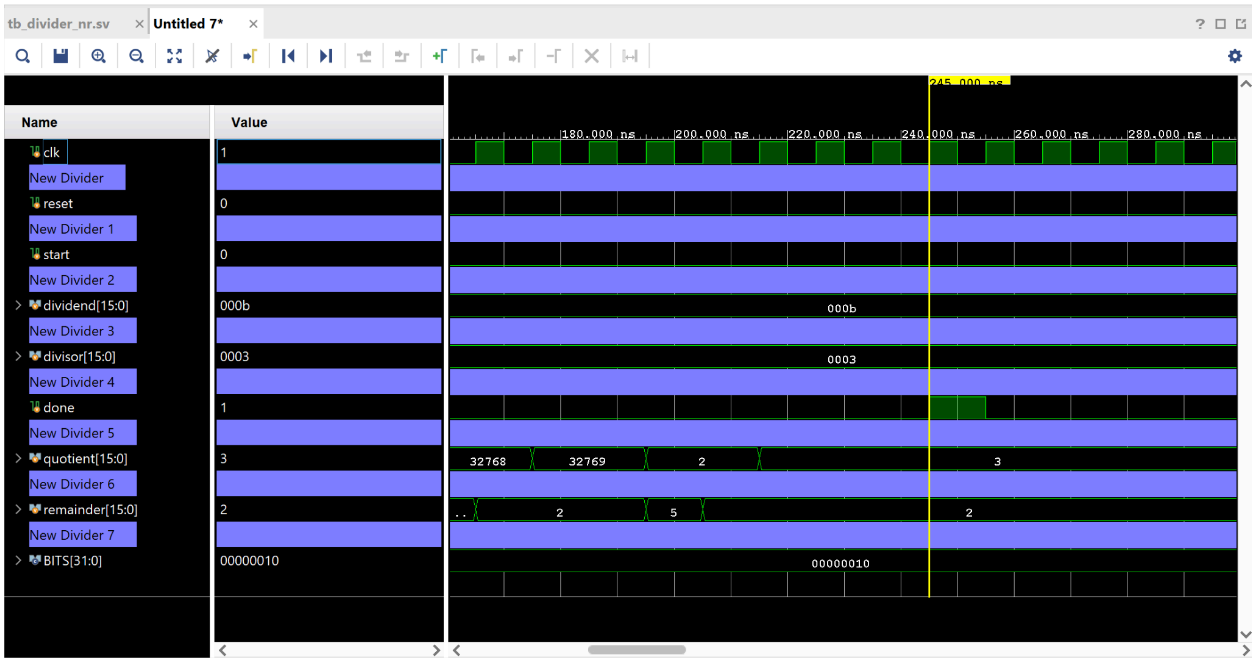
Type	Signals
reg	clk
reg	dutyCycle[7:0]
reg	modeBurst
wire	outRST
reg	period[15:0]
wire	pwmOut
reg	rst
reg	typeBurst

Figure 2.9: 2MHz with 90/10 duty cycle, burst on (16)

Deliverables

A:

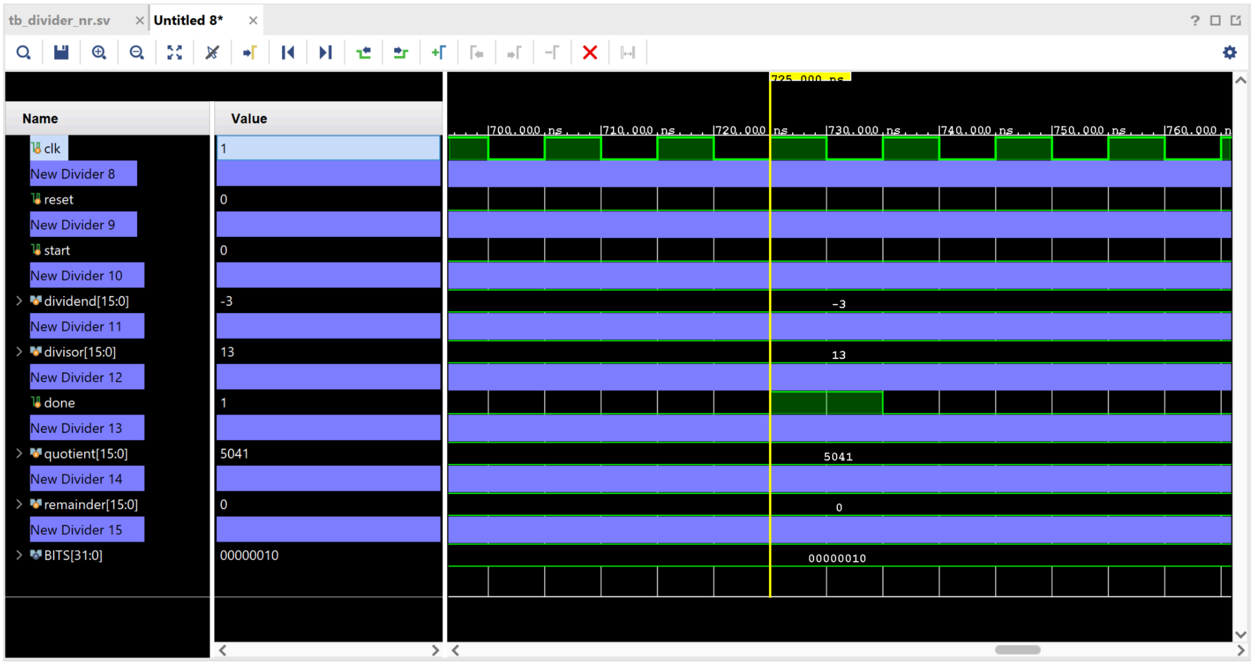
Show simulation of the divider page 181-182. You can show a screenshot passing and force a fail to show an error.



```
Name              Value      Waveform
clk               1
New Divider
reset             0
New Divider 1
start             0
New Divider 2
dividend[15:0]    000b       000b
New Divider 3
divisor[15:0]     0003       0003
New Divider 4
done              1
New Divider 5
quotient[15:0]    3          32768 | 32769 | 2 | 3
New Divider 6
remainder[15:0]   2          .. | 2 | 5 | 2
New Divider 7
BITS[31:0]        00000010   00000010
```

FIgure: Vivado Simulation of 11/3; Resulting in Passing Conditions

The figure above shows the test case for the divider using 000b (11) as the **dividend** and 0003 (3) as the **divisor**. The divider program uses integer division, so there are two outputs, the **quotient** and the **remainder**. Using integer division, 11 divided by 3 will result in a quotient of 3 and remainder of 2. The waveforms above show when the division is completed both the quotient and remainder hold the values previously determined. The divider program shows a passing case for this division.



Name	Value
clk	1
New Divider 8	
reset	0
New Divider 9	
start	0
New Divider 10	
dividend[15:0]	-3
New Divider 11	
divisor[15:0]	13
New Divider 12	
done	1
New Divider 13	
quotient[15:0]	5041
New Divider 14	
remainder[15:0]	0
New Divider 15	
BITS[31:0]	00000010

Figure: Vivado Simulation of -3/13; Resulting in Failing Conditions

The above figure is the result of another test case, this time one that produces an error. On first glance it may not look like there is anything wrong. However, the divider program is only built for positive integer division. Using the same portion of **tb_divider** code as the previous scenario, the **dividend** and **divisor** have been changed to -3 and 13, respectfully. The waveform above shows the same outputs (**quotient** and **remainder**). The program uses the sign-extended notation of -3 (FFFD) and divides 13 into that large number. For the unsigned division that would equal a **quotient** of 5041 and **remainder** of 0, but that is not the correct output for the signed division that was used at input. This case provides an error, as the output is incorrect as previously stated.

B:

Complete the calculator build through pg 183. Show resources utilized from Vivado on the build. Provide a video of your calculator working. Show add, subtract, multiply.

The link provided shows basic operations (addition, subtraction, multiplication) through the FPGA board switches and buttons.

https://drive.google.com/file/d/1YpPZj5_KPNArlo0t2ANGwIrbyTLr8bxC/view?usp=drive_link

C:

The divider uses a non-restoring division algorithm. Another method to use is a restoring division algorithm. Describe the difference between the two algorithms. Be clear on this.

The primary difference between restoring and non-restoring division algorithms lies in how they handle the remainder during the division process. In the restoring division algorithm, if a subtraction results in a negative remainder, the algorithm immediately adds back the divisor to restore the remainder to a non-negative value and sets the corresponding quotient bit to zero. This ensures that the remainder remains non-negative throughout the computation but requires an additional addition operation whenever a negative remainder occurs. In contrast, the non-restoring division algorithm does not immediately correct a negative remainder. Instead, it continues the process without restoring and adjusts the remainder in a subsequent step, typically by adding the divisor if necessary. By eliminating the need for immediate restoration, the non-restoring division algorithm reduces the number of operations, making it more efficient for hardware implementations.

D:

The division algorithms are for positive numbers only. What could you do to make the algorithms work for negative numbers? Just describe the change or method you would add to the algorithms.

To extend the non-restoring division algorithm to support signed numbers in Verilog, several key modifications are necessary. First, a **sign detection mechanism** must be introduced to determine whether the quotient should be negative. This can be achieved by XORing the sign bits (MSB) of the dividend and divisor. Next, before performing the division, both operands must be **converted to their absolute values** using two's complement if they are negative. The existing division logic then operates on these positive values without modification. After the division completes, the **quotient's sign must be adjusted** based on the sign flag—if only one of the original inputs was negative, the quotient is converted back to negative using two's complement. Additionally, the **remainder must retain the same sign as the original dividend** to ensure consistency with signed division rules. These modifications allow the non-restoring division algorithm to correctly handle both positive and negative operands while maintaining computational efficiency.

References

FPGA Handbook:
https://github.com/PacktPublishing/The-FPGAProgramming-Handbook-Second-Edition