# COMPUTER ARCHITECTURE PROJECT

**GitHub repo link** : https://github.com/Hossam-Sayed/architecture-project

**Video Link** : https://drive.google.com/file/d/1MPUDwgiO5n9kK0rlBJ--ZrWg051jRRBN/view

| Name | ID |
|---|---|
| Abdelrahman Mohamed Mohamed Ibrahim | 1900922 |
| Omar Mohamed Ibrahim Bayoumi | 1900409 |
| Omar Mohamed Kholeif Ahmed | 1900957 |
| Hossam Sayed Nasr Ibrahim | 1900883 |
| Amr Atef Ahmed Hessain | 1900430 |
| Mustafa Mahmoud Ahmed el sayed | 1900475 |

## Table of Contents

## Table of figures

# The Advanced Peripheral Bus (APB Bus)

## Module Diagram

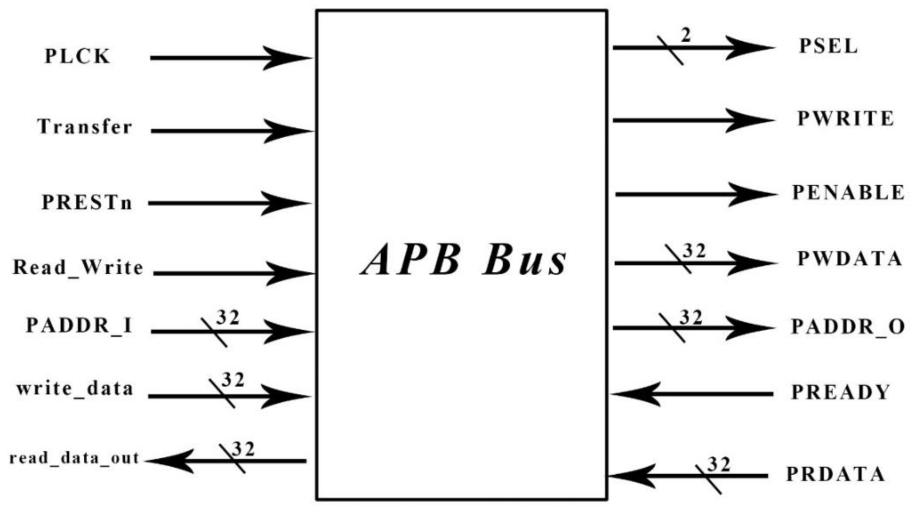


*Figure 1 : APB Diagram*

## Module Description

I. Inputs

| Signal | Description |
|---|---|
| PCLK | The input Clock. The rising edge of PCLK times all transfers on the APB |
| Transfer | APB enable signal. If high APB is activated else APB is disabled |
| PRESTn | The reset signal , An active low signal . |
| Read_Write | Input signal indicate the type of transfer , if high is write , low is read |
| PADDR_I | The APB input address bus |
| Write_data | The input data want to be written |
| PREADY | Form slave , it`s used to enter the access state |
| PRDATA | From slave , The selected slave drives this bus during read operation |

## II. Outputs

| Signal | Description |
|---|---|
| read_out_data | The output Data from APB to master (Read Access) |
| PSEL | The selection signal , There is a PSEL for each slave. It’s an active high signal. |
| PWRITE | The Data transfer direction , if high indicates master to slave (write access) , if low indicates slave to master (read access) |
| PENABLE | It indicates the 2nd cycle of a data transfer. It’s an active high signal. |
| PWDATA | Write data. The data want to be written during write cycles when PWRITE is high |
| PADDR_O | The ouput address form APB to slave to access the right location |

## III. Finite state machine

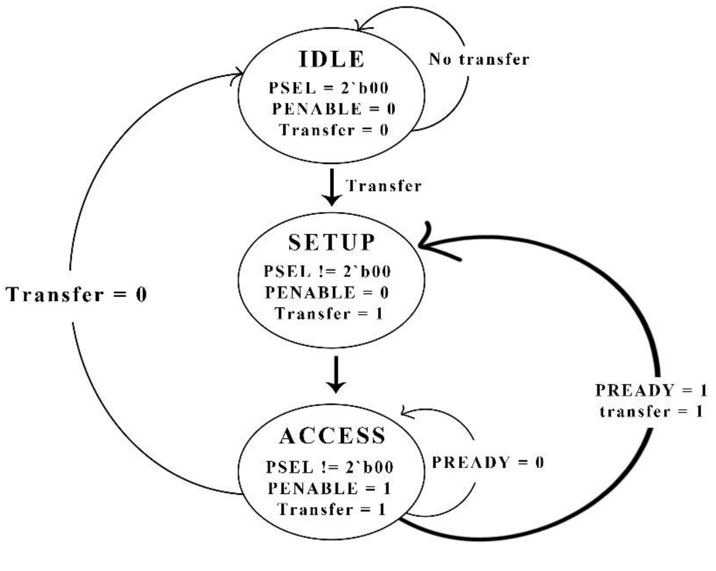


*Figure 2: Finite state machine for APB Bus*

## IV. Brief description

APB is low bandwidth and low performance bus. So, the components requiring lower bandwidth like the peripheral devices such as UART, Keypad and GPIO (General purpose input output) devices are connected to the APB .

## V. Test benches

- Write operation

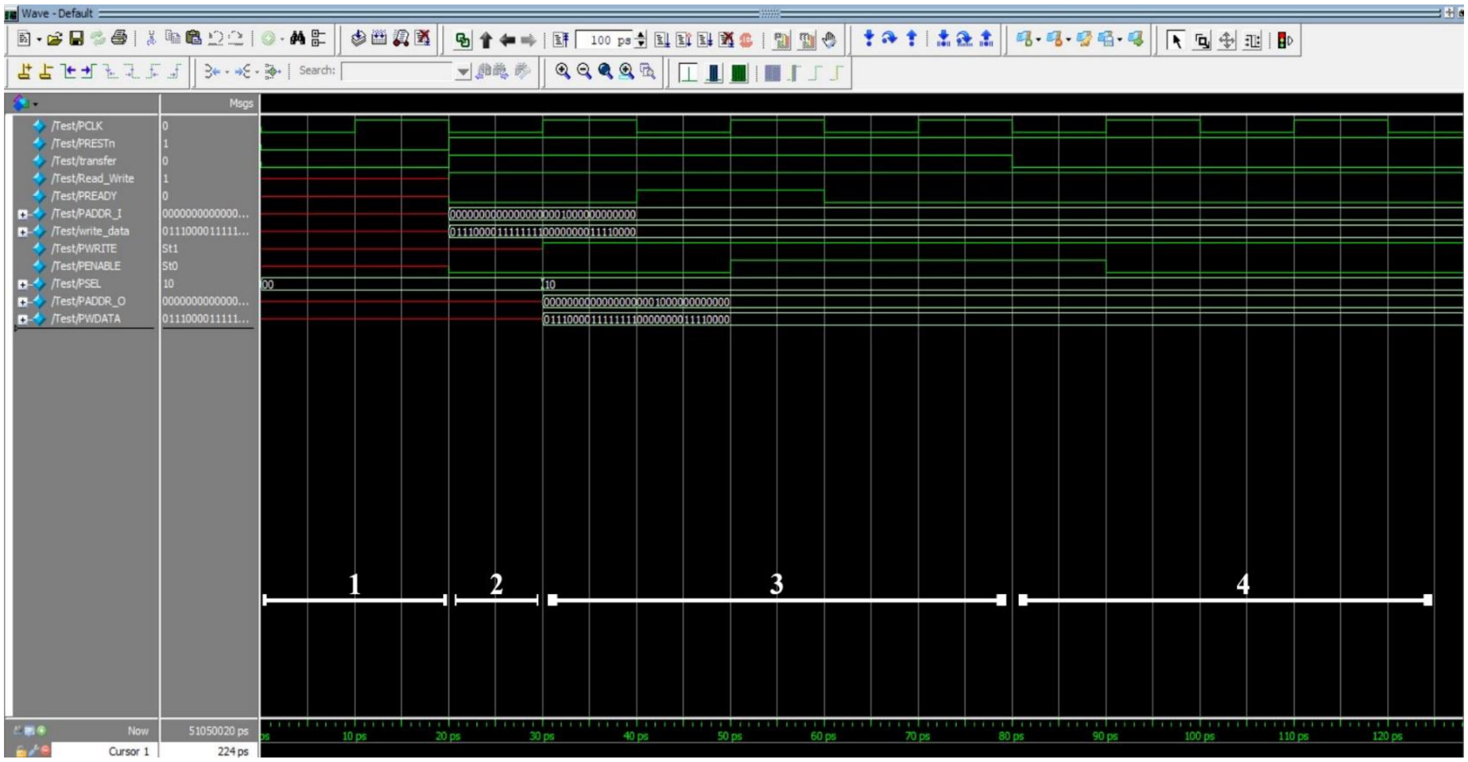


*Figure 3: APB bus Write operation*

```
always
begin
#10
PCLK = ~PCLK;
end
initial
begin
PCLK = 0 ; PRESTn = 0 ; transfer = 0 ;
@(negedge PCLK)
```

```
PRESTn = 1 ; transfer = 1 ; Read_Write=1 ; PREADY = 0 ;

PADDR_I = 31'h1000 ; write_data = 31'hF0FF00F0;

@(negedge PCLK)

PRESTn = 1 ; transfer = 1 ; Read_Write=1 ; PREADY = 1 ;

PADDR_I = 31'h1000 ; write_data = 31'hF0FF00F0;

@(negedge PCLK)

PRESTn = 1 ; transfer = 1 ; Read_Write=1 ; PREADY = 0 ;

PADDR_I = 31'h1000 ; write_data = 31'hF0FF00F0;

@(negedge PCLK)

PRESTn = 1 ; transfer = 0 ; Read_Write=1 ; PREADY = 0 ;

end
```

1- First **PRESTn** and **transfer** signals are low , so the APB in **idle** state which characterized by **PSEL** signal = 00 and other signals are don't cares .

2- When **PRESTn** and **transfer** signals are high , so the APB moves form **idle** to **setup** state which characterized by **PREADY** = 0 and will receive form master (Ex. **CPU**) the slave's address (Ex. *PADDR_I = 31'h1000*), data want to be written in slave (*Ex. write_data = 31'hF0FF00F0* ) and **Read_Write** signal is high to write access .

3- When rising edge and, **PSEL** will take specific value to select which of slaves is selected (*Ex. PSEL = 2'10 to access GPIO module* ) , **PADDR_O** will take the input address to access the correct location (Ex. PADDR_O = *31'h1000* ) and **Write_date** will take the input data wanted to be written (Ex. *31'hF0FF00F0* ) . when **PREADY** and **PENABLE** are high the apb moves from **setup** state to **access** state and will take the address and data form bus to slave .

4- When **transfer** is low which that master end the transaction , the APB bus moves to **idle** state again in next rising edge

- Read operation

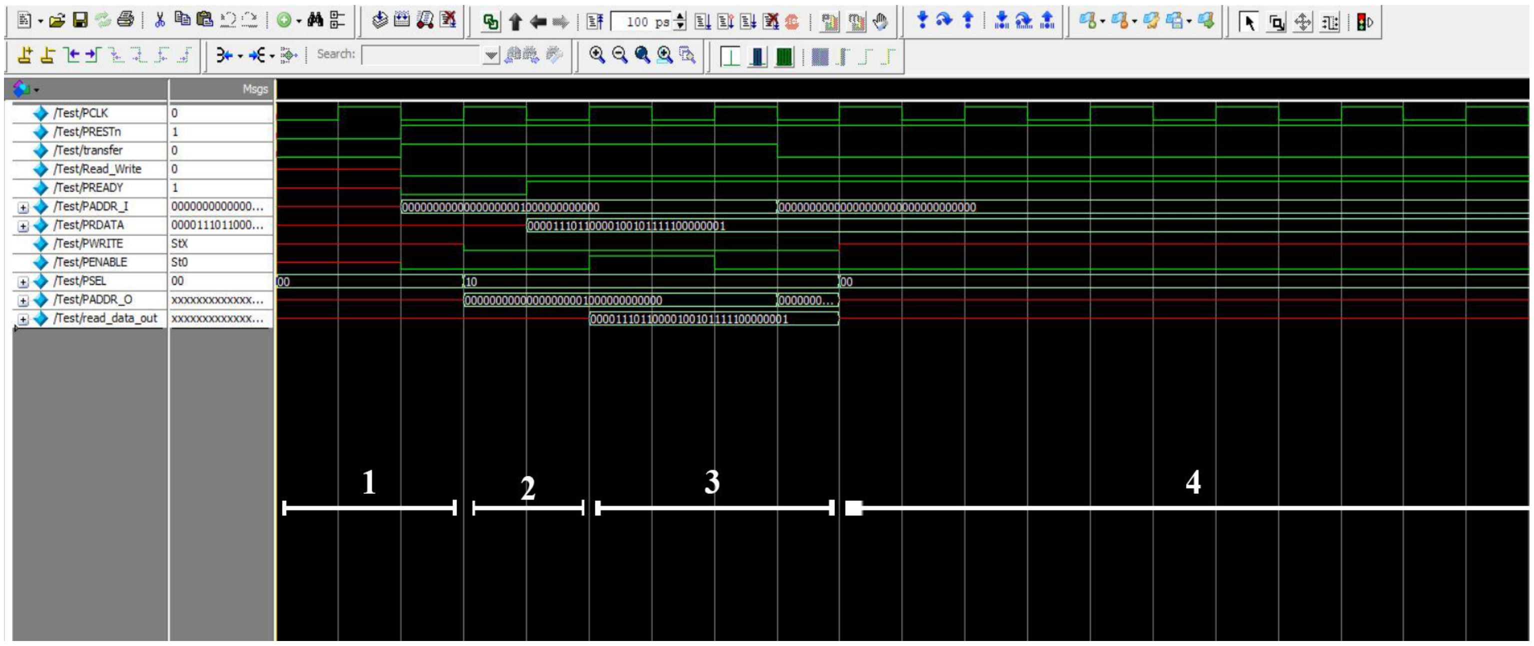

```
/Test/PCLK           0
/Test/PRESTn         1
/Test/transfer       0
/Test/Read_Write     0
/Test/PREADY         1
/Test/PADDR_I        0000000000000...
/Test/PRDATA         0000111011000...
/Test/PWRITE         StX
/Test/PENABLE        St0
/Test/PSEL           00
/Test/PADDR_O        xxxxxxxxxxxxx...
/Test/read_data_out  xxxxxxxxxxxxx...
```

*Figure 4: APB bus read operation*

```
always
begin
#10
PCLK = ~PCLK;
end
initial
begin
PCLK = 0 ; PRESTn = 0 ; transfer = 0 ;
@(negedge PCLK)
PRESTn = 1 ; transfer = 1 ; Read_Write=0 ; PREADY = 0 ;
PADDR_I = 31'h1000 ;
@(negedge PCLK)
PRESTn = 1 ; transfer = 1 ; Read_Write=0 ; PREADY = 1 ;
PADDR_I = 31'h1000 ; PRDATA = 32'h0EC25F01;
@(negedge PCLK)
```

```
PRESTn = 1 ; transfer = 1 ; Read_Write=0 ; PREADY = 1 ;

PADDR_I = 31'h1000 ; PRDATA = 32'h0EC25F01;

@(negedge PCLK)

PADDR_I = 31'h00 ; transfer = 0 ;

end
```

1- First **PRESTn** and **transfer** signals are low , so the APB in **idle** state which characterized by **PSEL** signal = 00 and other signals are don't cares .
2- When **PRESTn** and **transfer** signals are high , so the APB moves form **idle** to **setup** state which characterized by **PREADY** = 0 and will receive form master (Ex. **CPU**) the slave's address (Ex. *PADDR_I = 31'h1000*), at next rising edge **PSEL** will take specific value to select which of slaves is selected (*Ex. PSEL = 2'10 to access GPIO module* ), and **Read_Write** signal is **low** to read access .
3- When **PREADY is** high , the slave will put the data stored at location *31'h1000 in **PRDATA** bus* , at next rising edge , when **ENABLE** is high **read_data_out** will take the input data (Ex. *Read_out_data = 32'h0EC25F01*) .
4- When **transfer** is low which that master end the transaction , the APB bus moves to **idle** state again in next rising edge which characterized by **PSEL** signal = 00 and other signals are don't cares .

# UART Module

## Block Diagram

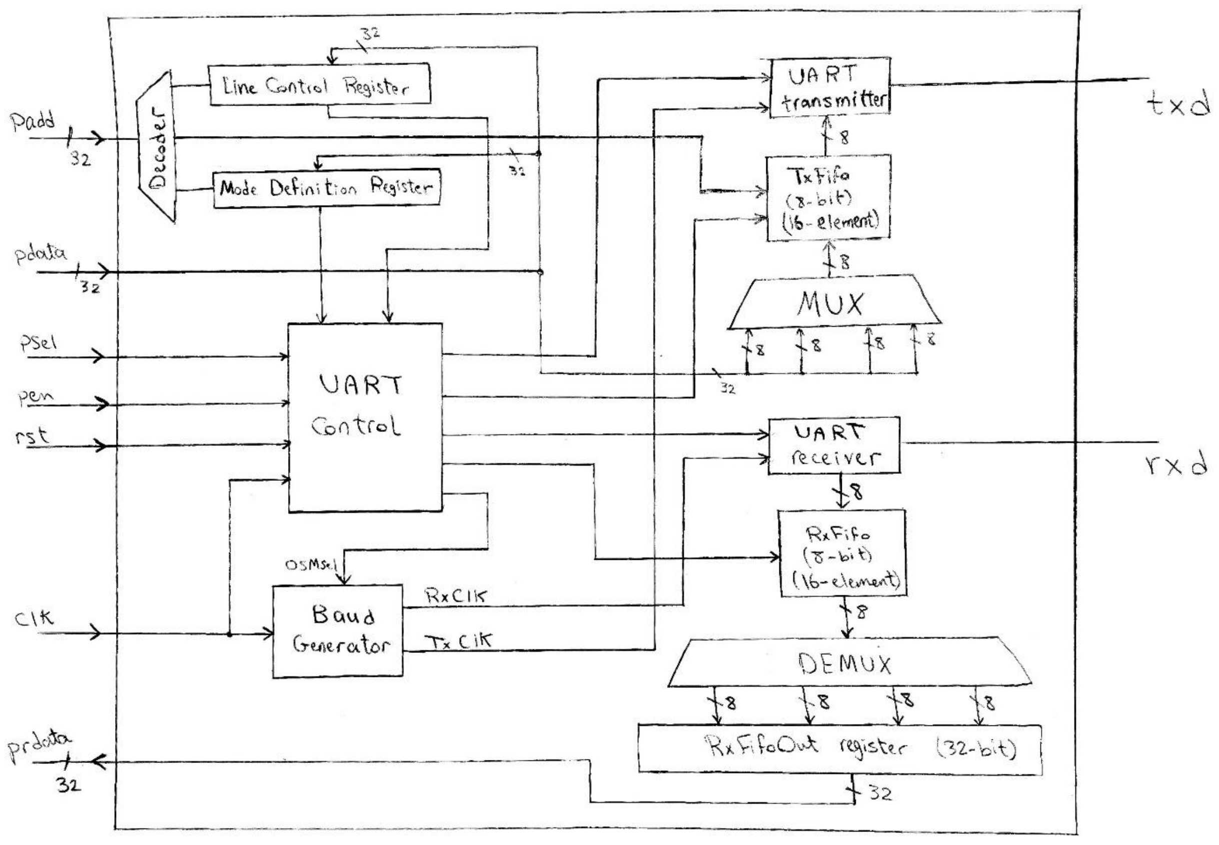


*Figure 5:Uart Module diagram*

## Module signals and parameters description

I. Input parameters

| Parameter | Description |
|---|---|
| ***LCR_ADDRESS*** | Address of the Line Control Register |
| ***MDR_ADDRESS*** | Address of the Mode Definition Register |
| ***TX_FIFO_ADDRESS*** | Address of the transmitter FIFO element |

## II. Inputs

| Signal | Description |
|---|---|
| $\boldsymbol{padd}$ | The input address to the UART module. |
| $\boldsymbol{pdata}$ | Input 32-bit parallel data to the UART module to be added to the control registers ($LCR, MDR$) or the transmitter FIFO. |
| $\boldsymbol{psel}$ | When set, it means that the APB bus chose the UART to start a transaction (active high). |
| $\boldsymbol{pen}$ | This signal indicates the second clock cycle of an APB transfer which means the data is valid or APB is ready to receive the data (active high). |
| $\boldsymbol{pwr}$ | Indicates write or read operations (active high for a write transaction). |
| $\boldsymbol{rst}$ | A reset signal (active high), resets the whole UART operations. |
| $\boldsymbol{clk}$ | CPU clock. |
| $\boldsymbol{rxSerial}$ | The serial input received by the $Rx$ and goes to the $Rx\ FIFO$. |

## III. Outputs

| Signal | Description |
|---|---|
| $\boldsymbol{prdata}$ | Output 32-bit parallel data on the APB bus from the $Rx\ FIFO$. |
| $\boldsymbol{pready}$ | Is set by the UART to tell the APB bus that it’s ready for a transaction. |
| $\boldsymbol{txSerial}$ | The serial output of the $Tx$. |

## IV. Brief description

UART module connects the UART transmitter, UART receiver, baud generator and control registers together, and controls the data flow among them. It also interfaces the APB bus and makes transactions with.

## V. Detailed description

The UART module contains 7 always blocks which will be described in detail as follows

1. The first always block handles the reset signal ($\boldsymbol{rst}$). It gets triggered with the positive edge of the reset signal. When it gets triggered, the UART empty the transmitter and the receiver FIFOs, resets the control registers ($\boldsymbol{LCR}$ and $\boldsymbol{MDR}$), resets the FIFOs read and write indices, brings the $\boldsymbol{r\_rx\_done}$ and $\boldsymbol{rxDataReady}$ signals low, and aborts any running transactions. This ensures a fresh start for the UART module.

2. The second always block handles the control registers operations. It gets triggered with the positive clock edge. It loads the control register bits values into their corresponding transmitter and receiver input signals only if the transmitter or receiver are not busy (idle) to avoid changing their mode during an active transaction. It loads the $\boldsymbol{OSM\_SEL}$ signal ($\boldsymbol{MDR[0]}$) into the $\boldsymbol{osmSel}$ receiver input signal, loads parity enable signal ($\boldsymbol{LCR[3]}$) and the even parity select signal ($\boldsymbol{LDR[4]}$) into the $\boldsymbol{pen}$ and the $\boldsymbol{eps}$ signals of $Tx$ and $Rx$ respectively.

3. The third always block handles the read and write transactions with the APB bus. It gets triggered with the negative edge of the clock. It checks if there was a recent transaction in the previous clock cycle, and the $\boldsymbol{pready}$ signal is already high, it sets the $\boldsymbol{pready}$ signal to low. It checks if $\boldsymbol{psel}$ is high meaning that the APB bus chose the UART to start a transaction with it. If $\boldsymbol{psel}$ is high, then it checks if the pen is also high meaning that the UART is ready to start the transaction. If $\boldsymbol{psel}$ is also high, it checks if the transaction is a write transaction to the UART or a read transaction from the UART using $\boldsymbol{pwr}$ signal. If the transaction was a write transaction, then it checks the $\boldsymbol{padd}$ signal for the address to write the data in. According to the address, the data is written in the LCR, MDR or the $Tx\ FIFO$ buffer. If it's a read transaction and there is available data ($rxDataReady = 1$), then the first four bytes (32 bits) in the $Rx\ FIFO$ are loaded in the $\boldsymbol{prdata}$ output of the UART to the APB. The data won't be written to the $Tx\ FIFO$ if it was full. During any transaction, the $\boldsymbol{pready}$ is set to high.

4. The fourth always block handles the read and write operations with the $Rx\ FIFO$. It gets triggered with the positive clock edge. It checks if there is a complete word (32-bits) in the $Rx\ FIFO$, if so, it loads the first word into the $\boldsymbol{rxFifoOut}$ register, increment the $\boldsymbol{rxFifoReadIndex}$ by four to point to the next word in the $Rx\ FIFO$, mark the $Rx\ FIFO$ as not full by resetting the $\boldsymbol{rxFifoFull}$ flag zero, and finally mark the $Rx\ FIFO$ as ready to send data (loaded word in $\boldsymbol{rxFifoOut}$) by setting the $\boldsymbol{rxDataReady}$ flag to one. If there is no at least one complete word in the $Rx\ FIFO$, it checks if the $Rx\ FIFO$ isn't full and a new byte has been received from the receiver, if so, it saves the newly received byte to the $Rx\ FIFO$ in the $\boldsymbol{rxFifoWriteIndex}$, increment the $\boldsymbol{rxFifoWriteIndex}$ by one to point to the next byte location in the $Rx\ FIFO$, reset the $\boldsymbol{r\_rx\_done}$ to zero, and mark the $Rx\ FIFO$ as not empty by resetting the $\boldsymbol{rxFifoEmpty}$ flag to zero. It marks the $Rx\ FIFO$ as full if there was a space for only one byte in it as it will be taken by the current write to the $Rx\ FIFO$. If new bytes came while the $Rx\ FIFO$ was full, they are discarded.

5. The fifth always block handles setting the flags after receiving a new byte from the Rx. It gets triggered with the positive $\boldsymbol{rxDone}$ edge. If a new byte arrived form the Rx, it sets the $\boldsymbol{r\_rx\_done}$ to one as the Rx FIFO now has newly arrived data, and loading the newly received byte into the $\boldsymbol{r\_rx\_byte}$.

6. The sixth always block handles the write operations to the $Tx\ FIFO$. It gets triggered with the positive clock edge. It checks if the $Tx$ receives a start signal to receive one word, if so, it loads the input word into the $Tx\ FIFO$, increment the $\boldsymbol{txFifoWriteIndex}$ by four to point to the next word in the $Tx\ FIFO$, mark the $Tx\ FIFO$ as not empty by resetting $\boldsymbol{txFifoEmpty}$ flag to zero, reset the $\boldsymbol{txFifoStart}$ and $\boldsymbol{txStart}$ flags to zero.

7. The seventh always block handles the read operations to the $Tx\ FIFO$. It gets triggered with the negative transmitter's clock edge, which is slower than the CPU clock. It checks if the $Tx\ FIFO$ is empty ($txFifoEmpty\ =\ 0$) and the $Tx$ is not busy ($txBusy\ =\ 0$), if so, it sets $\boldsymbol{txStart}$ to one and sends a byte to the $Tx$ module, loads a byte from the $Tx\ FIFO$ into $\boldsymbol{txByte}$ register, increments the $\boldsymbol{txFifoReadIndex}$ by one to point to the next byte in the Tx FIFO, and mark the Tx FIFO as not full by resetting the $\boldsymbol{txFifoFull}$ flag to zero. It checks if the last byte in the $Tx\ FIFO$ is taken, mark the $Tx\ FIFO$ as empty.

## I. Test bench output

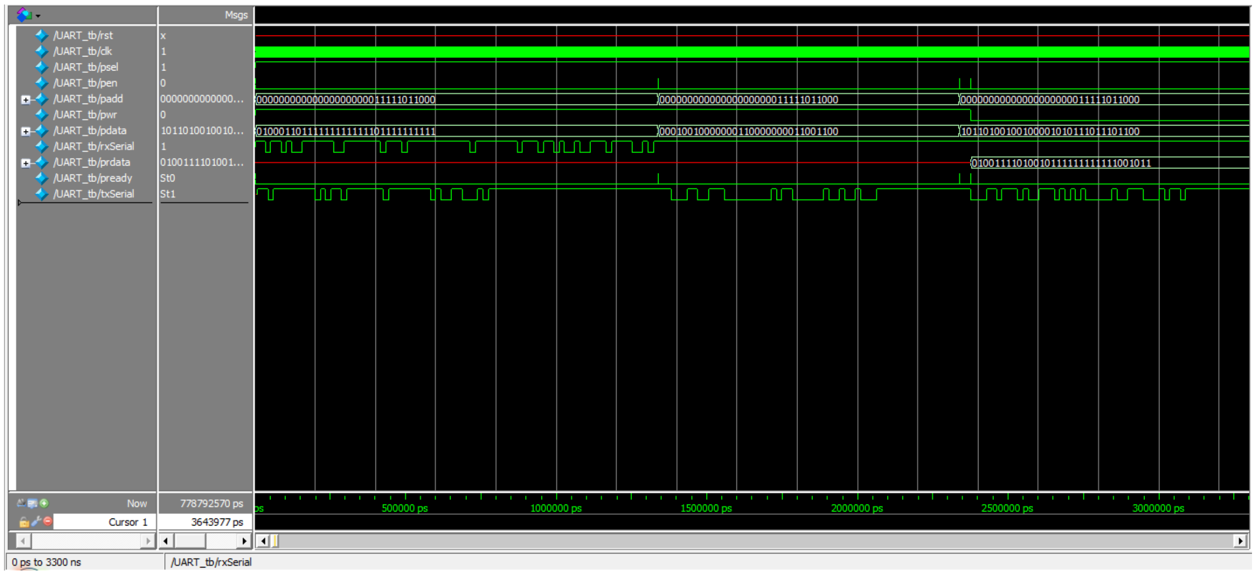


*Figure 6:UART transmit and receive*

At first, the $padd$ signal targets the Line Control Register address (LCR) to allow for even parity bit ($eps\ =\ 1, pen\ =\ 1$). Then, it targets the Mode Definition Register (MDR) to set the sampling mode to 16x ($osmSel\ =\ 0$). After that both receiver and transmitter are tested in parallel.

**Receiver**

The following data bits are received on the …RX serial…. With even parity and 16x sampling mode

| Serial input (left first) | Start bit | Data bits | Parity bit | Stop bit | Validity |
|---|---|---|---|---|---|
| 0 1 1 0 1 0 0 1 1 1 1 | 0 | 1100 1011 | 1 | 1 | Valid |
| 0 0 1 1 1 1 1 1 1 0 1 | 0 | 1111 1110 | 0 | 1 | Not valid |
| 0 1 1 1 1 1 1 1 1 1 1 | 0 | 1111 1111 | 1 | 1 | Not valid |
| 0 1 1 1 1 1 1 1 1 0 1 | 0 | 1111 1111 | 0 | 1 | Valid |

| | | | | | |
|---|---|---|---|---|---|
| 0 1 1 0 1 0 0 1 0 0 1 | 0 | 0100 1011 | 0 | 1 | Valid |
| 0 1 1 1 1 0 1 0 0 1 1 | 0 | 0010 1111 | 1 | 1 | Valid |

Note: The data on the serial starts with start bit followed by least significant bit is until the most significant bit, parity bit then stop bit. So the Data bits is extracted from serial input then inverted.

The validity of the start, parity and stop bits determines the validity of the data received. Only the valid data bits are sent to the $Rx\ FIFO$ and stored until requested by the APB bus. When the APB bus sent a read request ($pwr = 0$), four bytes are loaded from the $Rx\ FIFO$ and sent over the $\boldsymbol{prdata}$.

**Transmitter**

When $padd$ signal targets the $TxFIFO$ to write 32-bit parallel input byte-by-byte, least significant byte is stored first. And the $Tx$ starts transmission as soon as the $Tx$ is not busy.

The transmitter is tested in 3 modes, with even parity, with no parity and the third one with

At first 32-bit input parallel input is transmitted with even parity bit (initial state), then after the transmission the $padd$ targets the LCR and disables the parity bit ($eps = x, pen\ = 1$), then targets the $TxFIFO$ again with new 32-bit parallel input. Finally, it targets the LCR again and enable parity bit with odd parity ($eps\ = \ 0, pen\ = \ 1$), then targets the $TxFIFO$ with the last 32-bit input.

| **32-bit input** | **Data on serial (left first)** |
|---|---|
| 01000110 11111111 11111011 11111111 | 0 1 1 1 1 1 1 1 1 0 1<br>0 1 1 0 1 1 1 1 1 1 1<br>0 1 1 1 1 1 1 1 1 0 1<br>0 0 1 1 0 0 0 1 0 1 1 |
| 00010010 00000011 00000000 11001100 | 0 0 0 1 1 0 0 1 1 1<br>0 0 0 0 0 0 0 0 0 1<br>0 1 1 0 0 0 0 0 0 1<br>0 0 1 0 0 1 0 0 0 1 |
| 10110100 10010000 10101110 11101100 | 0 0 0 1 1 0 1 1 1 0 1<br>0 0 1 1 1 0 1 0 1 0 1<br>0 0 0 0 0 1 0 0 1 1 1<br>0 0 0 1 0 1 1 0 1 1 1 |

It takes the bytes stored in the $TxFIFO$ (least significant byte first) and adds a start bit at the beginning, and a parity bit (if enabled) and a stop bit at the end. And start sending the 8 bits bit by bit with the least significant bit first.

# Baud Generator Module

## Module signals and parameters description

### I. Input parameters

| Parameter | Description |
|---|---|
| $CLOCK\_RATE$ | input clock frequency (default 100MHZ) |
| $BAUD\_RATE$ | output baud rate of UART clocks (default 115200) |

### II. Inputs

| Signal | Description |
|---|---|
| $Clk$ | is the CPU clock |
| $osmSel$ | Over Sampling Mode Select to choose from 13x and 16x oversampling modes |

### III. Outputs

| Signal | Description |
|---|---|
| $rxClk$ | Receiver clock |
| $txClk$ | Transmitter clock |

### IV. Brief Description

Baud rate generator takes the CPU clock as input and generates two clock signals with different frequencies one for the UART transmitter and the other for the UART receiver. The frequency of the transmitter clock equals the baud rate (2nd parameter). The receiver clock is faster than the transmitter's clock by a factor (13x or 16x) depending on the $osmSel$ input signal.

### V. Detailed Description

Main local parameters

a. $MAX\_RATE\_TX$: A computed constant that represents the maximum CPU clock counts in each half cycle of $TX$ clock.

$$MAX\_RATE\_TX = \frac{CPU\ Clock\ Rate}{2 \times Baud\ Rate}$$

b. $MAX\_RATE\_13x / MAX\_RATE\_16x$: Computed constants that represent the maximum CPU clock counts in each half cycle in $RX$ clock while sampling in 13x/16x mode

$$MAX\_RATE\_13x = \frac{CPU\ Clock\ Rate}{2 \times Baud\ Rate \times 13}$$

$$MAX\_RATE\_16x = \frac{CPU\ Clock\ Rate}{2 \times Baud\ Rate \times 16}$$

$txCounter/rxCounter$: Two counters (registers) defined and initialized to 0 (one for $Tx$ and the other for $Rx$).

At first $rxClk$ and $txClk$ are initialized to 0 in an initial block.

At every positive CPU clock edge

- The $osmSel$ is checked to determine the used $MAX\_RATE$ for $Rx$. If $osmSel = 0$, the $MAX\_RATE\_16x$ is used which is the smaller, producing faster clock for sampling. If $osmSel = 1$, the $MAX\_RATE\_13x$ is used which is the greater, producing slower clock.
- The $rxCounter$ is incremented until it reaches the $MAX\_RATE\_13x/MAX\_RATE\_16x$, this means that it reached half the $Rx$ clock cycle, so we negate the $rxClk$ and reset the counter to zero to start counting till the next half cycle.
- The $txCounter$ is incremented until it reaches the $MAX\_RATE\_TX$, this means that it reached half the $Tx$ clock cycle, so we negate the $txClk$ and reset the counter to zero to start counting till the next half cycle.

## VI. Test bench Output

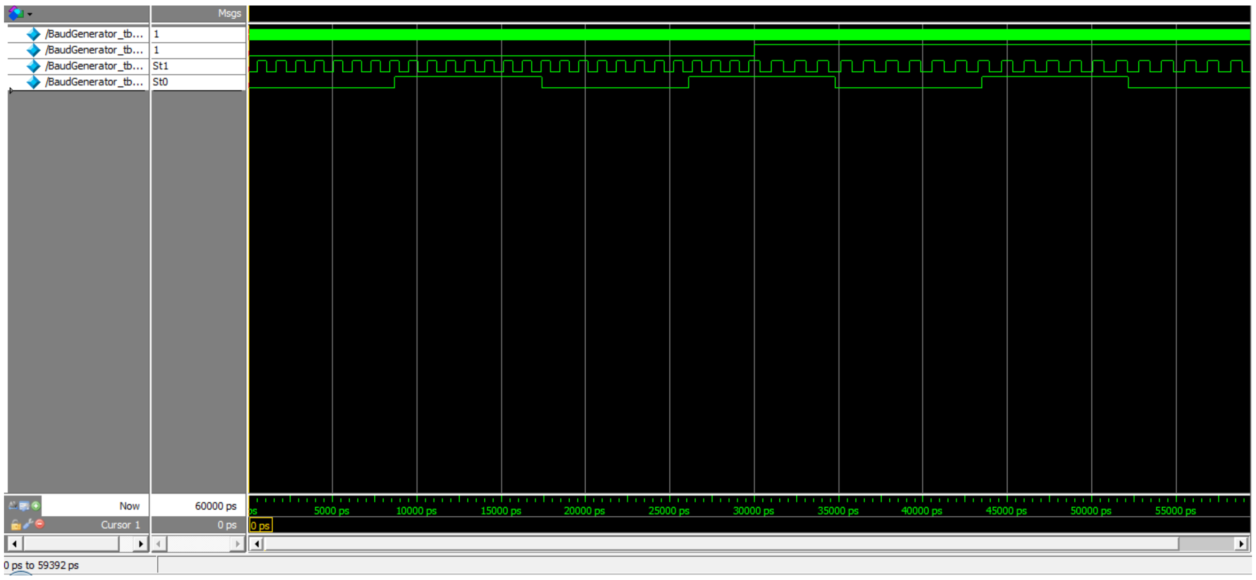


*Figure 7:Baud Generator Testbench*

In baud rate generator test bench output image above, we start by $osmSel = 0$ (16x over sampling mode) which is the faster $Rx$ clock, then toggles the $osmSel$ signal to 1 (13x oversampling mode) which is a slower $Rx$ clock. The $Tx$ clock rate doesn't depend on the $osmSel$ signal (over sampling mode) so it is constant in both cases and equals the baud rate.

# UART Transmitter Module

## Module Finite State Machine

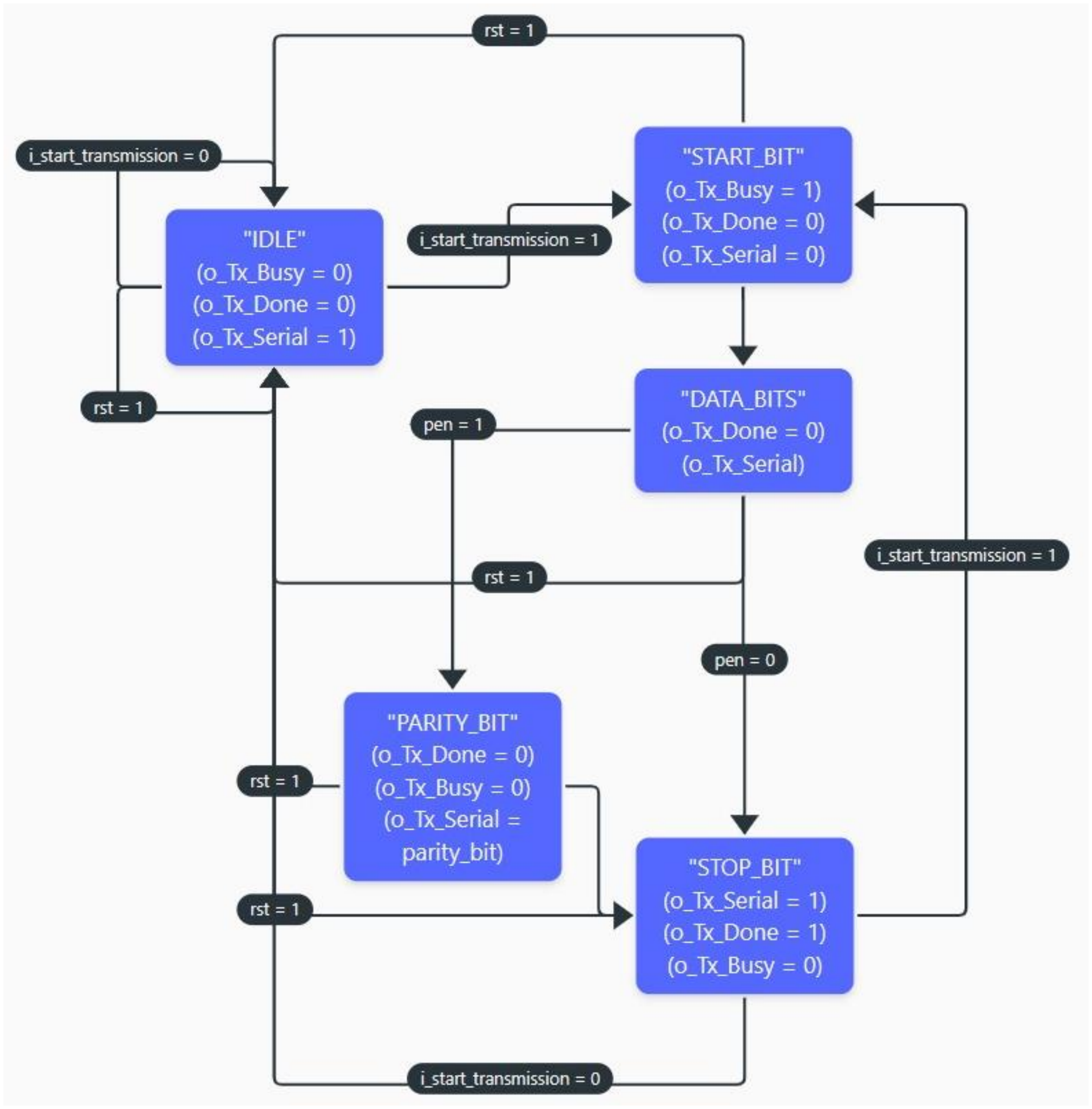


*Figure 8: UART Transmitter FSM*

## Module Inputs and Outputs Description

### I. Inputs

| Signal | Description |
|---|---|
| $\boldsymbol{i\_Clock}$ | The input clock from the baud generator |
| $\boldsymbol{eps}$ | Output baud rate of UART clocks (default 115200) |
| $\boldsymbol{pen}$ | Parity Enable signal which enables/disables the parity bit (active high) |
| $\boldsymbol{i\_start\_transmission}$ | Signal to tell the transmitter to start transmitting a byte from parallel input |
| $\boldsymbol{i\_Tx\_Byte}$ | The byte to be transmitted by the transmitter to the serial output |

### II. Outputs

| Signal | Description |
|---|---|
| $\boldsymbol{o\_Tx\_Busy}$ | A Signal to tell whether the transmitter is currently busy or not (active low) |
| $\boldsymbol{o\_Tx\_Serial}$ | The serial output of the transmitter |
| $\boldsymbol{o\_Tx\_Done}$ | Signal to tell whether the transmitter has finished sending a complete byte on the serial output or not (active high) |

### III. Brief Description

UART transmitter converts parallel input data from its FIFO into serial output data on its serial output terminal after adding a zero (start bit), followed by the 8 data bits, a parity bit (if enabled) and finally a one (stop bit).

### IV. Detailed description

Main local registers

a. $\boldsymbol{r\_Bit\_Index}$: Is used to access the data byte bit-by-bit.
b. $\boldsymbol{r\_Tx\_Byte}$: The parallel 8-bit data loaded from the FIFO.
c. $\boldsymbol{r\_Tx\_Done}$: It's driven high when the $Tx$ finishes sending a complete byte on the serial output. This signal is always assigned to the $\boldsymbol{o\_Tx\_Done}$ signal.
d. $\boldsymbol{r\_Tx\_Busy}$: It's driven high when the $Tx$ is busy sending a byte on the serial output. This signal is always assigned to the $\boldsymbol{o\_Tx\_Busy}$ signal.

The UART transmitter has five states

1. $\boldsymbol{IDLE}$: The default transmitter state in which the serial output ($\boldsymbol{o\_Tx\_Serial}$) is continuously high and the $Tx$ keeps checking the $\boldsymbol{i\_start\_transmission}$ signal, if it is set to 1, the transmitter becomes busy (to send data on its serial output) and moves on to the next state, the $\boldsymbol{START\_BIT}$ state. Else, it remains in the $\boldsymbol{IDLE}$ state.

2. ***START_BIT***: The $Tx$ sends 0 (start bit) on the serial output, loads a byte from its $FIFO$ (8-bits parallel input), then moves on to the ***DATA_BITS*** state.
3. ***DATA_BITS***: The $Tx$ sends the loaded data byte on the serial output, bit-by-bit, starting by the least significant bit until sending the most significant one. Then it checks the parity enable signal (***pen***), if it's high, it calculates the parity bit then moves on to the ***PARITY_BIT*** state to send it, else it moves to the ***STOP_BIT*** state.
4. ***PARITY_BIT***: The $Tx$ sends the calculated parity bit, bring the ***r_Tx_Busy*** down to 0, and moves to the ***STOP_BIT*** state.
5. ***STOP_BIT***: Finally, the $Tx$ outputs 1 on its serial output (stop bit), sets the ***r_Tx_Done*** signal to 1, and checks whether there is another transmission awaits or not, if so it becomes busy (to send data again on its serial output) and moves to the ***START_BIT*** state. If no transmission awaits, the $Tx$ moves to the ***IDLE*** state.

## V. Test bench Output

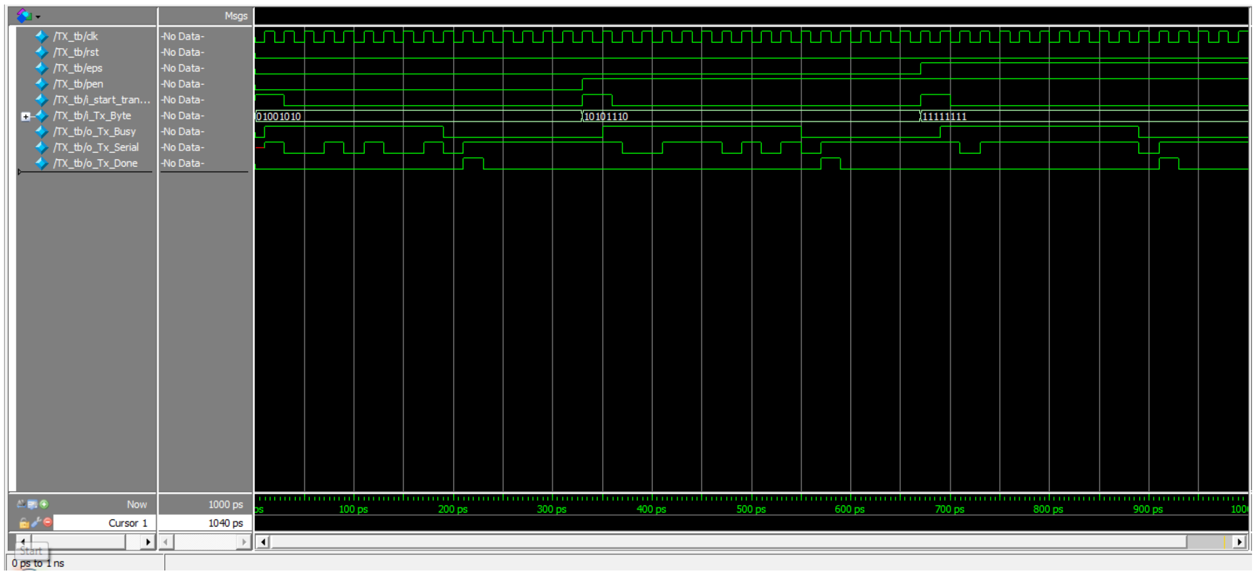


*Figure 9:UART Transmitter Testbench*

The first input byte is 01001010 and parity enable (***pen***) equal 0, and the ***i_start_transmission*** signal is driven high. So the $Tx$ drives the ***o_Tx_Busy*** signal high, and sends a start bit (zero) on the serial output, followed by the data bits, bit-by-bit, starting by the least significant bit until sending the most significant bit. Then it sends the stop bit (one), driving the ***o_Tx_Busy*** low, and the ***o_Tx_Done*** high for one bit duration.

We repeat the whole process again sending 10101110 as data and setting the parity enable (***pen***) to 1 and the even parity select (***eps***) to 0 (odd parity). So the $Tx$ sends a zero parity bit after the data bits as the number of ones in the data are odd.

We repeat the whole process for the third time sending 11111111 as data and setting the parity enable ($\boldsymbol{pen}$) to 1 and the even parity select ($\boldsymbol{eps}$) to 1 (even parity). So the $Tx$ sends a zero parity bit after the data bits as the number of ones in the data are even.

# UART Receiver module

## Module Finite State Machine

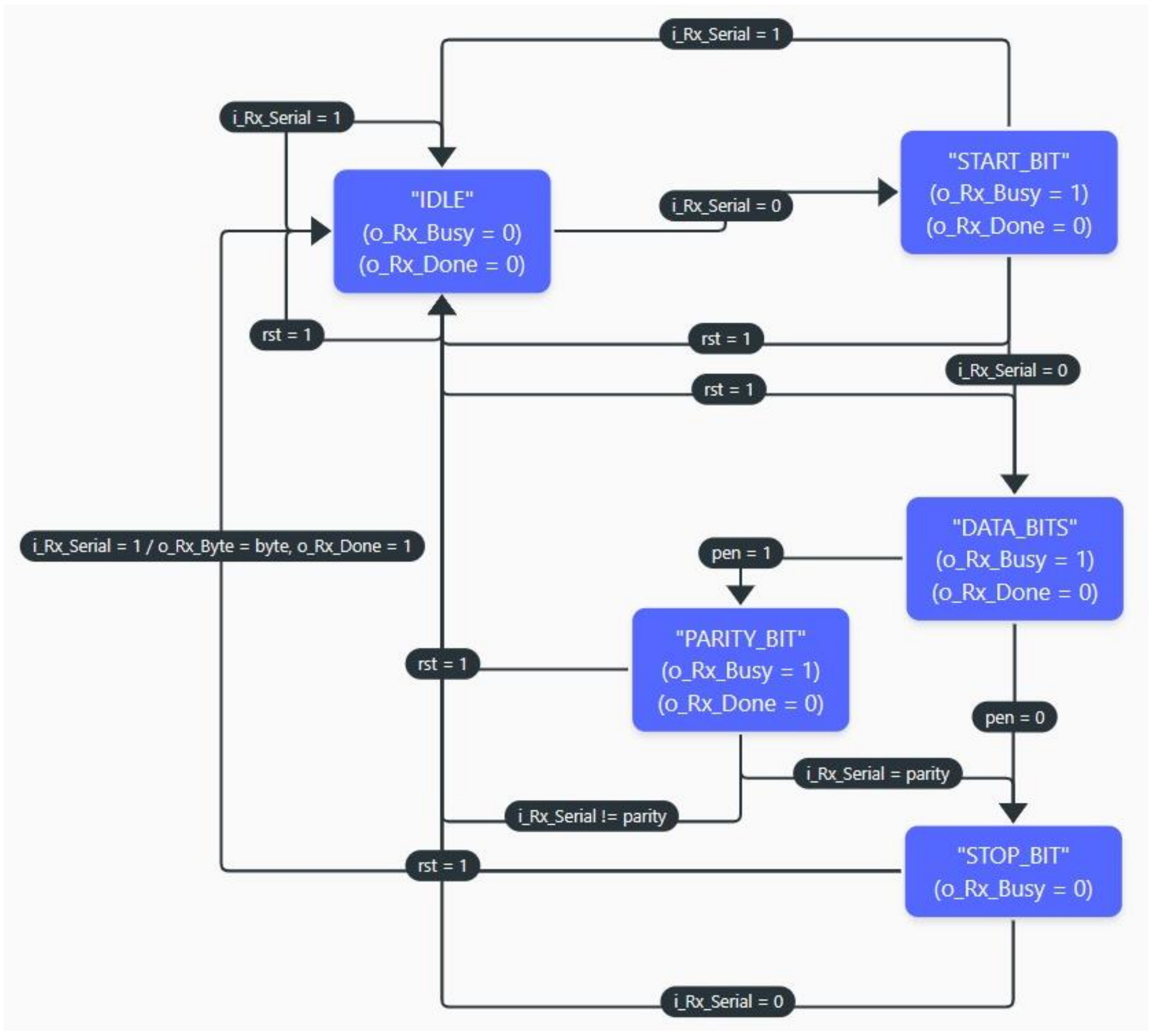


*Figure 10: UART Receiver  FSM*

## Module Inputs and Outputs description

### I. Inputs

| Signal | Description |
|---|---|
| $\boldsymbol{i\_Clock}$ | The input clock from the baud generator which is faster than the Baud rate by the sampling factor |
| $\boldsymbol{i\_Rx\_Serial}$ | The input serial signal of the UART |
| $\boldsymbol{i\_Clks\_Per\_Bit}$ | The number of clocks per bit which is the same as sampling factor (used to determine the sampling mode of the receiver). |
| $\boldsymbol{eps}$ | Even Parity select to toggle parity mode (even=1,odd=0) |
| $\boldsymbol{pen}$ | Parity Enable signal which enables/disables the parity bit (active high) |

### II. Outputs

| Signal | Description |
|---|---|
| $\boldsymbol{o\_Rx\_Busy}$ | An output signal used to indicate that the receiver is already receiving an input, it is driven high when receiving a start bit. |
| $\boldsymbol{o\_Rx\_Done}$ | An output signal used to indicate that a complete correct byte is received (the correctness of the byte is determined by the parity bit). |
| $\boldsymbol{o\_Rx\_Byte}$ | The parallel 8-bit output signal. |

### III. Brief description

UART receiver continuously samples the serial input for zero (start bit), after the start bit it samples and stores the input serial 8-bits on the $\boldsymbol{i\_Rx\_Serial}$ signal, then sampling the parity bit (if enabled) to check the correctness of the received byte using parity, then checks for one (stop bit). If the data is correct and the stop bit has arrived, the received 8-bit data is sent on the parallel 8-bit output $\boldsymbol{o\_Rx\_Byte}$.

### IV. Detailed description

Main local parameters

a. $\boldsymbol{r\_Bit\_Index}$: Is used to access the data byte bit-by-bit.
b. $\boldsymbol{r\_Rx\_Byte}$: The parallel 8-bit register used to store the output data. This signal is always assigned to the $\boldsymbol{o\_Rx\_Byte}$ signal.
c. $\boldsymbol{r\_Rx\_Done}$: It's driven high when the $Rx$ receives a complete valid byte from its serial input. This signal is always assigned to the $\boldsymbol{o\_Rx\_Done}$ signal.
d. $\boldsymbol{r\_Rx\_Busy}$: It's driven high when the $Rx$ is busy receiving a byte from its serial input. This signal is always assigned to the $\boldsymbol{o\_Rx\_Busy}$ signal.
e. $\boldsymbol{rx\_Data\_Bits}$: Used to store the serial input bits, bit-by-bit, in order.

The UART receiver has five states

1. $\boldsymbol{IDLE}$: The default receiver state in which it continuously sample the $\boldsymbol{i\_Rx\_Serial}$ input to detect a zero (start bit) with a rate faster than the baud rate 13x or 16x, depending on the sampling mode of the UART. If the start bit is detected, the $Rx$ becomes busy and moves on to the next state, the $\boldsymbol{START\_BIT}$ state. Else, it remains in the $\boldsymbol{IDLE}$ state.
2. $\boldsymbol{START\_BIT}$: The $Rx$ waits half the bit duration (to sample the bits at their center) then checks the validity of the start bit on the $\boldsymbol{i\_Rx\_Serial}$. If the start bit is valid, it moves on to the $\boldsymbol{DATA\_BITS}$ state, else it returns back to the $\boldsymbol{IDLE}$ state.
3. $\boldsymbol{DATA\_BITS}$: The $Rx$ waits for the center of each data bit then samples it from the $\boldsymbol{i\_Rx\_Serial}$ and stores it into the $\boldsymbol{rx\_Data\_Bits}$ register. It repeats this process 8 times to sample the 8-bit serial data, bit-by-bit, starting by the least significant bit until receiving the most significant one. Then it checks the parity enable signal ($\boldsymbol{pen}$), if it's high, it calculates the parity bit then moves on to the $\boldsymbol{PARITY\_BIT}$ state to send it, else it moves to the $\boldsymbol{STOP\_BIT}$ state. The parity bit is calculated by XORing the received data bits (using reduction XOR operator) then XOR the result with the negation of the even parity select bit.
4. $\boldsymbol{PARITY\_BIT}$: The $Rx$ waits for the center of the parity bit, samples it, then compares it with the calculated parity bit. If the samples parity bit matches the calculated one, it moves to the $\boldsymbol{STOP\_BIT}$ state, else, it brings the $\boldsymbol{r\_Rx\_Busy}$ down to 0, and returns back to the $\boldsymbol{IDLE}$ state, neglecting the wrong data bits received.
5. $\boldsymbol{STOP\_BIT}$: Finally, the $Rx$ waits for the center of the stop bit and samples it. If it equals one (stop bit), the $Rx$ sets the $\boldsymbol{r\_Byte\_Done}$ signal to 1, outputs the received data byte, stored in $\boldsymbol{rx\_Data\_Bits}$, on the 8-bit parallel output signal $\boldsymbol{r\_Rx\_Byte}$. Finally it brings the $\boldsymbol{r\_Rx\_Busy}$ down to 0, and returns back to the $\boldsymbol{IDLE}$ state regardless the value of the sampled stop bit.

## V. Test bench output

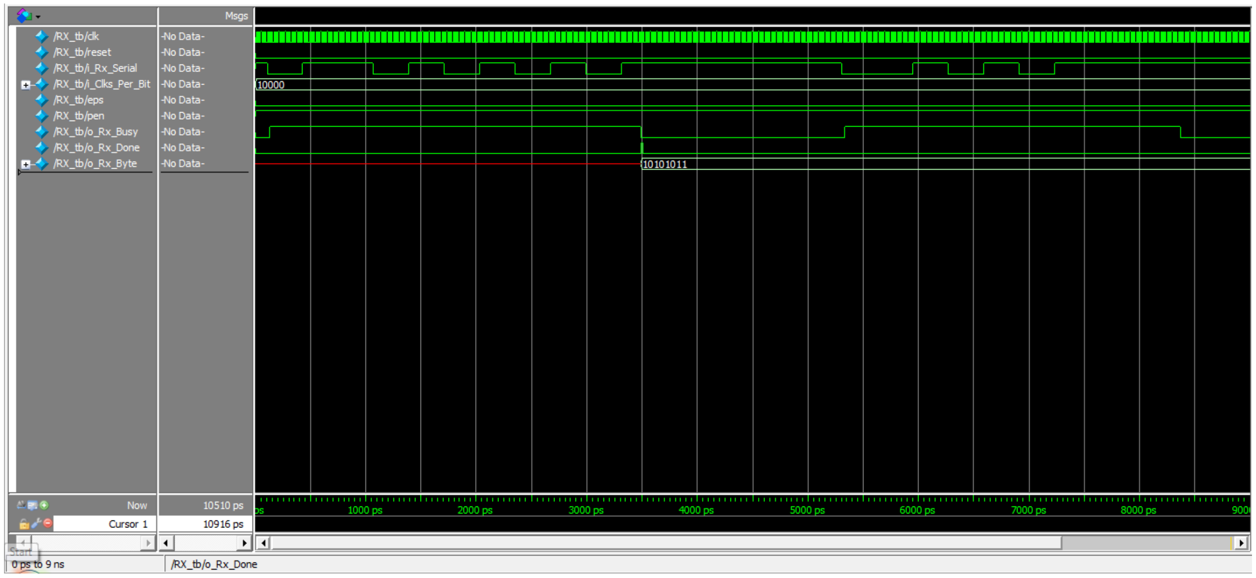


*Figure 11:UART Receiver Testbench*

The following data bits are received on the 16x sampling mode with odd parity ($pen = 1, eps = 0$).

| **Serial input (left first)** | **Start bit** | **Data bits** | **Parity bit** | **Stop bit** | **Validity** |
|---|---|---|---|---|---|
| 0 1 1 0 1 0 1 0 1 0 1 | 0 | 1010 1011 | 0 | 1 | Valid |
| 0 0 1 0 1 0 1 1 1 1 1 | 0 | 1110 1010 | 1 | 1 | Not valid |

Note: The data on the serial starts with start bit followed by least significant bit is until the most significant bit, parity bit then stop bit. So the Data bits is extracted from serial input then inverted.

In the $\boldsymbol{IDLE}$ state the receiver keeps sampling the $\boldsymbol{i\_Rx\_Serial}$ for start bit.

When the start bit of the first input is sampled, the $o\_Rx\_Busy$ signal is driven high, and starts receiving the data then the parity bit which is correct, then it samples the stop bit, then sends the valid data on the parallel 8-bit output $\boldsymbol{o\_Rx\_Byte}$, drive the $\boldsymbol{o\_Rx\_Busy}$ low and drive $\boldsymbol{o\_Rx\_Done}$ high for 1 clock cycle and returns to $\boldsymbol{IDLE}$.

When the start bit of the second input is sampled, the $\boldsymbol{o\_Rx\_Busy}$ signal is driven high, and starts receiving the data then the parity bit which is not correct, then it neglects the received bits and drive the $\boldsymbol{o\_Rx\_Busy}$ low and return to $\boldsymbol{IDLE}$.

# BONUS

## 1. Line Control Register (LCR)

Line Control Register controls the parity bits of the UART module.

| | | | | |
|---|---|---|---|---|
| Reserved [31:5] | EPS | PEN | STB | WLS [1:0] |

EPS (Even Parity Select) set to one for even parity and zero for odd parity.

PEN (Parity ENable) active high to enable parity bit.

STB and WLS are not supported in our project.

LCR can be modified by targeting its address at any time.

## 2. Mode Definition Register (MDR)

Mode definition Register controls the over sampling mode of the UART receiver.

| | |
|---|---|
| Reserved [31:1] | OSM_SEL |

OSM_SEL (Over Sampling Mode SELect)

0 = 16× oversampling

1 = 13× oversampling

MDR can be modified by targeting its address at any time.

## 3. Baud Generator Module With Two Over Sampling Modes

A baud generator that takes the CPU clock as input and generates a transmitter clock equals to the baud rate and a receiver clock with two over sampling modes depending on its OSM_SEL signal that can be modified during its operation as shown in the baud generator test bench below.

# GPIO Module

## Module diagram

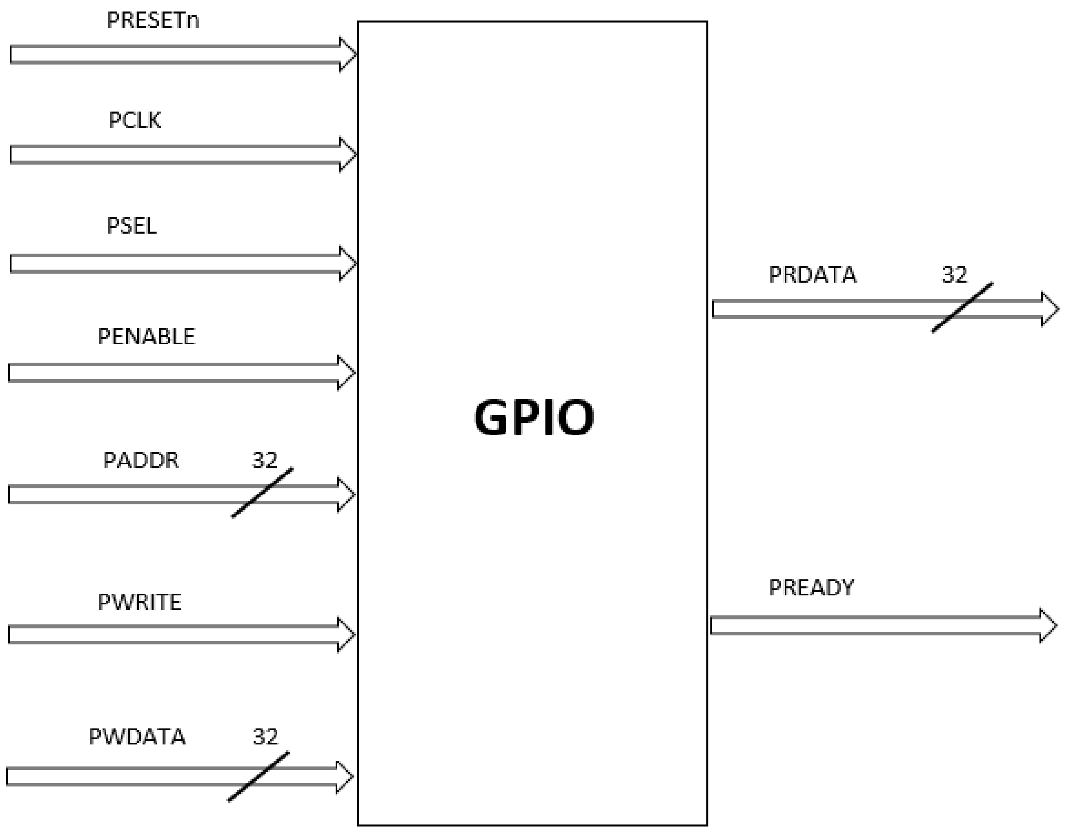


*Figure 12: GPIO diagram*

## Module Description

I. Inputs

| SIGNAL | DISCREPTION |
|---|---|
| PRESETn | Resets the GPIO module to its default values(active low) |
| PCLK | The input Clock. The rising edge of PCLK times all READS & WRITES on the GPIO |
| PSEL | Selection signal to inform GPIO that it is the slave(active high) |
| PENABLE | It indicates the 2nd cycle of a data transfer. It's an active high signal. |
| PADDR | The input address form APB to GPIO to access the right location |

| | |
|---|---|
| PWRITE | The Data transfer direction , if high indicates APB to GPIO (write access) , if low indicates GPIO to APB (read access) |
| PWDATA | Write data. The data want to be written during write cycles when PWRITE is high. |

I. Ouputs

| | |
|---|---|
| PRDATA | signal that drives this bus during read operation to APB. |
| PREADY | Signal indicating that GPIO is ready. |

## III. Brief description:

general-purpose input output (GPIO) is a signal pin on an integrated circuit or electronic circuit board which may be used as an input or output, or both, and is controllable by software. GPIO is also connected to the APB bus to take (write) and give (read) data.

## IV. FINITE STATE MACHINE:

The GPIO module code implementation depends on no state representations as it only reads the data from the bus into the reg_data and reads data from register to the APB bus.

## V. Test benchs

- Write operation:

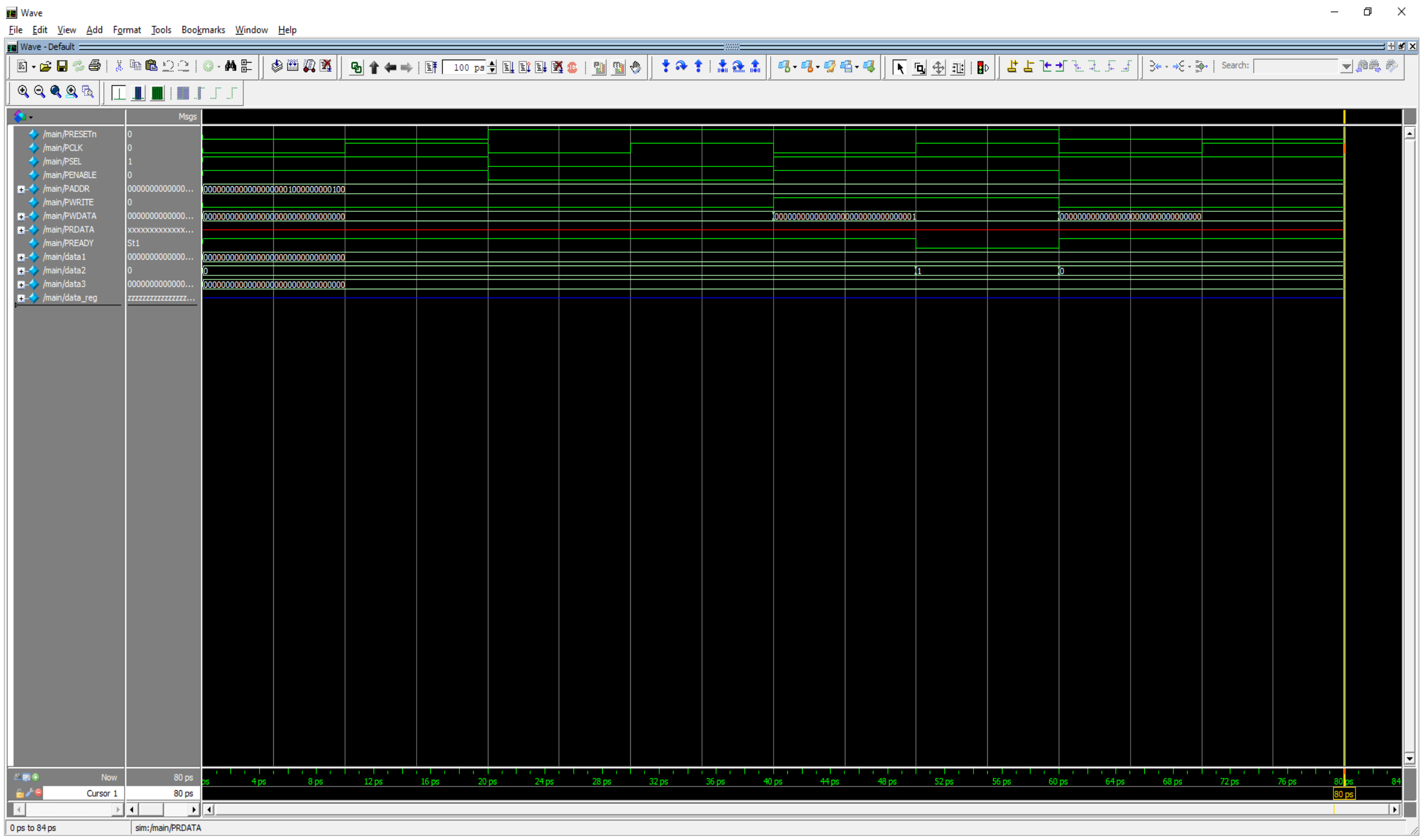

Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Msgs
/main/PRESETn 0
/main/PCLK 0
/main/PSEL 1
/main/PENABLE 0
/main/PADDR 0000000000000...
/main/PWRITE 0
/main/PWDATA 0000000000000...
/main/PRDATA xxxxxxxxxxxxx...
/main/PREADY St1
/main/data1 0000000000000...
/main/data2 0
/main/data3 0000000000000...
/main/data_reg zzzzzzzzzzzzzzz...
Now 80 ps
Cursor 1 80 ps
0 ps to 84 ps
sim:/main/PRDATA

*Figure 13:GPIO write operation*

```
initial
begin
PRESETn = 0 ; PCLK = 0 ; PSEL = 1 ; PENABLE = 1 ; PADDR = 32'h1004 ; PWRITE = 0 ; PWDATA = 0 ; // check reset
@(negedge PCLK)
PRESETn = 1 ; PSEL = 1 ; PENABLE = 0 ; PADDR = 32'h1004 ; PWRITE = 0 ; PWDATA = 0 ;  // check sel
@(negedge PCLK)–
PRESETn = 1 ; PSEL = 1 ; PENABLE = 1 ; PADDR = 32'h1004; PWRITE = 1; PWDATA = 1 ;
@(negedge PCLK)
PRESETn = 0  ; PCLK = 0 ; PSEL = 1 ; PENABLE = 0 ; PADDR = 32'h1004 ; PWRITE = 0 ; PWDATA = 0 ;
$stop;
end
```

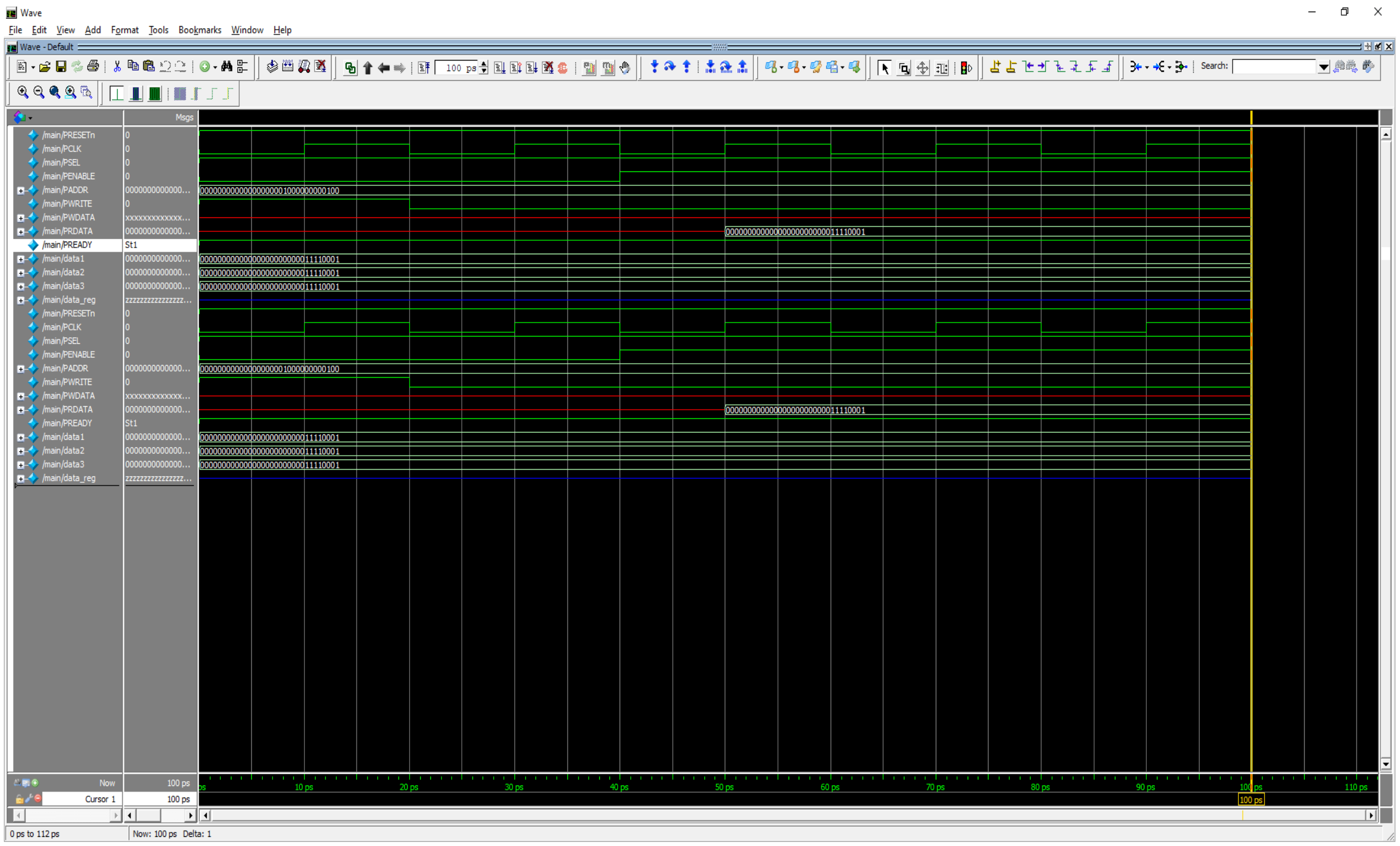
1- At the first initial condition, we check the reset signal to be low active. (**PRESETn = 0**), the rest signals are don't care. We predict that at the next +ve edge CLK, all control signals will be reset to zeroes. Which is already satisfied as shown in the image.

2- At the second initial condition, the (**PRESETn = 1**) to be Off. While the PSEL is set to 1(**PSEL = 1**).the enable signal is still equal to Zero (**PENABLE = 0**), so there will be no Response at the output (**data2**) in the next +ve edge CLK.

3- At the third initial condition, the (**PRESETn = 1**) to be Off. While the PSEL is set to 1(**PSEL = 1**).the enable signal is now set to 1(**PENABLE = 1**) to test the enable signal. All control signals are set to 1,the PADD is set to point at "32'h1004" (**PADDR = 32'h1004)** ,which is the address of the output(**data2**).we will be expecting the output(**data2**) in the next +ve edge CLK to have the value (**PWDATA = 1**).

4- At the final initial condition, the RESET is set to zero (**PRESETn = 0**),to change the values again to their default value(zero) in the following +ve edge CLK. Which is already satisfied as shown in the image.

- Read operation:


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
100 ps
Search:
Msgs
Now 100 ps
Cursor 1 100 ps
0 ps to 112 ps
Now: 100 ps Delta: 1
ps 10 ps 20 ps 30 ps 40 ps 50 ps 60 ps 70 ps 80 ps 90 ps 100 ps 110 ps

| | Msgs | |
|---|---|---|
| /main/PRESETn | 0 | |
| /main/PCLK | 0 | |
| /main/PSEL | 0 | |
| /main/PENABLE | 0 | |
| /main/PADDR | 0000000000000... | 0000000000000000001000000000100 |
| /main/PWRITE | 0 | |
| /main/PWDATA | xxxxxxxxxxxxx... | |
| /main/PRDATA | 0000000000000... | 0000000000000000000000011110001 |
| /main/PREADY | St1 | |
| /main/data1 | 0000000000000... | 0000000000000000000000011110001 |
| /main/data2 | 0000000000000... | 0000000000000000000000011110001 |
| /main/data3 | 0000000000000... | 0000000000000000000000011110001 |
| /main/data_reg | zzzzzzzzzzzzzzzz... | |
| /main/PRESETn | 0 | |
| /main/PCLK | 0 | |
| /main/PSEL | 0 | |
| /main/PENABLE | 0 | |
| /main/PADDR | 0000000000000... | 0000000000000000001000000000100 |
| /main/PWRITE | 0 | |
| /main/PWDATA | xxxxxxxxxxxxx... | |
| /main/PRDATA | 0000000000000... | 0000000000000000000000011110001 |
| /main/PREADY | St1 | |
| /main/data1 | 0000000000000... | 0000000000000000000000011110001 |
| /main/data2 | 0000000000000... | 0000000000000000000000011110001 |
| /main/data3 | 0000000000000... | 0000000000000000000000011110001 |
| /main/data_reg | zzzzzzzzzzzzzzzz... | |

*Figure 14:GPIO read operation*

```
initial

begin

data_reg = 32'hF1 ;

data_reg2 = 32'hF1 ;

data_reg3 = 32'hF1 ;

PREADY = 1 ;

End


initial

 begin

 PRESETn = 1  ;  PSEL = 1 ; PENABLE = 1 ; PADDR = 32'h1004 ; PWRITE = 1;  PCLK = 0;

 @(negedge PCLK)

 PRESETn = 1 ; PSEL = 1 ; PENABLE = 0 ; PADDR = 32'h1004 ; PWRITE = 0 ;

 @(negedge PCLK)

 PRESETn = 1 ; PSEL = 1 ; PENABLE —= 1 ; PADDR = 32'h1004 ; PWRITE = 0 ;

 @(negedge PCLK)

 PRESETn = 1 ; PSEL = 1; PENABLE = 1 ; PADDR = 32'h1004 ; PWRITE = 0 ;

@(negedge PCLK)

 PRESETn = 0 ; PSEL = 0 ; PENABLE = 0 ; PADDR = 32'h1004 ; PWRITE = 0 ;

end
```

**NOTE**: before starting the test bench, we set the registers value to be 32'hF1.this data will be written into the **PRDATA** register when **(PWRITE=0).**

1. At the first initial condition, the (**PRESETn = 1**) to be Off. While the PSEL is set to 1(**PSEL = 1**). We intend to test the PWRITE signal, so It is set to 1 which indicates writing **(PWRITE = 1)**, so there will be no Response at the output (**PRDATA**) in the next +ve edge CLK.
2. At the second initial condition, the (**PRESETn = 1**) to be Off. While the PSEL is set to 1(**PSEL = 1**).the enable signal is still equal to Zero (**PENABLE = 0**), PWRITE is set to 0 which indicates reading **(PWRITE = 0),** yet there is no response at the output (**PRDATA**) in the next +ve edge CLK because PENABLE is set to zero.

**3-** At the third initial condition, the (**PRESETn = 1**) to be Off. While the PSEL is set to 1(**PSEL = 1**).the enable signal is now set to 1(**PENABLE = 1**). All control signals are set to 1,the PADD is set to point at “32’h1004” (**PADDR = 32'h1004)** ,which is the address of the data_reg2 (**data2**).we will be expecting the output(**PRDATA**) in the next +ve edge CLK to have the value (**PRDATA=32’h1**).

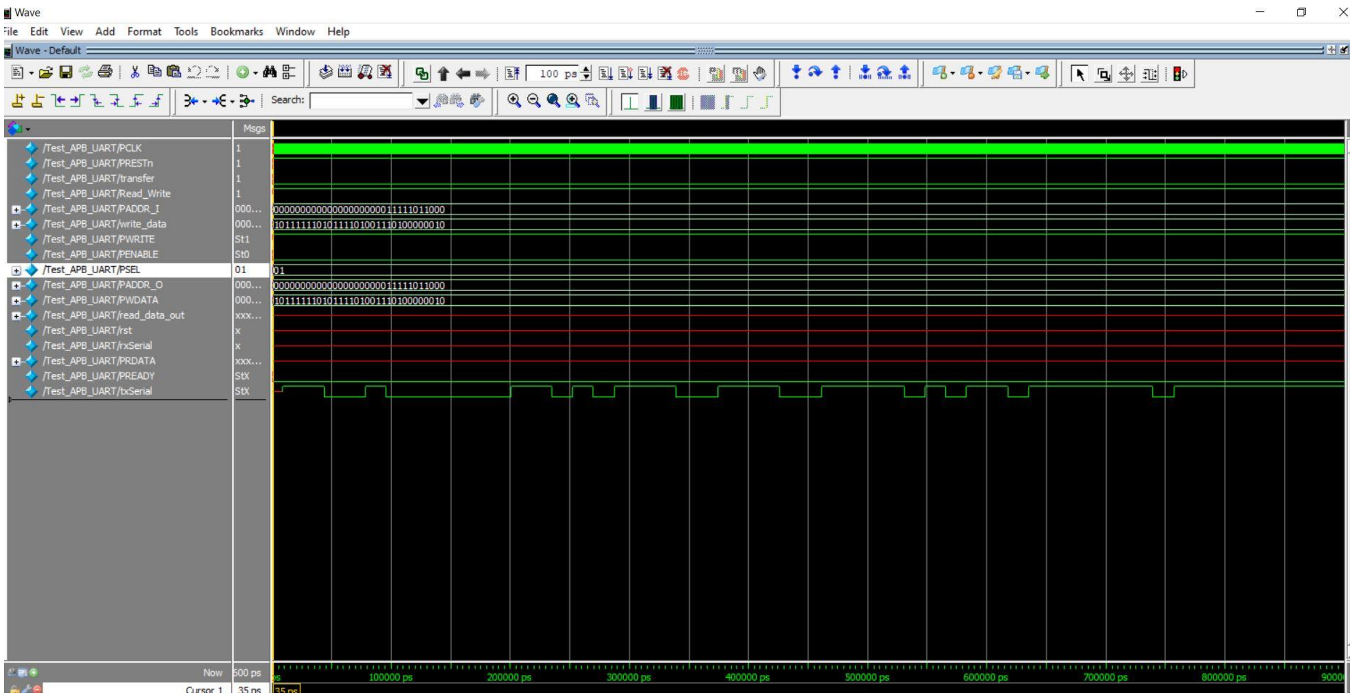
**4-** at the fourth initial condition, all signals have the same value as the previous test condition, to make sure that it is working properly.

**5-** At the final initial condition, the RESET is set to zero (**PRESETn = 0**),to change the values again to their default value(zero) in the following +ve edge CLK. Which is already satisfied as shown in the image.

# APB Bus With slaves

## APB bus with UART

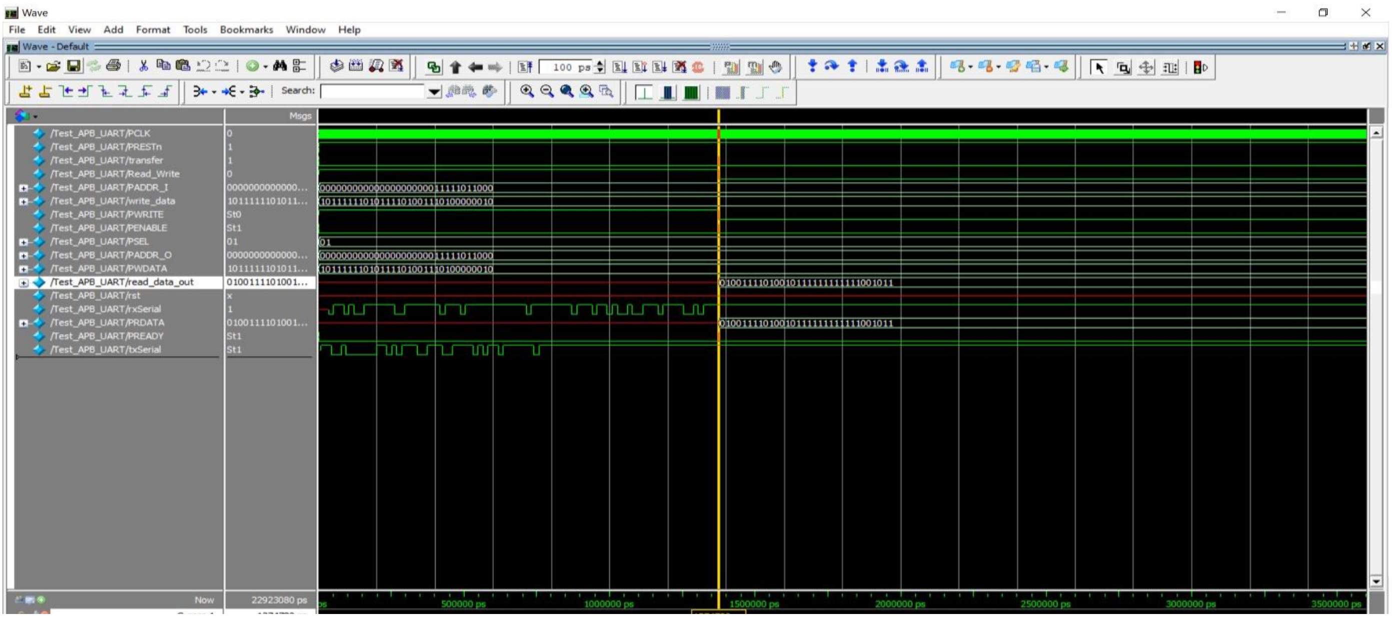
### I. Transmit



```
/Test_APB_UART/PCLK           1
/Test_APB_UART/PRESTn         1
/Test_APB_UART/transfer       1
/Test_APB_UART/Read_Write     1
/Test_APB_UART/PADDR_I        000...    00000000000000000000011111011000
/Test_APB_UART/write_data     000...    10111111010111101001110100000010
/Test_APB_UART/PWRITE         St1
/Test_APB_UART/PENABLE        St0
/Test_APB_UART/PSEL           01        01
/Test_APB_UART/PADDR_O        000...    00000000000000000000011111011000
/Test_APB_UART/PWDATA         000...    10111111010111101001110100000010
/Test_APB_UART/read_data_out  xxx...
/Test_APB_UART/rst            x
/Test_APB_UART/rxSerial       x
/Test_APB_UART/PRDATA         xxx...
/Test_APB_UART/PREADY         StX
/Test_APB_UART/txSerial       StX
Now 500 ps
Cursor 1 35 ps
```

Figure 15: APB Connect with Uart at Transmit operation

### II. Receive

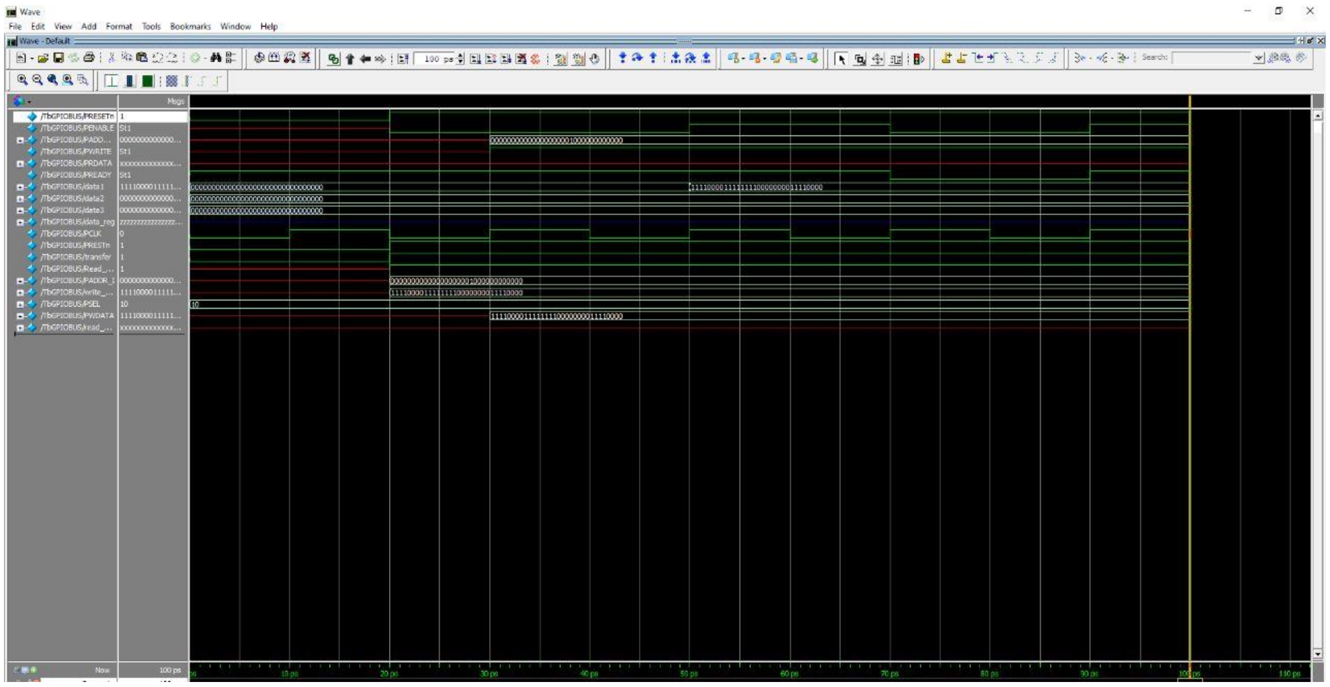


```
/Test_APB_UART/PCLK           0
/Test_APB_UART/PRESTn         1
/Test_APB_UART/transfer       1
/Test_APB_UART/Read_Write     0
/Test_APB_UART/PADDR_I        0000000000000...   00000000000000000000011111011000
/Test_APB_UART/write_data     1011111101011...   10111111010111101001110100000010
/Test_APB_UART/PWRITE         St0
/Test_APB_UART/PENABLE        St1
/Test_APB_UART/PSEL           01                 01
/Test_APB_UART/PADDR_O        0000000000000...   00000000000000000000011111011000
/Test_APB_UART/PWDATA         1011111101011...   10111111010111101001110100000010
/Test_APB_UART/read_data_out  0100111101001...   0100111101001011111111111001011
/Test_APB_UART/rst            x
/Test_APB_UART/rxSerial       1
/Test_APB_UART/PRDATA         0100111101001...   0100111101001011111111111001011
/Test_APB_UART/PREADY         St1
/Test_APB_UART/txSerial       St1
Now 22923080 ps
```

Figure 16: APB Connect with Uart at Receive operation

APB bus with GPIO

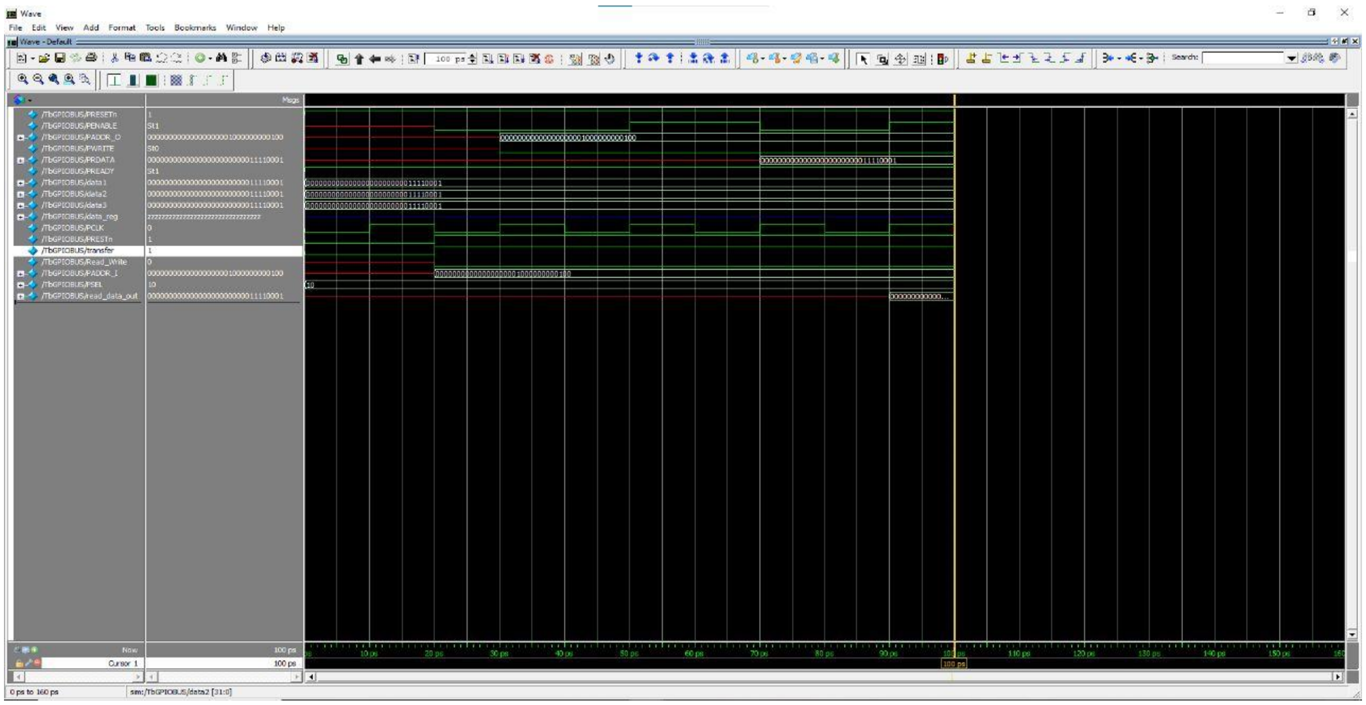
I. Write operation



| Signal | Msgs |
|---|---|
| /TbGPIOBUS/PRESETn | 1 |
| /TbGPIOBUS/PENABLE | St1 |
| /TbGPIOBUS/PADD... | 0000000000000... |
| /TbGPIOBUS/PWRITE | St1 |
| /TbGPIOBUS/PRDATA | xxxxxxxxxxxxx... |
| /TbGPIOBUS/PREADY | St1 |
| /TbGPIOBUS/data1 | 1111000011111... |
| /TbGPIOBUS/data2 | 0000000000000... |
| /TbGPIOBUS/data3 | 0000000000000... |
| /TbGPIOBUS/data_reg | zzzzzzzzzzzzzzz... |
| /TbGPIOBUS/PCLK | 0 |
| /TbGPIOBUS/PRESTn | 1 |
| /TbGPIOBUS/transfer | 1 |
| /TbGPIOBUS/Read_... | 1 |
| /TbGPIOBUS/PADDR_I | 0000000000000... |
| /TbGPIOBUS/write_... | 1111000011111... |
| /TbGPIOBUS/PSEL | 10 |
| /TbGPIOBUS/PWDATA | 1111000011111... |
| /TbGPIOBUS/read_... | xxxxxxxxxxxxx... |

*Figure 17:APB Connect with GPIO at write operation*

II. Read operation



| Signal | Msgs |
|---|---|
| /TbGPIOBUS/PRESETn | 1 |
| /TbGPIOBUS/PENABLE | St1 |
| /TbGPIOBUS/PADDR_O | 00000000000000000001000000000100 |
| /TbGPIOBUS/PWRITE | St0 |
| /TbGPIOBUS/PRDATA | 00000000000000000000000011110001 |
| /TbGPIOBUS/PREADY | St1 |
| /TbGPIOBUS/data1 | 00000000000000000000000011110001 |
| /TbGPIOBUS/data2 | 00000000000000000000000011110001 |
| /TbGPIOBUS/data3 | 00000000000000000000000011110001 |
| /TbGPIOBUS/data_reg | zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz |
| /TbGPIOBUS/PCLK | 0 |
| /TbGPIOBUS/PRESTn | 1 |
| /TbGPIOBUS/transfer | 1 |
| /TbGPIOBUS/Read_Write | 0 |
| /TbGPIOBUS/PADDR_I | 00000000000000000001000000000100 |
| /TbGPIOBUS/PSEL | 10 |
| /TbGPIOBUS/read_data_out | 00000000000000000000000011110001 |

*Figure 18:APB Connect with GPIO at read operation*