UART TRANSMITTER

This lab has you implement a UART Transmitter module.

BACKGROUND

Review the notes on UART transmission.

TASK DESCRIPTION

You are to implement a UART transmitter that has the following features:

- Uses a hardware FIFO
- Supports multiple baud rates
- Data format is 8 data + 1 start + 1 stop bit

The module interface is given below:

```
module uart_tx(clk, reset, din, dout, wren, rden, txout, addr)
input clk, reset, wren, rden;
input [7:0] din;
output [7:0] dout;
output txout;       //serial data out
input [2:0] addr;

//This is the initial value for the period register for the baud rate
//generator. This timeout value is one-half of the 16x baud rate clock
//period for shifting out bits
parameter PERIOD = 8'h1A;
```

The input/output definition is:

- *clk* – clock input
- *reset* – high true reset
- *rden* – read enable for dout bus
- *din* – data input bus
- *dout* – data output bus, reflects the contents addressed by the *addr* input when *rden* is high, else *dout* is 0.
- *addr* – address bus for internal registers
- *wren* – write line for a register. The register selected by *addr* is written on the rising clock edge when *wren* is high.
- *txout* – serial data out

Register addressing and reset behavior:

Register	Address	Comments	Value at reset

Period Register for baud rate timer	0b000	This is an 8-bit register, can be read and written.	Set to parameter `PERIOD`
TX Reg	0b001	Write port for TX FIFO. This is a write only port, cannot read this port.	Undefined
Undefined (dout = 0 for this)	0b010	unimplemented	undefined
Status/Control	0b011	This is an 8-bit register, can be read and written.	See comments below

Status/Control register definition:

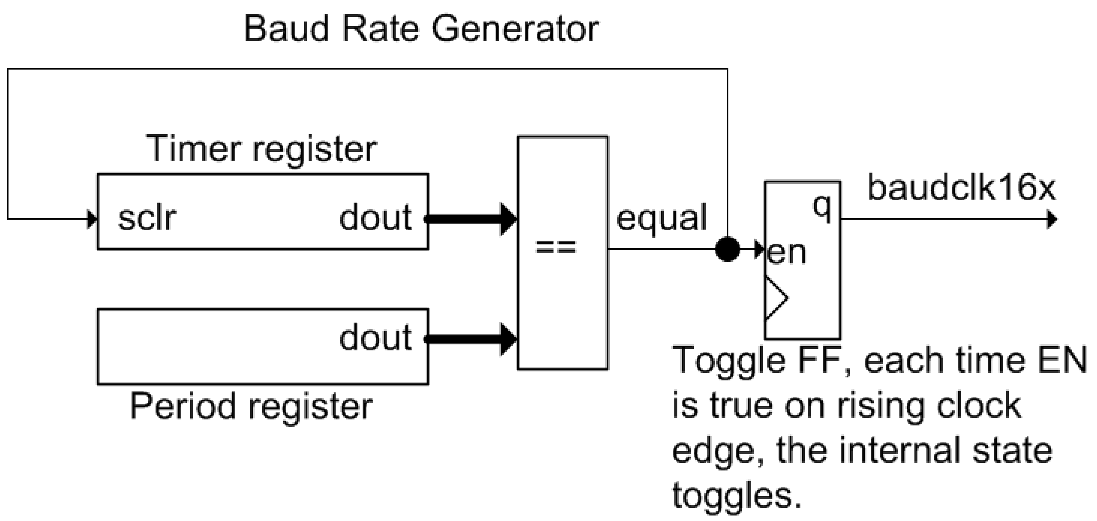
- Bit 0: TXFULL – this is set to a '1' when the transmit FIFO is full. This is a read-only bit, it cannot be cleared by writing to the status register. It is only cleared when the control logic reads the TX FIFO to send data to the internal TX Shift register. To implement this bit, simply pass the FIFO FULL signal out as this bit. The value of this bit at reset is '1'.
- Bit 1: TXDONE – this bit is set to a '1' when a byte transmission is finished (including the stop bit!). It can only be cleared by a write to the status/control register
- Bits 2-7: These are currently unimplemented, and should read as 0.

Implementation: TX FIFO

To implement the UART TX block, you must first generate a FIFO using the Xilinx CoreGen tool. See the appendix for the options required to generate this FIFO. This FIFO will be used within your design.

Implementation: Baud Rate Generator

For baud rate control, implement some logic that generates a clock with a period that is 1/16 of a bit time (16 cycles of this clock gives one bit time). A timer with a period register (like what you have previously done) is the approach that must be used for this as shown below.

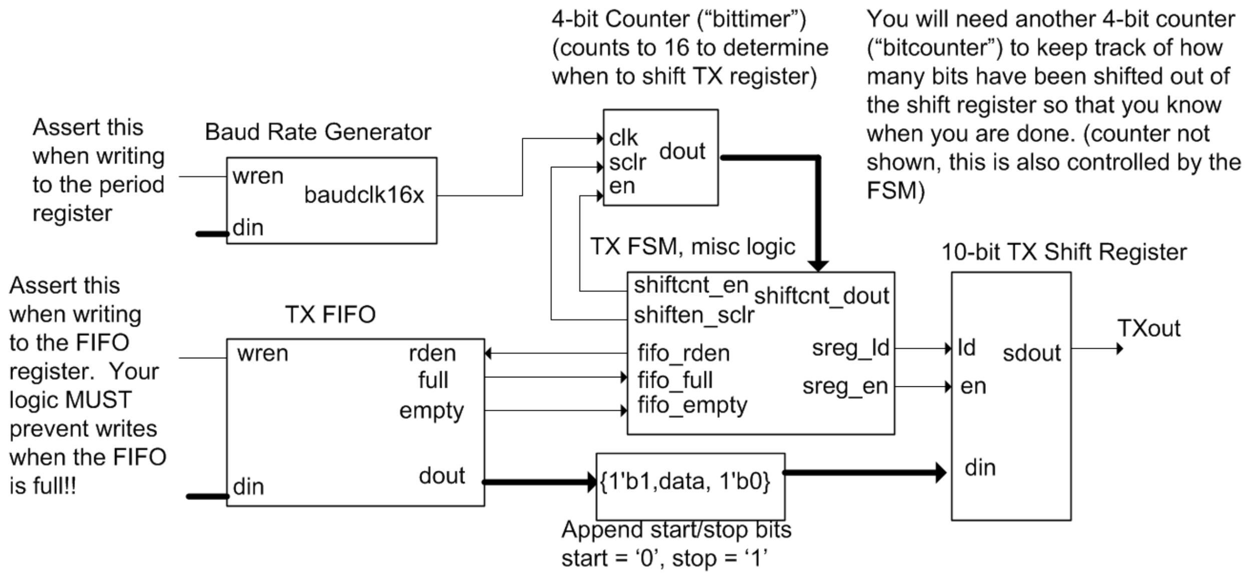

Baud Rate Generator
Timer register
sclr
dout
dout
Period register
==
equal
q
en
baudclk16x
Toggle FF, each time EN is true on rising clock edge, the internal state toggles.

The 16X baud rate clock should be driven from the toggle FF that is toggled each time the timer and period register match. This means that the timeout period will be one-half of the 16X baud rate clock period as two timer matches will generate one complete clock cycle of the 16X baud rate clock.

This is the first time that we are using a generated signal as a clock within our design. The 16X baud rate clock will be used as the clock for a 4-bit counter that is used to count 16 of these cycles for measuring one bit time.

Implementation: Overall Design

The block diagram below is very rough diagram of the needed hardware (I have left it 'rough' to give you design freedom, other signals, interconnections, and logic is definitely needed).



4-bit Counter ("bittimer") (counts to 16 to determine when to shift TX register)
You will need another 4-bit counter ("bitcounter") to keep track of how many bits have been shifted out of the shift register so that you know when you are done. (counter not shown, this is also controlled by the FSM)
Assert this when writing to the period register
Baud Rate Generator
wren
baudclk16x
din
clk
sclr
en
dout
TX FSM, misc logic
10-bit TX Shift Register
Assert this when writing to the FIFO register. Your logic MUST prevent writes when the FIFO is full!!
TX FIFO
wren
rden
full
empty
din
dout
shiftcnt_en
shiften_sclr
shiftcnt_dout
fifo_rden
fifo_full
fifo_empty
sreg_ld
sreg_en
ld
en
din
sdout
TXout
{1'b1,data, 1'b0}
Append start/stop bits start = '0', stop = '1'

This is a rough diagram, more signals and additional components may be needed.

You will need a FSM to control reading data from the TX FIFO, writing it to the TX shift register, and then shifting data out. You can get by with only three states in your FSM. The following is a textual description of the states, you will need to transform this into an implementation:

- INPUT_WAIT_STATE: Stay in this state while the FIFO EMPTY signal is '1'. When FIFO EMPTY becomes '0', assert the FIFO rden (read enable) signal and the TX shift register *ld* (load) signal to transfer data from the TX FIFO to the Shift register, and go to state SHIFT1. This state should keep the timer disabled and cleared until the FIFO becomes non-empty, and on the transition to the next state, start the timer ticking.
- SHIFT1_STATE: If the "bitcounter" count value is 10, then you have shifted all 10 bits and transition back to the INPUT_WAIT_STATE, else wait until the baudclk16x clock is HIGH, and state SHIFT2 (waiting until the baudclk16x means that one half of a bit time has passed). This state should also ensure that the "bittimer" counter will increment when the rising edge of baudclk16x occurs.
- SHIFT2_STATE: Wait until the baudclk16x is low (this means the entire bit time has passed). If baudclk16x is low and the "bittimer" counter is 0, then one entire bit time has passed, so

increment the "bitccounter" counter by one, shift the shift register by one position, and then go back to the SHIFT1_STATE to start timing the next bit.

This FSM continuously loops until the FIFO is non-empty. So, several writes to the FIFO stores data in the FIFO, and the FSM fetches bytes from the FIFO each time it is finished shifting all of the bits out.

Some other implementation tips:

1. The shift register should be reset to all "1s" on power up reset so that *txou*t starts high.
2. The *txout* line is connected to the LSb of the shift register, and the shift operation is a right shift (from MSb to LSb). The new bit shifted into the MSb should be a '1'.
3. The FSM description above assumes the baudclk16x clock should always start out as '0'.
4. The *baudclk16x* clock is only being generated when you are sending data – this means you enable and disable the timer appropriately.
5. The *baudclk16x* signal is only used as a clock signal for the "bittimer" counter – the system clock is used as the clock for all other components.
6. The *baudclk16x* signal is used as a control input to the FSM (it tests whether it is high or low). The *baudclk16x* signal is NOT a clock signal for this FSM, the system clock is used for the FSM. The block diagram is incomplete in that the *baudclk16x* signal is not shown as an input to the FSM.
7. The address interface only allows writes to the PERIOD register; writes to the TIMER register are not needed (the timer register should be cleared while waiting for the FIFO to become non-empty).
8. You may be surprised by how many system clock cycles it will take to send just one byte of data. The system clock is 50 MHz, a period of 20 ns. Each transmission is 10 bit times; so a baud rate of 115,200 takes 10 bit times x 1/115,200 = 86,805 ns. The number of system clocks is then 86,805 ns/20 ns = 4341 clocks! Asynchronous serial transmission is SLOW!

Associated files

The ZIP archive associated with this lab contains the following files:

- *uart_tx.v* -- complete this module
- *tb_uart_tx.v* – test bench for the FIFO
- *report.doc* – report file that needs to be filled out

uart_tx.v

Create a project named *lab8_uart_tx* and finish implementing the *uart_tx.v* module (use the Spartan Family Xilinx device you have used for previous labs). Verify both behavioral simulation and Post-route simulation using the *tb_uart_txt.v* testbench. The testbench contains a parameter that sets the baud rate of the serial link as shown below. You will need to test your design for two different baud rates.

```
//parameter UUT_PERIOD=8'h1A;    //57600 baudrate
parameter UUT_PERIOD=8'h0C;    //115200 baudrate
```

The testbench should be run for long enough to transmit 31 characters. This time is calculated as:

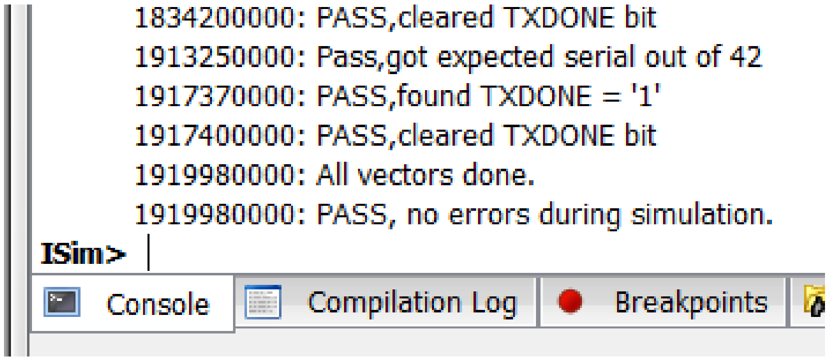
31 character times = 1/baudrate x 10 bits x 31 characters.

For a baud rate of 115,200 this time would be:

31 character times = 1/115200 x 10 bits x 31 characters = 0.00269 seconds, approximately 3 ms.

To run for 3 ms, type the command 'run 3 ms'. Note the number of system clocks this requires is 3 ms/ 20 ns = 3000 us/20 ns = 3,000,000 ns / 20 ns = 150,000 clocks!

You will know that all vectors passed when you get the following message:



```
        1834200000: PASS,cleared TXDONE bit
        1913250000: Pass,got expected serial out of 42
        1917370000: PASS,found TXDONE = '1'
        1917400000: PASS,cleared TXDONE bit
        1919980000: All vectors done.
        1919980000: PASS, no errors during simulation.
ISim>
```

Fill out the requested information in the *report.doc* file.

Submission

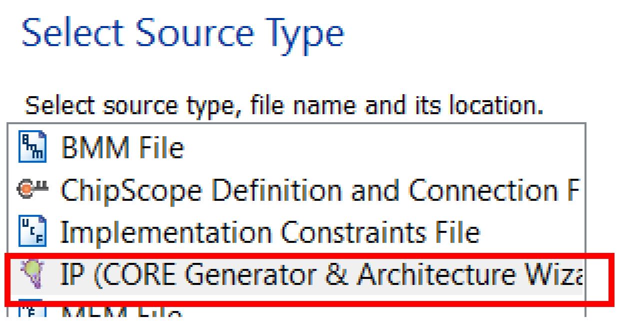
For submission, create a directory named 'lab8_*netid*', i.e., (lab8_rbr5).

Copy your *lab8_uart_tx* project directory to this directory.
Copy your completed *report.doc* to this directory.

Create a ZIP archive of this directory and submit it.

Appendix: Parameters for generating the FIFO memory

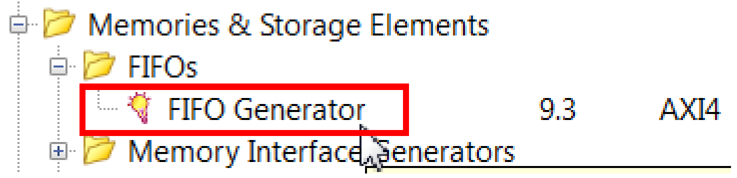
You need to generate a FIFO that will be used within your UART TX module. This FIFO will have room for 18 bytes (even though the parameter you use to specify the number of locations will be 16).
The following screenshots show the parameters to use.


Select Source Type

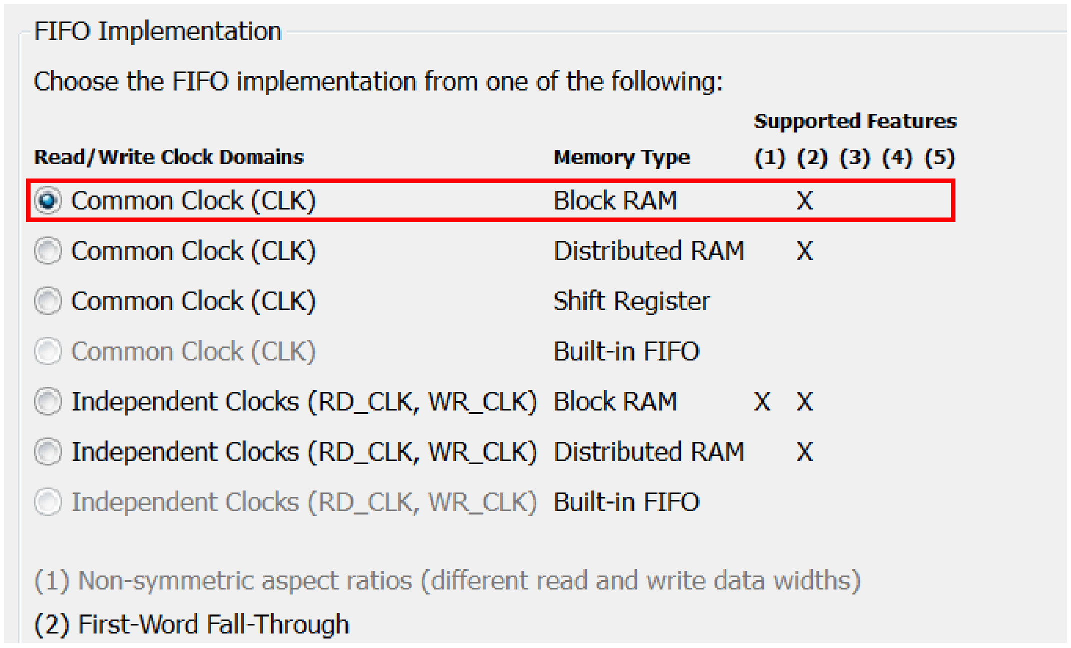
Select source type, file name and its location.

- BMM File
- ChipScope Definition and Connection F
- Implementation Constraints File
- IP (CORE Generator & Architecture Wiz
- MEM File

Select FIFO Generator



Parameters: Page 1, use all defaults.
Parameters: Page 2, select 'Common Clock, Block RAM'


FIFO Implementation

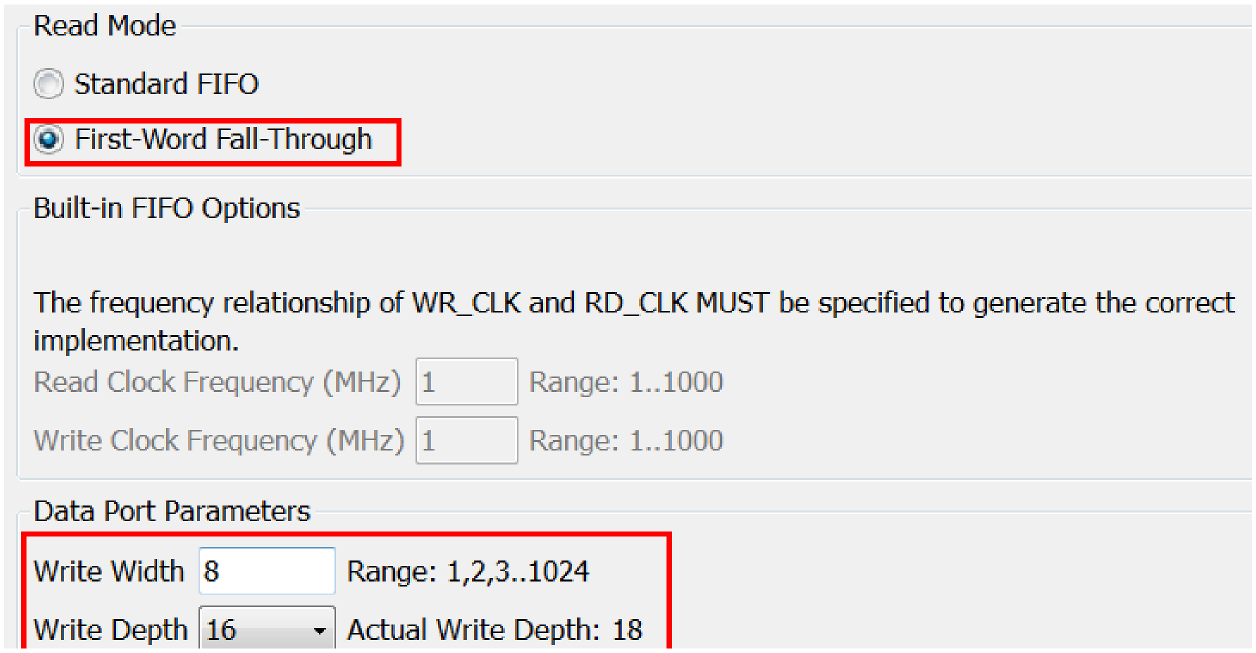
Choose the FIFO implementation from one of the following:

		Supported Features				
Read/Write Clock Domains	Memory Type	(1)	(2)	(3)	(4)	(5)
[✓] Common Clock (CLK)	Block RAM		X			
Common Clock (CLK)	Distributed RAM		X			
Common Clock (CLK)	Shift Register					
Common Clock (CLK)	Built-in FIFO					
Independent Clocks (RD_CLK, WR_CLK)	Block RAM	X	X			
Independent Clocks (RD_CLK, WR_CLK)	Distributed RAM		X			
Independent Clocks (RD_CLK, WR_CLK)	Built-in FIFO					

(1) Non-symmetric aspect ratios (different read and write data widths)
(2) First-Word Fall-Through

Parameters: Page 3, select 'First-Word Fall-Through', Write width=8 (each element is 8 bits), and write-depth = 16 (this FIFO will actually hold18 elements as shown on the screenshot).

'First-Word Fall-Through' means that when an element is written to the FIFO, it immediately becomes available on the output port.



Read Mode

Standard FIFO [✓ First-Word Fall-Through]

Built-in FIFO Options

The frequency relationship of WR_CLK and RD_CLK MUST be specified to generate the correct implementation.

Read Clock Frequency (MHz) 1 Range: 1..1000

Write Clock Frequency (MHz) 1 Range: 1..1000

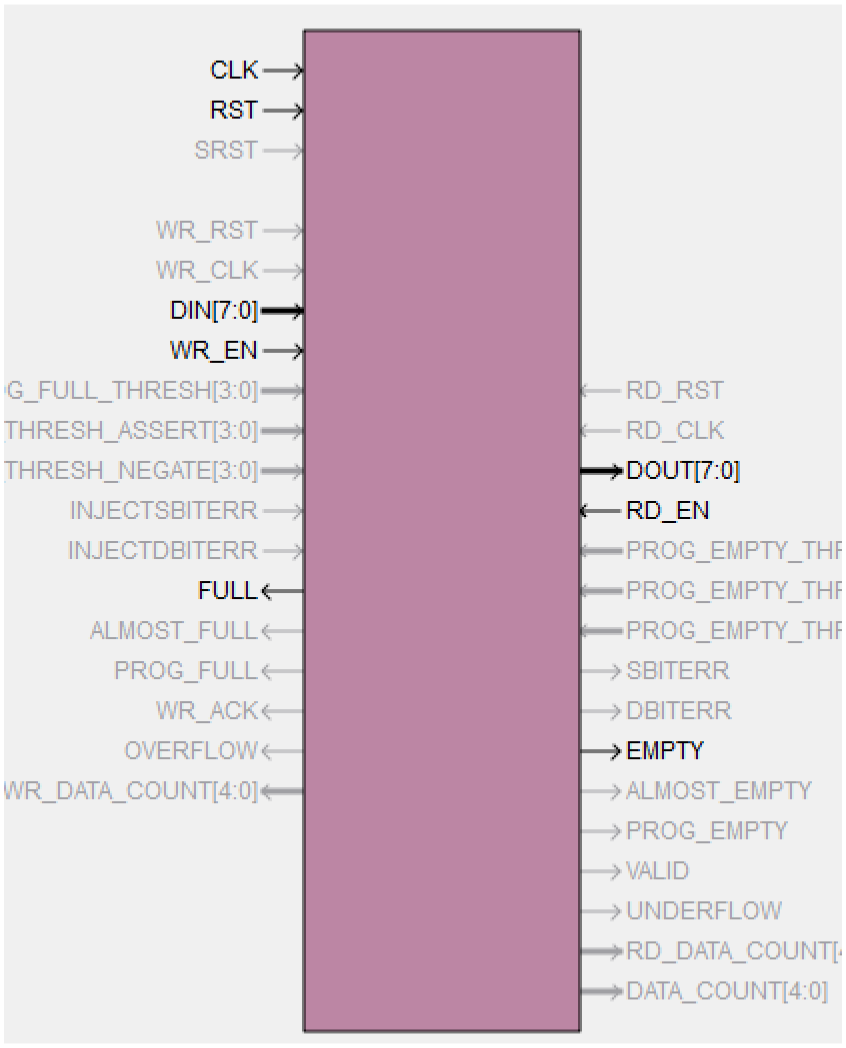
Data Port Parameters

Write Width 8 Range: 1,2,3..1024

Write Depth 16 Actual Write Depth: 18

For the remaining parameter pages, use all of the defaults.

Final interface looks like (bolded names are the actual ports that are created). The *RST* input is an asynchronous reset and should be tied to the *reset* input.


CLK
RST
SRST
WR_RST
WR_CLK
DIN[7:0]
WR_EN
G_FULL_THRESH[3:0]
THRESH_ASSERT[3:0]
THRESH_NEGATE[3:0]
INJECTSBITERR
INJECTDBITERR
FULL
ALMOST_FULL
PROG_FULL
WR_ACK
OVERFLOW
WR_DATA_COUNT[4:0]
RD_RST
RD_CLK
DOUT[7:0]
RD_EN
PROG_EMPTY_THF
PROG_EMPTY_THF
PROG_EMPTY_THF
SBITERR
DBITERR
EMPTY
ALMOST_EMPTY
PROG_EMPTY
VALID
UNDERFLOW
RD_DATA_COUNT[
DATA_COUNT[4:0]