

# Creating and simulating an AXI lite IP

---

## Goals

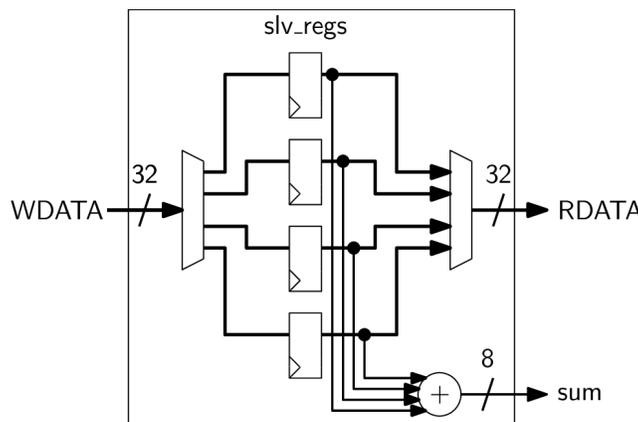
- Build a simple AXI-Lite Slave Peripheral
- Use low-level drivers to write to the IP
- Simulate and analyze a MicroBlaze system Vivado Simulator

## Requirements

- Xilinx Vivado software
- Xilinx SDK software
- Enough disk space for the project files

## Background

In this tutorial, we will create a simple AXI lite slave IP and simulate its behaviour in Vivado. The datapath of the IP is shown below:



It contains four slave registers that can be read or written to by an AXI master and a sum port. Sum is an 8-bit value that is the result of adding the low byte of all four slave registers.

## Creating the AXI Lite IP

Start Vivado and select Manage IP > New IP Location... under the Tasks heading. Select the xc7a100tcsq324-1 part and choose Verilog for both the Target and Simulator languages.

There is a bug in Vivado 2014.1 with the IP Packager, to work around this select:

Tools > Project Settings > IP and open the Packager tab

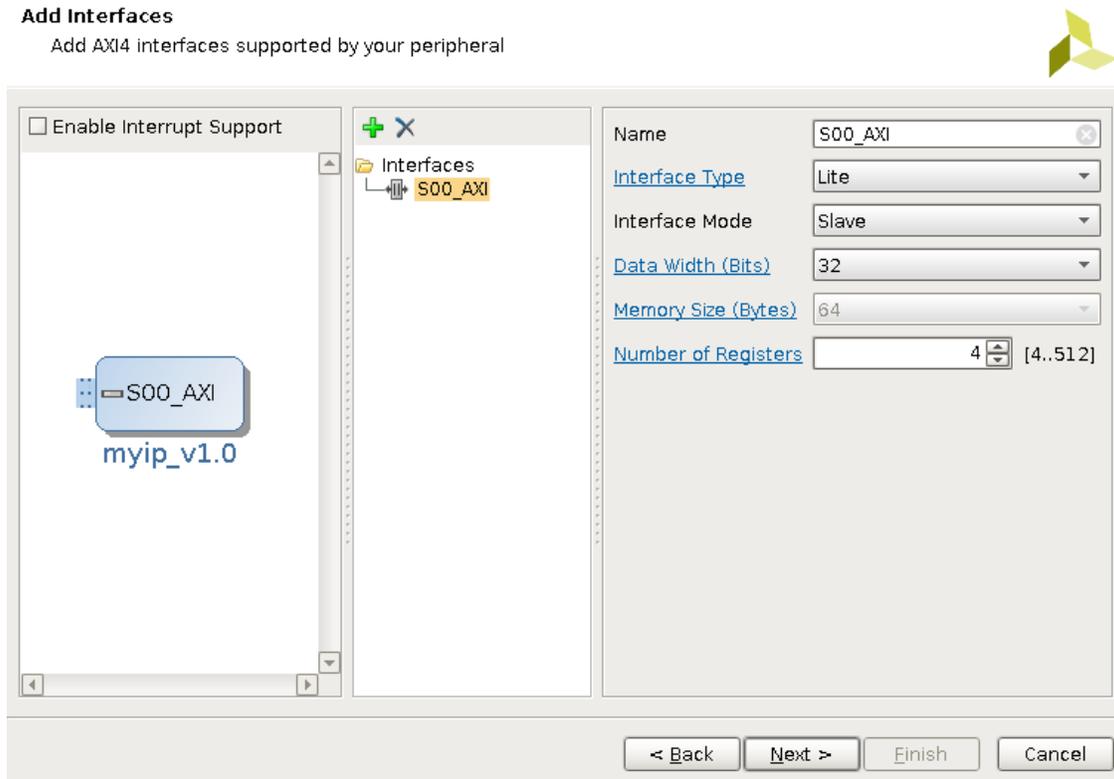
Change “(none)” in the Vendor field to “user”

We will use the wizard provided by Xilinx to create a skeleton IP, select:

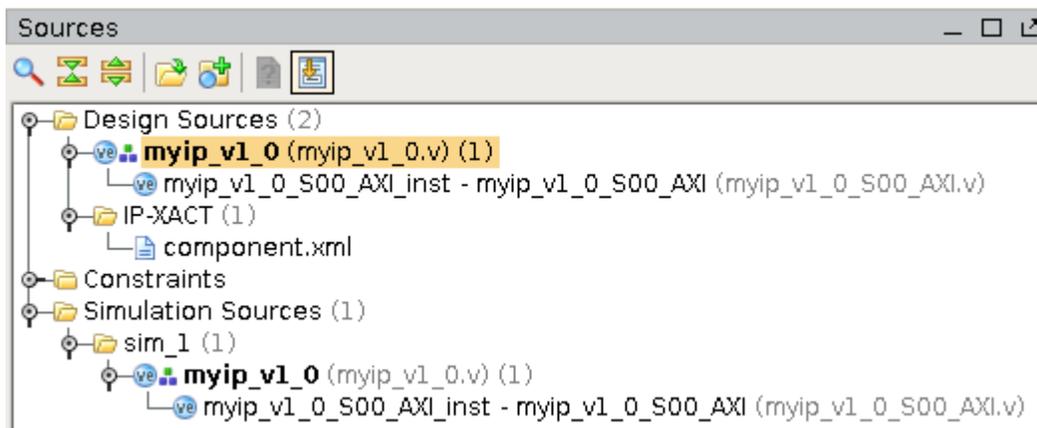
Tools > Create and Package IP...

Create a new AXI4 peripheral and give it a name, we will keep the defaults “myip”.

Click next and keep the default as shown to create an IP with one AXI Lite slave.



Under Next Steps: select “Edit IP”. This will open a new Vivado project for the IP. Under sources you will see both the Verilog source and a component.xml that describes your IP in IP-XACT format that the IP Integrator can understand.



There are two sources, a top level “myip\_v1\_0.v” and an interface module “my\_ip\_v1\_0\_S00\_AXI”. For this design the top-level module just wraps the S00\_AXI module.

# Understanding the AXI Lite interface

The template core implements a simple AXI lite slave interface, the reader is encouraged to delve into the code in more detail. Here we will describe a few basics of the AXI interface.

The AXI protocol consists of five channels: **read address (AR)**, **read data (R)**, **write address (AW)**, **write data (W)** and **write response (B)**, the Verilog signals are prefixed with the parenthesized characters. In general all channels communicate using READY/VALID handshaking. Only when the sender asserts and valid signal and the receiver asserts a ready signal can data flow.

## Write Operations

Write operations use the write address, write data and write response channels. Since this core is a slave, the address and data are inputs given by the master telling the slave where and what to write. The response channel is an output for acknowledging the transfer.

Look for the assignment of the AWREADY, WREADY and AWADDR (write address data) signals. This implementation sets the ready signals high and gets the address when both address and data channels are valid. In the next cycle, the data to be written (WDATA) is stored in the addressed slave register and BVALID is asserted.

## Read Operations

Read operations only use an address and data channel. Like the write channel, the address is registered one cycle before the data is transferred. However, in this case the data is an output from our slave containing the value of the addressed register. The logic for read operations is below the write operations, starting at line 357.

# Modifying the AXI Lite IP

The skeleton core handles reading and writing the slave registers. We will now insert an adder to produce the **sum** output.

1. Modify the S00\_AXI Module:

Add a sum port to the module

```
// Users to add ports here  
output [7: 0] sum,  
// User ports ends
```

Create the sum value by adding the lower eight bits of each slave register:

*// Add user logic here*

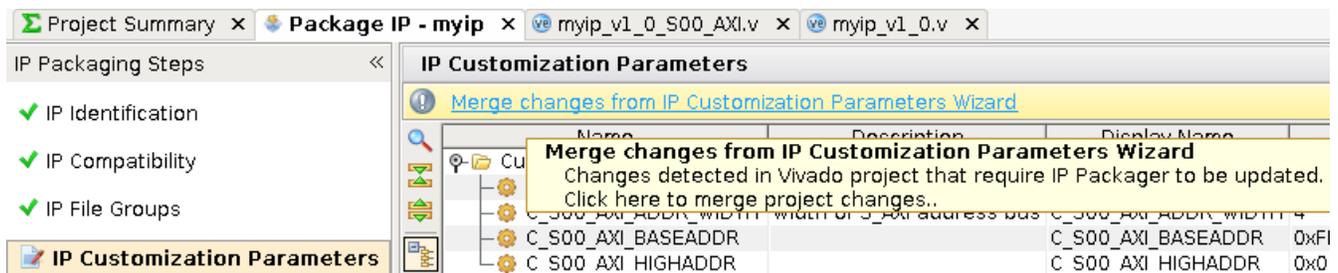
```
assign sum = slv_reg0[7:0] + slv_reg1[7:0] + slv_reg2[7:0] + slv_reg3[7:0];
```

*// User logic ends*

2. Modify the top level module to carry the sum output as an external port. This is done by adding a top level sum output and connecting it to the S00\_AXI instance sum port.

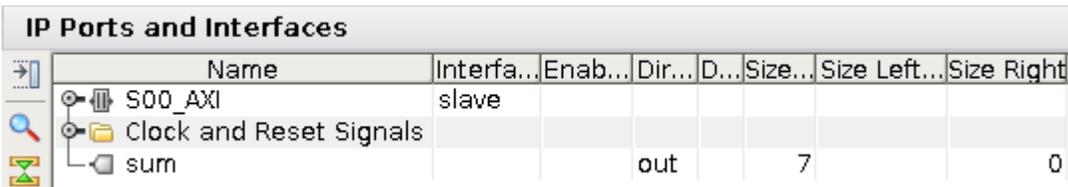
## Package the IP

Select the Package IP tab. The packaging wizard detects changes to the top level module changing the icon for a number of steps from a green check mark to . To automatically merge the changes, select an IP Packaging Step with an edited icon and click the Merge changes from IP Customization Parameters Wizard.



You will now see the sum port added to the IP description in the IP Ports and Interfaces step.

Examine the rest of the



Name	Interfa...	Enab...	Dir...	D...	Size...	Size Left...	Size Right
S00_AXI	slave						
Clock and Reset Signals							
sum			out		7		0

Packaging Steps then on the Review and Package step click Re-Package IP.

## Integrating the IP

File > Close Project to return back to the Vivado start screen

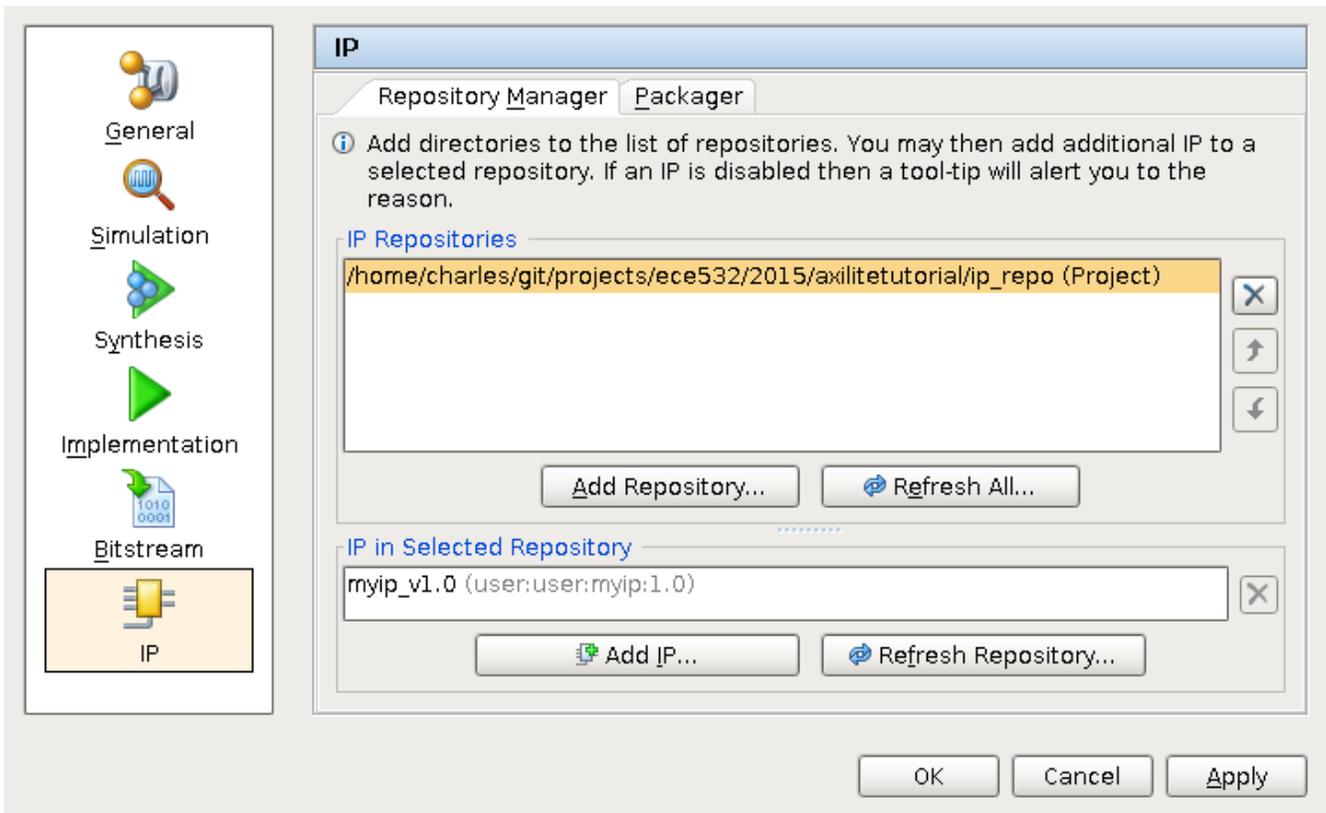
Start a new project selecting the xc7a100tcsg324-1 part

Open the IP Settings by following:

Tools > Project Settings > IP

In the Repository Manager tab click "Add Repository..." and find the ip\_repo folder. This should be located where you created the IP location in the previous steps.

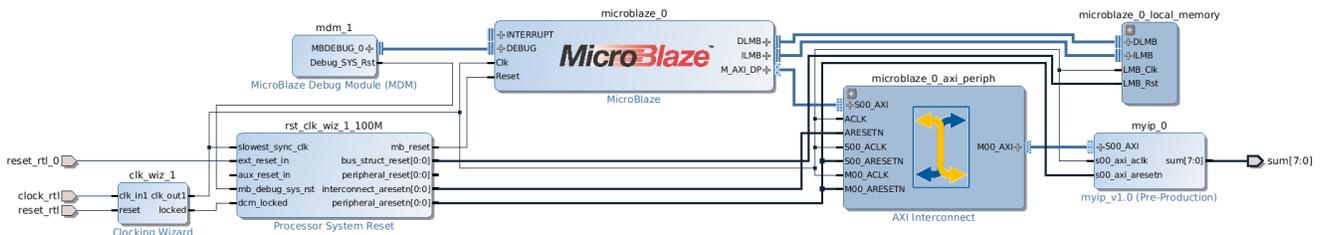
You now should see your IP appear in the IP in Selected Repository box



Press Ok.

Your IP can now be used in a Block Design through the Add IP dialog.

Create a new Block Design with a MicroBlaze and your IP. Use the default block automation on the MicroBlaze and set the clock wizard input to single ended as in the MicroBlaze tutorial. Use connection automation on all ports and finally make sum[7:0] on your IP block external. You should have a design similar to:



Validate your design to ensure there are no errors.

Create an HDL Wrapper and click Generate Block Design under IP Integrator.

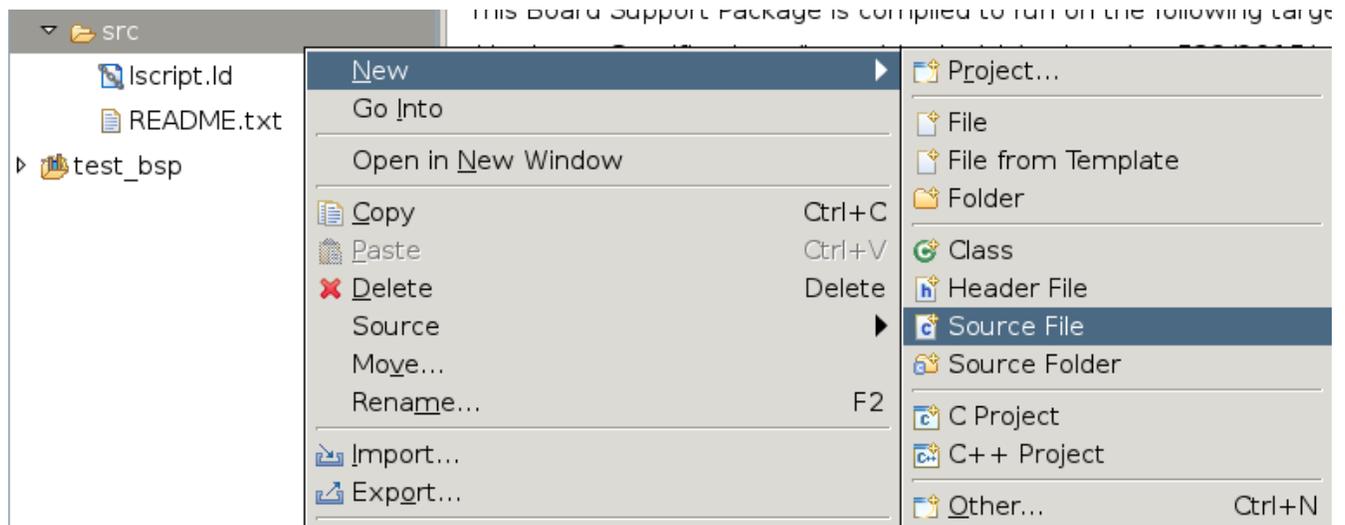
## MicroBlaze Code

With the system generated, we now need the MicroBlaze to control our slave peripheral. Export the design to SDK:

File > Export > Export to SDK and select Launch SDK

With SDK open, create a new Empty Application, this example will use the name “test” for the application.

Since this is an empty application, we will need to add source files to it. Expand the test application in the Project Explorer, right click the src folder and select New > Source File



Give it a name, for instance main.c

Xilinx automatically creates and packages **low level drivers** for our IP. They are compiled in the board support package (BSP). These drivers wrap reading and writing the slave registers in convenience functions **mReadReg** and **mWriteReg**. Expand the BSP for your application and find the header and source files associated with your IP.

The example program below uses the low level driver for the IP “myip” to write 0xDEADBEEF into the first slave register and a pointer to directly write 0xBAADF00D into slave register 1.

```
#include "xparameters.h"
#include "xil_io.h"
#include "myip.h"

volatile unsigned int *myip_ptr = (unsigned int*) XPAR_MYIP_0_S00_AXI_BASEADDR;

int main()
{
    // Low Level Driver
    MYIP_mWriteReg(XPAR_MYIP_0_S00_AXI_BASEADDR, 0, 0xDEADBEEF);

    // Direct Access
    *(myip_ptr + 1) = 0xBAADF00D;

    return 0;
}
```

Save the code into your source file, modifying the name of the IP as necessary. This will compile the program into an executable in your project SDK folder.

## System Simulation

Now that we have a program to drive our IP, we will use it in simulation. Return to the Vivado GUI and select

Tools > Associate ELF Files...

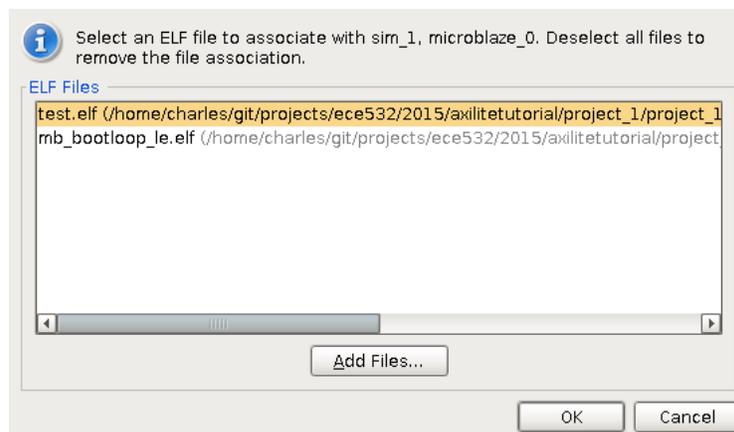


Click the ... button under Simulation Sources to associate a new executable.

In the new window click Add Files... and browse to your SDK folder, it will be located in your Vivado project, for instance project\_1.sdk. Your ELF file will be in a path similar to

<project>.sdk/SDK/SDK\_Export/<application>/Debug/<application>.elf

Highlight the elf file and click OK



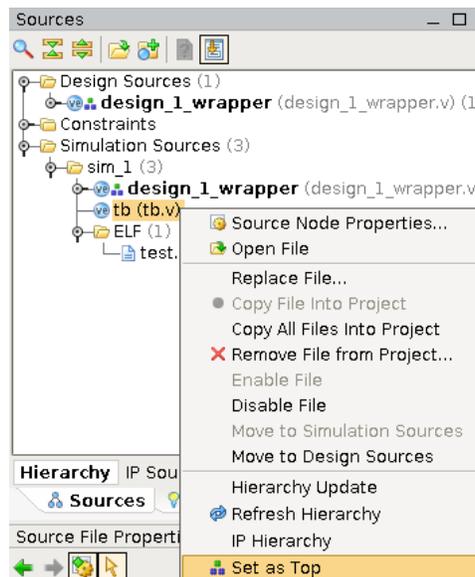
To simulate our system we will need a testbench to drive the external ports. First, add a new Verilog file to contain the testbench:

Project Manager > Add Sources > Add or Create Simulation Sources > Create File...

Name the file tb.v

In the Module Definition dialog, leave the I/O Port Definitions empty and click Ok

The new tb.v will appear in the Sources window next to your HDL wrapper. Since the testbench wraps and drives our design, we want to make it the top-level module. Right click the tb module and select Set as Top



Next we need to modify our test bench to drive our system. Recall that our design has just four external ports: one clock, two resets and the sum port. Open your HDL wrapper to examine the ports.

The testbench below instantiates the design, creates a 100MHz clock and resets the system. Make sure you understand how it works.

```
`timescale 1ns / 1ps
```

```
module tb();
```

```
reg sys_clk;
```

```
reg reset;
```

```
wire [7:0] sum;
```

```
// Instantiate DUT
```

```
design_1_wrapper dut
```

```
  (.clock_rtl(sys_clk),    // system clock
```

```
   .reset_rtl(reset),    // active high
```

```
.reset_rtl_0(~reset), // active low
.sum(sum)); // sum output
```

```
always #5 sys_clk = ~sys_clk;
```

**initial**

**begin**

```
sys_clk = 1'b0;
```

```
reset = 1'b1;
```

```
#45
```

```
reset = 1'b0;
```

**end**

**endmodule**

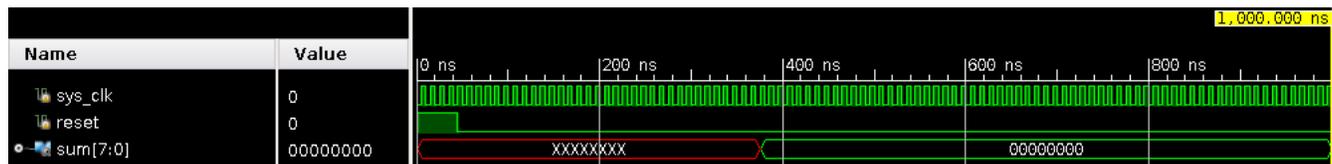
The testbench assumes you used the default reset settings during connection automation. In other words active high reset\_rtl and active low reset\_rtl\_0.

Finally, run simulation:

Simulation > Run Simulation > Run Behavioural Simulation.

This step may take some time to compile the simulation libraries.

You should see the following waveform after pressing Zoom Fit :

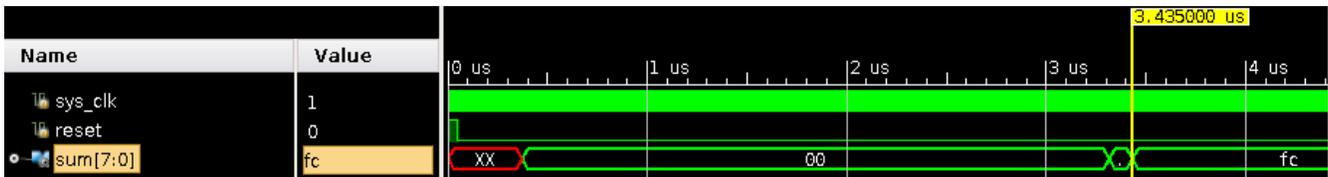


We can see that the clock is running and reset is deasserted at 45ns. However, the value of sum is not correct. It should be the addition of the bottom 8 bits of: 0xDEADBEEF and 0xBAADF00D or 0xEF + 0x0D which is 0xFC.

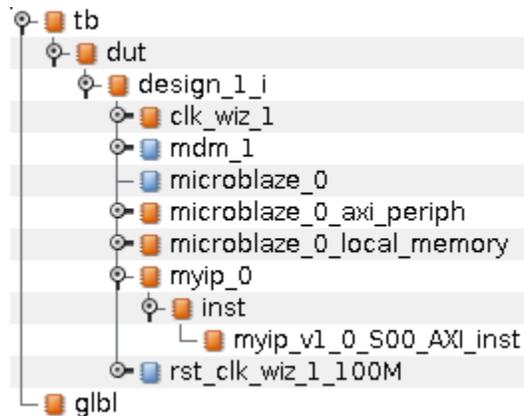
The default simulation time is 1us, which is too short for the MicroBlaze to complete execution of the program we wrote. At the top of the window, change the time units to 4 us and click continue simulation:



Pressing Zoom Fit  will show that the correct value appears after about 3.4us



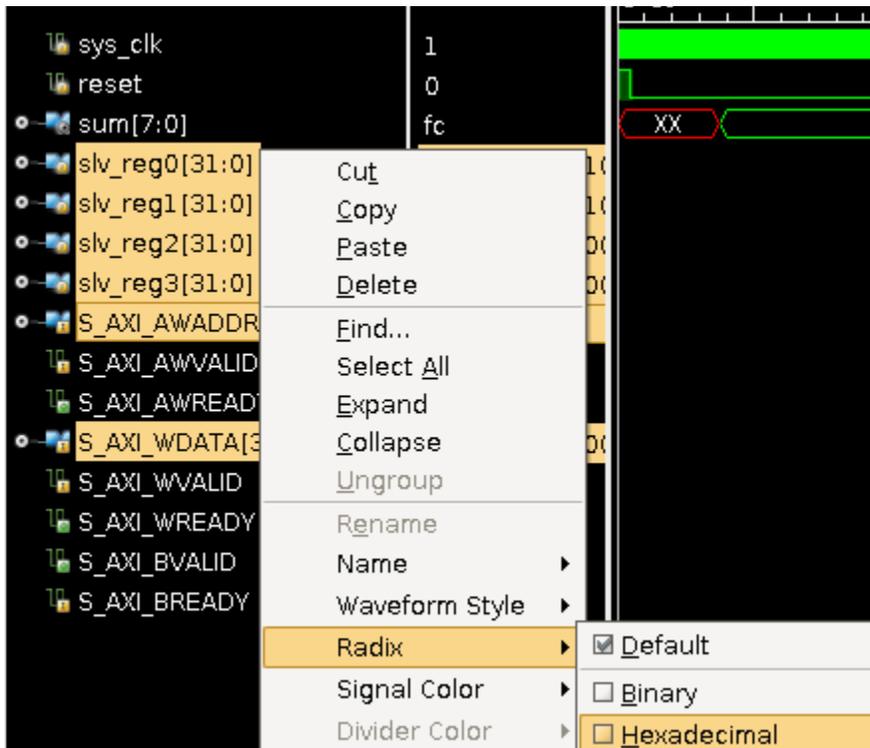
To investigate a little further, let's probe some of the internals of our IP. In the Scopes window, expand the dut and find the S00\_AXI module of your IP:



In the Objects window you will see the signals that you can probe within the module. Add the four slv\_regs, AWADDR, WDATA as well as the READY and VALID signals for AW, W and B channels, by selecting them and dragging them into the waveform. If you have trouble adding signals to the waveform with error [Wavedata 42-43], try increasing the trace limit in the Tcl Console:

```
set_property trace_limit 65535 [current_sim]
```

Change the radix of the slv\_regs, AWADDR and WDATA to hexadecimal by right clicking on those signals and selecting Radix > Hexidecimal



Finally, to see the signals click Restart  and Run . In the screenshot below of the two write transactions, the AXI channels have been colour coded.



Try to follow the sequence of events, looking at the Verilog code for the IP, why does BVALID come up a cycle after WREADY and AWREADY are asserted?

## Running in Hardware

To run the system in hardware, assign pins to the four ports of the block design. For instance, assign the sum port to the LEDs and run through to implementation.