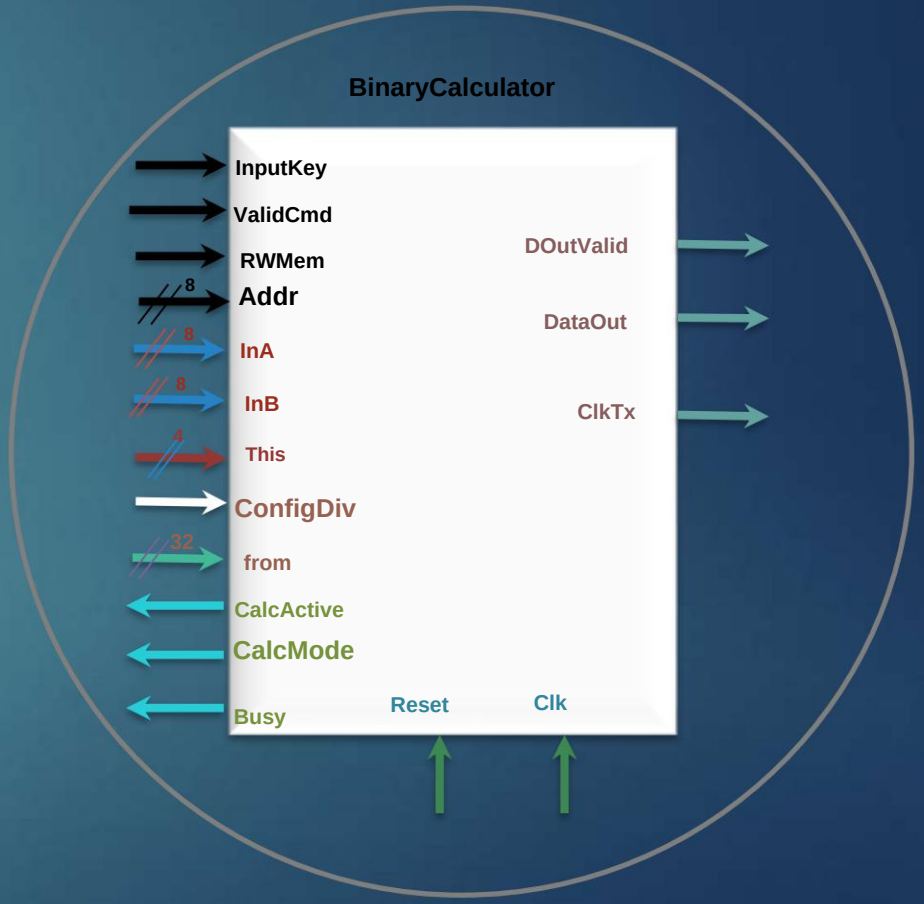
Binary calculator

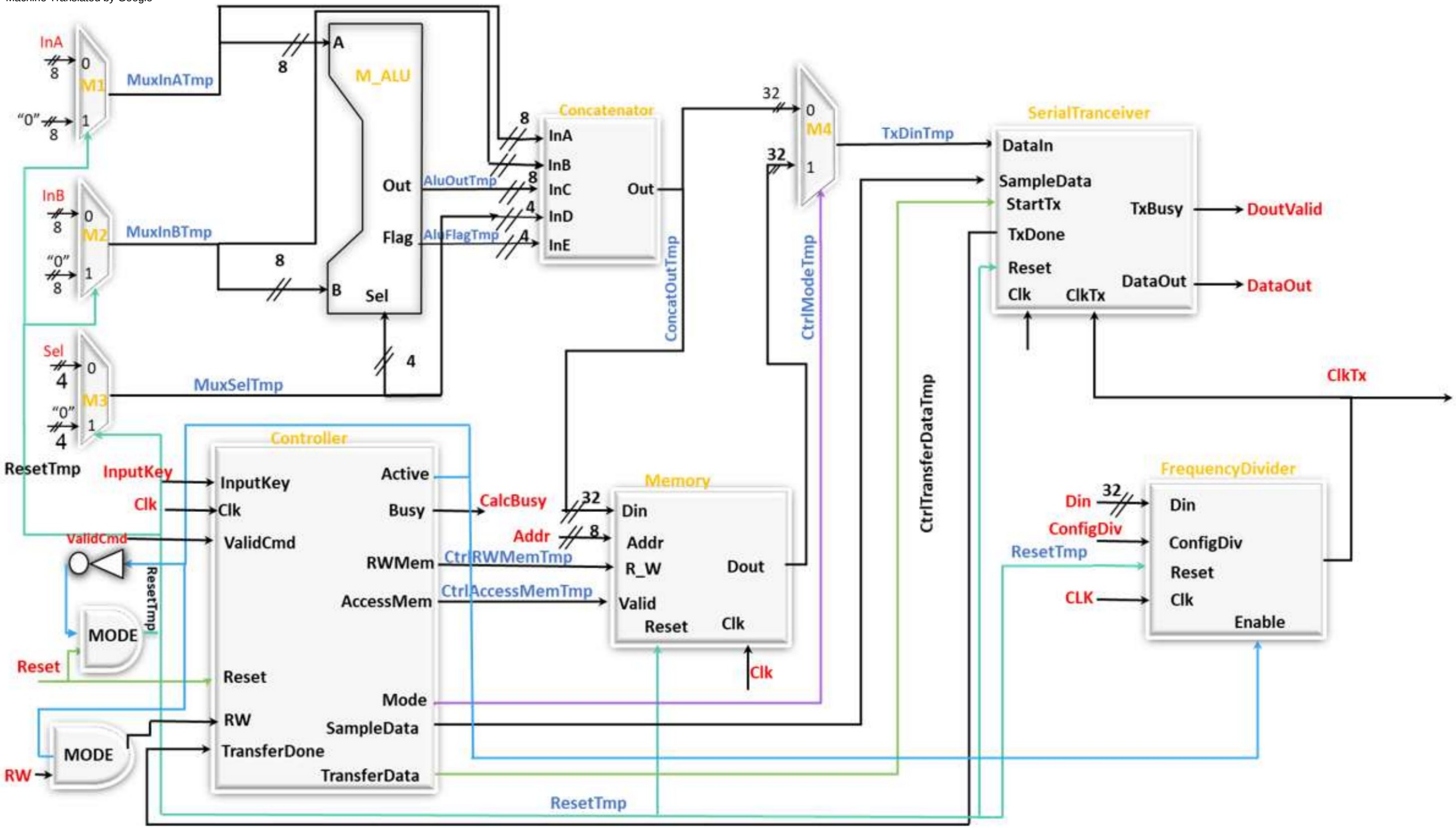
NAME: IONESI SAMUEL

The purpose of the project

Creation of a binary computer with the following functionalities:

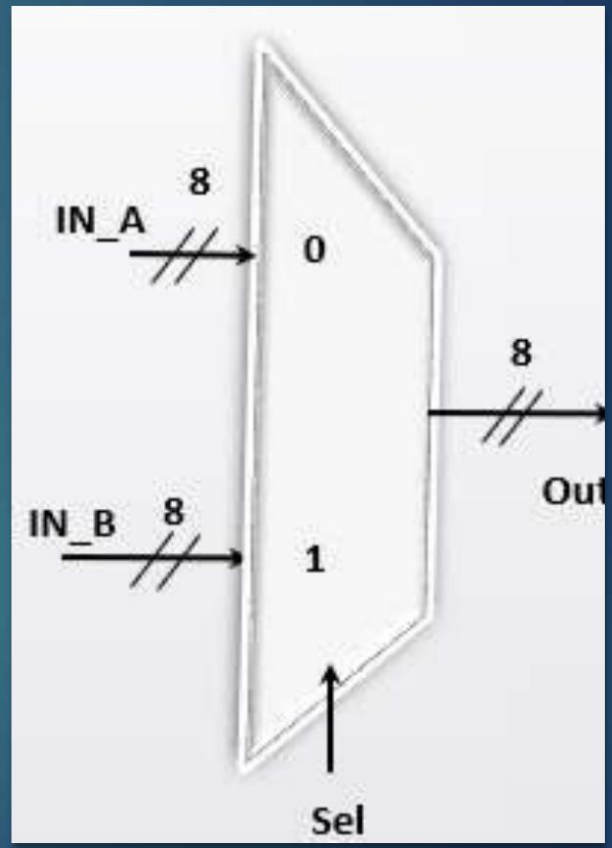
- 2 operating modes:
  - Calculation -> *Memorization* -> Transfer.
  - Calculation + Transfer.
- Activation using a ***passphrase.***
- Possibility of storing up to 256 calculations.
- Serial transfer at ***variable frequencies.***





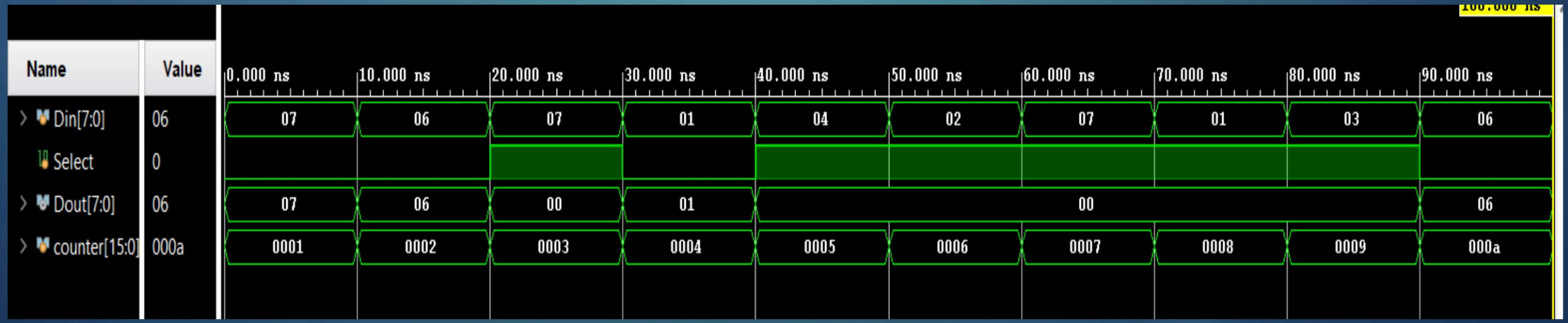
Multiplexors

- They are CLC with 3 inputs and one output. It can also be understood as a switch.
- **M1 and M2** are used as inputs of the two operands in the ALU. a operand is one byte long.
- **M3** is used as a selector for ALU. It is on 4 bits, so they will be able to select 16 operations that the ALU can do.
- When ResetTmp is active, the operation inputs will be connected to "0" logic.
- **M4** is used to connect a block (ALU, Memory) to its input SerialTranceiver
  - If the calculator is not in memorization mode (CalcMode= 0) the input data will be taken from the ALU.
  - If the calculator is in storage mode (CalcMode= 1), the data input will be retrieved from Memory.



```
module Mux_8b #(parameter LENGTH = 8)
(
input [LENGTH - 1:0] Din,
input Select,
output reg [LENGTH - 1:0] Dout
);

    always @(*) begin
        if(Select) begin
            Dout <= 8'h00;
        end
        else begin
            Dout <= Din;
        end
    end
endmodule
```

```
module Mux_32b #(parameter LENGTH = 32)
(
input wire [LENGTH - 1:0] Din_A,
input wire [LENGTH - 1:0] Din_B,
input wire Select,
output reg [LENGTH - 1:0] Dout
);

    assign Dout = Select ? Din_B : Din_A;

endmodule
```

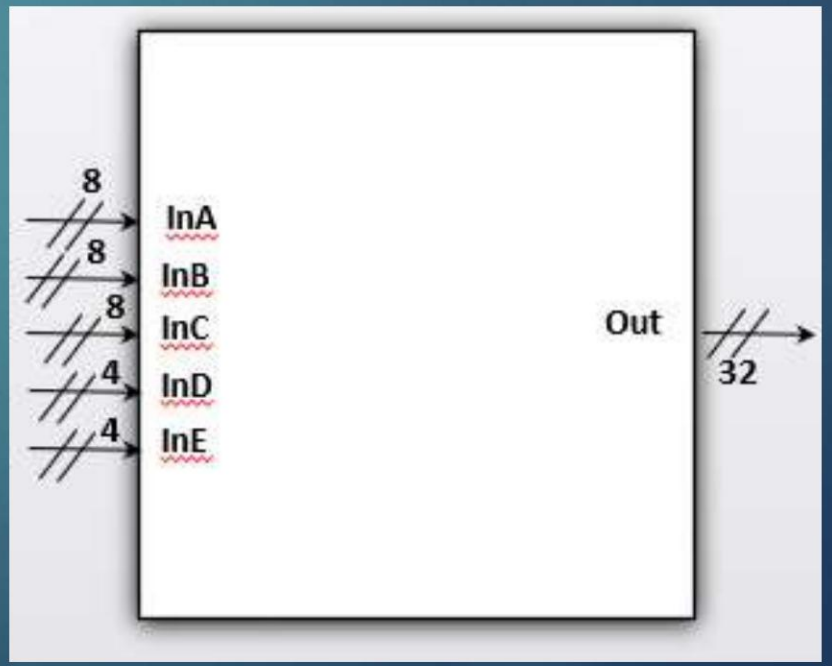


Concatenator

ÿ It is a CLC with 5 inputs and one output

ÿ Out = {InE, InD, InC, InB, InA} -> concatenation of the 5 inputs

- InA -> the first operand of the operation (InA from Mux M1)
- InB -> the second operand of the operation (InB from Mux M2)
- InC -> output from ALU
- InD -> selection input from M3
- InE -> Flag output from ALU

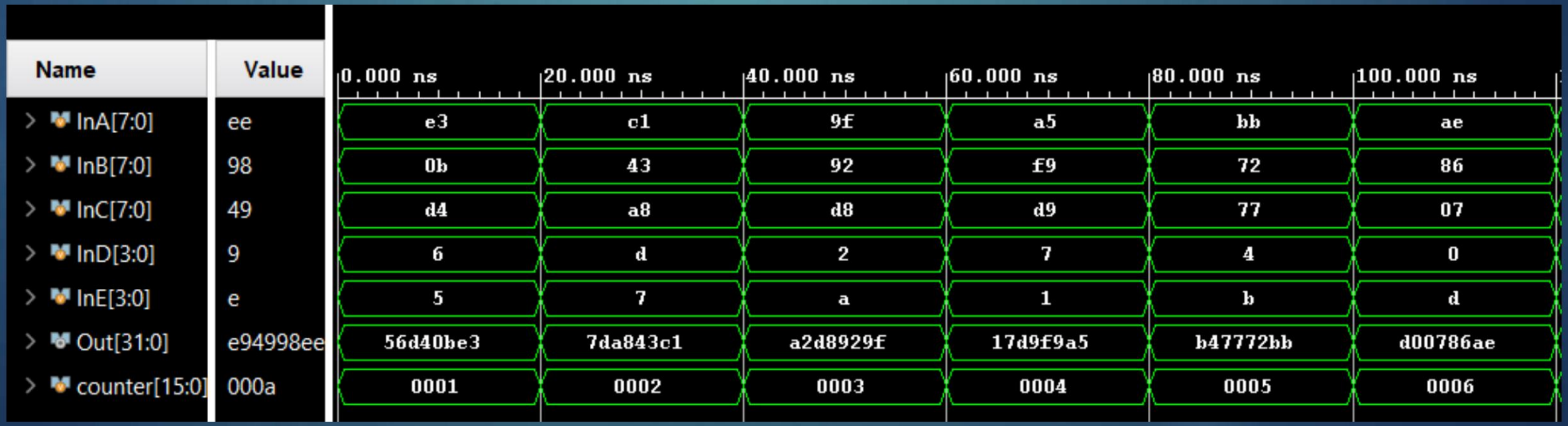


ÿ Example:

- InA = 8'h25
- InB = 8'hA
- InC = 8’h4B
- InD = 8'h22
- InE = 8'hF

ÿ Out = 32’hF_22_4B_A_25

```
module Concatenator
(InA, InB, InC, InD, InE, Out);
    input [7:0] InA, InB, InC;
    input [3:0] InD, InE;
    output reg [31:0] Out;

    assign Out = {InE, InD, InC, InB, InA};
endmodule
```



GO

ÿ It is a combinational circuit with 3 inputs and two outputs.

ÿ Inputs A and B are used instead of the two operands that we must use.

ÿ The entry Sel designates the type of operation that we will do. ÿ The

Out output is represented by the result of the 8-bit operation. ÿ The

Flag output assigns the following signals:

```
reg ZeroFlag;
reg CarryFlag;
reg OverflowFlag;
reg UnderflowFlag;
```

```
Flag[0] = ZeroFlag;
Flag[1] = CarryFlag;
Flag[2] = OverflowFlag;
Flag[3] = UnderflowFlag;
```

ÿ It is used to perform an operation with two operands and then the result and the flag are sent to the concatenator

For the addition operation, we made an 8-bit adder made of 2 4-bit adders and the latter made of 1-bit full adders.
To realize the full adder, I used the default structural description.

```
module Full_Adder
(
    input A,
    input B,
    input Ci,
    output S,
    output C0
);

    assign S = A ^ B ^ Ci;
    assign C0 = (A & B) | (Ci & (A ^ B));

endmodule
```

```
module Four_bit_adder
(
    input [3:0] A,
    input [3:0] B,
    input wire Cin,
    output wire[3:0] S,
    output wire Cout
);

    wire [2:0] Cin_Cout;

    Full_Adder FA0 (.A(A[0]), .B(B[0]), .Ci(Cin), .S(S[0]), .C0(Cin_Cout[0]));
    Full_Adder FA1 (.A(A[1]), .B(B[1]), .Ci(Cin_Cout[0]), .S(S[1]), .C0(Cin_Cout[1]));
    Full_Adder FA2 (.A(A[2]), .B(B[2]), .Ci(Cin_Cout[1]), .S(S[2]), .C0(Cin_Cout[2]));
    Full_Adder FA3 (.A(A[3]), .B(B[3]), .Ci(Cin_Cout[2]), .S(S[3]), .C0(Cout));

endmodule
```

```
module Eight_bit_adder #(parameter WIDTH = 8)(
    input [WIDTH - 1:0] A,
    input [WIDTH - 1:0] B,
    input wire Cin,
    output wire[WIDTH - 1:0] S,
    output wire Cout
    );

    wire connection;

    Four_bit_adder FBA0 (.A(A[3:0]), .B(B[3:0]), .Cin(Cin), .S(S[3:0]), .Cout(connection));
    Four_bit_adder FBA1 (.A(A[7:4]), .B(B[7:4]), .Cin(connection), .S(S[7:4]), .Cout(Cout));
endmodule
```

```
4'h0: begin
    if(AdderCarry) begin
        CarryFlag <= AdderCarry;
        Out <= AdderOut;
        UnderflowFlag <= 1'b0;
        OverflowFlag <= 1'b0;

    end
    else begin
        CarryFlag <= 1'b0;
        UnderflowFlag <= 1'b0;
        OverflowFlag <= 1'b0;
        Out <= AdderOut;
    end
end
```

All operations were covered

ÿ Decrease:

```
4'h1: begin
    //Zero();
    if(A < B) begin
        UnderflowFlag <= 1'b1;
        Out <= ~(A - B) + 1; /
    end
    else begin
        UnderflowFlag <= 1'b0;
        Out <= A - B;
    end
  CarryFlag <= 1'b0;
  OverflowFlag <= 1'b0;
end
```

Impart:

```
4'h3: begin
    if(B != 8'h00)  begin
        if(A < B) begin
            UnderflowFlag <= 1'b1;
        end
        else begin
            UnderflowFlag <= 1'b0;
        end
        Out <= A / B;
        CarryFlag <= 1'b0;
        OverflowFlag <= 1'b0;
    end
    else begin
        Flag <= 8'h1;
        Out <= 8'h00;
    end
end
```

ÿ Shift-Left si Shift-Right:

```
4'h4: begin
    Out <= A << B;
    CarryFlag <= 1'b0;
    OverflowFlag <= 1'b0;
    UnderflowFlag <= 1'b0;
end

//Shift-Right
4'h5: begin
    Out <= A >> B;
    CarryFlag <= 1'b0;
    OverflowFlag <= 1'b0;
    UnderflowFlag <= 1'b0;
end
```

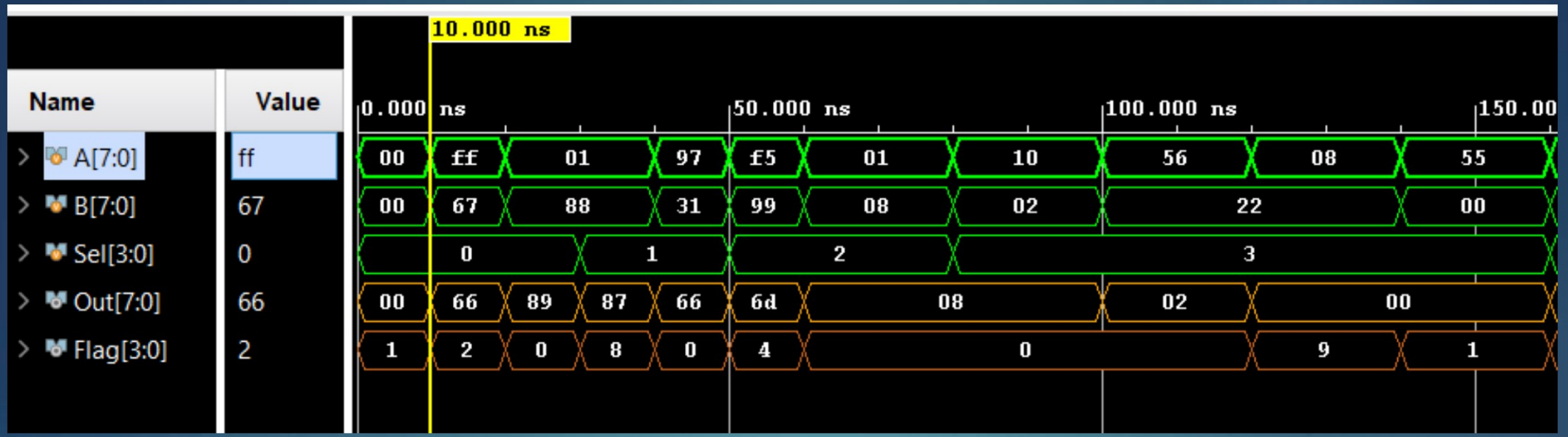
ÿ AND, OR, XOR, NXOR, NAND, NOR

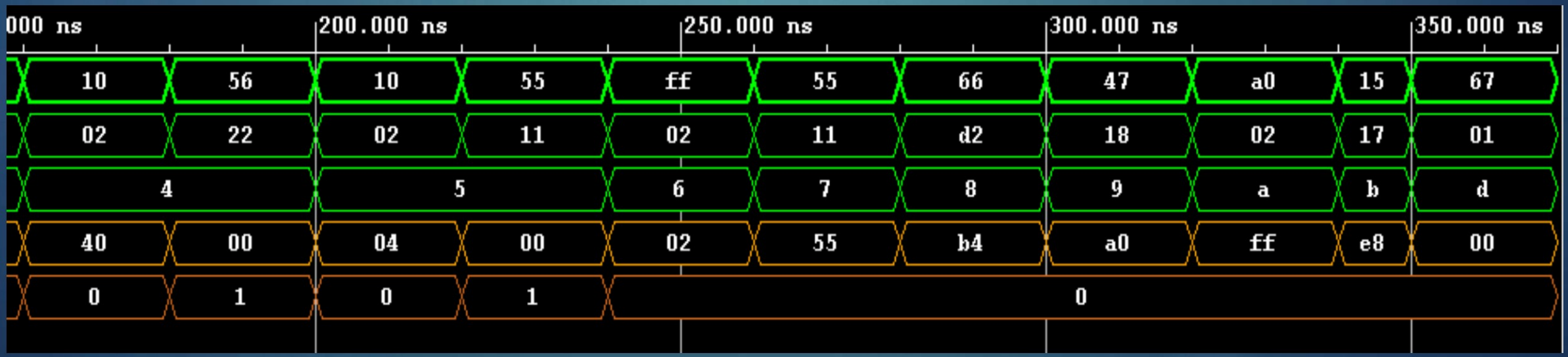
Multiplication:

```
4'h2: begin
    //Zero();
    MultiplyTemp = A * B;
    if(MultiplyTemp[15:8] == 8'h00) begin
        OverflowFlag <= 1'b0;
        Out <= MultiplyTemp[7:0];
    end
    else begin
        OverflowFlag <= 1'b1;
        Out <= MultiplyTemp[7:0];
    end
    CarryFlag <= 1'b0;
    UnderflowFlag <= 1'b0;
end
```

ÿ If Out == 0 && Sel is valid(0:B) -> ZeroFlag = 1

```
if(Out == 8'h00 && (Sel >= 4'h0 && Sel <= 4'hB)) begin
    ZeroFlag <= 1'b1;
end
else begin
    ZeroFlag <= 1'b0;
end
```





Memory

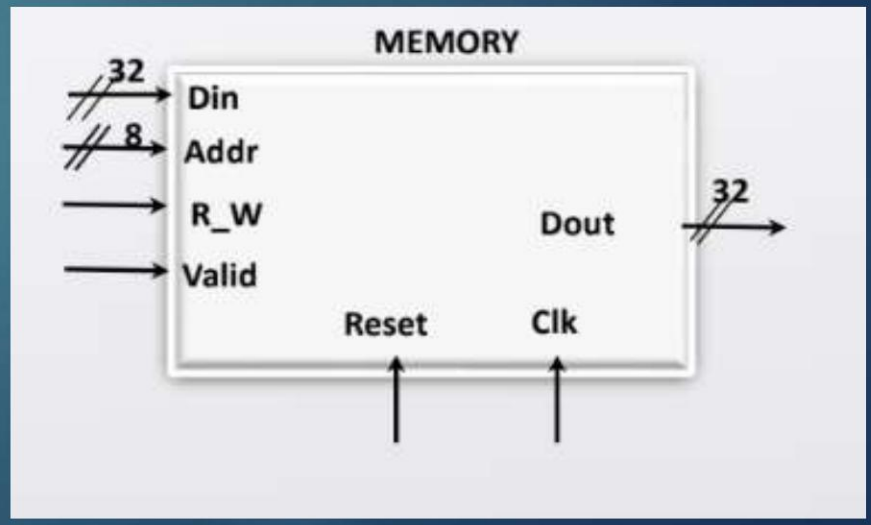
ÿ Parameterizable sequential circuit active at the positive front of Clk.

- WIDTH -> the address bus, with this we select the addresses.
- FromLENGTH -> length of input data.
- MemorySIZE -> the memory will have 2^WIDTH memory locations, that is 256.
- Memory is an array of 256 memory addresses with a length of one egala cu DinLENGTH to the address.

ÿ From is the data input coming from the concatenator.

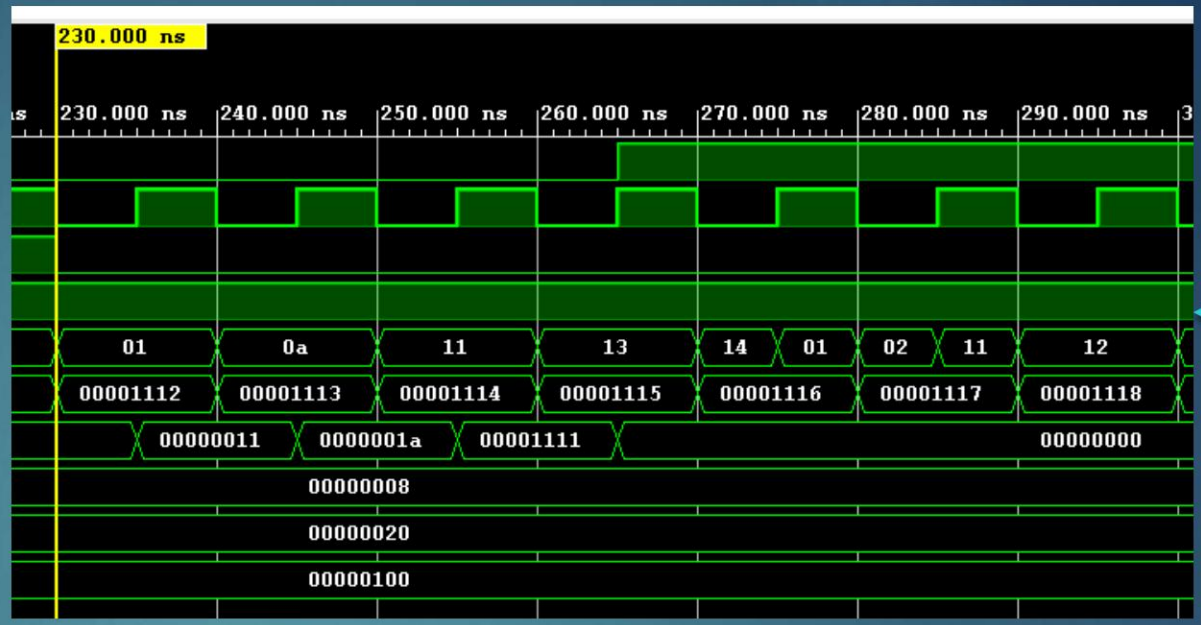
ÿ The reset is asynchronous to the positive front.

```
always @(posedge Clk or posedge Reset) begin
    if(Reset) begin
        while(counter < MemorySIZE) begin
            memory[counter] <= 32'h0;
            counter++;
        end
        Dout <= 32'h0;
    end
```

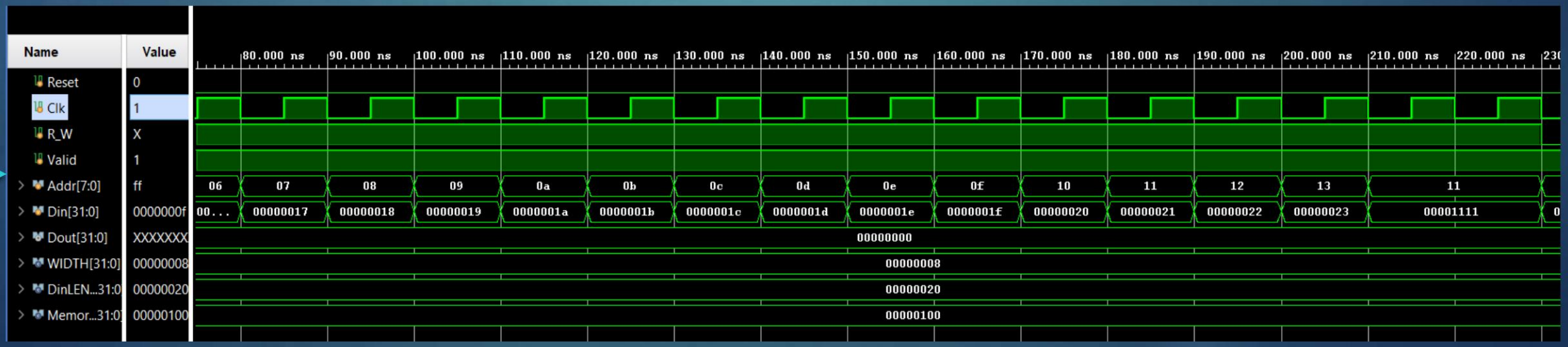


ÿ This module assumes writing (R_W == 1) and reading (R_W == 0) from memory.

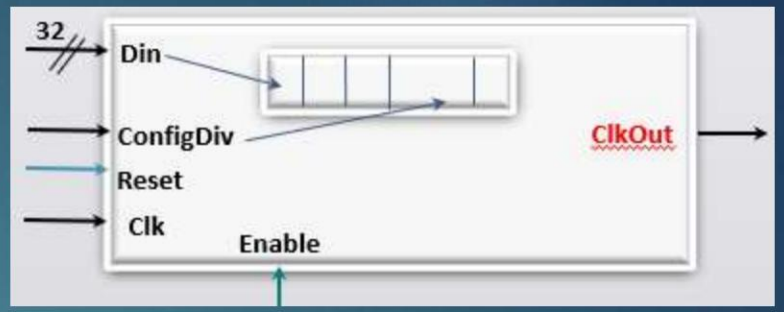
```
        else begin
            if(Valid) begin
                if(R_W) begin
                    memory[Addr] <= Din;
                    Dout <= 32'h0;
                end
                else begin
                    Dout <= memory[Addr];
                end
            end
            else begin
                Dout <= 32'h0;
            end
        end
    end
```



Successive writes in memory and overwriting. Reading from memory and what happens when it is reset.

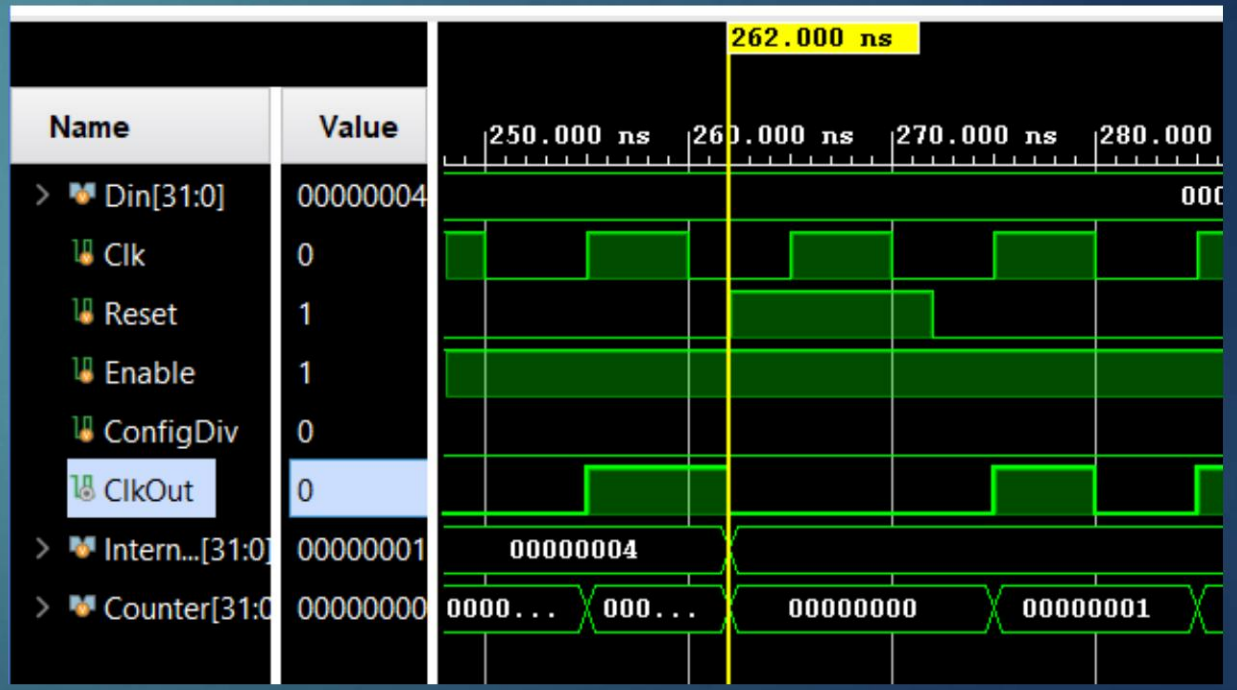


Frequency Divider



- Sequential circuit with 5 inputs and one output.
- Din is the input by which the frequency will be divided.
- When the reset is active, then the internal configuration register and everything is reset this time the output will be in 0.

```
task ResetValues();
    InternRegister <= 1;
    ClkOut <= 0;
    Counter <= 0;
endtask
always @(posedge Clk or negedge Clk or posedge Reset) begin
    if(Reset) begin
        ResetValues();
    end
```



ÿ When Reset == 0 && Enable == 0:

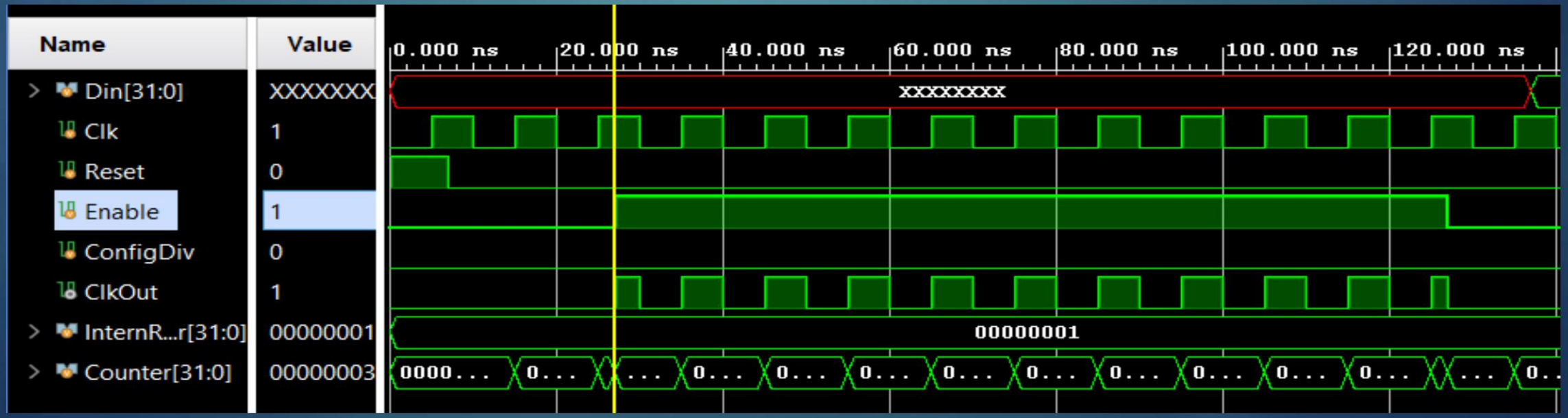
- ConfigDiv == 0 -> is stored in the internal registry that was previously saved in the registry
- ConfigDiv == 1 -> the data from the input Din is stored in the internal register

```
if(!Enable) begin
    ClkOut <= 0;
    if(ConfigDiv) begin
        //store to intern register the value that we are given as Din
        InternRegister <= Din;
    end
    else begin
        //store to intern register the value that is already stored
        InternRegister <= InternRegister;
    end
end
```

ÿ When Reset == 0 && Enable == 1 (the divider works)

- When we divide by 1:

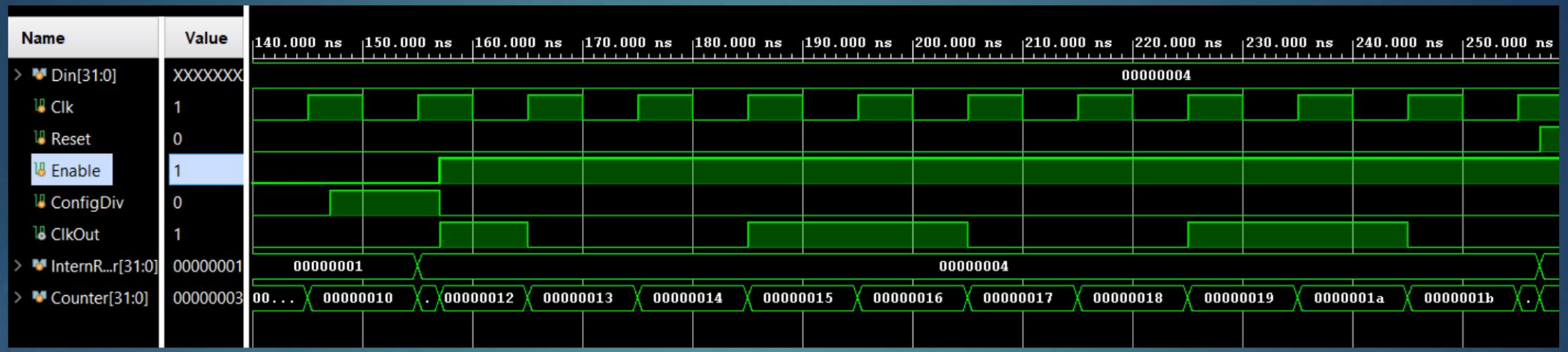
```
//this case is used when we want to divide with 1
always @(posedge Clk or negedge Clk or posedge Reset or Enable) begin
    if(Reset) begin
        ResetValues();
    end
    else begin
        if(Enable) begin
            if(InternRegister == 1) begin
                if(Clk)
                    ClkOut <= 1'b1;
                else
                    ClkOut <= 1'b0;
            end
        end
        else begin
            ClkOut <= 1'b0;
        end
    end
end
```



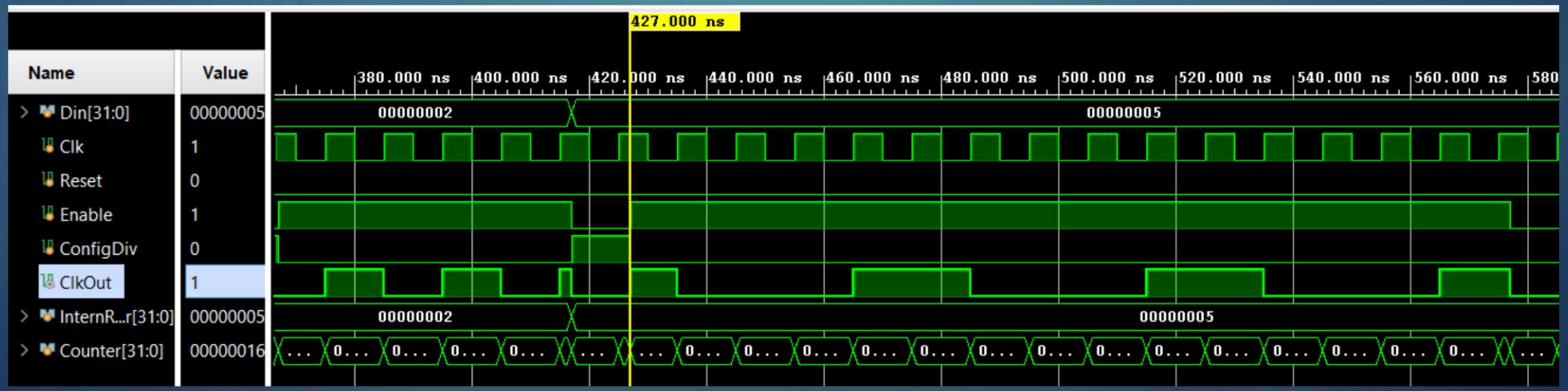
ÿ Division with numbers >= 2.

```
else begin
//this case is used when we want to divide with more than 2
    if(InternRegister >= 2) begin
        if(Counter % InternRegister < InternRegister/2)
          ClkOut <= 1;
        else
          ClkOut <= 0;
    end
end
```

- With even numbers:



- With odd numbers:



Serial Tranceiver

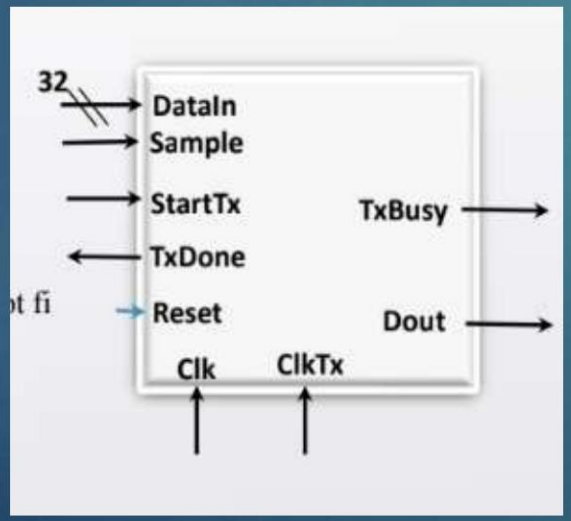
ÿ Parameterizable sequential circuit with 6 inputs and 3

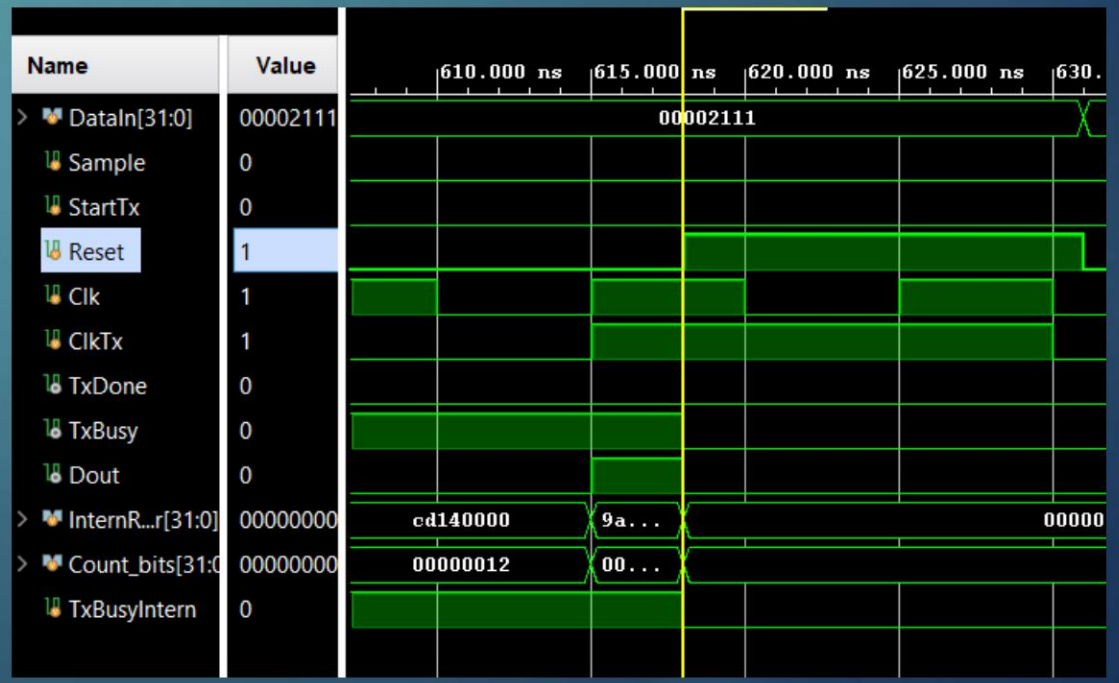
outputs. ÿ The Clk signal is used for internal operation (the DataIn, Sample, StartTx and TxDone signals will be evaluated).

ÿ The ClkTx signal is used for serial data transmission

sampled (TxBusy and Dout signals will be evaluated). This Clk is taken from the frequency divider

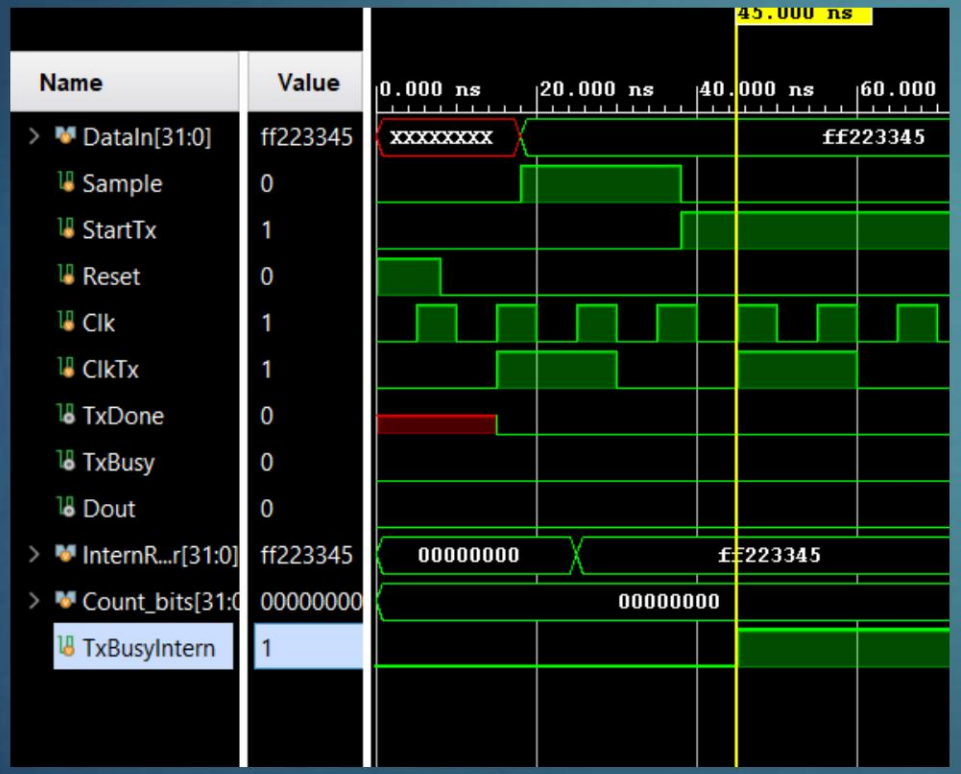
ÿ The reset is asynchronous: it resets internal registers and commands the end of a transfer in progress.





It works like a PISO.

ÿ Sample and StartTx cannot be active at the same time. When Sample is active, the data from the entry in the register is saved internal. It will be evaluated at the positive edge of the internal signal Clk.




```
//in this case we are saving the DataIn inside the intern register
if(Sample && !StartTx) begin
    if(!TxBusyIntern) begin
        InternRegister <= DataIn;
    end
    else begin
        InternRegister <= InternRegister;
    end
end
```

When TxBusy == 1, the transmission of bits from the internal register to the serial output will begin, starting with the MSB.
I used an internal register TxBusyIntern to synchronize the time when the transmission starts and also to synchronize the time when the transmission ends

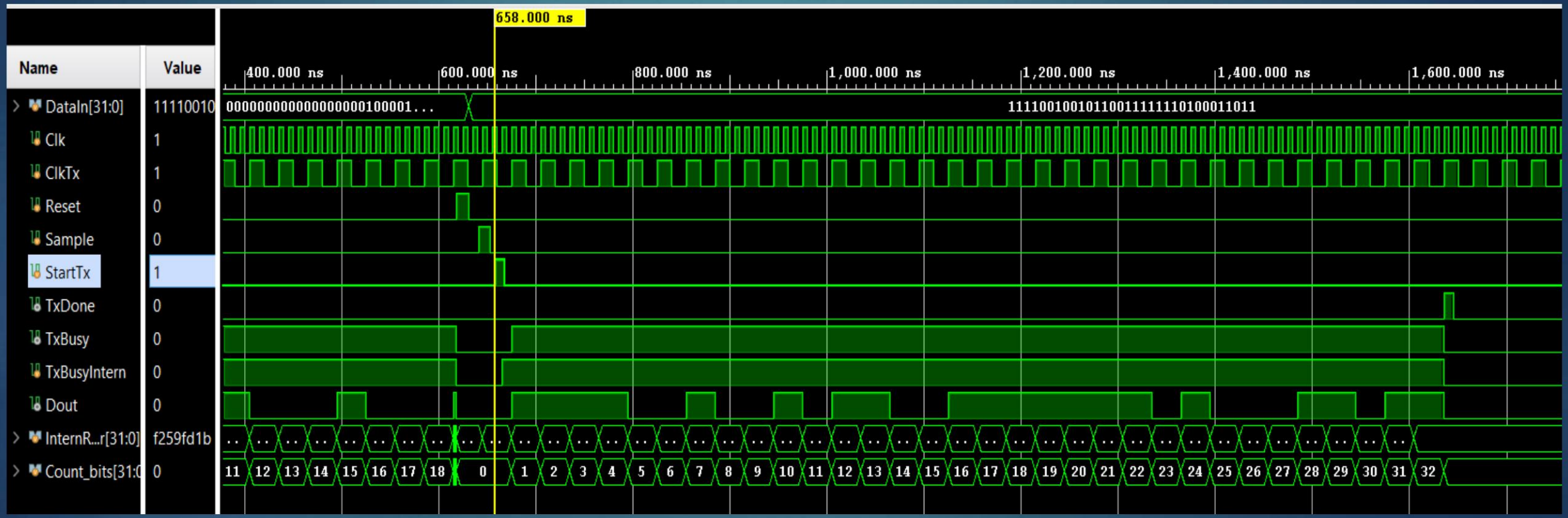
Rated at Clk

```
else if(!Sample && StartTx) begin
    if(!TxDone) begin
        TxBusyIntern <= 1;
    end
    else begin
        TxBusyIntern <= 0;
    end
end
else if(Sample && StartTx) begin
    $display("This case is not possible");
end

if(Count_bits == (SIZE + 1)) begin
    TxDone <= 1'b1;
    TxBusyIntern <= 1'b0;
    Dout <= 1'b0;
    Count_bits <= 0;
end
else begin
    TxDone <= 1'b0;
end
```

Rated at ClkTx

```
  //in this case we will start to sent data to output, bit by bit
  if (Count_bits <= SIZE) begin
        if(TxBusyIntern) begin
            TxBusy <= 1'b1;
            Dout <= InternRegister[SIZE -1];
            InternRegister <= InternRegister << 1;
            Count_bits++;
        end
    end
  if(Count_bits == (SIZE + 1)) begin
        TxBusy <= 1'b0;
    end
end
```

I used a counter that will count the transferred bits. The first bit that is transmitted is MSB and will start with counter = 1, the last bit transferred is when counter = 32, and when the counter is 33, then the transfer will end, that is TxDone = 1 and TxBusy = 0.

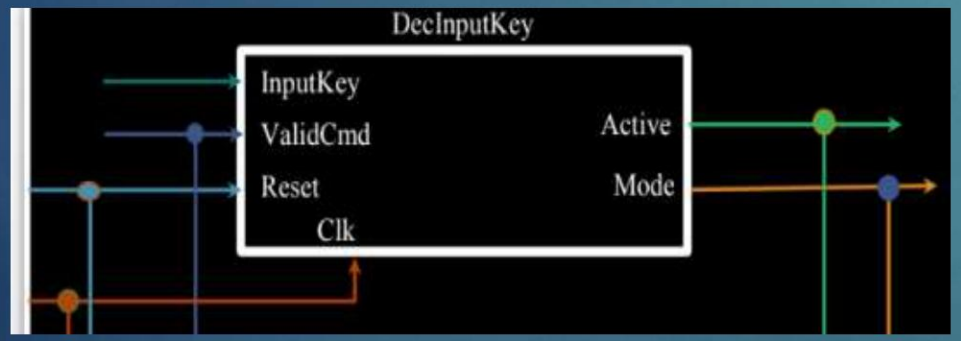


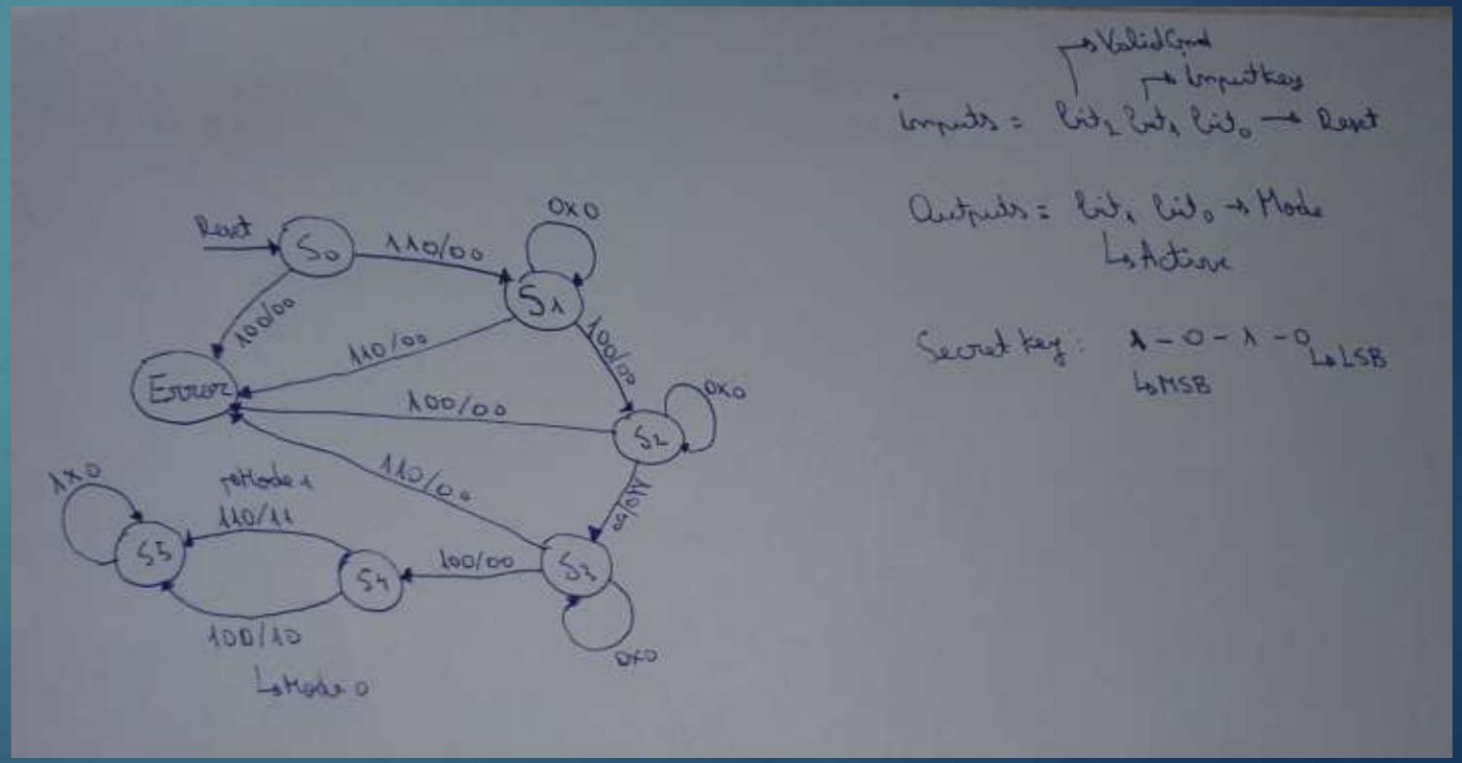
DecInputKey

ÿ Sequential circuit with 4 inputs and 2 outputs. It works on the positive front of the Clk.

ÿ For the circuit to work, a phrase must be entered secret (LSB 1-0-1-0 MSB), after entering it, the next bit will be Mode.

ÿ It works like an FSM.





When entering the secret key, you can reach the desired state or an error state from which you can only exit with a reset.

```
always @(InputKey or Present_state) begin
    case(Present_state)
            S0: begin
                Active <= 1'b0;
                Mode <= 1'b0;
                if(InputKey == 1'b1) begin
                    Next_state <= S1;
                end
                else begin
                    Next_state <= Error;
                end
            end

            S1: begin
                Active <= 1'b0;
                Mode <= 1'b0;
                if(InputKey == 1'b0) begin
                    Next_state <= S2;
                end
                else begin
                    Next_state <= Error;
                end
            end

            S2: begin
                Active <= 1'b0;
                Mode <= 1'b0;
                if(InputKey == 1'b1) begin
                    Next_state <= S3;
                end
                else begin
                    Next_state <= Error;
                end
            end
```

```
            S3: begin
                Active <= 1'b0;
                Mode <= 1'b0;
                if(InputKey == 1'b0) begin
                    Next_state <= S4;
                end
                else begin
                    Next_state <= Error;
                end
            end

            S4: begin
                Active <= 1'b0;
                Mode <= 1'b0;
                Next_state <= S5;
            end

            S5: begin
                Active <= 1'b1;
                //Mode <= Mode;
                Next_state <= S5;
            end

            Error: begin
                Active <= 1'b0;
                Mode <= 1'b0;
                Next_state <= Error;
            end

            default: begin
                Active <= 1'bx;
                Mode <= 1'bx;
                Next_state <= Error;
            end
    endcase
```

Way of operation:

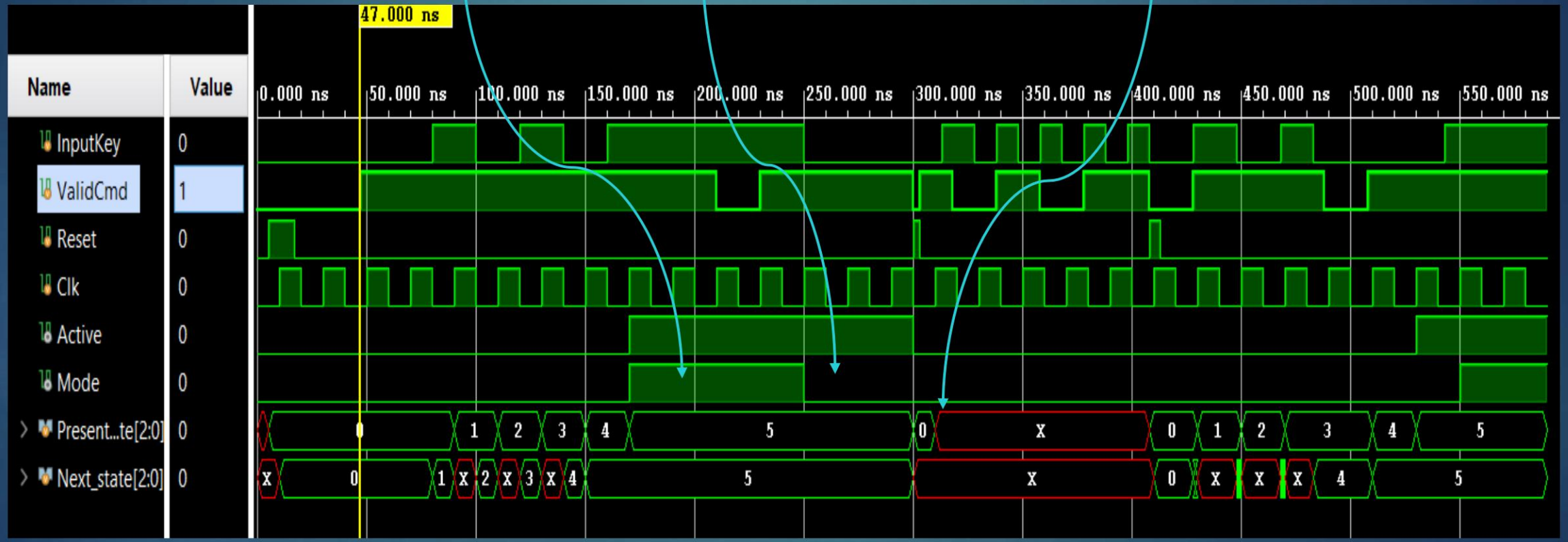
```
always @(posedge Clk or posedge Reset) begin
    if(Reset) begin
        Active <= 1'b0;
        Mode <= 1'b0;
        Present_state <= S0;
        Next_state <= S0;
    end
    else begin
        if(ValidCmd) begin
            Present_state = Next_state;
            if(Present_state == 5) begin
                Mode = InputKey;
            end
        end
    end
end
```

Normal operation:

Mode 1

Mode 0

Error status:

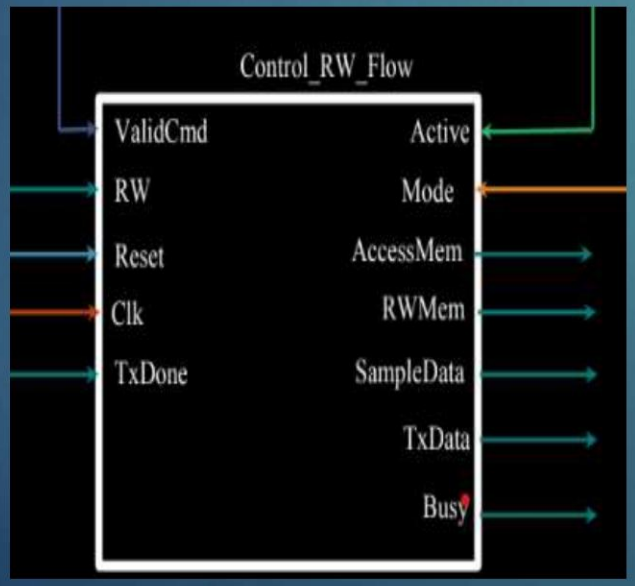


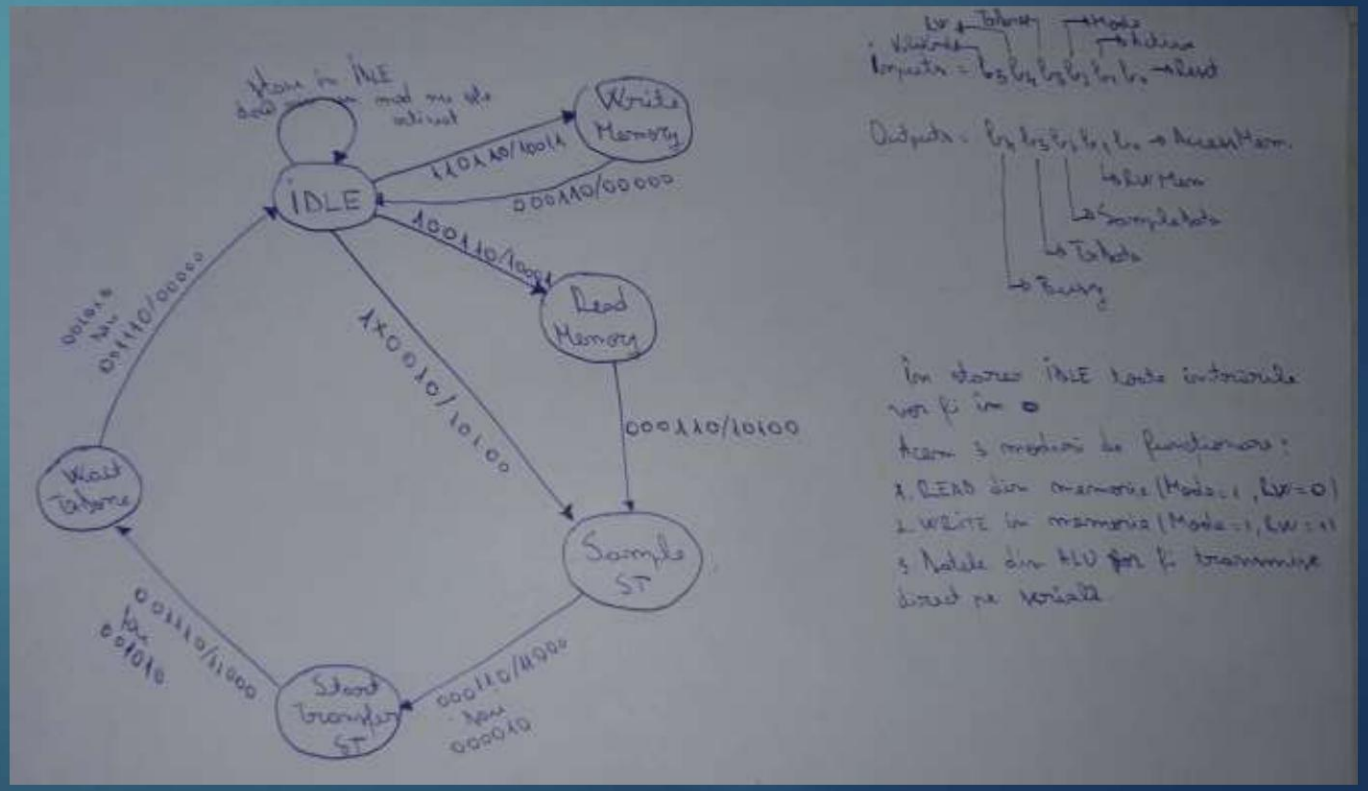
Control_RW_Flow

ÿ It is a sequential circuit with 7 inputs and 5 outputs.

ÿ It has two modes of operation:

- Mode = 0: **Idle -> Sample SerialTransceiver (with data provided by ALU) -> Start Transfer Serial Transceiver -> Wait Transfer Done -> Idle**
- Mode = 1:
  - READ: **Idle -> ReadMemory -> Sample SerialTranceiver -> Start Transfer Serial Tranceiver -> Wait Transfer Done -> Idle**
  - WRITE: **Idle -> WriteMemory -> Idle**





If ValidCmd == 0 then it will not be possible to retrieve any command. But when ValidCmd == 1, the command can be retrieved and it is valid only in the first state, then it can be neglected.
Once a transfer has started, you must wait for the completion of the transfer to be able to take over another order.

When they are in the WaitTxDone state, they wait for the transfer to finish, after which they switch to the IDLE state.

```
always @(*) begin
    case(Present_state)
        IDLE: begin
            AccessMem <= 1'b0;
            RWMem <= 1'b0;
            SampleData <= 1'b0;
            TxData <= 1'b0;
            Busy <= 1'b0;

            if(ValidCmd && Mode && Active && !RW) begin
                Next_state <= ReadMemory;
            end
            else if(ValidCmd && Mode && Active && RW) begin
                Next_state <= WriteMemory;
            end
            else if(ValidCmd && !Mode && Active)begin
                Next_state <= SampleST;
            end
            else begin
                Next_state <= IDLE;
            end
        end

        ReadMemory: begin
            AccessMem <= 1'b1;
            RWMem <= 1'b0;
            SampleData <= 1'b0;
            TxData <= 1'b0;
            Busy <= 1'b1;

            if(ValidCmd && Mode && Active && !RW) begin
                Next_state <= SampleST;
            end
        end
```

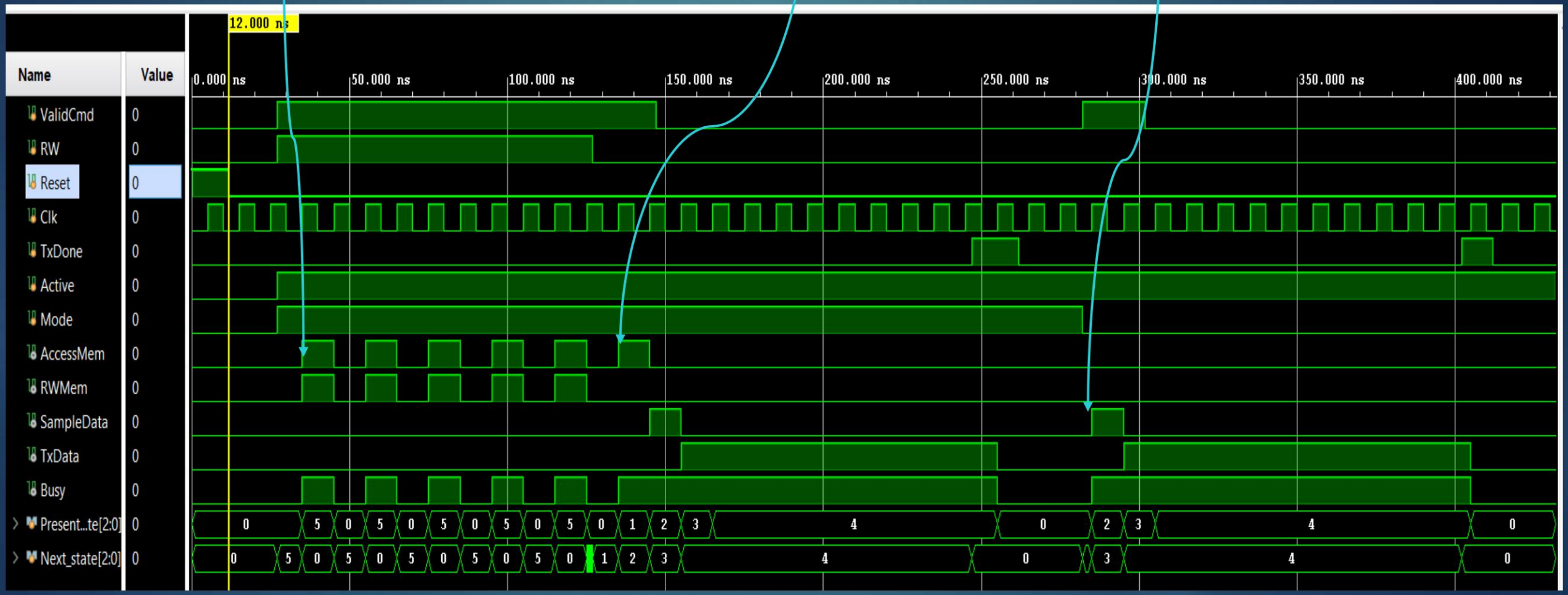
```
StartTransferST: begin
    AccessMem <= 1'b0;
    RWMem <= 1'b0;
    SampleData <= 1'b0;
    TxData <= 1'b1;
    Busy <= 1'b1;

    if(Active && !TxDone && (Mode || !Mode)) begin
        Next_state <= WaitTransferDone;
    end
end

WaitTransferDone: begin
    AccessMem <= 1'b0;
    RWMem <= 1'b0;
    SampleData <= 1'b0;
    TxData <= 1'b1;
    Busy <= 1'b1;

    if(TxDone) begin
        Next_state <= IDLE;
    end
    else begin
        $display("Waiting for trasfer to be done!");
    end
end
```

```
WriteMemory: begin
    AccessMem <= 1'b1;
    RWMem <= 1'b1;
    SampleData <= 1'b0;
    TxData <= 1'b0;
    Busy <= 1'b1;

    Next_state <= IDLE;
end

default: begin
    AccessMem <= 1'b0;
    RWMem <= 1'b0;
    SampleData <= 1'b0;
    TxData <= 1'b0;
    Busy <= 1'b0;

    Next_state <= Error;
end
```

Successive writes in memory

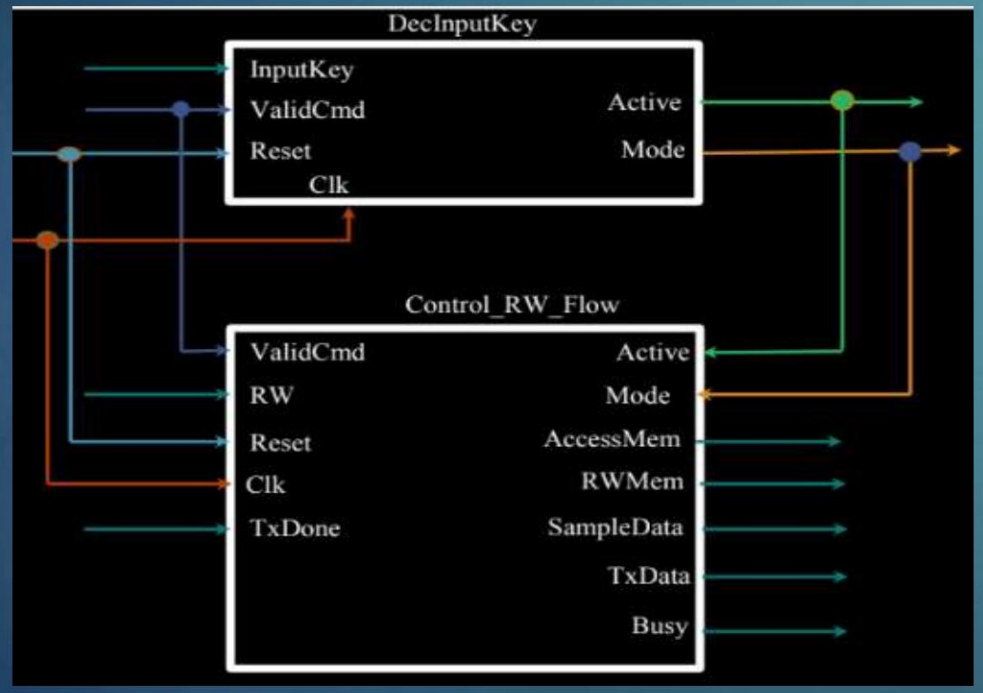
Reading from memory

Mode 0



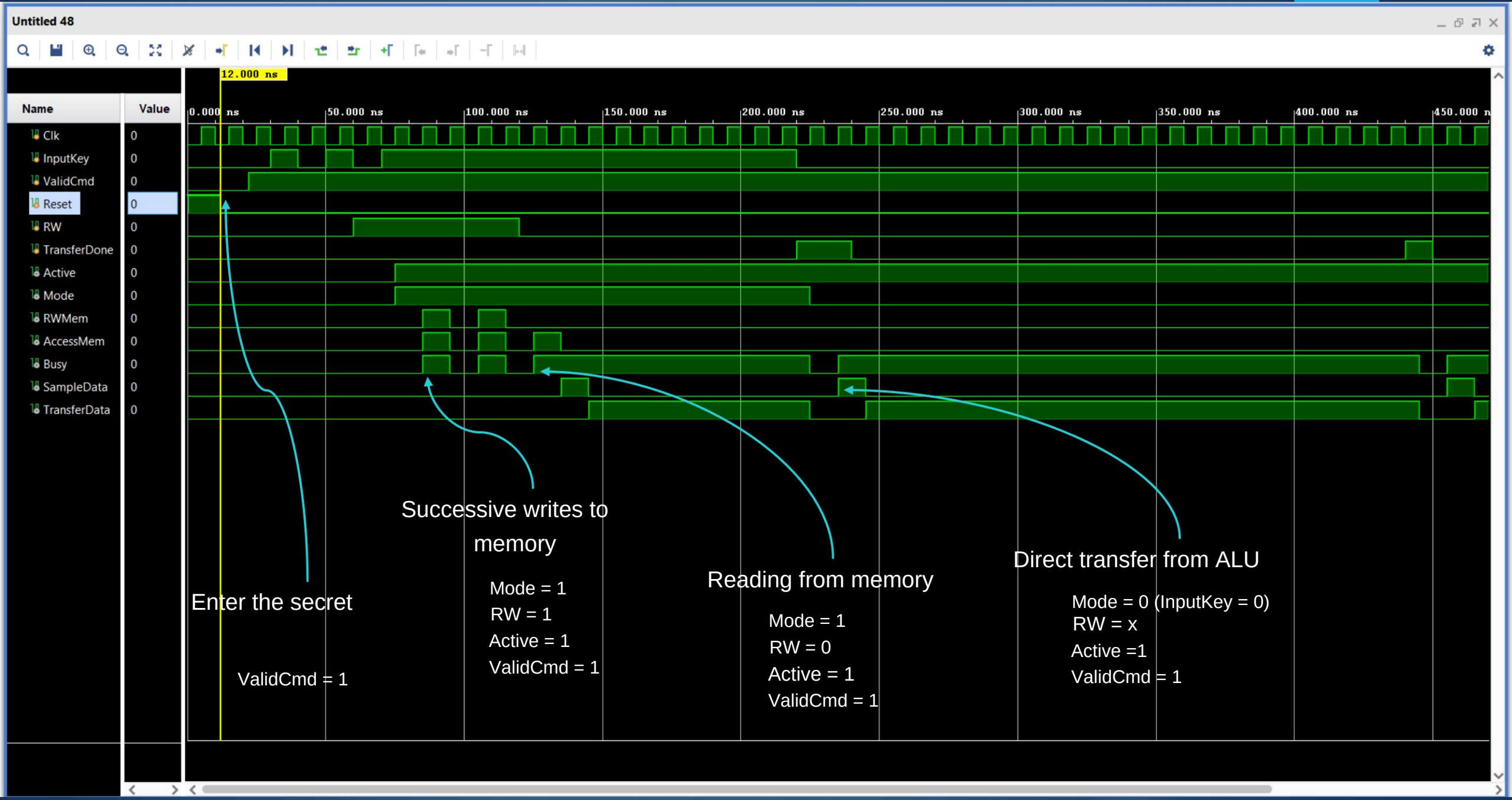
Controller

- Sequential circuit created to incorporate the two modules from imagine.
- Active and Mode outputs from DecInputKey are constituted as inputs for Control_RW_Flow




```
module Controller
(
    input InputKey,
    input Clk,
    input ValidCmd,
    input Reset,
    input RW,
    input TransferDone,
    output Active,
    output Mode,
    output RWMem,
    output AccessMem,
    output Busy,
    output SampleData,
    output TransferData
);
```

```
wire Active_wire;
wire Mode_wire;

DecInputKey DUT_DIK(
.InputKey(InputKey),
.ValidCmd(ValidCmd),
.Reset(Reset),
.Clk(Clk),
.Active(Active_wire),
.Mode(Mode_wire)
);

assign Active = Active_wire;
assign Mode = Mode_wire;

Control_RW_Flow DUT_CRWF(
.ValidCmd(ValidCmd),
.RW(RW),
.Reset(Reset),
.Clk(Clk),
.TxDone(TransferDone),
.Active(Active_wire),
.Mode(Mode_wire),
.AccessMem(AccessMem),
.RWMem(RWMem),
.SampleData(SampleData),
.TxData(TransferData),
.Busy(Busy)
);
```

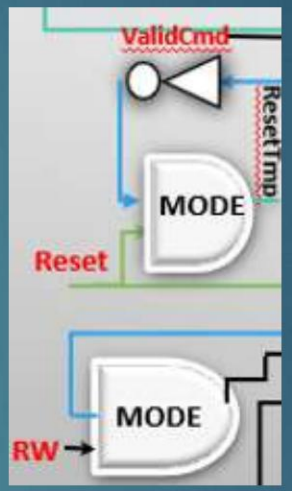


integration

ÿ This involves uniting all the modules in such a way that perform the desired functionalities correctly.

ÿ Aspect important:

- RestTmp = Reset & ~ActiveCalcTmp
- RWTmp = RWMem & ActiveCalcTmp




```
wire ResetTmp, ActiveCalcTmp, RWTmp, CtrlModeTmp;
wire SampleDataTmp, TxDoneTmp, CtrlRWMemTmp, CtrlAccessMemTmp;
wire [7:0] MuxInATmp;
wire [7:0] MuxInBTmp;
wire [3:0] MuxSelTmp;
wire [7:0] AluOutTmp;
wire [3:0] AluFlagTmp;
wire [31:0] ConcatOutTmp, TxDinTmp, MemoryOutTmp;

assign ResetTmp = Reset & (~ActiveCalcTmp);
assign RWTmp = RWMem & ActiveCalcTmp;
assign CalcActive = ActiveCalcTmp;
assign CalcMode = CtrlModeTmp;
```

```
module Binary_Calculator
(
    input InputKey,
    input ValidCmd,
    input RWMem,
    input [7:0] Addr,
    input [7:0] InA,
    input [7:0] InB,
    input [3:0] Sel,
    input ConfigDiv,
    input [31:0] Datain,
    input Reset,
    input Clk,
    output CalcActive,
    output CalcMode,
    output CalcBusy,
    output DOutValid,
    output DataOut,
    output ClkTx
);
```

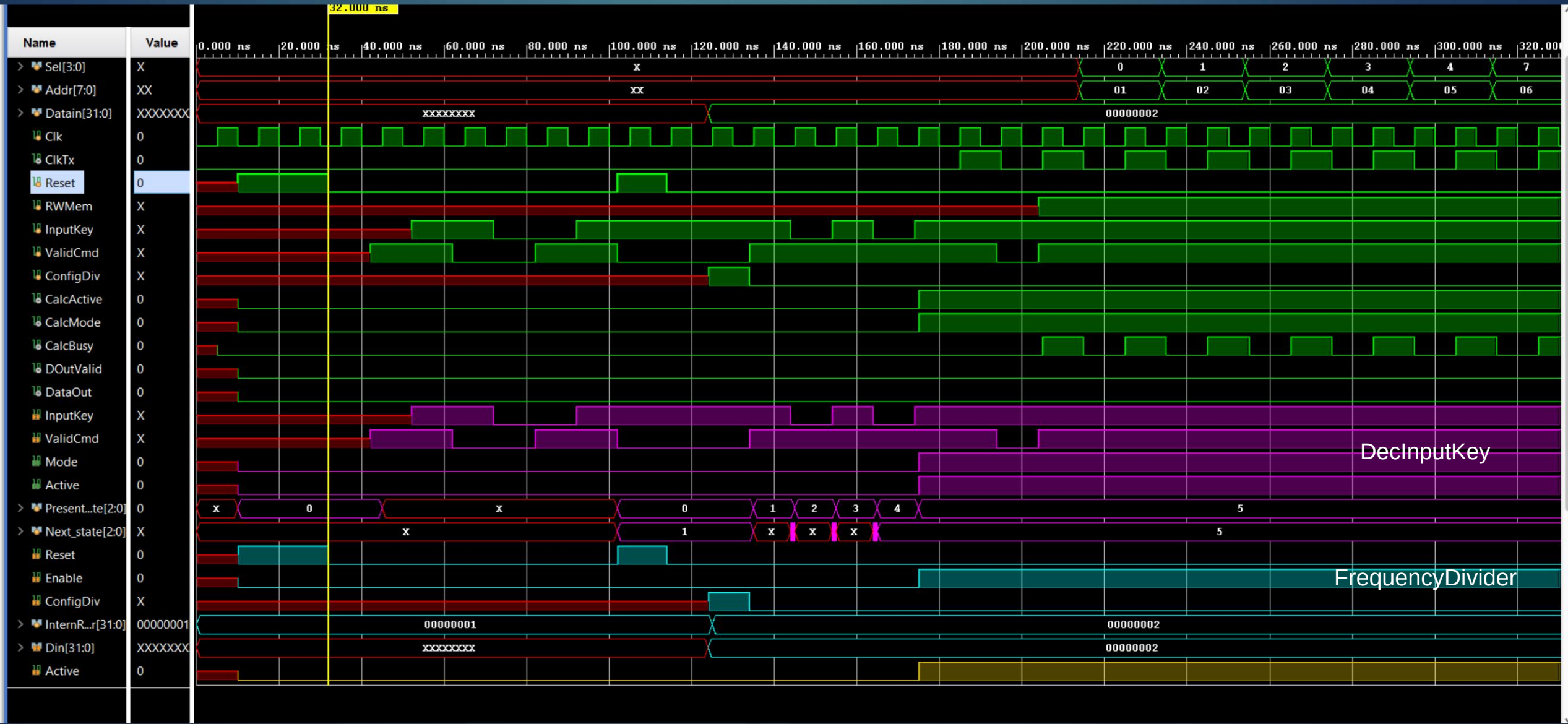
```
//initialize the ALU
ALU ALU(
    .A(MuxInATmp),
    .B(MuxInBTmp),
    .Sel(MuxSelTmp),
    .Out(AluOutTmp),
    .Flag(AluFlagTmp)
);

//initialize the Concatenator
Concatenator Concatenator(
    .InA(MuxInATmp),
    .InB(MuxInBTmp),
    .InC(AluOutTmp),
    .InD(MuxSelTmp),
    .InE(AluFlagTmp),
    .Out(ConcatOutTmp)
);

//initialize the Controller
Controller Controller(
    .InputKey(InputKey),
    .Clk(Clk),
    .ValidCmd(ValidCmd),
    .Reset(Reset),
    .RW(RWTmp),
    .TransferDone(TxDoneTmp),
    .Active(ActiveCalcTmp),
    .Busy(CalcBusy),
    .RWMem(CtrlRWMemTmp),
    .AccessMem(CtrlAccessMemTmp),
    .Mode(CtrlModeTmp),
    .SampleData(SampleDataTmp),
    .TransferData(TransferDataTmp)
);
```

```
//initialize the Memory
Memory Memory(
    .Din(ConcatOutTmp),
    .Addr(Addr),
    .R_W(CtrlRWMemTmp),
    .Valid(CtrlAccessMemTmp),
    .Reset(ResetTmp),
    .Clk(Clk),
    .Dout(MemoryOutTmp)
);

//initialize the FrequencyDivider
FrequencyDivider FrequencyDivider(
    .Din(Datain),
    .ConfigDiv(ConfigDiv),
    .Reset(ResetTmp),
    .Clk(Clk),
    .Enable(ActiveCalcTmp),
    .ClkOut(ClkTx)
);

//initialize the SerialTranceiver
SerialTranceiver SerialTranceiver(
    .DataIn(TxDinTmp),
    .Sample(SampleDataTmp),
    .StartTx(TransferDataTmp),
    .TxDone(TxDoneTmp),
    .Reset(ResetTmp),
    .Clk(Clk),
    .ClkTx(ClkTx),
    .TxBusy(DOutValid),
    .Dout(DataOut)
);
```

```
//initialize the Muxs
Mux_8b M1(
    .Din(InA),
    .Select(ResetTmp),
    .Dout(MuxInATmp)
);

Mux_8b M2(
    .Din(InB),
    .Select(ResetTmp),
    .Dout(MuxInBTmp)
);

Mux_4b M3(
    .Din(Sel),
    .Select(ResetTmp),
    .Dout(MuxSelTmp)
);

Mux_32b M4(
    .Din_A(ConcatOutTmp),
    .Din_B(MemoryOutTmp),
    .Select(CtrlModeTmp),
    .Dout(TxDinTmp)
);
```

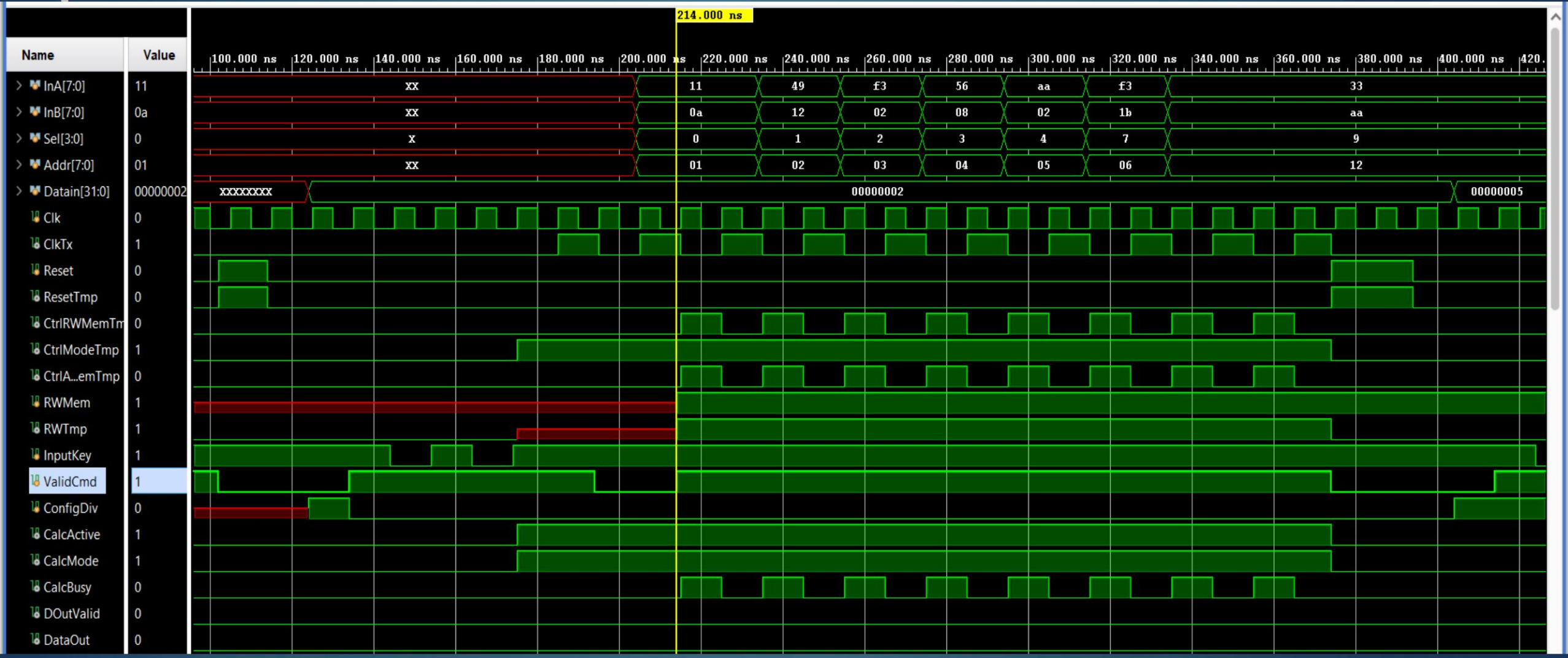
scenarios

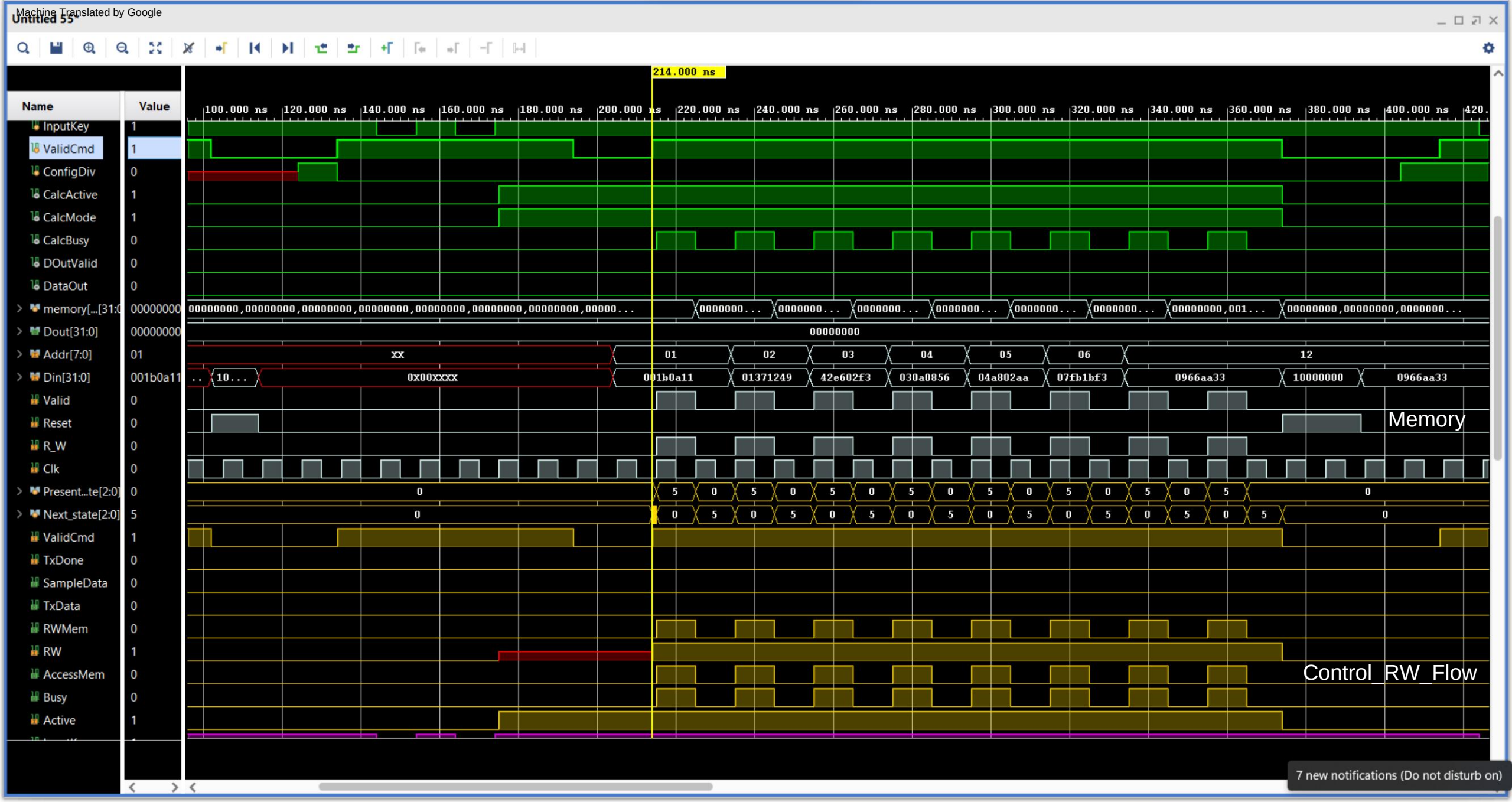
ÿ Case 1: One scenario must check the functionality of the DUT when we cross between INVALID INPUT_KEY and VALID INPUT_KEY. Also we divide ClkTx by 2.



ÿ Case 2: One scenario must make 7 different operations and write the different 7 results to memory.

Case 3: One scenario must check what happens when the RTL is in Mode 1 write to memory, but a reset is randomly activated.



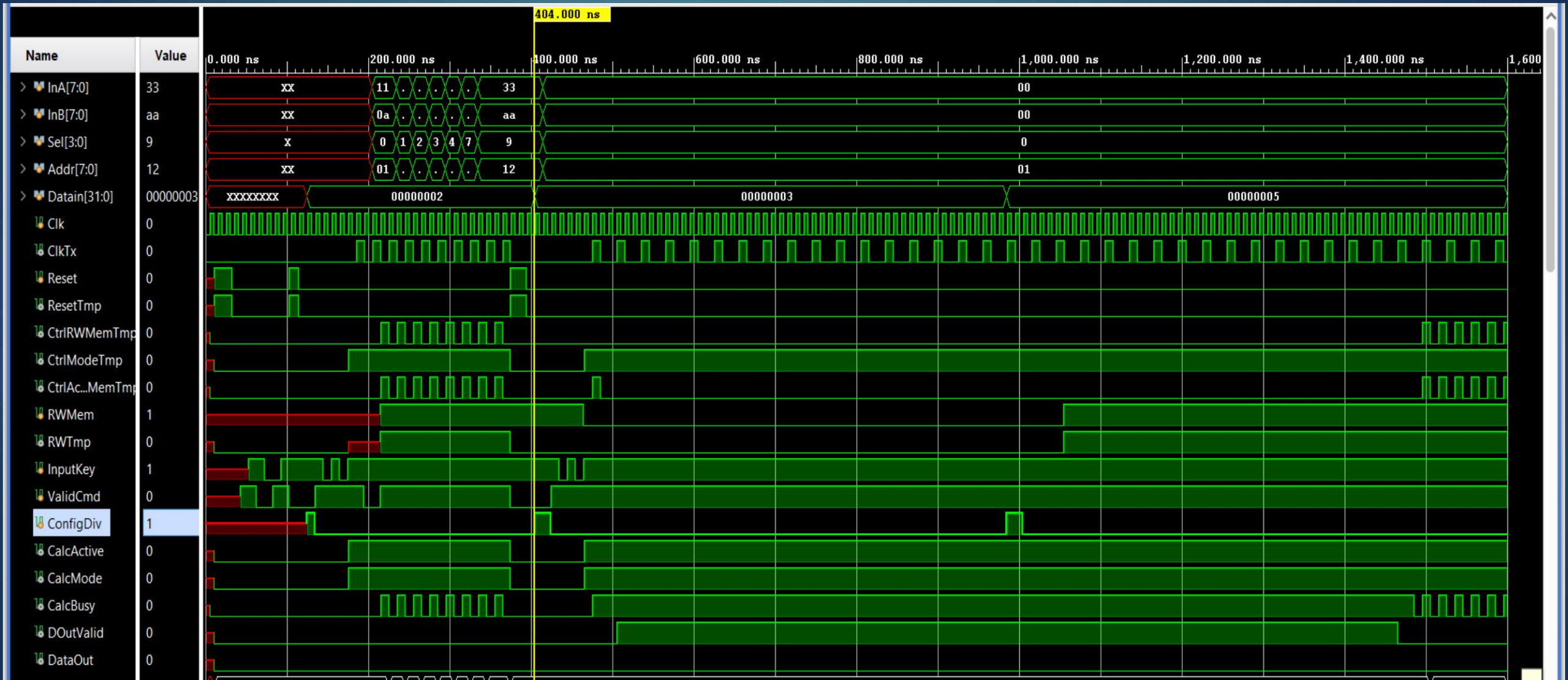


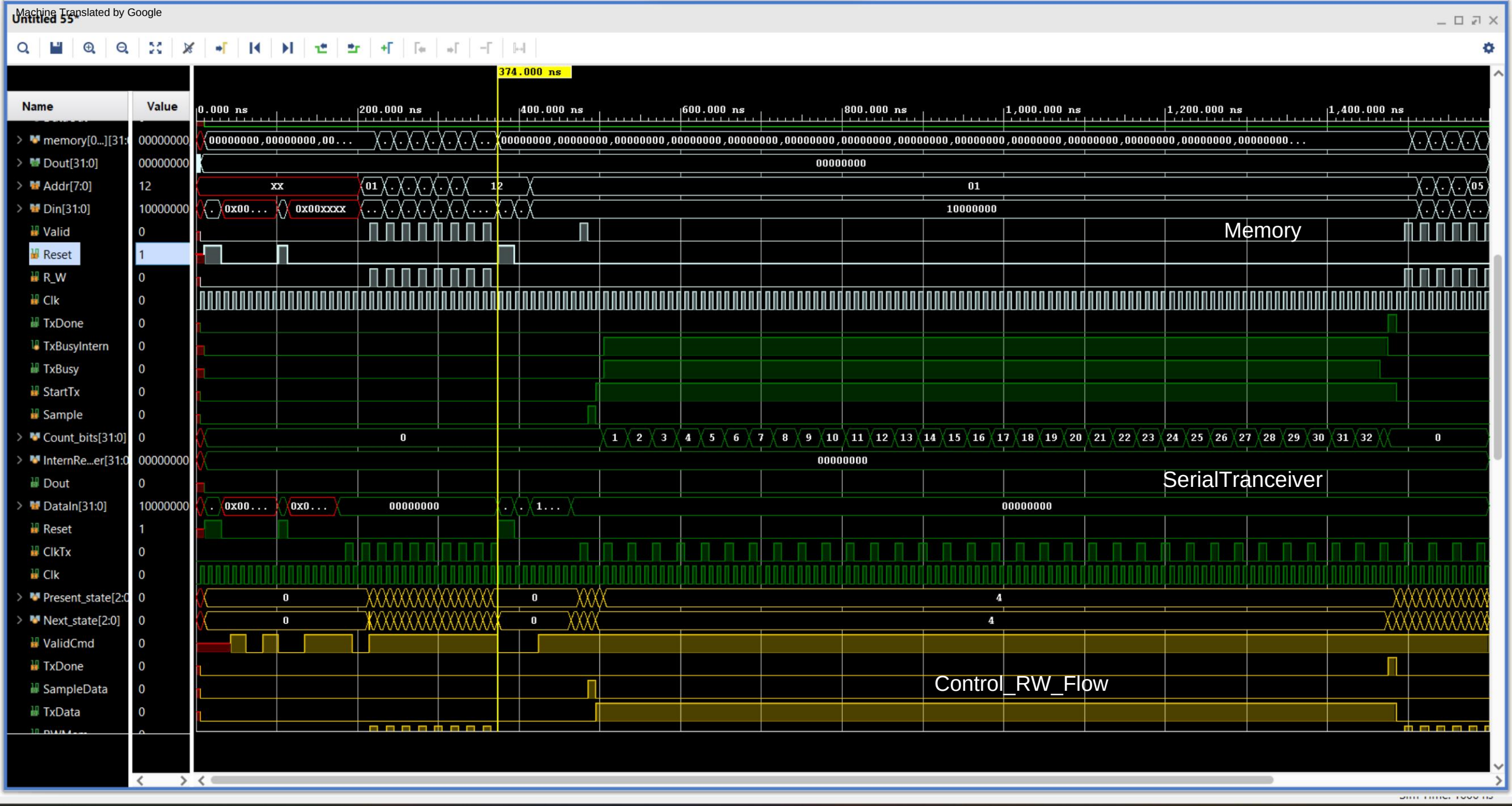
7 new notifications (Do not disturb on)

- Case 4: One scenario must check if, after reset, we enter in Mode 1 and we try to read something, if we can get any valid data.

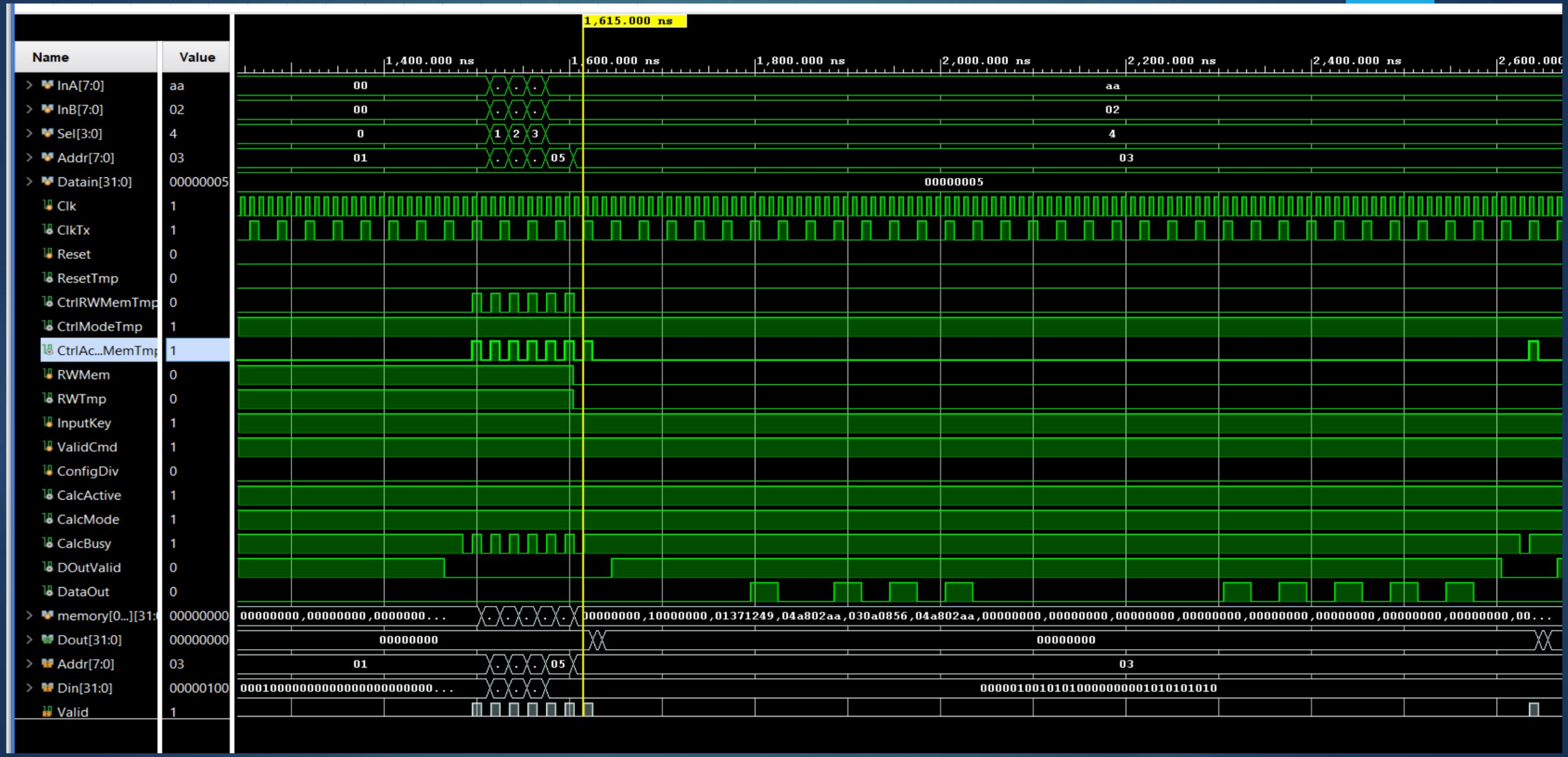
Case 5: One scenario must check the functionality of the DUT when we cross from a CLK frequency (ClkX is divided by 2) to a different one (ClkX is divided by 5) and we are in Mode 1 Read from memory, if we have any errors.

Case 6: One scenario must check the functionality of the DUT when we cross from Mode 1 read to Mode 1 write, if we have any errors.

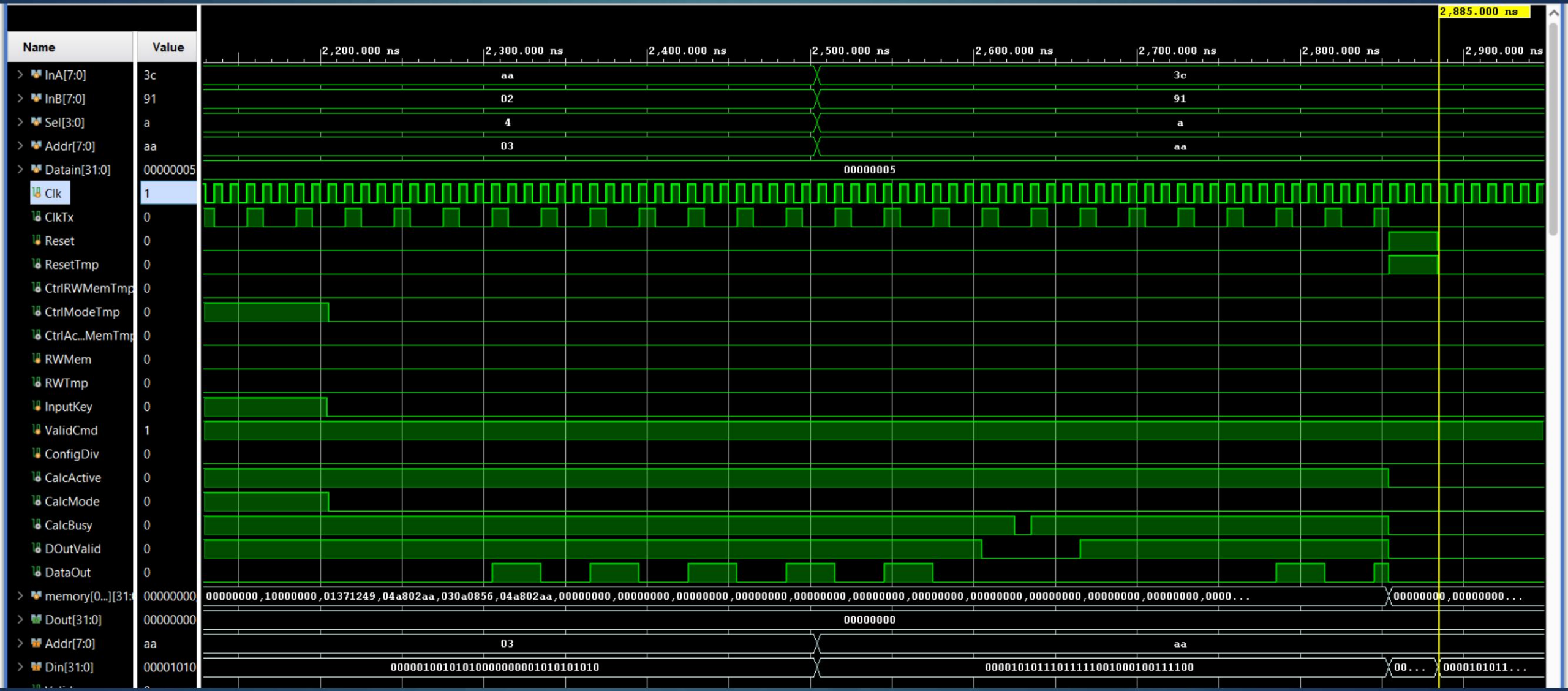




Case 7: Write a valid data from a address



- Case 8 : One scenario must check the functionality of the DUT when we cross between Mode 0 and Mode 1.
- Case 9: One scenario must check the functionality of the DUT when the mode 0 is active, and a reset is activated for 3 clock cycles.



UVM – SystemVerilog

Memory

```
interface Memory_itf;
  logic Clk, Reset;
  logic R_W, Valid;
  logic [7:0] Addr;
  logic [31:0] Din;
  logic [31:0] Dout;
endinterface
```

```
class generator;
  transaction trans;
  mailbox #(transaction) mbx;
  event next;
  event done;
  int count;

  function new(mailbox #(transaction) mbx);
    this.mbx = mbx;
    trans = new();
  endfunction

  task main();
    repeat(count) begin
      assert(trans.randomize()) else $display("Randomize failed!");
      mbx.put(trans);
      trans.display("GEN");
      @(next);
    end
    ->done;
  endtask

endclass
```

```
class transaction;
  rand bit Operation;
  bit R_W;
  bit Valid;
  randc bit [7:0] Addr;
  randc bit [31:0] Din;
  bit [31:0] Dout;

  function void display(input string tag);
    $display("[%0s] -> Operation: %0b, Valid: %0b, Addr: %0h, Din: %0h, Dout: %0h",
tag, Operation, Valid, Addr, Din, Dout);
  endfunction

  constraint operation_contraint{
    Operation dist {1 :/ 70, 0 :/ 30};
  }

  constraint Addr_min{
    Addr inside {[0 : 5]};
  }
endclass
```

```
class driver;
  virtual Memory_itf itf;
  transaction trans;
  mailbox #(transaction) mbx;

  function new(mailbox #(transaction) mbx);
    this.mbx = mbx;
  endfunction

  task Reset();
    itf.Reset <= 1'b0;
    itf.R_W <= 1'b0;
    itf.Valid <= 1'b0;
    itf.Addr <= 8'b0;
    itf.Din <= 32'b0;
    @(posedge itf.Clk);
    itf.Reset <= 1'b1;
    repeat(2) @(posedge itf.Clk);
    itf.Reset <= 1'b0;
    $display("[DRV] -> RESET DONE.");
    $display("---------------------");
  endtask
```

```
task Read();
  @(posedge itf.Clk);
  itf.Reset <= 1'b0;
  itf.R_W <= 1'b0;
  itf.Valid <= 1'b1;
  itf.Addr <= trans.Addr;
  itf.Din <= trans.Din;
  @(posedge itf.Clk);
  itf.R_W <= 1'b0;
  itf.Valid <= 1'b0;
  $display("[DRV] -> READ.");
  $display("[DRV] -> Data read from Address: %0h", itf.Addr);
  @(posedge itf.Clk);
endtask
```

```
task main();
  forever begin
    mbx.get(trans);
    if(trans.Operation == 1)
      Write();
    else
      Read();
  end
endtask
```

```
class monitor;
  virtual Memory_itf itf;
  transaction trans;
  mailbox #(transaction) mbx;

  function new(mailbox #(transaction) mbx);
    this.mbx = mbx;
  endfunction

  task main();
    trans = new();

    forever begin
      repeat(2) @(posedge itf.Clk);
      trans.R_W = itf.R_W;
      trans.Valid = itf.Valid;
      trans.Addr = itf.Addr;
      trans.Din = itf.Din;
      @(posedge itf.Clk);
      trans.Dout = itf.Dout;

      mbx.put(trans);
      $display("[MON] -> R_W: %0b, Valid: %0b, Addr: %0h, Din: %0h, Dout: %0h",
trans.R_W, trans.Valid, trans.Addr, trans.Din, trans.Dout);
    end

  endtask

endclass
```

```
task Write();
  @(posedge itf.Clk);
  itf.Reset <= 1'b0;
  itf.R_W <= 1'b1;
  itf.Valid <= 1'b1;
  itf.Addr <= trans.Addr;
  itf.Din <= trans.Din;
  @(posedge itf.Clk);
  itf.R_W <= 1'b0;
  itf.Valid <= 1'b0;
  $display("[DRV] -> WRITE.");
  $display("[DRV] -> Data: %0h saved to Address: %0h", itf.Din, itf.Addr);
  @(posedge itf.Clk);
endtask
```

```
class scoreboard;
  transaction trans;
  mailbox #(transaction) mbx;
  event next;
  bit [31:0] memory[255:0];

  function new(mailbox #(transaction) mbx);
    this.mbx = mbx;
  endfunction

  task main();
    forever begin
      mbx.get(trans);
      $display("[SCO] -> R_W: %0b, Valid: %0b, Addr: %0h, Din: %0h, Dout: %0h", trans.R_W,
trans.Valid, trans.Addr, trans.Din, trans.Dout);

      if(trans.Valid == 1)
        if(trans.R_W == 1) begin
          memory[trans.Addr] = trans.Din;
          $display("[SCO] -> Data: %0h saved to Address: %0h", trans.Din, trans.Addr);
        end
        else begin
          if(trans.Dout == memory[trans.Addr]) begin
            $display("[SCO] -> Dout(%0h) == memory[%0h](%0h).", trans.Dout, trans.Addr,
memory[trans.Addr]);
            $display("[SCO] -> Data matched.");
          end
          else begin
            $display("[SCO] -> Dout(%0h) != memory[%0h](%0h).", trans.Dout, trans.Addr,
memory[trans.Addr]);
            $display("[SCO] -> Data mismatched.");
          end
        end
      else
        $display("No command in proccess.");

      $display("------------------------------");
      -> next;
    end

  endtask
```

```
class environment;
  generator gen;
  driver drv;
  monitor mon;
  scoreboard sco;
  mailbox #(transaction) gdmbx;  // Gen
  mailbox #(transaction) msmbx;  // Mon
  event nextgs;
  virtual Memory_itf itf;

  function new(virtual Memory_itf itf);
    gdmbx = new();
    gen = new(gdmbx);
    drv = new(gdmbx);
    msmbx = new();
    mon = new(msmbx);
    sco = new(msmbx);
    this.itf = itf;
    drv.itf = this.itf;
    mon.itf = this.itf;
    gen.next = nextgs;
    sco.next = nextgs;
  endfunction

  task pre_test();
    drv.Reset();
  endtask

  task test();
    fork
      gen.main();
      drv.main();
      mon.main();
      sco.main();
    join_any
  endtask

  task post_test();
    wait(gen.done.triggered);
    $finish();
  endtask

  task main();
    pre_test();
    test();
    post_test();
  endtask
endclass
```

```
module testbench();

  Memory_itf itf();
  environment env;

  Memory DUT(
     .Clk(itf.Clk),
     .Reset(itf.Reset),
     .R_W(itf.R_W),
     .Valid(itf.Valid),
     .Addr(itf.Addr),
     .Din(itf.Din),
     .Dout(itf.Dout)
  );

  initial begin
     itf.Clk <= 1'b0;
  end

  always #5 itf.Clk <= ~itf.Clk;

  initial begin
     env = new(itf);
     env.gen.count = 30;
     env.main();
  end

  initial begin
     $dumpfile("dump.vcd");
     $dumpvars;
  end
endmodule
```

```
[DRV] -> RESET DONE.
----------------------
[GEN] -> Operation: 0, Valid: 0, Addr: 4, Din: 1da21f, Dout: 0
[DRV] -> READ.
[DRV] -> Data read from Address: 4
[MON] -> R_W: 0, Valid: 1, Addr: 4, Din: 1da21f, Dout: 0
[SCO] -> R_W: 0, Valid: 1, Addr: 4, Din: 1da21f, Dout: 0
[SCO] -> Dout(0) == memory[4](0).
[SCO] -> Data matched.
--------------------------------
```

```
[GEN] -> Operation: 1, Valid: 0, Addr: 5, Din: 1d21d7b2, Dout: 0
[DRV] -> WRITE.
[DRV] -> Data: 1d21d7b2 saved to Address: 5
[MON] -> R_W: 1, Valid: 1, Addr: 5, Din: 1d21d7b2, Dout: 0
[SCO] -> R_W: 1, Valid: 1, Addr: 5, Din: 1d21d7b2, Dout: 0
[SCO] -> Data: 1d21d7b2 saved to Address: 5
--------------------------------
[GEN] -> Operation: 1, Valid: 0, Addr: 1, Din: 6418962c, Dout: 0
[DRV] -> WRITE.
[DRV] -> Data: 6418962c saved to Address: 1
[MON] -> R_W: 1, Valid: 1, Addr: 1, Din: 6418962c, Dout: 0
[SCO] -> R_W: 1, Valid: 1, Addr: 1, Din: 6418962c, Dout: 0
[SCO] -> Data: 6418962c saved to Address: 1
--------------------------------
[GEN] -> Operation: 1, Valid: 0, Addr: 3, Din: c30e5c40, Dout: 0
[DRV] -> WRITE.
[DRV] -> Data: c30e5c40 saved to Address: 3
[MON] -> R_W: 1, Valid: 1, Addr: 3, Din: c30e5c40, Dout: 0
[SCO] -> R_W: 1, Valid: 1, Addr: 3, Din: c30e5c40, Dout: 0
[SCO] -> Data: c30e5c40 saved to Address: 3
--------------------------------
[GEN] -> Operation: 0, Valid: 0, Addr: 5, Din: dceac0ed, Dout: 0
[DRV] -> READ.
[DRV] -> Data read from Address: 5
[MON] -> R_W: 0, Valid: 1, Addr: 5, Din: dceac0ed, Dout: 1d21d7b2
[SCO] -> R_W: 0, Valid: 1, Addr: 5, Din: dceac0ed, Dout: 1d21d7b2
[SCO] -> Dout(1d21d7b2) == memory[5](1d21d7b2).
[SCO] -> Data matched.
--------------------------------
```

SerialTransceiver

```
class transaction;

  //inputs
  rand logic [31:0] DataIn;
  logic Sample;
  logic StartTx;
  //outputs
  logic TxDone;
  logic TxBusy;
  logic Dout;

  function void display(input string tag);
    $display("[%0t]:[%0s] -> DataIn: %0h, Sample: %0b, StartTx: %0b, TxDone: %0b, TxBusy:
%0b, Dout: %0b", $time(), tag, DataIn, Sample, StartTx, TxDone, TxBusy, Dout);
  endfunction

  function transaction copy();
    copy = new();
    copy.DataIn = this.DataIn;
    copy.Sample = this.Sample;
    copy.StartTx = this.StartTx;
    copy.TxDone = this.TxDone;
    copy.TxBusy = this.TxBusy;
    copy.Dout = this.Dout;
  endfunction

endclass
```

```
class generator;

  transaction trans;
  mailbox #(transaction) mbx_gd;
  event done;
  event drv_next;
  event sco_next;
  int count = 0;

  function new(mailbox #(transaction) mbx_gd);
    this.mbx_gd = mbx_gd;
    trans = new();
  endfunction

  task main();
    repeat(count) begin
      assert(trans.randomize) else $display("Randomize failed.");
      trans.display ("GEN");
      mbx_gd.put(trans.copy);
      @(drv_next);
      @(sco_next);
    end
    ->done;
  endtask

endclass
```

```
class driver;
  virtual SerialTranceiver_itf itf;
  transaction trans;
  mailbox #(transaction) mbx_gd;
  mailbox #(logic [31:0]) mbx_ds;
  event drv_next;

  logic [31:0] DataIn;

  function new(mailbox #(transaction) mbx_gd, mailbox #(logic [31:0]) mbx_ds);
    this.mbx_gd = mbx_gd;
    this.mbx_ds = mbx_ds;
  endfunction

  task reset();
    itf.DataIn <= 32'h0;
    itf.Sample <= 1'b0;
    itf.StartTx <= 1'b0;
    itf.Reset <= 1'b1;
    repeat(3) @(posedge itf.Clk);
    itf.Reset <= 1'b0;
    repeat(2) @(posedge itf.Clk);

    $display("[DRV] -> Reset Done.");
    $display("------------------------------------");
  endtask
```

Driver

```
task main();
  forever begin
    mbx_gd.get(trans);
    itf.DataIn <= 32'h00;
    itf.Sample <= 1'b0;
    itf.StartTx <= 1'b0;
    @(negedge itf.Clk);
    itf.DataIn <= trans.DataIn;
    itf.Sample <= 1'b1;
    itf.StartTx <= 1'b0;
    mbx_ds.put(trans.DataIn);
    @(negedge itf.Clk);
    itf.Sample <= 1'b0;
    itf.StartTx <= 1'b1;
    wait(itf.TxBusy == 1'b1);
    $display("[%0t]:[DRV] -> Transfer started! Data Sent to Serial: %0h", $time(),
trans.DataIn);
    wait(itf.TxDone == 1'b0);
    itf.DataIn <= 32'h00;
    itf.Sample <= 1'b0;
    itf.StartTx <= 1'b0;
    ->drv_next;
  end
endtask
```

Monitor

```
task main();
  forever begin
    wait(itf.TxBusy == 1'b1);

    for(int i = 0; i <= 31; i++) begin
      @(negedge itf.ClkTx);
      SerialData[31 - i] = itf.Dout;
    end

    wait(itf.TxDone == 1'b1);
    @(posedge itf.ClkTx);

    $display("[%0t]:[MON] -> Transfer Ended! Data Sent = %0h.", $time(), SerialData);
    mbx_ms.put(SerialData);
  end

endtask
```

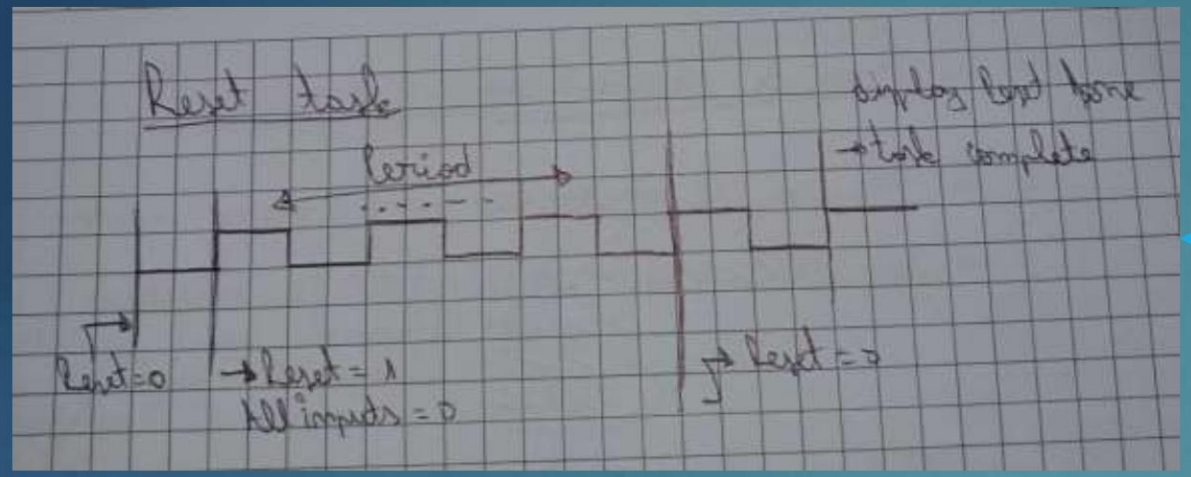
```
class scoreboard;
  mailbox #(logic [31:0]) mbx_ds;
  mailbox #(logic [31:0]) mbx_ms;
  event sco_next;

  logic [31:0] Data_ds; //golden data from the driver
  logic [31:0] Data_ms; //data that we receaved serialy from the DUT

  function new(mailbox #(logic [31:0]) mbx_ds, mailbox #(logic [31:0]) mbx_ms);
    this.mbx_ds = mbx_ds;
    this.mbx_ms = mbx_ms;
  endfunction

  task main();
    mbx_ds.get(Data_ds);
    mbx_ms.get(Data_ms);

    $display("[%0t]:[SCO] -> Data from DRV = %0h, Data from MON = %0h.", $time(), Data_ds,
Data_ms);

    if(Data_ds == Data_ms)
      $display("[%0t]:[SCO] -> DATA MATCHED", $time());
    else
      $display("[%0t]:[SCO] -> DATA MISMATCHED", $time());

    $display("------------------------------------");

    ->sco_next;
  endtask

endclass
```

```
[DRV] -> Reset Done.
------------------------------------
[45000]:[GEN] -> DataIn: d31823e4, Sample: x, StartTx: x, TxDone: x, TxBusy: x, Dout: x
[100000]:[DRV] -> Transfer started! Data Sent to Serial: d31823e4
[1420000]:[MON] -> Transfer Ended! Data Sent = d31823e4.
[1420000]:[SCO] -> Data from DRV = d31823e4, Data from MON = d31823e4.
[1420000]:[SCO] -> DATA MATCHED
```

BinaryCalculator

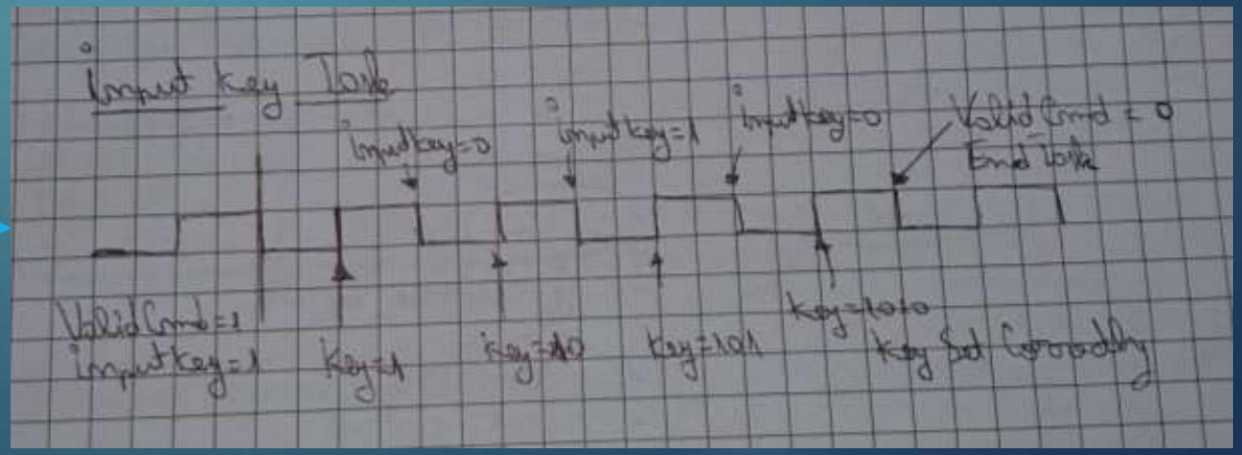
```
task main();
  repeat(counter) begin
    assert(trans.randomize) else $display("Randomize failed.");
    mbx.put(trans.copy);
    $display("---------------------------------");
    trans.display_inputs("GEN");
    @(next_drv);
    @(next_sco);
  end
  -> done;
endtask
```

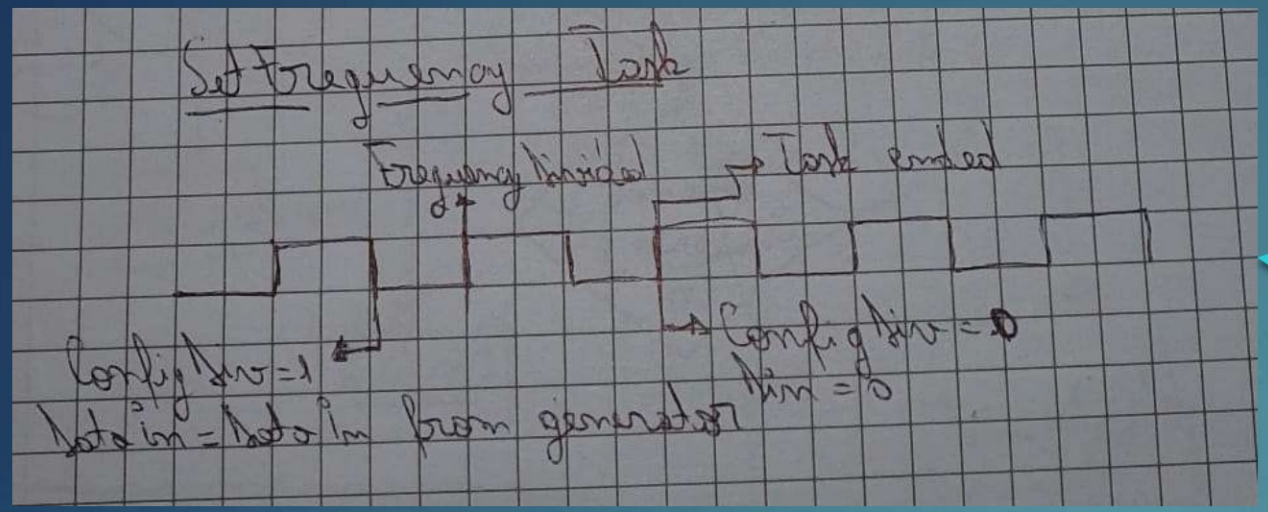
```
//make a constraint for Sel
constraint CorrectSel {
  Sel inside {[0:12]};
}
```




```
task Reset(input int Period);

  BC_itf.Reset <= 1'b0;

  @(posedge BC_itf.Clk);
  BC_itf.Reset <= 1'b1;
  BC_itf.ValidCmd <= 1'b0;
  BC_itf.InputKey <= 1'b0;
  BC_itf.RWMem <= 1'b0;
  BC_itf.InA <= 8'h00;
  BC_itf.InB <= 8'h00;
  BC_itf.Sel <= 4'h0;
  BC_itf.Addr <= 8'h00;
  BC_itf.ConfigDiv <= 1'b0;
  BC_itf.Datain <= 32'h0;

  repeat(Period) @(posedge BC_itf.Clk);
  BC_itf.Reset <= 1'b0;

  @(posedge BC_itf.Clk);

  $display("[DRV] -> Reset done.");

endtask
```

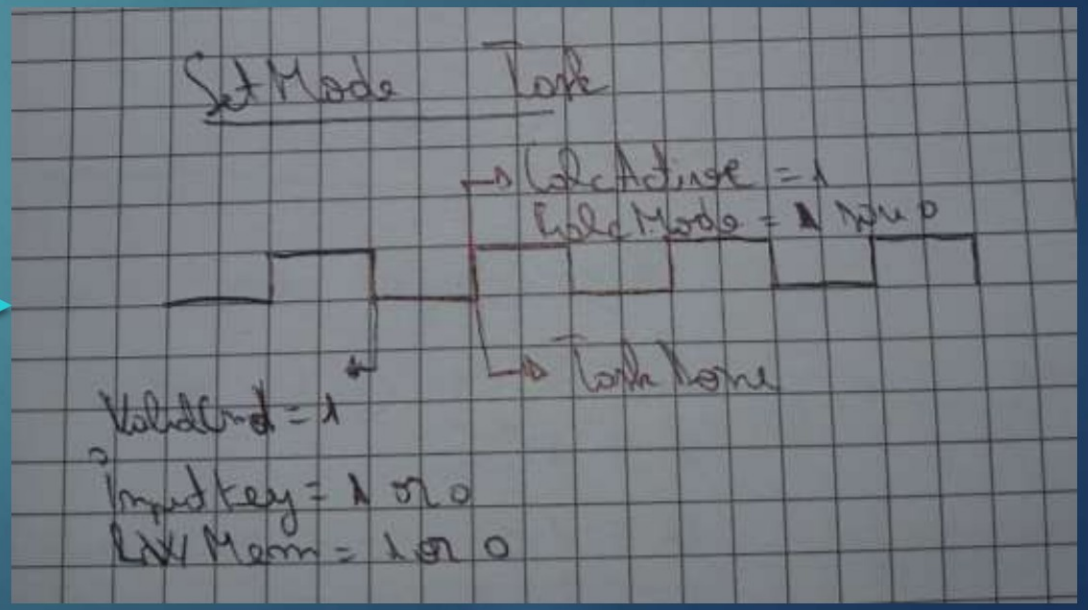
```
task InputKey();

  @(negedge BC_itf.Clk);
  BC_itf.InputKey <= 1'b1;
  BC_itf.ValidCmd <= 1'b1;

  @(negedge BC_itf.Clk);
  BC_itf.InputKey <= 1'b0;

  @(negedge BC_itf.Clk);
  BC_itf.InputKey <= 1'b1;

  @(negedge BC_itf.Clk);
  BC_itf.InputKey <= 1'b0;

  @(negedge BC_itf.Clk);
  BC_itf.ValidCmd <= 1'b0;

  $display("[DRV] -> InputKey set corectly.");

endtask
```






```
task SetFrequency();

  @(negedge BC_itf.Clk);
  BC_itf.ConfigDiv <= 1'b1;
  BC_itf.Datain <= data.Datain;

  repeat(2) @(posedge BC_itf.Clk);
  BC_itf.ConfigDiv <= 1'b0;
  BC_itf.Datain <= 0;

  $display("[DRV] -> Frequency divided with %0d.", data.Datain);

endtask
```

```
task SetMode(input bit Mode, input bit RWMem);

  @(negedge BC_itf.Clk);
  BC_itf.ValidCmd <= 1'b1;
  BC_itf.InputKey <= Mode;
  BC_itf.RWMem <= RWMem;

  $display("[DRV] -> Mode set to %0b", Mode);
  $display("[DRV] -> RWMem set to %0b", RWMem);

endtask
```

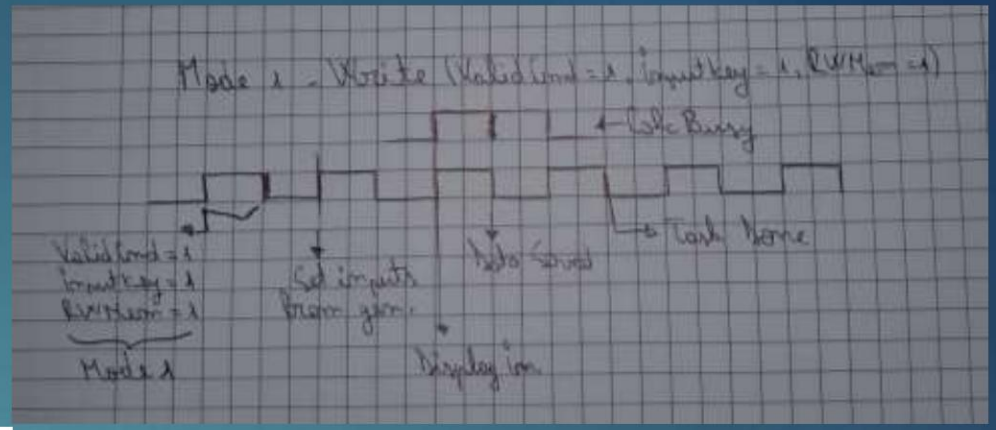


Mode1_Write

Driver

```
mbx.get(data);
SetMode(1, 1);
Mode1_Write();
```

```
task Mode1_Write();

  @(posedge BC_itf.Clk);
  BC_itf.InA <= data.InA;
  BC_itf.InB <= data.InB;
  BC_itf.Sel <= data.Sel;
  BC_itf.Addr <= data.Addr;

  wait(BC_itf.CalcBusy == 1'b1);
  data.display_inputs("DRV");

endtask
```



Monitor

```
wait(BC_itf.CalcBusy == 1'b1);
@(negedge BC_itf.Clk);

trans.InA = BC_itf.InA;
trans.InB = BC_itf.InB;
trans.Addr = BC_itf.Addr;
trans.Sel = BC_itf.Sel;
trans.Datain = BC_itf.Datain;
trans.InputKey = BC_itf.InputKey;
trans.ValidCmd = BC_itf.ValidCmd;
trans.RWMem = BC_itf.RWMem;
trans.ConfigDiv = BC_itf.ConfigDiv;
trans.Reset = BC_itf.Reset;
trans.DOutValid = BC_itf.DOutValid;
trans.DataOut = BC_itf.DataOut;
trans.ClkTx = BC_itf.ClkTx;
trans.CalcActive = BC_itf.CalcActive;
trans.CalcMode = BC_itf.CalcMode;
trans.CalcBusy = BC_itf.CalcBusy;

mbx.put(trans);
display_in();
$display("[MON] -> CalcBusy: %0h. Data Saved in memory.", trans.CalcBusy);

@(negedge BC_itf.Clk);
```

```
[GEN] -> InA: ef, InB: 3e, Sel: 6, Addr: de, Datain: 3.
[DRV] -> Mode set to 1
[DRV] -> RWMem set to 1
[DRV] -> InA: ef, InB: 3e, Sel: 6, Addr: de, Datain: 3.
[MON] -> InA: ef, InB: 3e, Sel: 6, Addr: de, Datain: 0.
[MON] -> CalcBusy: 1. Data Saved in memory.
[SCO] -> InA: ef, InB: 3e, Sel: 6, Addr: de, Datain: 0.
[SCO] -> Data: 62e3eef saved to Address: de.
--------------------------------
--------------------------------
[GEN] -> InA: 84, InB: 94, Sel: 4, Addr: d0, Datain: 3.
[DRV] -> Mode set to 1
[DRV] -> RWMem set to 1
[DRV] -> InA: 84, InB: 94, Sel: 4, Addr: d0, Datain: 3.
[MON] -> InA: 84, InB: 94, Sel: 4, Addr: d0, Datain: 0.
[MON] -> CalcBusy: 1. Data Saved in memory.
[SCO] -> InA: 84, InB: 94, Sel: 4, Addr: d0, Datain: 0.
[SCO] -> Data: 14009484 saved to Address: d0.
```

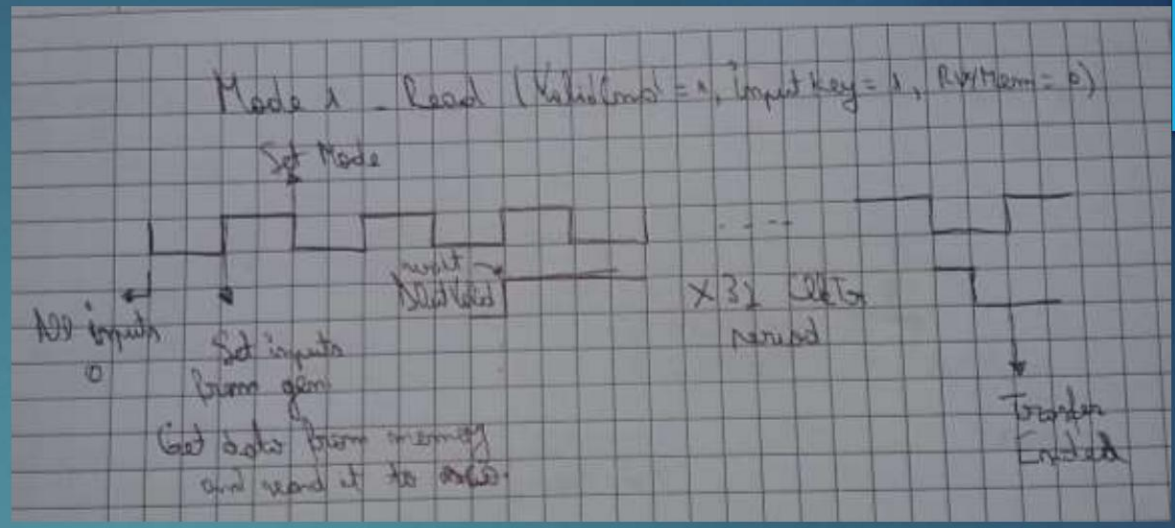
Scoreboard

```
      mbx.get(trans);
        memory_tmp[trans.Addr] = ConcatResult(trans.InA, trans.InB, trans.Sel);
        display_in();
        $display("[SCO] -> Data: %0h saved to Address: %0h", ConcatResult(trans.InA,
trans.InB, trans.Sel), trans.Addr);
```

Driver

```
        mbx.get(data);

        //set all inputs to 0
        BC_itf.ValidCmd <= 1'b0;
        BC_itf.InputKey <= 1'b0;
        BC_itf.RWMem <= 1'b0;
        BC_itf.InA <= 8'h00;
        BC_itf.InB <= 8'h00;
        BC_itf.Sel <= 4'h0;
        BC_itf.Addr <= 8'h00;

        //set random inputs
        @(posedge BC_itf.Clk);
        BC_itf.InA <= data.InA;
        BC_itf.InB <= data.InB;
        BC_itf.Sel <= data.Sel;
        BC_itf.Addr <= data.Addr;
        mbx_ds.put(memory_tmp[data.Addr]);

        //set the mode
        SetMode(1, 0);

        //wait for start the transfer
        wait(BC_itf.DOutValid == 1'b1);
        $display("[DRV] -> Transfer started! Data Sent to Serial:
%0h, from Addr: %0h.", memory_tmp[data.Addr], data.Addr);

        //wait for transfer to be done
        wait(BC_itf.DOutValid == 1'b0);

        //set all inputs to 0
        BC_itf.ValidCmd <= 1'b0;
        BC_itf.InputKey <= 1'b0;
        BC_itf.RWMem <= 1'b0;
        BC_itf.InA <= 8'h00;
        BC_itf.InB <= 8'h00;
        BC_itf.Sel <= 4'h0;
        BC_itf.Addr <= 8'h00;
```

Mode1_Read




```
[GEN] -> InA: e2, InB: 57, Sel: 4, Addr: 58, Datain: 3.
[DRV] -> Mode set to 1
[DRV] -> RWMem set to 0
[DRV] -> Transfer started! Data Sent to Serial: 0, from Addr: 58.
[MON] -> Transfer ended! Data Receaved from Serial: 0.
[SCO] -> DRV : 0, MON : 0.
[SCO] -> DATA MATCHED!
```

Monitor

```
wait(BC_itf.DOutValid == 1'b1);

for (int i = 0; i <= 31; i++) begin
  @(negedge BC_itf.ClkTx);
  SerialData[31 - i] = BC_itf.DataOut;
end

wait(BC_itf.DOutValid == 1'b0);
@(posedge BC_itf.ClkTx);

$display("[MON] -> Transfer ended! Data Receaved from Serial: %0h.", SerialData);
mbx_ms.put(SerialData);
```

Scoreboard

```
mbx_ds.get(ConcatData);
mbx_ms.get(Data_ms);

$display("[SCO] -> DRV : %0h, MON : %0h.", ConcatData, Data_ms);

if(ConcatData == Data_ms)
  $display("[SCO] -> DATA MATCHED!");
else
  $display("[SCO] -> DATA MISMATCHED!");
```

Mode0

Driver

```
mbx.get(data);
//set all inputs to 0
BC_itf.ValidCmd <= 1'b0;
BC_itf.InputKey <= 1'b0;
BC_itf.RWMem <= 1'b0;
BC_itf.InA <= 8'h00;
BC_itf.InB <= 8'h00;
BC_itf.Sel <= 4'h0;
BC_itf.Addr <= 8'h00;

//set random inputs
@(posedge BC_itf.Clk);
BC_itf.InA <= data.InA;
BC_itf.InB <= data.InB;
BC_itf.Sel <= data.Sel;
BC_itf.Addr <= data.Addr;
mbx_ds.put(ConcatResult(data.InA, data.InB, data.Sel));

//set the mode
SetMode(0, 0);
data.display_inputs("DRV");

//wait to start the transfer
wait(BC_itf.DOutValid == 1'b1);
$display("[DRV] -> Transfer started! Data Sent to Serial: %0h.",
ConcatResult(data.InA, data.InB, data.Sel));

//wait for transfer to be done
wait(BC_itf.DOutValid == 1'b0);

//set all inputs to 0
BC_itf.ValidCmd <= 1'b0;
BC_itf.InputKey <= 1'b0;
BC_itf.RWMem <= 1'b0;
BC_itf.InA <= 8'h00;
BC_itf.InB <= 8'h00;
BC_itf.Sel <= 4'h0;
BC_itf.Addr <= 8'h00;
```

```
[DRV] -> Reset done.
[DRV] -> Frequency divided with 5.
[DRV] -> InputKey set corectly.
---------------------------------
[GEN] -> InA: 1f, InB: 8, Sel: 4, Addr: 8b, Datain: 5.
[DRV] -> Mode set to 0
[DRV] -> RWMem set to 0
[DRV] -> InA: 1f, InB: 8, Sel: 4, Addr: 8b, Datain: 5.
[DRV] -> Transfer started! Data Sent to Serial: 1400081f.
[MON] -> Transfer ended! Data Recived from Serial: 1400081f.
[SCO] -> DRV : 1400081f, MON : 1400081f.
[SCO] -> DATA MATCHED!
---------------------------------
```

Monitor

```
wait(BC_itf.DOutValid == 1'b1);

for (int i = 0; i <= 31; i++) begin
  @(negedge BC_itf.ClkTx);
  SerialData[31 - i] = BC_itf.DataOut;
end

wait(BC_itf.DOutValid == 1'b0);
@(posedge BC_itf.ClkTx);

$display("[MON] -> Transfer ended! Data Recived from Serial: %0h.", SerialData);
mbx_ms.put(SerialData);
```

Scoreboard

```
mbx_ds.get(ConcatData);
mbx_ms.get(Data_ms);

$display("[SCO] -> DRV : %0h, MON : %0h.", ConcatData, Data_ms);

if(ConcatData == Data_ms)
  $display("[SCO] -> DATA MATCHED!");
else
  $display("[SCO] -> DATA MISMATCHED!");
```

Thank you!