Electronic System Design with FPGAs Coursework 2(ELC054)

Abstract- **This research paper contains three circuits. The cnt1_wrap, mac_wrap and bram_wrap, this following circuits are detailed in this report showing their microarchitecture and SoC-FPGA implementation on Vivado. The SoC-Fpga Implementation shows in detail the design flow followed & timing constraints. This circuits highlight the master and slave link which is used in the AXI4 control interface and its intended to evaluate the Hardware utilization, timing constraints and amount of power used at different clock speeds(PS). Also it includes the analysis of the design simulation validation using QuestaSim and Zedboard validation using Xilinx SDK, showing the modifications of the peripheral drivers as well for the zedboard validation.**

Index Terms—**Accumulates, algorithm, Bitstream, configurations, correspondence, Field Programmable gate array(FPGA),Gnome-terminal, handshaking ,implementation, microarchitecture, parametric, peripheral, protocols, PS-PL, Random access memory(RAM), Register Transfer Level (RTL), Sequential, Silicon, Testbench, Utilization, Verification, Vivado, Xilinx.**

I. INTRODUCTION

Processors and FPGAs are the hardworking cores of most embedded systems. Integrating the high-level management functionality of processors and the stringent, real-time operations, extreme data processing, or interface functions of an FPGA into a single device forms an even more powerful embedded computing platform.[1]SoC FPGAs provide a familiar processing system, that is the cpu to execute sequential processing algorithms[2], they give higher integration, less power, smaller board size, and higher transmission capacity correspondence between the processor and FPGA. They likewise incorporate a rich set of peripherals, on-chip memory, a FPGA-style logic array, and fast transceivers[1] .This new type of programmable devices abuses exploits the flexibility of the FPGA engineering, to keep pace with changing measures and end-client requests, joined with the administration usefulness of a processor, to meet execution desires, without risk penalties.

The design flow used to create these circuits is as follows.

- A directory is created for the project
- Created a baseline system called base design
- The RTL project is chosen and then the required zedboard (Zedboard Zynq evaluation & development kit is selected)
- The vivado cockpit is gotten into
- Project parameters are set
- A block diagram for the circuit is created and silicon IP added.
- Clock configurations are set for Zynq hardened ARM system after the block automation is ran
- Design is validated and checked for errors

  The target device used is the Zync-7020 Soc-FPGA from Xilinx. It features a Dual-Core ARM cortex-A9 MP Core PS operating at a max of 866 MHz It has 85K logic cells, 53,200 LUTs, 106,400 Flip -Flops, 220 DSP slices and a 4.9 Megabyte Block Ram[9].

  The design flow showed which targets this device is done in the Vivado Design suite and XSDK. Vivado being a user friendly software package that supports many features for example, auto mapping memory for AXI4 blocks. Also it provides a tool for block diagram validation, TCL console and debugging information through the Vivado logs. lastly it also gives warning and errors messages.

II. MICROARCHITECTURE

Microarchitecture is the internal organization of a design. The three circuits in this report all use and require the hardened ARM cpu /AXI4 based sub system. This is shown as each circuit is integrated to the base design and reference system.

Cnt1_wrap. The cnt1_wrap shows the count_no_of_ones integrated with the axi4lite, this is created as a hierarchy and given a different name(cnt1_wrap_axi4) and is connected to the base system consisting of the ZYNQ7 processing system (processing_system7_0), axi_interconnect and the processor system reset. The design consists of a count no of ones and a programmer's version of the design to interface the AXI4LITE slave regsif. The design provides a read/ write interface to the cnt1 connected to the regsif. This consists of the clk, reset,1-bit read strobe (rd.), 1bit write strobe, 32 bit write data, 32-bit read data , 8 bit address. The AXI4 lite has five channels which make up the specification; the read address channel, the write address channel, the read data channel, the write data channel, and the write acknowledge channel. The read address channel allows the processor to signal that it wishes to initiate a read transaction. This channel carries addressing information and some handshaking signals. The read data channel carries the data values that are transferred during a transaction, along with their associated handshaking signals. Together, these two channels contain everything that is needed for a successful read transaction.

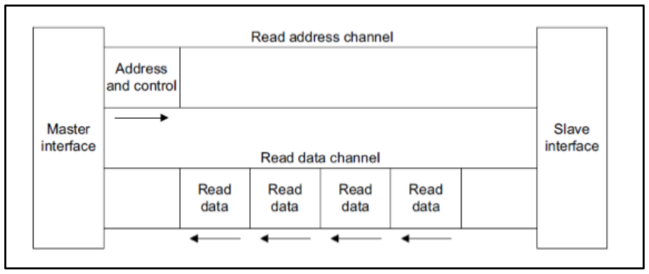


Fig 1. Read address channel diagram. [2]

The write address channel allows the processor to signal that it wishes to initiate a write transaction. The write data channel performs an equivalent function to the read data channel, except that the data flows towards the slave. For write transactions an additional channel is used in the form of the Write Response Channel. This last channel is used to allow the slave peripheral to acknowledge receipt of the data, or to signal that an error has occurred.

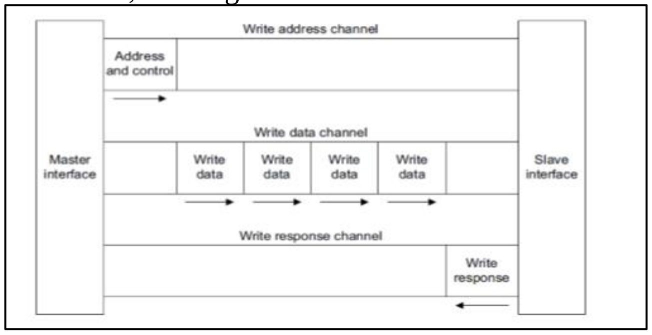


Fig 2. Write address channel diagram. [2]

The handshaking signals are consistent across the five channels, and offer the user a simple yet powerful way to control all read and write transactions. The signals are based on a simple "Ready" and "Valid" principle; "Ready" is used by the recipient to indicate that it is ready to accept a transfer of a data or address value, and "Valid" is used to clarify that the data (or address) provided on that channel by the sender is valid so that the recipient can then sample it.

The signals involved in both the read and write transactions. The axi4lite_slave_regsif links the axi4lite protocol to drive these connection configuration .This mechanism consists of the write address signals; AWADDR[width-1:0] – write address bus giving write transaction address, AWVALID – indicates the write address is available, AWREADY – output indicating the slave is ready to accept an address, the write channel signals; WDATA[width-1:0] – write data, AWVALID – indicates valid write data and write strobes available, AWREADY – indicates slave can accept data, the write response channel signals. The read address channel signals; ARADDR[width-1:0] – address of read transaction, ARVALID – indicates read address is valid and remains stable until ARREADY is set HIGH, ARREADY – read address ready indicates slave can accept address, and lastly the read data channel signals;, ARVALID – read data is available and read transfer can complete, ARREADY – read ready indicates the master can accept the read data and response information. See appendix A & B for full diagram on the signals involved in this transaction.

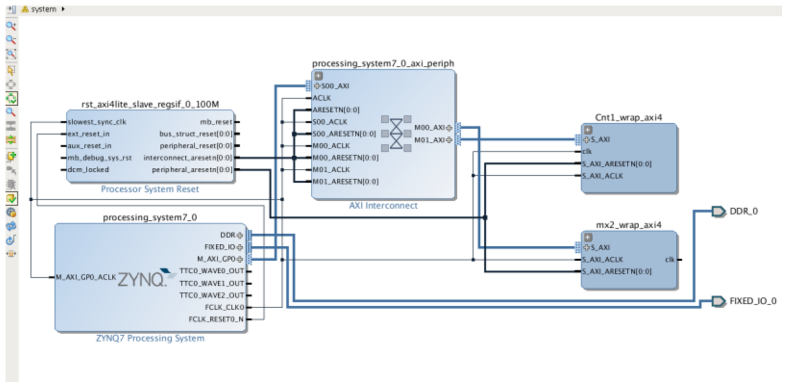


Fig 3. Cnt1_wrap design schematic(AXI4LITE SLAVE REGSIF AND CNT1_WRAP integrated into reference system).

This design is shown with the mx2_wrap as each integration with the slave regsif for the three circuits are all linked and connected via the Clk and Reset Signals to the mx2_wrap. In The further circuit diagrams shown subsequently, the links would also be shown.

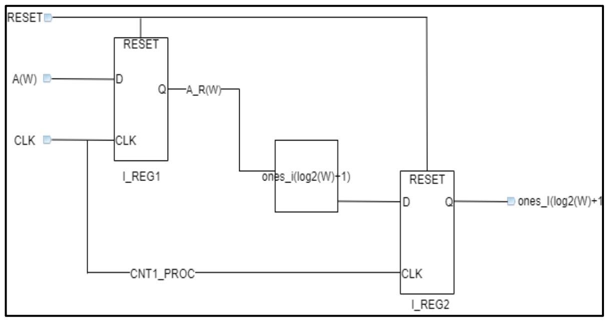


Fig 4. design schematic for the cnt1[10].

This Consists of a parametric input register, a count block and a parametric output register[10]. As said earlier this design provides a read/ write interface to the cnt1 connected to the regsif. Below shown is the top level schematic. This schematic is similar for every other circuit.
The high level view of the axi4 induced with the cnt1_wrap design interface is shown below in fig5.

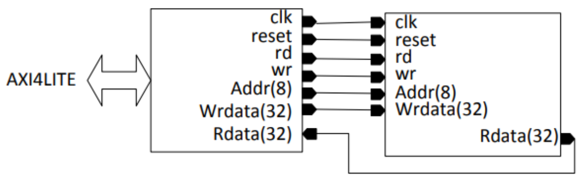


Fig 5. High level view of the cnt1_wrap_axi4(Vassilios chouliaras Zedboard experiment book,2017).

A testbench "tb_cnt1_wrap" was created for the verification of the design. This was shown in the RTL simulation and validation. The following traces for the verification where gotten and is shown in Modelsim.

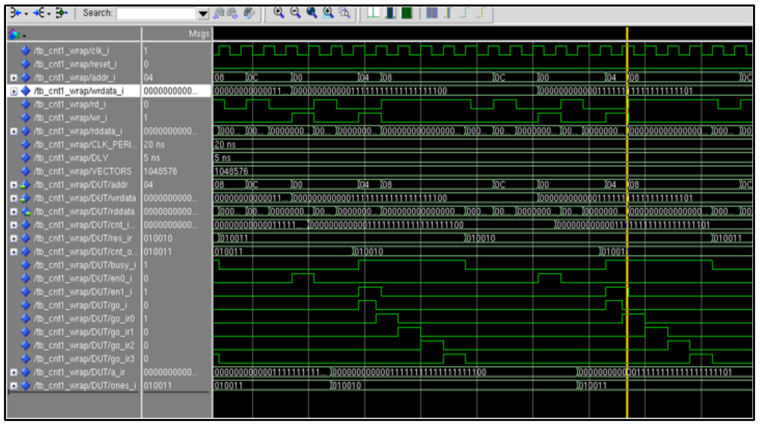


Fig 6. Cnt1_wrap timing diagram(Testbench).

The modelsim waveform which is shown in fig 6 verifies and shows that the cnt1_wrap functions as expected. The designs needed to be packaged using TCL scripts, for vivado to operate with them. The TCL scripts used to package them are shown in Appendix C. This scripts where compiled as well as the .vhd files in the gnome terminal using the command "vsim -c do scripts/compile.do".

After the design was verified the bitstream was generated in vivado and the implemented design showed two levels of hierarchy being the (axi4lite slave regsif and cnt1_wrap). The hierarchies colour coded are shown below. The cnt1_wrap colour code red and the axi4lite slave regsif colour coded purple.

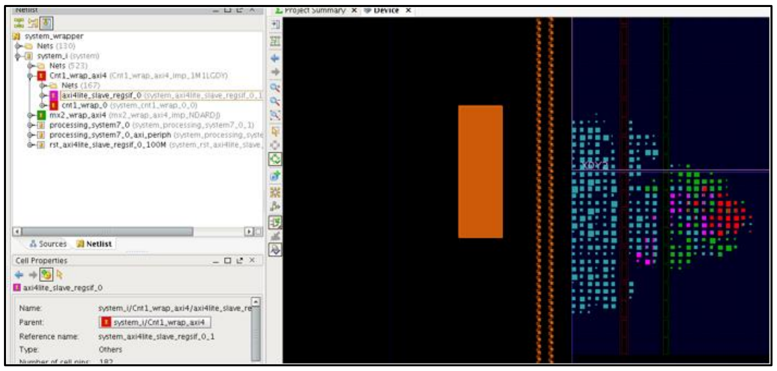


Fig 6. CNT1 device layout(Axi4lite slave regsif combined with cnt1_wrap).
The following colour coded small boxes which are seen are from the fpga editor tool and are called slices. Slices are what make up a configurable logic block. One CLB consists of 4 slices. A **s**lice contains a set number of LUTs, flip-flops and multiplexers. A LUT is a collection of logic gates hard-wired on the FPGA. [11].

Mac_wrap. The mac_wrap is also connected to the axi4lite_slave_regsif and put in a hierarchy. It uses the same method of protocols on how they are linked as the cnt1_wrap. The design provides a read/ write interface to the mac connected to the regsif. The same AXI4LITE channels and signals are used for the three circuits, although as the design interfaces are different. That is this multiplies and accumulates, and is a three-input operations in which the first two operands are multiplied together and the accumulator register is added to the product[10].The verification produces a different timing simulation but is as expected.

This peripheral combines the axi4liteslave regsif with the mac_wrap, this is then connected to the AXI interconnect by creating another master interface on it. The hierarchy mac_wrap_axi4 is then linked to cnt1_wrap and mx2_wrap via the Clk signal on the mac_wrap and reset signal from the axi4lite slave regsif. The diagram below in Fig 7 shows the MAC units instantiation on the reference system.

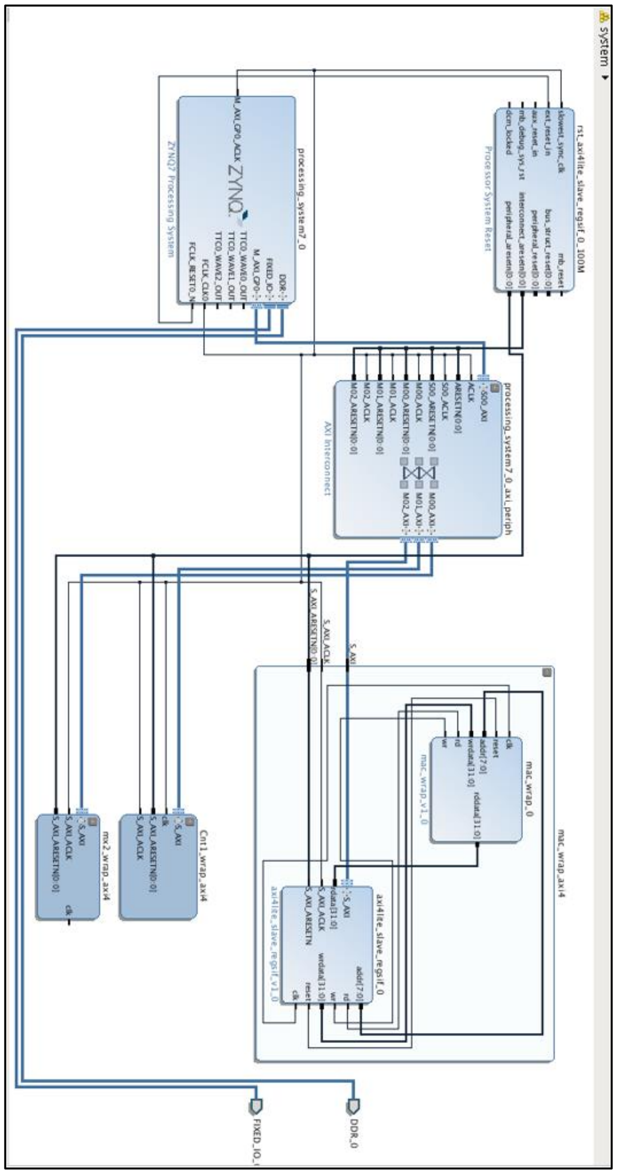


Fig 7.Mac_wrap & AXI4LITE_SLAVE_REGSIF integrated in base design.

A testbench "tb_mac_wrap" was created for the verification of the design. This was shown in the RTL simulation and validation. The modelsim waveform verified and shows that the mac_wrap functions as expected. Also the designs where packaged using TCL scripts, for vivado to operate with them as well. The TCL scripts used to package them are shown in Appendix D. The modelsim wave form is shown below. The simulation was ran using the command vsim"run -step", then run"1000ns".

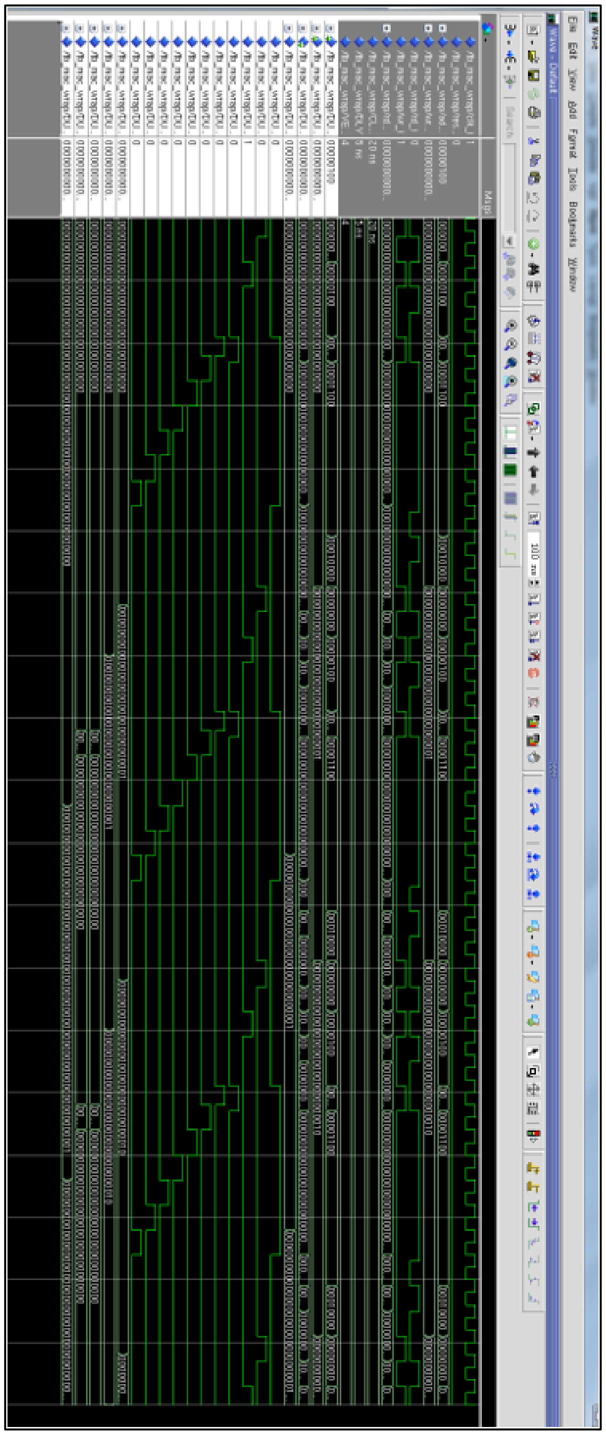


Fig 8. Mac_wrap Verification and timing diagram.

The design which was integrated was assigned and addresses and verified. The bitstream was then generated in vivado and the implemented design showed the hierarchy. The hierarchies colour coded are shown below.

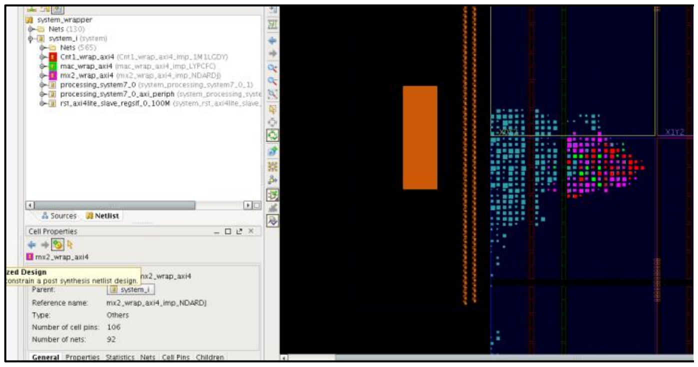


Fig 9. MAC_wrap device layout(colour code green).

Bram_wrap. The single port RAM can be thought as 2D arrays and are used for dense storage in multitude of applications for example register files in embedded processors[10]. The Bram_wrap peripheral is the last circuit looked at in this report and it also connected to the axi4lite_slave_regsif. Its put in hierarchy with the axi4lite slave regsif and then called(bram_wrap_axi4). Another Master interface is further configured to be on the Axi interconnect, the bram_wrap hierarchy is then connected to the base design from the axi4lite_slave_regsif. Below is the Bram_wrap shown integrated to the base design system.

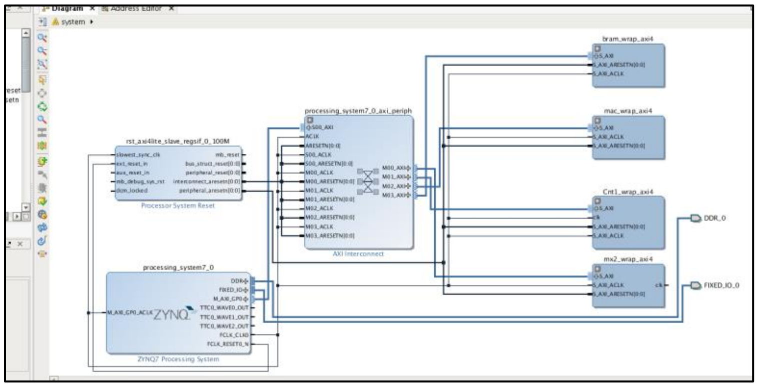


Fig 10. AXI4LITE_SLAVE_REGSIF & Bram_Wrap integrated with base design.

The diagram below shows the further detail of the hierarchy.

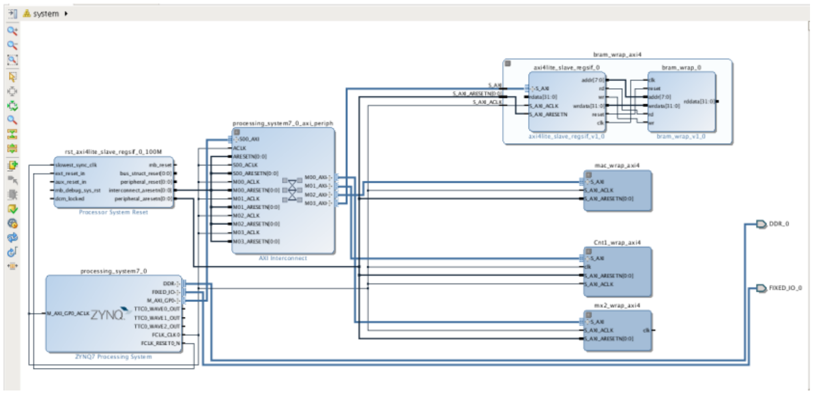


Fig 12. Bram_wrap integrated to base design(Shows inside hierarchy).

Finally the bram_wrap hierarchy is then connected to the rest of the peripherals using the clk signal from the bram_wrap and reset signal from the axi4lite_slave _regsif. A testbench is created called “tb_bram_wrap” and is used to run an RTL simulation and verifies that the bram_wrap works likes it supposed to. The modelsim waveform which is shown below in fig 11further shows that the bram_wrap functions as expected.

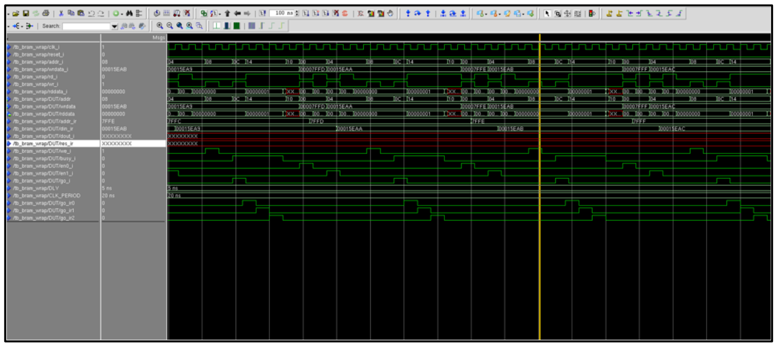


Fig 11. Bram verification campaign.

The design was then post placed and routed. As the bitstream was then generated in vivado and the implemented design showed the hierarchy and BRAM primitives used. The hierarchies colour coded are shown below.

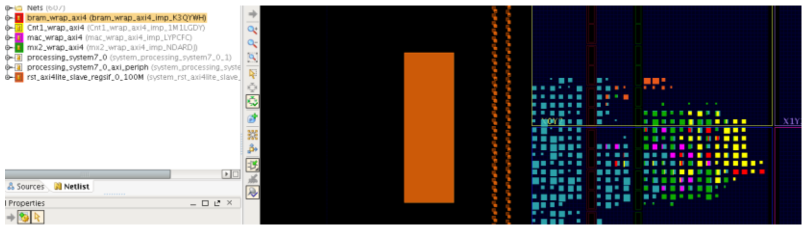


Fig 12. Bram_Wrap device layout.(Colour code red).

III. SOC FPGA IMPLEMENTATION

The fpga flow which has been followed is shown in the diagram below.

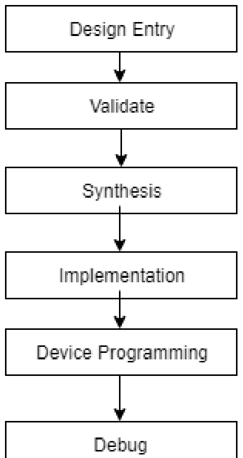


Fig 13. FPGA design Flow followed.

The circuits are designed in the vivado cockpit. Then the PL clock is set to either 50,100 or 200 MHZ. the completed circuit design is now validated and checked for errors after the base address has been set. The bitstream of the design would now be generated and the implemented design is shown on the layout. After the bitstream and hardware are now exported to the computer, Using WinSCP it is copied from the project directory to a new folder on the computer(EXPORT). Then on the Xilinx SDK an application is created using the exported (. hdf and .bit)files. That is, the hardware and bit stream. The FPGA is programmed and then the peripheral is debugged using a serial connection to PUTTY.

Each circuit had a report for the timing of the circuit after the flow had been completed for them and had certain other resources and reports such as Power utilization. The Tables below show the report on circuit timings, power report and Device utilization.

The **Cnt1_wrap** report on Hardware Utilization and circuit timing.

Parameter	50MHZ	100MHZ	200MHZ
Total on chip Power(w)	1.686	1.694	1.707
Slice LUTs	1.17%	1.56%	1.72%
Slice Registers	0.67%	0.92%	0.92%
RAMB18	0.00%	0	0
RAM36/FIFO	0.0%	0	0
DSps	0.00%	0	0

Table1. Hardware Utilization at 50, 100 and 200 MHZ.

A slice contains a set number of LUTs, flip-flops and multiplexers. The LUT is a collection of logic gates hard-wired on the FPGA. LUTs store a predefined list of outputs for every combination of inputs and provide a fast way to retrieve the output of a logic operation.[5]

FIFO (First in , First out)can be made of registers or BRAM, a register based fifo is small and a BRAM is big, they have width & depth and are synchronous as well. FiFos are used for crossing block domain, Buffering data before sending it to a chip and also storing data for later processing.

Parameter	50MHZ	100MHZ	200MHZ
Worst negative Slack(WNS) ns	13.685	2.392	-0.682
Total Negative Slack(TNS) ns	0	0	-48.501
TNS failing Endpoints	0	0	166
TNS Total Endpoints	1300	2248	2248
WHS(ns)	0.020	0.050	0.061
THS(ns)	0.0	0	0
THS failing Endpoints	0	0	0
THS Total Endpoints	1300	2248	2248
WPWS(ns)	9.020	4.020	1.520
TPWS(ns)	0.0	0	0
TPWS failing End points	0	0	0
TPWS Total Endpoints	793	1059	1059

Table 2. Cnt1_wrap Circuit timing at 50, 100 and 200MHz(Design timing summary).

The user specified timing constraint are met for both the 50 and 100MHz , but the 200 MHZ timing constraints aren’t met. Increasing the frequency for the clock makes it go faster. Which in the 200 MHz case below caused the design not to meet the timing constraints unlike the others.

The Key things in the timing summary WNS and TNS. WNS is the worst negative slack If this is positive then it means that the path passes. If it is negative, then it means the path fails. The "Total Negative Slack (TNS)" is the sum of the (real) negative slack in your design. If 0, then the design meets timing. If it is a positive number, then it means that there is negative slack in the design which means the design fails.[7] .Its shown from the table that the 200 MHz doesn’t meet the timing constraints as the WNS having a negative number tells that the path has failed.

The **mac_wrap** report on Hardware Utilization and circuit timing.

Parameter	50MHZ	100MHZ	200MHZ
Total on chip Power(w)	1.688	1.695	1.706
Slice LUTs	1.61%	1.61%	1.66%
Slice Registers	0.96%	0.96%	0.96%
RAMB18	0.00%	0	0
RAM36/FIFO	0.0%	0	0
DSps	0.00%	0	0

Table3. Hardware Utilization at 50, 100 and 200 MHZ.

Parameter	50MHZ	100MHZ	200MHZ
Worst negative Slack(WNS) ns	12.119	2.106	-0.751
Total Negative Slack(TNS) ns	0	0	-40.090
TNS failing Endpoints	0	0	145
TNS Total Endpoints	2323	2323	2323
WHS(ns)	0.081	0.101	0.029
THS(ns)	0.0	0	0
THS failing Endpoints	0	0	0
THS Total Endpoints	2323	2323	2323
WPWS(ns)	9.020	4.020	1.520
TPWS(ns)	0.0	0	0
TPWS failing End points	0	0	0
TPWS Total Endpoints	1097	1097	1097

Table 4. Mac_wrap Circuit timing at 50, 100 and 200MHz(Design timing summary).
Timing constraints are met for the 50 and 100 as well here and not the 200MHz for the same reasons as explained earlier.

The **Bram_wrap** report on Hardware Utilization and circuit timing.

Parameter	50MHZ	100MHZ	200MHZ
Total on chip Power(w)	1.689	1.695	1.709
Slice LUTs	1.78%	1.78%	1.83%
Slice Registers	1%	1%	1%
RAMB18	0.00%	0	0
RAM36/FIFO	0.0%	0	0
DSps	0.00%	0	0

Table5. Hardware Utilization at 50, 100 and 200 MHZ.

Parameter	50MHZ	100MHZ	200MHZ
Worst negative Slack(WNS) ns	1.046	12.119	-0.728
Total Negative Slack(TNS) ns	0	0	-23.625
TNS failing Endpoints	0	0	108
TNS Total Endpoints	3686	2323	2397
WHS(ns)	0.026	0.081	0.038
THS(ns)	0.0	0	0
THS failing Endpoints	0	0	0
THS Total Endpoints	3686	2323	2397
WPWS(ns)	4.020	9.020	1.520
TPWS(ns)	0.0	0	0
TPWS failing End points	0	0	0
TPWS Total Endpoints	1528	1097	1136

Table 6. Bram_wrap Circuit timing at 50, 100 and 200MHz(Design timing summary).
Timing constraints are met for the 50 and 100 as well here and not the 200MHz for the same reasons as explained earlier.

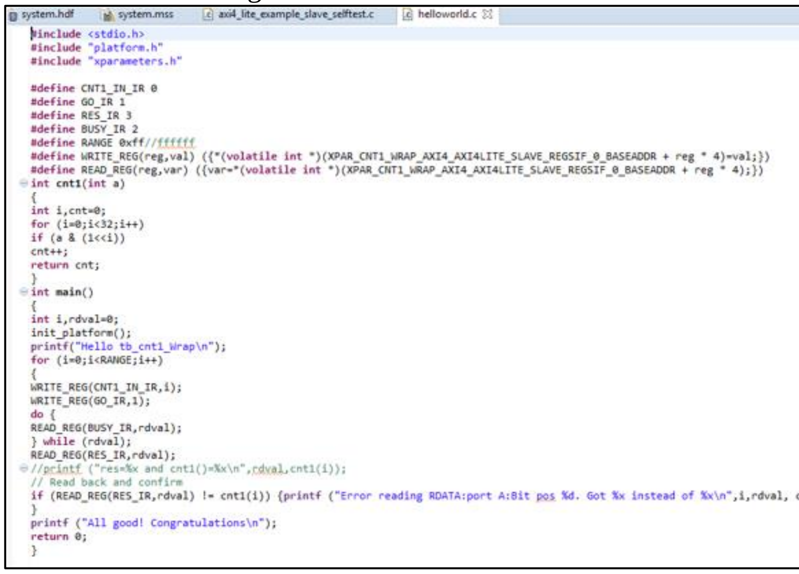
The timing constraints used for each circuit.

	Timing Constraints	
Circuits	Clock name	Period(ns)
Cnt1_wrap	Clk_fpga_0	10.000
Mac_wrap	Clk_fpga_0	20.000
Bram_wrap	Clk_fpga_0	20.000

Table 7. Timing constraints of circuits.

IV. ZEDBOARD VALIDATION

After Implementation of the designs, creating a top-level HDL wrapper and generating the bitstream. A system wrapper file was generated and was added to the hierarchy of the auxiliary pane. Once the bitstream was generated successfully the implemented design is opened. Then the bitstream and hardware are exported to XSDK and the FPGA programmed. A couple of C-application drivers are used to help interact with the hardware as well as peripheral drivers. Then the C code is modified to test the peripherals. The "testperipheral.c" file is then executed and debugged and the output is shown.
There are several drivers used for the different circuits. The **Cnt1_wrap.** A test software design is created which is shown below in Fig 14.

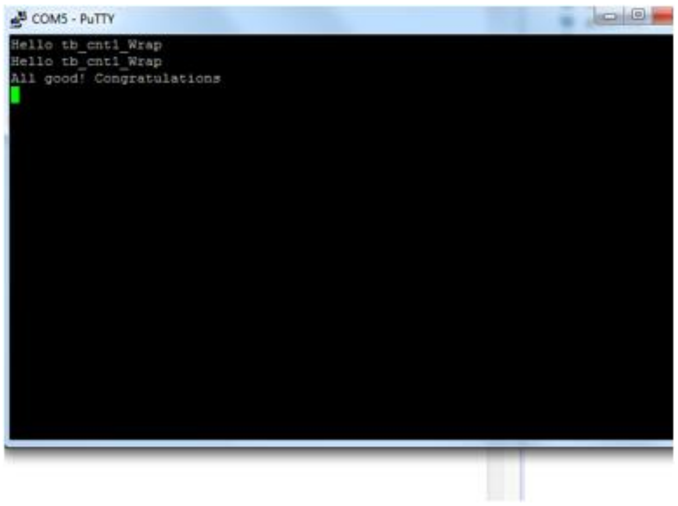


```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"

#define CNT1_IN_IR 0
#define GO_IR 1
#define RES_IR 3
#define BUSY_IR 2
#define RANGE 0xff//ffffff
#define WRITE_REG(reg,val) ({*(volatile int *)(XPAR_CNT1_WRAP_AXI4_AXI4LITE_SLAVE_REGSIF_0_BASEADDR + reg * 4)=val;})
#define READ_REG(reg,var) ({var=*(volatile int *)(XPAR_CNT1_WRAP_AXI4_AXI4LITE_SLAVE_REGSIF_0_BASEADDR + reg * 4);})
int cnt1(int a)
{
int i,cnt=0;
for (i=0;i<32;i++)
if (a & (1<<i))
cnt++;
return cnt;
}
int main()
{
int i,rdval=0;
init_platform();
printf("Hello tb_cnt1_Wrap\n");
for (i=0;i<RANGE;i++)
{
WRITE_REG(CNT1_IN_IR,i);
WRITE_REG(GO_IR,1);
do {
READ_REG(BUSY_IR,rdval);
} while (rdval);
READ_REG(RES_IR,rdval);
//printf ("res=%x and cnt1()=%x\n",rdval,cnt1(i));
// Read back and confirm
if (READ_REG(RES_IR,rdval) != cnt1(i)) {printf ("Error reading RDATA:port A:Bit pos %d. Got %x instead of %x\n",i,rdval, 
}
printf ("All good! Congratulations\n");
return 0;
}
```

Code 1. Cnt1_wrap C-level Testbench.

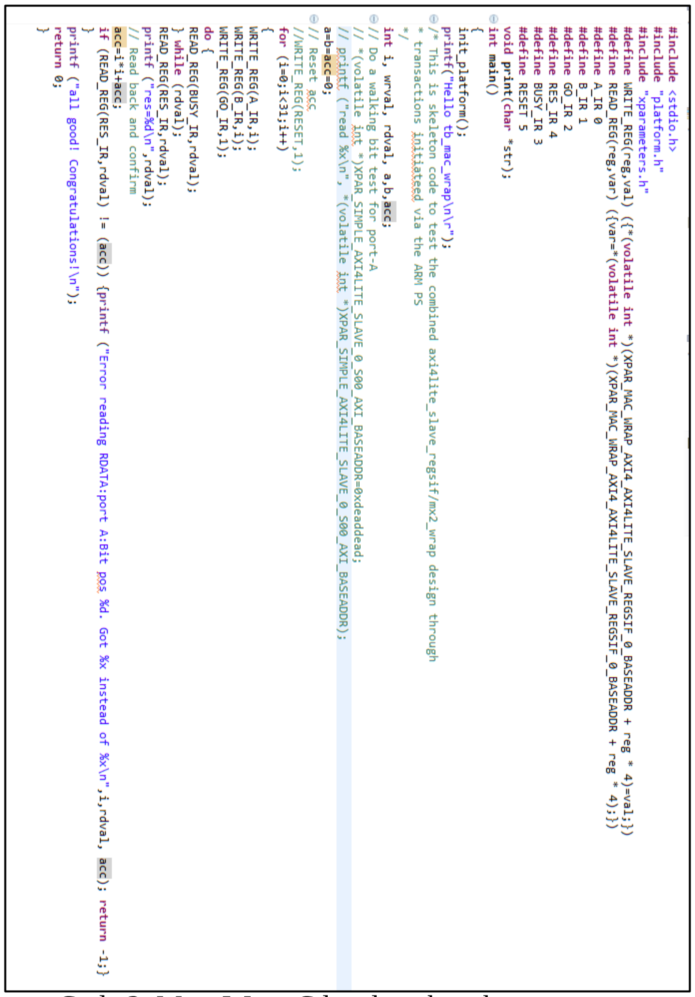
After the cnt1 test software design has been executed on silicon and the fpga has been programmed the following ouput is shown which shows correct execution of the design on silicon.



```
Hello tb_cnt1_Wrap
Hello tb_cnt1_Wrap
All good! Congratulations
```

Fig 15.The passed output of Cnt1_wrap.

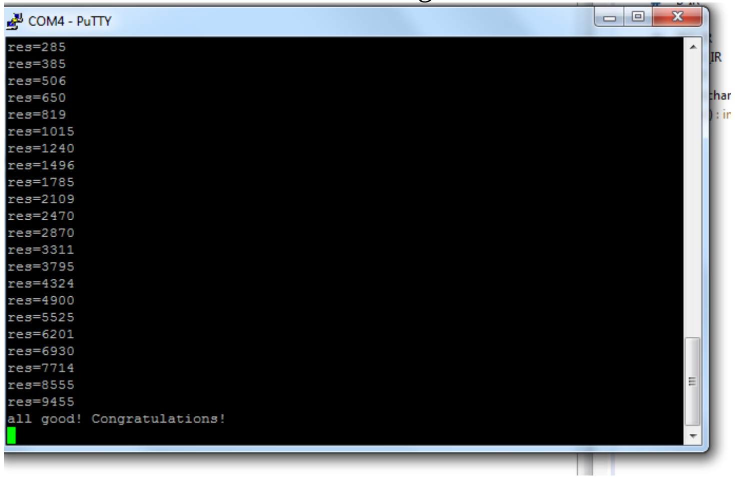
The **Mac_wrap.** The Test software design created for the mac_wrap is shown below.



```c
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#define WRITE_REG(reg,val) ({*(volatile int *)(XPAR_MAC_WRAP_AXI4_AXI4LITE_SLAVE_REGSIF_0_BASEADDR + reg * 4)=val;})
#define READ_REG(reg,var) ({var=*(volatile int *)(XPAR_MAC_WRAP_AXI4_AXI4LITE_SLAVE_REGSIF_0_BASEADDR + reg * 4);})
#define A_IR 0
#define B_IR 1
#define GO_IR 2
#define RES_IR 4
#define BUSY_IR 3
#define RESET 5
void print(char *str);
int main()
{
init_platform();
printf("Hello tb_mac_wrap\n\r");
/* This is skeleton code to test the combined axi4lite_slave_regsif/mx2_wrap design through
 * transactions initiateed via the ARM PS
 */
int i, wrval, rdval, a,b,acc;
// Do a walking bit test for port-A
// *(volatile int *)XPAR_SIMPLE_AXI4LITE_SLAVE_0_S00_AXI_BASEADDR=0xdeaddead;
// printf ("read %x\n", *(volatile int *)XPAR_SIMPLE_AXI4LITE_SLAVE_0_S00_AXI_BASEADDR);
a=b=acc=0;
// Reset acc
//WRITE_REG(RESET,1);
for (i=0;i<31;i++)
{
WRITE_REG(A_IR,i);
WRITE_REG(B_IR,i);
WRITE_REG(GO_IR,1);
do {
READ_REG(BUSY_IR,rdval);
} while (rdval);
READ_REG(RES_IR,rdval);
printf ("res=%d\n",rdval);
// Read back and confirm
acc=i*i+acc;
if (READ_REG(RES_IR,rdval) != (acc)) {printf ("Error reading RDATA:port A:Bit pos %d. Got %x instead of %x\n",i,rdval, acc); return -1;}
}
printf ("all good! Congratulations!\n");
return 0;
}
```

Code 2. Mac_Wrap C level testbench.

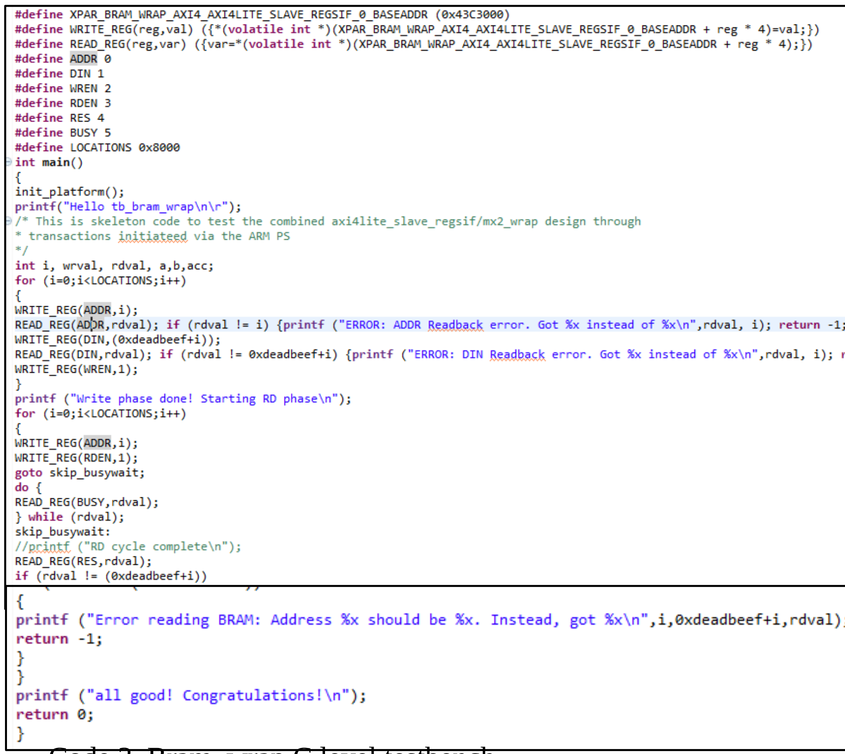
The FPGA programmed and the design executed on silicon Passes the following output shown below which shows correct execution of the design on silicon.



```
res=285
res=385
res=506
res=650
res=819
res=1015
res=1240
res=1496
res=1785
res=2109
res=2470
res=2870
res=3311
res=3795
res=4324
res=4900
res=5525
res=6201
res=6930
res=7714
res=8555
res=9455
all good! Congratulations!
```

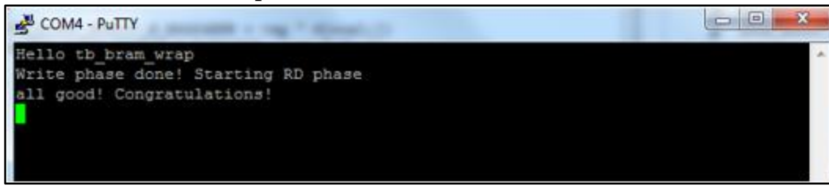
Fig 16. Passed output of the Mac_wrap.

The **Bram_wrap**. The Test software design created for the bram_wrap is shown below. No modifications where made to the created software designs.



```c
#define XPAR_BRAM_WRAP_AXI4_AXI4LITE_SLAVE_REGSIF_0_BASEADDR (0x43C3000)
#define WRITE_REG(reg,val) ({*(volatile int *)(XPAR_BRAM_WRAP_AXI4_AXI4LITE_SLAVE_REGSIF_0_BASEADDR + reg * 4)=val;})
#define READ_REG(reg,var) ({var=*(volatile int *)(XPAR_BRAM_WRAP_AXI4_AXI4LITE_SLAVE_REGSIF_0_BASEADDR + reg * 4);})
#define ADDR 0
#define DIN 1
#define WREN 2
#define RDEN 3
#define RES 4
#define BUSY 5
#define LOCATIONS 0x8000
int main()
{
init_platform();
printf("Hello tb_bram_wrap\n\r");
/* This is skeleton code to test the combined axi4lite_slave_regsif/mx2_wrap design through
* transactions initiateed via the ARM PS
*/
int i, wrval, rdval, a,b,acc;
for (i=0;i<LOCATIONS;i++)
{
WRITE_REG(ADDR,i);
READ_REG(ADDR,rdval); if (rdval != i) {printf ("ERROR: ADDR Readback error. Got %x instead of %x\n",rdval, i); return -1;
WRITE_REG(DIN,(0xdeadbeef+i));
READ_REG(DIN,rdval); if (rdval != 0xdeadbeef+i) {printf ("ERROR: DIN Readback error. Got %x instead of %x\n",rdval, i);
WRITE_REG(WREN,1);
}
printf ("Write phase done! Starting RD phase\n");
for (i=0;i<LOCATIONS;i++)
{
WRITE_REG(ADDR,i);
WRITE_REG(RDEN,1);
goto skip_busywait;
do {
READ_REG(BUSY,rdval);
} while (rdval);
skip_busywait:
//printf ("RD cycle complete\n");
READ_REG(RES,rdval);
if (rdval != (0xdeadbeef+i))
{
printf ("Error reading BRAM: Address %x should be %x. Instead, got %x\n",i,0xdeadbeef+i,rdval)
return -1;
}
}
printf ("all good! Congratulations!\n");
return 0;
}
```

Code 3. Bram_wrap C level testbench.



```
Hello tb_bram_wrap
Write phase done! Starting RD phase
all good! Congratulations!
```

Fig 17.Operation of Bram_wrap design on silicon.

V. CONCLUSIONS

The three circuits which have been designed cnt1_wrap, mac_wrap and bram_wrap all use the axislave and aximaster principle to work. They also all give different timing reports & utilize on-chip power differently. The bram_wrap uses the most power out of all of the other circuits at 200Mhz. Base addresses always have to be assigned to the circuits interfaces in other for them to be able to transfer data. The software design created for each circuit all tested and each one produces the expected outcome , each circuit passes right. Also, if the FPGA isn't programmed from the XSDK tool the debug wouldn't run properly. A lot of thought has gone into ensuring designs are correct hence the questasim testbenches given.

Suggestions are in future work; a better serial connection would be advisable. Validation should also have a precise verification method. Brian Kernighan, creator of the C language, stated that "Everyone knows debugging is twice as hard as writing a program in the first place." A lot of effort goes into specifying the requirements of the design. Given that verification is a larger task, more effort should go into specifying how to make sure the design is correct.

VI. REFERENCES

1. Altera.co.jp. (2014). What is an SoC FPGA? [online] Available at: https://www.altera.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/ab/ab1_soc_fpga.pdf [Accessed 1 Dec. 2017].

2. Rich Griffin. version 1.0, July (2014). Designing a custom AXI-lite slave peripheral. Silica EMEA[online] Available at: https://forums.xilinx.com/xlnx/attachments/xlnx/NewUser/34911/1/designing_a_custom_axi_slave_rev1.pdf 2.[Accessed 5 Jan.2018].

3. Intranet.birmingham.ac.uk. (2015). A short guide to referencing figures and tables for Postgraduate Taught students. [online] Available at: https://intranet.birmingham.ac.uk/as/libraryservices/library/skills/asc/documents/public/pgtreferencingtables.pdf [Accessed 1 Dec. 2017].

4. Vassilios Chouliaras. Zedboard Experiment book (2017). Wolfson school of mechanical, electrical and Manufacturing Engineering. Loughborough University.

5. Zone.ni.com. (2011). Introduction to FPGA Hardware Concepts (FPGA Module) - LabVIEW 2011 FPGA Module Help - National Instruments. [online]

6. AXI Reference Guide. UG 761 (v13.1). (2011). Available at :https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

7. Forums.xilinx.com. (2018). Total Negative Slack vs Worst Negative Slack. [online] Available at: https://forums.xilinx.com/t5/Welcome-Join/Total-Negative-Slack-vs-Worst-Negative-Slack/td-p/308077 [Accessed 11 Jan. 2018].

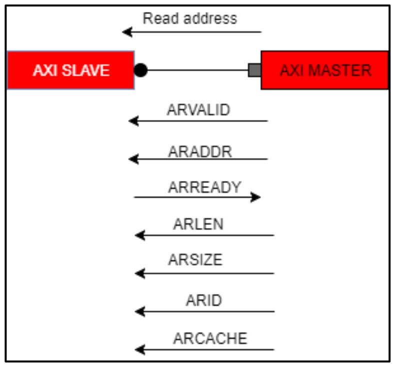
8. Digikey.co.uk. (2015). MCUs or SoC FPGAs? Which is the Best Solution?. [online] Available at:https://www.digikey.co.uk/en/articles/techzone/2015/nov/mcus-or-soc-fpgas-which-is-the-best-solution-for-your-application [Accessed 12 Apr. 2018]. 2

9. Xilinx.com. (2017). Zyqn-700 All Programmable SoC Family Product Tables and Product Selection Guide. [online] Available at: https://www.xilinx.com/support/doc .3

10. Vassilios Chouliarias.ELB020_labs(2016). Wolfson school of mechanical, electrical and Manufacturing Engineering. Loughborough University.

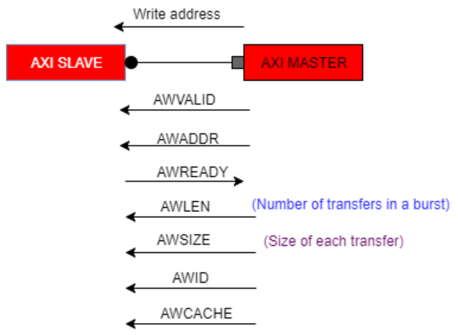
VII. APPENDIX

APPENDIX A

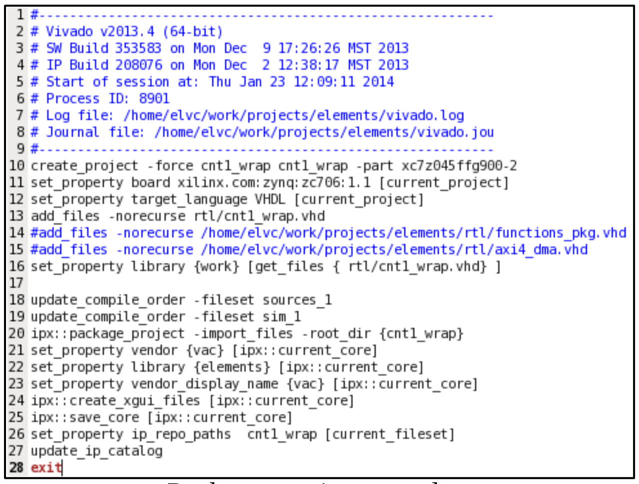


Signals involved in read transactions

APPENDIX B



APPENDIX C

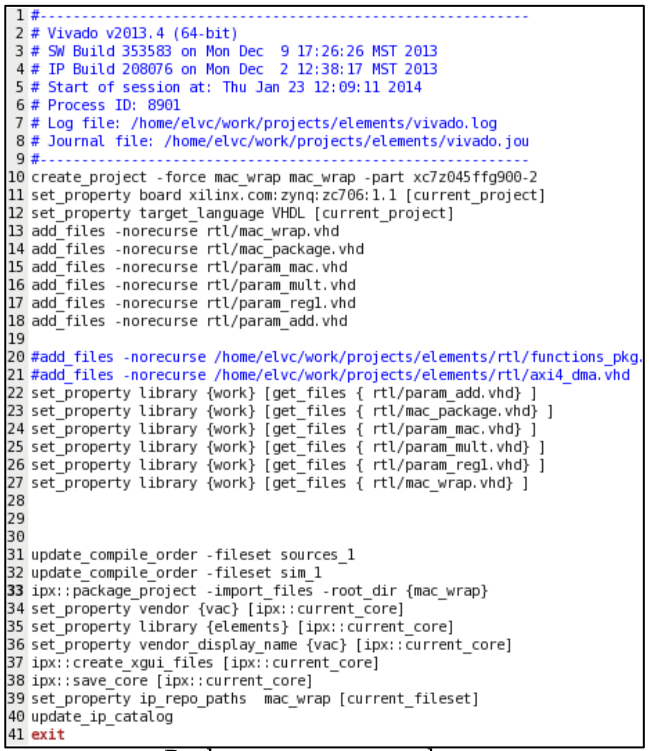


```
#------------------------------------------------------------
# Vivado v2013.4 (64-bit)
# SW Build 353583 on Mon Dec  9 17:26:26 MST 2013
# IP Build 208076 on Mon Dec  2 12:38:17 MST 2013
# Start of session at: Thu Jan 23 12:09:11 2014
# Process ID: 8901
# Log file: /home/elvc/work/projects/elements/vivado.log
# Journal file: /home/elvc/work/projects/elements/vivado.jou
#------------------------------------------------------------
create_project -force cnt1_wrap cnt1_wrap -part xc7z045ffg900-2
set_property board xilinx.com:zynq:zc706:1.1 [current_project]
set_property target_language VHDL [current_project]
add_files -norecurse rtl/cnt1_wrap.vhd
#add_files -norecurse /home/elvc/work/projects/elements/rtl/functions_pkg.vhd
#add_files -norecurse /home/elvc/work/projects/elements/rtl/axi4_dma.vhd
set_property library {work} [get_files { rtl/cnt1_wrap.vhd} ]

update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
ipx::package_project -import_files -root_dir {cnt1_wrap}
set_property vendor {vac} [ipx::current_core]
set_property library {elements} [ipx::current_core]
set_property vendor_display_name {vac} [ipx::current_core]
ipx::create_xgui_files [ipx::current_core]
ipx::save_core [ipx::current_core]
set_property ip_repo_paths  cnt1_wrap [current_fileset]
update_ip_catalog
exit
```

Package_cnt1_wrap.tcl

APPENDIX D



```
#------------------------------------------------------------
# Vivado v2013.4 (64-bit)
# SW Build 353583 on Mon Dec  9 17:26:26 MST 2013
# IP Build 208076 on Mon Dec  2 12:38:17 MST 2013
# Start of session at: Thu Jan 23 12:09:11 2014
# Process ID: 8901
# Log file: /home/elvc/work/projects/elements/vivado.log
# Journal file: /home/elvc/work/projects/elements/vivado.jou
#------------------------------------------------------------
create_project -force mac_wrap mac_wrap -part xc7z045ffg900-2
set_property board xilinx.com:zynq:zc706:1.1 [current_project]
set_property target_language VHDL [current_project]
add_files -norecurse rtl/mac_wrap.vhd
add_files -norecurse rtl/mac_package.vhd
add_files -norecurse rtl/param_mac.vhd
add_files -norecurse rtl/param_mult.vhd
add_files -norecurse rtl/param_reg1.vhd
add_files -norecurse rtl/param_add.vhd

#add_files -norecurse /home/elvc/work/projects/elements/rtl/functions_pkg.
#add_files -norecurse /home/elvc/work/projects/elements/rtl/axi4_dma.vhd
set_property library {work} [get_files { rtl/param_add.vhd} ]
set_property library {work} [get_files { rtl/mac_package.vhd} ]
set_property library {work} [get_files { rtl/param_mac.vhd} ]
set_property library {work} [get_files { rtl/param_mult.vhd} ]
set_property library {work} [get_files { rtl/param_reg1.vhd} ]
set_property library {work} [get_files { rtl/mac_wrap.vhd} ]



update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
ipx::package_project -import_files -root_dir {mac_wrap}
set_property vendor {vac} [ipx::current_core]
set_property library {elements} [ipx::current_core]
set_property vendor_display_name {vac} [ipx::current_core]
ipx::create_xgui_files [ipx::current_core]
ipx::save_core [ipx::current_core]
set_property ip_repo_paths  mac_wrap [current_fileset]
update_ip_catalog
exit
```

Package_mac_wrap.tcl

APPENDIX E

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity tb_cnt1_wrap is
end tb_cnt1_wrap;
architecture exercise of tb_cnt1_wrap is
component cnt1_wrap
port
(
clk : in std_logic;
reset : in std_logic;
addr : in std_logic_vector(7 downto 0);
wrdata : in std_logic_vector(31 downto 0);
rd : in std_logic;
wr : in std_logic;
rddata : out std_logic_vector(31 downto 0)
);
end component;
signal clk_i : std_logic;
signal reset_i : std_logic;
signal addr_i : std_logic_vector(7 downto 0);
signal wrdata_i : std_logic_vector(31 downto 0);
signal rd_i : std_logic;
signal wr_i : std_logic;
signal rddata_i : std_logic_vector(31 downto 0);
constant CLK_PERIOD : time := 20 ns;
constant DLY : time := CLK_PERIOD/4;
constant VECTORS : integer := 16#100000#;
begin
clkmeProc : process
begin
clk_i <= '1';
wait for CLK_PERIOD/2;
clk_i <= '0';
wait for CLK_PERIOD/2;
end process;
resetmeProc : process
begin
wait for DLY;
reset_i <= '1';
wait for CLK_PERIOD;
reset_i <= '0';
wait;
end process;
DUT :cnt1_wrap
port map
(
clk =>clk_i,
reset => reset_i,
addr => addr_i,
wrdata => wrdata_i,
rd => rd_i,
wr => wr_i,
rddata => rddata_i
);
stimProc : process
variable goodonesv : integer;
begin
wait on reset_i until reset_i = '0';
addr_i <= conv_std_logic_vector(0,8);
wrdata_i <= conv_std_logic_vector(0,32);
rd_i <= '0';
```

```
wr_i <= '0';
wait for CLK_PERIOD;
-- wait for a clock and then start writing to registers
-- This is part of the whole verification campaign which
checks the mux.
--
**************************************************
********
-- Iterate over 8 numbers
for i in 0 to VECTORS-1 loop
-- Write R0 (opr1)
addr_i <= conv_std_logic_vector(0 , 8);
wrdata_i <= conv_std_logic_vector(i , 32);
wr_i <= '1';
wait for CLK_PERIOD;
-- Read back and confirm
rd_i <= '1';
wr_i <= '0';
wait for CLK_PERIOD;
rd_i <= '0';
assert (conv_integer(rddata_i) = i) report "cnt1:cnt1_in_ir
read error" severity failure;

-- Write go
wait for CLK_PERIOD;
wr_i <= '1';
addr_i <= conv_std_logic_vector(4 , 8);
wait for CLK_PERIOD;
wr_i <= '0';

--wait for 16 * CLK_PERIOD;
-- Busy wait

BUSY_WAIT_A : loop
  rd_i <= '1';
   addr_i <= conv_std_logic_vector(16#8# , 8);
     wait for CLK_PERIOD;
  exit BUSY_WAIT_A when rddata_i(0) = '0';
end loop;
rd_i <= '0';

wait for CLK_PERIOD;

-- extract result----------------
rd_i <= '1';
addr_i <= conv_std_logic_vector(16#c# , 8);
wait for CLK_PERIOD;
-- Now, check !!
goodonesv := 0;
for j in 0 to 31 loop
if conv_std_logic_vector(i,32)(j) = '1' then
goodonesv := goodonesv + 1;
end if;
end loop;
assert (goodonesv = rddata_i) report "Failure on cnt1_wrap"
severity Failure;
rd_i <= '0';
wait for CLK_PERIOD;
end loop;
assert (false) report "SUCCESS - Ending with 'Failure'
assertion" severity Failure;
   end process;
end exercise;
```

Tb_cnt1_wrap.vhd

APPENDIX F

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity tb_mac_wrap is
end tb_mac_wrap;
architecture exercise of tb_mac_wrap is
component mac_wrap
port
(
clk : in std_logic;
reset : in std_logic;
addr : in std_logic_vector(7 downto 0);
wrdata : in std_logic_vector(31 downto 0);
rd : in std_logic;
wr : in std_logic;
rddata : out std_logic_vector(31 downto 0)
);
end component;
signal clk_i : std_logic;
signal reset_i : std_logic;
signal addr_i : std_logic_vector(7 downto 0);
signal wrdata_i : std_logic_vector(31 downto 0);
signal rd_i : std_logic;
signal wr_i : std_logic;
signal rddata_i : std_logic_vector(31 downto 0);
constant CLK_PERIOD : time := 20 ns;
constant DLY : time := CLK_PERIOD/4;
constant VECTORS : integer := 16#4#;
begin
clkmeProc : process
begin
clk_i <= '1';
wait for CLK_PERIOD/2;
clk_i <= '0';
wait for CLK_PERIOD/2;
end process;
resetmeProc : process
begin
wait for DLY;
reset_i <= '1';
wait for CLK_PERIOD;
reset_i <= '0';
wait;
end process;
DUT :mac_wrap
port map
(
clk =>clk_i,
reset => reset_i,
addr => addr_i,
wrdata => wrdata_i,
rd => rd_i,
wr => wr_i,
rddata => rddata_i
);
stimProc : process
variable a,b,acc : integer;
begin
wait on reset_i until reset_i = '0';
addr_i <= conv_std_logic_vector(0,8);
wrdata_i <= conv_std_logic_vector(0,32);
```

```
a := 0;
b := 0;
Acc := 0;
wait for CLK_PERIOD;
-- wait for a clock and then start writing to registers
-- This is part of the whole verification campaign which checks
the mux.
*****************************************************
*****
-- Iterate over 8 numbers
for i in 0 to VECTORS-1 loop
-- Write R0 (a_ir)
addr_i <= conv_std_logic_vector(0 , 8);
wrdata_i <= conv_std_logic_vector(i , 32);
wr_i <= '1';
wait for CLK_PERIOD;
-- Read back and confirm
rd_i <= '1';
wr_i <= '0';
wait for CLK_PERIOD;
rd_i <= '0';
assert (conv_integer(rddata_i) = i) report "mac:a_ir read error"
severity failure;
-- Write R1 (b_ir)
addr_i <= conv_std_logic_vector(4 , 8);
wrdata_i <= conv_std_logic_vector(i , 32);
wr_i <= '1';
wait for CLK_PERIOD;
-- Read back and confirm
rd_i <= '1';
wr_i <= '0';
wait for CLK_PERIOD;
rd_i <= '0';
assert (conv_integer(rddata_i) = i) report "mac:b_ir read error"
severity failure;
-- Write go
wait for CLK_PERIOD;
wr_i <= '1';
addr_i <= conv_std_logic_vector(8 , 8);
wait for CLK_PERIOD;
wr_i <= '0';
--wait for 16 * CLK_PERIOD;
-- Busy wait
BUSY_WAIT_A : loop
rd_i <= '1';
addr_i <= conv_std_logic_vector(16#c# , 8);
wait for CLK_PERIOD;
exit BUSY_WAIT_A when rddata_i(0) = '0';
end loop;
rd_i <= '0';
wait for CLK_PERIOD;
-- extract result
rd_i <= '1';
addr_i <= conv_std_logic_vector(16#10# , 8);
wait for CLK_PERIOD;
-- Now, check !!
acc := i * i + acc;
assert (acc = rddata_i) report "Failure on mac_wrap:acc)_ir"
severity Failure;
rd_i <= '0';
wait for CLK_PERIOD;
end loop;
assert (false) report "SUCCESS - Ending with 'Failure' assertion"
severity Failure;
end process;
```

Tb_mac_wrap.vhd