MP2 Report

Nachuan Wang(nachuan3)

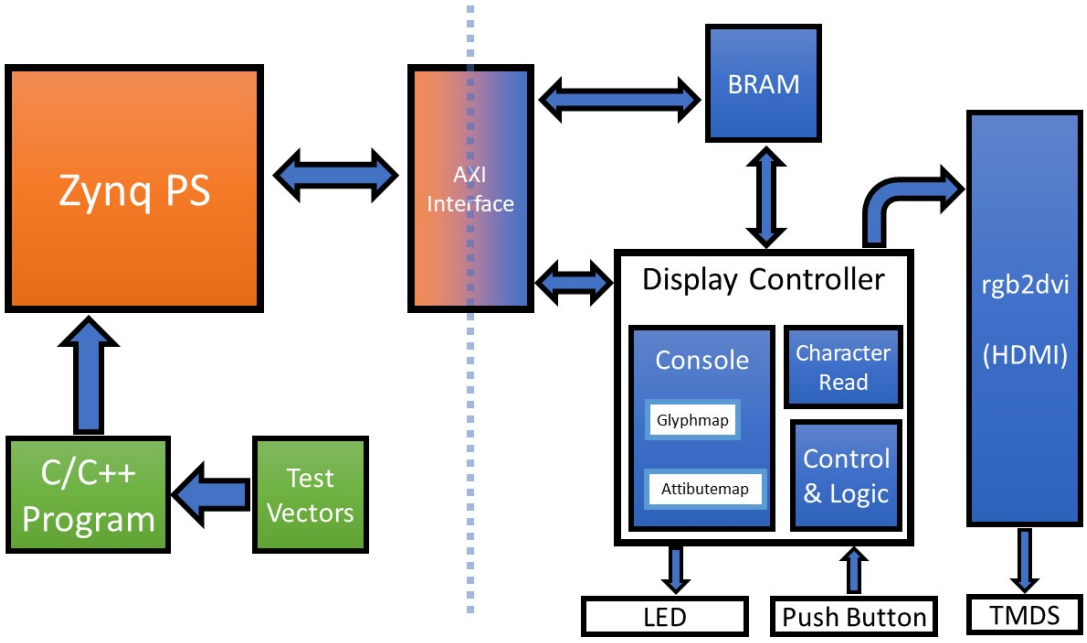
Liangyu Zhou(liangyu5)

Part A: Working with IPs and the HDMI

Assumptions:

For part one, we assume that the data is written to the BRAM immediately after the PS side is started. The test vector file has the same format as the sample test file. And the data transfer between PS and BRAM is based on AXI protocol. On PS side, we transfer 4 characters at a time, which is 32bit, while on the PL side, we only need to read 1 byte (8bit) at a time.

Block Diagram:



Our design uses the diagram above, where the AXI interface refers to the axi bram controller (AXI LITE protocol).

Entities/Modules:

In this part, we have Zynq PS to read data in testvectors.h and transfer all the data to BRAM with a data width of 32bit, which means transfer 4 characters at a time. AXI bram controller to actually responsible for the communication between PS and BRAM. BRAM generator is used to generate the bram on the pl side to store both number of test vector and test vector itself. The display controller is a RTL design. It responsible for read data from BRAM and deliver data to rgb2dvi IP. It can also read the input of push button to control which data to read from BRAM. The last part is rgb2dvi IP, which is used to encode the ASCII code to video signal that can be used by HDMI interface and output the signal to monitor.

Design process:

Firstly we divide the task into PS and PL, PS is responsible for read data from testvector.v and encode the characters into ASCII code. And then the PS should write the encoded data to BRAM via AXI. Since the data width is 32bit and ASCII code is 8 bit each, we can transfer and store 4 characters at a time. So, we compress the 4 ASCII into one 32-bit signal, and then use Xil_Out32() function to transfer data. And all the procedures are done during initialization process because in this way we don't need to care too much about the timing problem.

On the PL side, we read data from BRAM using a state machine, because each cycle we only need one byte that was on different index of the BRAM data. we use the remainder that the x position divided by 4. And each state read different part of the BRAM data. Last state should increase the address by one to read next four bytes. We use simple signal to read data from BRAM instead of a complex AXI protocol. The width of BRAM is 32 and the depth is automatically set because we use BRAM in BRAM controller mode, which means the width and depth is fixed. We didn't have a control signal on PS side. The button control is on PL side. We didn't use LED to show the status either. And we also use BRAM to store the number of test vectors because it is read easy to implement.

Latency calculations:

a. Initialization part: we need to write data to BRAM. In this calculation, we calculate the time needed to transfer one single test vector. 8*96/32 = 24 cycles are needed to transfer a single vector.

b. After press the button, all the process are pipelined, so we need 1 cycle to read data from BRAM, and 96 cycles to display the entire sentence on the screen, which is 97 in total.

Post-Implementation results:

Recourses	Utilization
FF	4710
IO	9
BUFG	5
LUT	5065
LUTRAM	645
BRAM	4

MMCM	1
PLL	1

Power	Utilization
Clocks	0.024W
Signals	0.020W
Logic	0.018W
BRAM	0.012W
I/O	0.136W
PS7	1.256W

Timing	
Setup WNS	1.259ns
Setup total number of Endpoints	15614
Hold WNS	0.052ns
Hold total number of Endpoints	15614

Difficulties/Bugs:

We have difficulties when reading data from the BRAM and writing data to BRAM. We Google a solution to write data into BRAM from PS side using AXI BRAM controller. And for PL side, we use the axi interface to read data, but we failed. We can write data and read data from PS side but not in PL side. we finally figure out that we need to buffer our address before output to the BRAM.

What we learnt from this

We learnt how to transfer data from PS to PL using BRAM. And we have knowledge about how to display sentence on screen.

Vivado version:

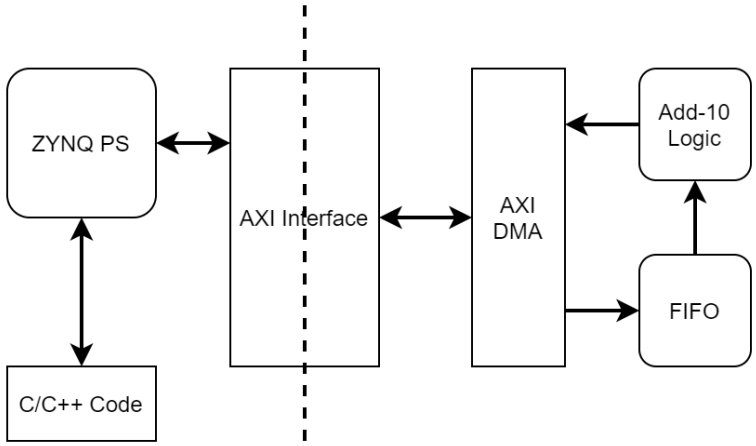
We used Vivado 2019.1 on our own personal computers with Windows.

Part B: Introduction to DMA and AXI4-Stream

Assumptions:

No assumption is made.

Block Diagram:



Entities/Modules:

1. ZYNQ Processing System: Embedded processor in FPGA which runs the C/C++ code.
2. AXI DMA: Transfers data to/from FIFO.
3. FIFO: Holds data from DMA and sends it back.
4. Add-10 Logic: Combinational logic that increment output of FIFO by 10.
5. AXI Interconnect: Communicates between ZYNQ Processor and AXI DMA.
6. AXI Smart Connect: Groups signals from AXI DMA to ZYNQ Processor.

Design process:

- Mechanism: The ZYNQ processor let DMA to transfer data from DDR to FIFO; the data is incremented by 10 and sent back to DMA; DMA then writes data back to the DDR; finally processor checks that the data in DDR is correctly incremented.
- PS and PL communicate through AXI interconnect.
- The part of sending/verifying data is implemented in software.
- Are all the test-vectors sent during initialization.
- The only control signal between PS and PL is the interrupt signal sent by DMA.

Latency calculations:

- Time to send and receive data with DMA and packet processing:
  Time for sending and receiving 32 8-bit unsigned data (using) = 2409 ns
  So Taxi cycle = 2409 / 8 ~= 256 ns

Post-Implementation results:

Recourses	Utilization
LUT	5711
LUTRAM	875
FF	8135
BRAM	3.5
BUFG	1

	Power
Clocks	0.028 W
Signals	0.011 W
Logic	0.01 W
BRAM	0.001 W
PS7	1.256 W
Device Static	0.136 W

Setup		Hold		Plus Width	
WNS	2.739 ns	WHS	0.014 ns	WPWS	3.750 ns
TNS	0 ns	THS	0 ns	TPWS	0 ns
Number of Failing Endpoints	0	Number of Failing Endpoints	0	Number of Failing Endpoints	0
Total Number of Endpoint	29010	Total Number of Endpoint	29010	Total Number of Endpoint	9780

Difficulties/Bugs:

One bug encountered: within four sent data only one is incremented by 10, finally we found that it is because the FIFO width is 32-bit but the data width is 8-bit, and we only incremented the 32-bit value by 10. After modifying the adder logic we got the correct result.

What we learnt:

We learnt about how the DMA is used to make memory transfer more efficient.

Vivado version:

We used Vivado 2019.1 as tutorial video on out own personal computers with Windows.