USER GUIDE

M.Abdul WAHAB

March 21, 2016

Contents

List of Figures

List of Tables

Nomenclature

ETB Embedded Trace Buffer is a Coresight component of sink class. It is a on chip RAM allowing to store the trace generated from trace source

ETM Embedded Trace Macrocell is a Coresight component of source class which generates trace for every executed instruction

Funnel Coresight component of link class and allows choosing which trace sources to send to the trace sinks

ITM Instrumentation Trace Macrocell is a Coresight component of source class which allows to add instructions in C code and to obtain the information that we are unable to obtain from other coresight components

PTM Program Trace Macrocell: Coresight component of source class which means that PTM generates trace only if the executed instruction changed the PC (e.g. Branch instructions, load pc ..., ..)

STM System Trace Macrocell is a Coresight component of source class which generates trace and is the most recent component of source class

TPIU Trace Port Interface Unit is also a Coresight component of sink class and allows to send generated trace data (from trace sources) towards PL or towards outside the chip thansk to MIO

Change log

Author	Version	Date	Change description
M.Abdul WAHAB	1.0	18/01/2016	1st version (basically a draft)
M.Abdul WAHAB	2.0	20/03/2016	Completed Programming CS components section (2.1) Change the structure of the document
M.Abdul WAHAB	2.1	21/03/2016	Added nomenclatures

1 Introduction

The first and the most important step of the Hardblare project is to implement DIFT (Dynamic Information Flow Tracking) also called DIFC (Dynamic Information Flow Control) on the Zedboard. Zedboard is a Xilinx Zynq SoC (System-on-Chip) which means that it contains a hardcore processor (called PS which stands for processing system) and an FPGA part (called PL which stands for programmable logic).

1.1 DIFT

DIFT/DIFC consists of adding a tag (or label) to data of interest (e.g. inputs) and keeps track of the propagation of tags throughout the system. If any tainted data is involved in potentially illegal activity (such as pointing inside the prohibited code), an alarm is triggered.

Software

Historically, the DIFT was first implemented in Software. This implementation is flexible but has the disadvantage of presenting huge overheads. For example, the least overhead while implementing DIFT in SW is 390% which means that the program runs slower 3.9 times than the original program. This means that a better way was needed to implement DIFT on real world system.

Hardware

The second solution which was proposed was use hardware to implement DIFT. The main advantage of this solution is that it is quicker than the pure software solution. The disadvantages of this solution is that it is not much flexible.

Mix solution : SW + HW

A third solution which was proposed was to mix the previously mentionned solutions to obtain advantages from both these solutions. Three types of solution exist in the related work.

1. In-core DIFT
2. Offloading DIFT
3. Off-core

In the context of this project, we will be implementing it as an off-core solution which means that the main core (or the application core) does not deal with the management of tags. The management of tags and their propagation is done on an off-core in HW. The main goals of this project are:

- Non-invasive flexible solution
- Implement DIFT on Zedboard
- Low performance overhead
- No false positives or negatives
- Approach based on a non-modified CPU with a standard Linux and generic binaries.

In order to implement DIFT on zedboard, we need to obtain some informations from the CPU (PS in Zynq). Mainly these informations consist of

- CPU to PL
  - Instruction
  - PC
  - Memory addresses from Load and Store
- PL to CPU
  - Stall the processor

As the existing solutions implemented DIFT on the soft cores, it was easy to obtain required informations. We needed to find a way of doing this with a hard core. A solution was proposed by Master students during their project. The solution was to use the ARM debug core (called **Coresight Components**) on the ARM Cortex A9 present on the PS of Zedboard.

2 Retrieving Trace

This chapter presents how the trace is retrieved from Coresight components. The chosen method is supposed to give all the informations required from CPU to PL. The proposed way is shown in figure 2.1. The red box shows the PTM (Prgoram Trace Macrocell) that generates the trace and the lines in red shows the path taken by the generated trace to get to ETB (Embedded Trace Buffer) or TPIU (Trace Port Interface Unit).

The following components belongs to the coresight components.

- DAP (Debug Access Port)
- CTI (Cross Trigger Interface)
- CTM (Cross Trigger Matrix)
- PTM (Program Trace Macrocell)
- ETM (Embedded Trace Macrocell)
- ITM (Instrumentation Trace Macrocell)
- ETB (Embedded Trace Buffer)
- TPIU (Trace Port Interface Unit)

These components are further divided into different categories listed in table 2.1.

Type	Examples	Function
Control & Access	CTI, CTM, DAP	Give control and access to CS components
Sources	PTM , ETM, ITM	collect trace from the CPU
Links	FUNNEL, REPLICATOR	link between the CS sources and CS sinks
Sinks	ETB, TPIU	Store or export the trace data

Table 2.1: Coresight Components types and function

We need to program the CS (CoreSight) components to obtain the trace from the CPU. The trace is generated once the instruction is committed for execution (P.39/252 of PFT Architecture V1.1). Once all the components are programmed, we can start capturing the trace and decoding it in order to understand what was executed.

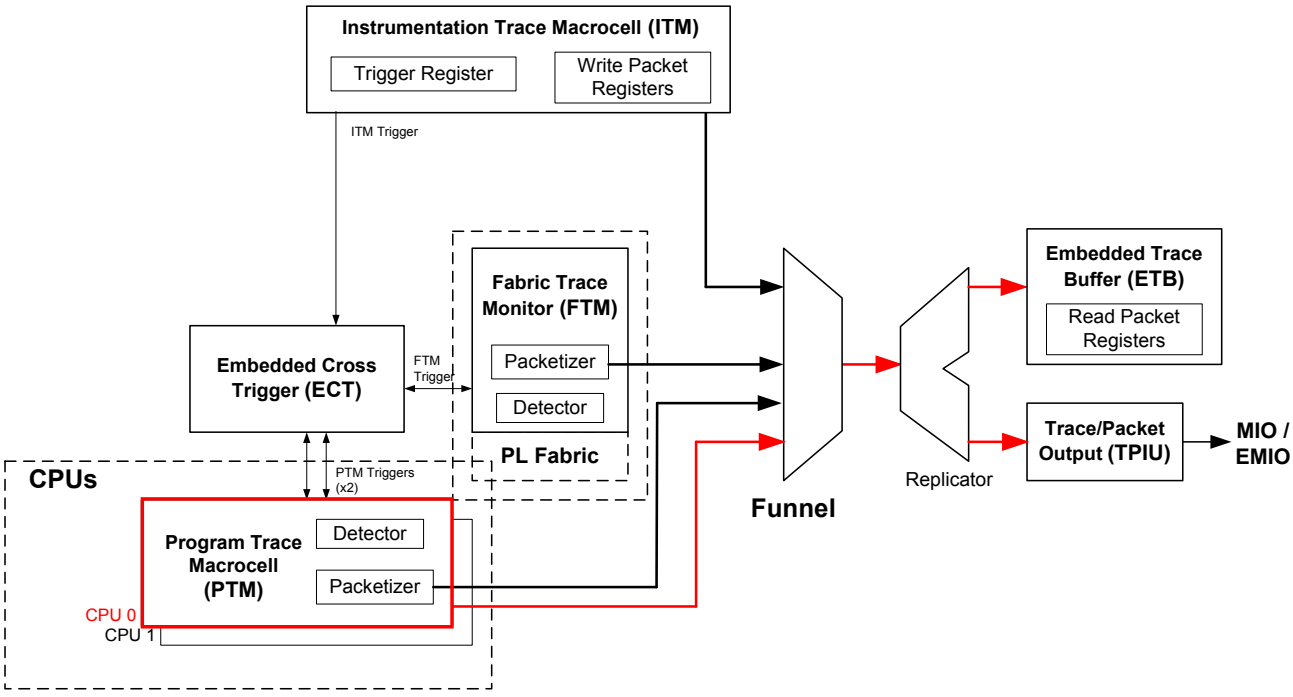

Figure 2.1: Trace retrieval from ARM Cortex-A9 through Coresight Components (PTM)

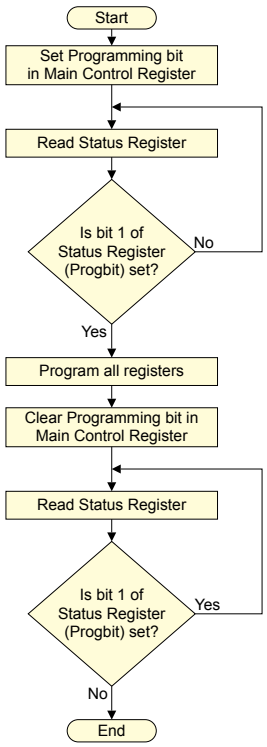

Figure 2.2: PTM registers programming order

Register	Value	Purpose
ETMLAR	0xC5ACCE55	Unlock PTM registers
ETMCR	1—(1¡¡8)—(1¡¡10)	Enable PTM features
ETMTRIGGER	0x6F	Events that capture trace
ETMTECR1	1¡¡24	Trace all code
ETMTEEVR	0x6F	TraceEnable Event
ETMTRACEID	0x0F	Trace ID
ETMLAR	0	lock PTM registers

Table 2.2: PTM Registers and Values

2.1 Programming CS components

To obtain the trace: we need to program :

1. Coresight source class component (PTM (on Zedboard) or it could be ETM, ITM or even STM (System Trace Macrocell) on other components)
2. Coresight link class component (Funnel)
3. Coresight sink class component (ETB, TPIU or both)

PTM

To program PTM, some registers must be programmed. Beware that these registers should be programmed in a specific order as presented in figure 2.2. For more information, please have a look at the c program written to program coresight components (file name is program_coresight_components.c and can be found on redaction_hw gforge github).

The table 2.2 presents the main registers to program in the PTM to activate it.

The following implementations for PTM are possible :

1. Trace all instructions
2. Trace range (in order to trace some functions only) or not to trace some range
3. Trace single instruction (This feature was not implemented as it is no interest to us)

Register	TRACE ALL INSTRUCTIONS	TRACE RANGE	TRACE ALL EXCEPT SOME REGIONS
ETMCR	1 ≪ 10 (Change programming bit alone)		
ETMCR	(1 ≪ 8)—(1 ≪ 12) (Activate other features)		
ETMTECR1	1 ≪ 24	0 ≪ 24	1 ≪ 24
ETMTEEVR	0x6F (Event ALWAYS TRUE)		
ETMACVR(n)	-	Start/stop address	
ETMACTR(n)	1		

Table 2.3: PTM Configuration register

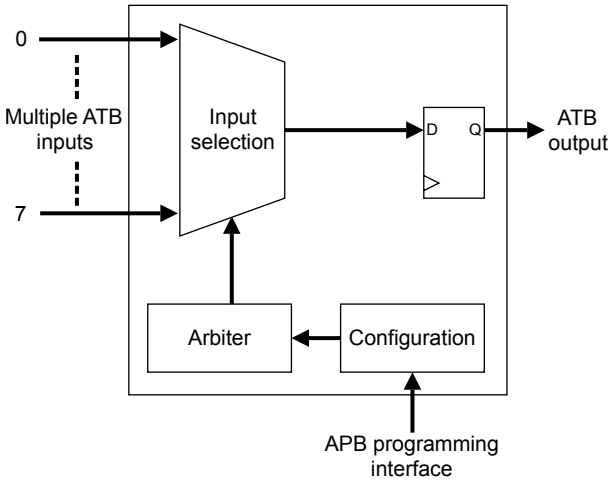

Figure 2.3: Coresight Funnel (Link class)

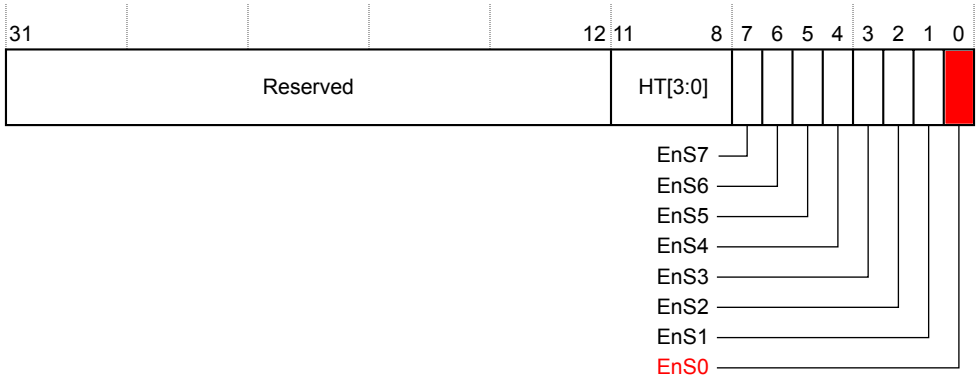

Figure 2.4: Coresight Funnel (Link class)

The table 2.3 presents the registers that need to be programmed and the bits that needs to be changed in order to program the different types of PTM implementations.

Here is the overview of the registers mentionned in the table 2.3.

FUNNEL

The funnel component is shown in figure 2.3 and only one register is needed to be programmed (the funnel control register shown in figure 2.4). In order to understand which bit to activate, we need to have a look at Zynq TRM (Technical Reference manual) which indicates which inputs are connected in the funnel. In our case, we need to activate the bit corresponding to PTM_0 (which generates the trace of what is executer on CPU_0). This bit is bit 0 as indicated in figure 2.4. The table 2.4 resumes the register to program in descending order.

ETB

To program ETB, following actions need to be taken and in the presented order. The table 2.5 shows the registers to program and the value to program with.

1. Program ETB registers

Register	Value	Purpose
CSTFLAR	0xC5ACCE55	Unlock FUNNEL registers
CSTF Control	1	Enable input for trace 0
CSTFLAR	0	lock FUNNEL registers

Table 2.4: FUNNEL Registers and Values

Register	Value	Purpose
ETBLAR	0xC5ACCE55	Unlock ETB registers
ETBFFCR	(1¡¡8—1¡¡9—1¡¡10)	Enable ETB features
ETBCONTROL	1	Enable ETB Trace Capture
ETBLAR	0	lock ETB registers

Table 2.5: ETB Registers and Values

2. Enable tracing
3. Wait until the AcqComp bit is set
4. CODE TO TRACE GOES HERE
5. Disable tracing
6. Wait until the DFEmpty bit is set
7. Read the trace

TPIU

To activate coresight components in the Zynq under vivado processing system IP, activate trace by choosing a trace width and deciding where to connect these traces (either to EMIO or MIO). If the connection is made to EMIO (Figure 3.14), it is a wire available to the PL part of ZYNQ and if the connection is made to MIO, it is available to the output ports and can be used and analysed by logic analyzers. A clock is needed by TPIU that allows to synchronize TPIU with trace analyzer. By looking at ARM Coresight Component User guide, two frequencies can be choosed (Figure 2.6).

1. @250 MHz : data should be received only at front end
2. @125 MHz : data should be received at both ends

The frequency of 250 MHz was chosen because working with dual edge is not easy and it is not adapted to Zynq FPGA. Dual edge clock can be used in CPLD's (like Xilinx's CoolRunner II which offers the possibility of using dual edge FF's). The design realized to test if the traces collected are right is presented in Figure **??**.

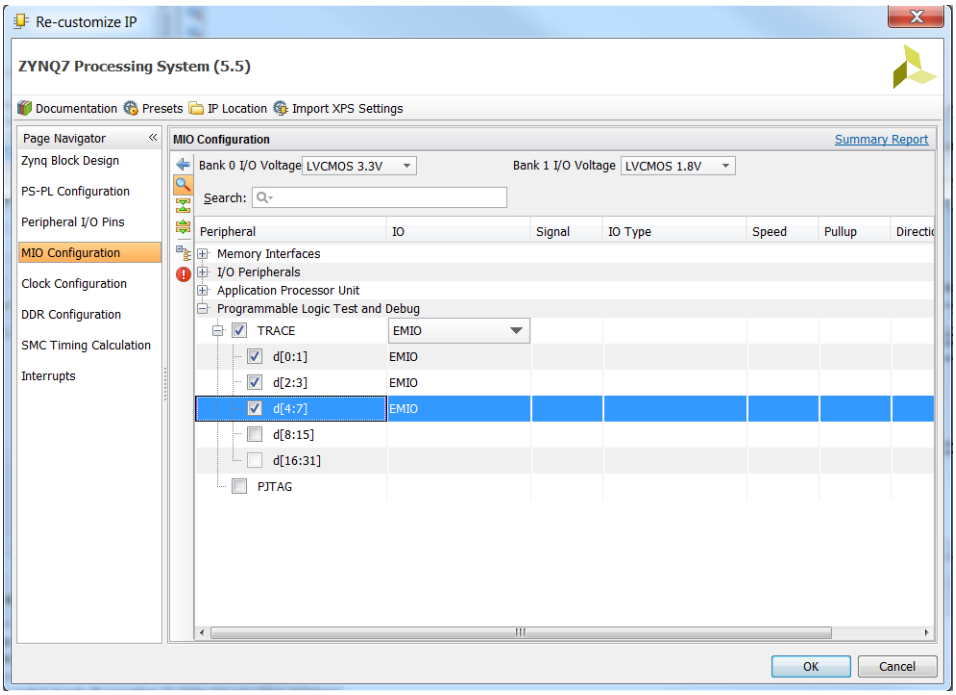

Figure 2.5: TPIU Traces connected to EMIO

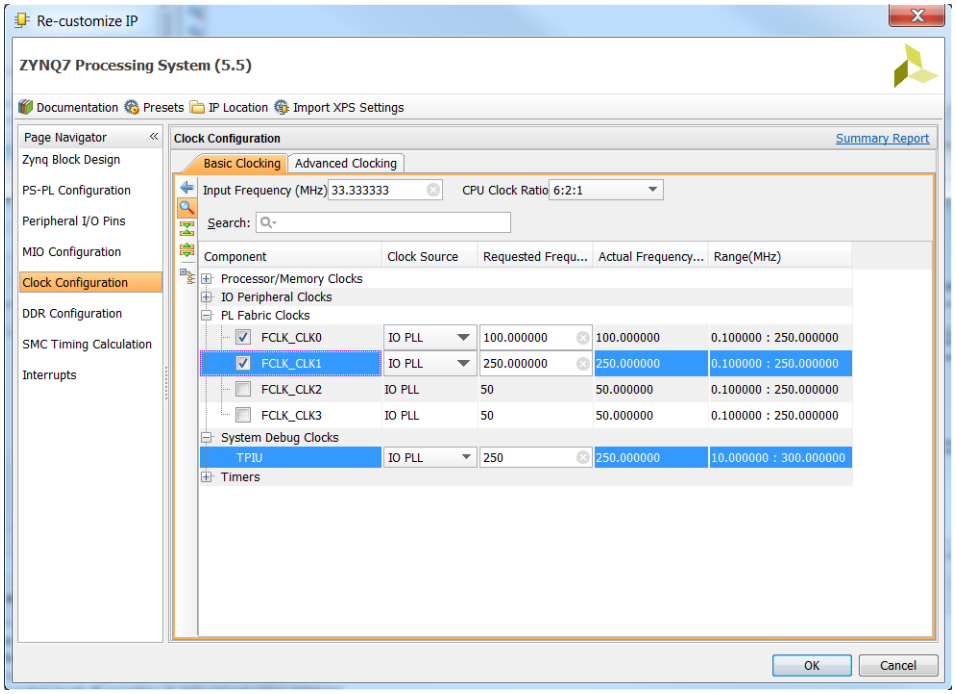

Figure 2.6: TPIU Trace CLOCK

3 IP development

This chapter will explain briefly some important things that we need to take into consideration when obtaining trace from the PL. These important things can be the errors I get or some important things that I read while reading a user guide. I will try to mention the source of my problem and the solution if I found one.

3.1 Goal

The aim of IP development section is to create an IP that will allow decode obtainted PFT packets from TPIU. The decode IP should be able to decode all type of packets the PFT protocol precise. At first, the IP will be created considering single implementation of the PTM such that we know exactly the format of generated packets.

3.2 Dividing the development of IP into multiple components

Once we programmed the TPIU, we can start getting trace on the PL. We need to store the trace sent by TPIU to PL and then decode it. We are going to need a FIFO to store the trace and to decode the trace, we will develop a custom IP in VHDL.

FIFO

The best solution to implement FIFO is to use an existing FIFO_generator IP provided by Xilinx otherwise we can replace it by a simple custom IP with following code for example.

PFT decoder

To decode PFT protcol, we need to understand the PFT packets first. The table 3.1 presents the packets and their corresponding headers.

Once we know how to recognize these packets, we need to understand the order in which a packet can come. After looking at different traces we obtained so far and documentation provided by ARM on Coresight components we find out that there is no such order. This means that in almost every state, we will need to identify packet header. In order to better understand, figure **??** is a finite state machine describing what we want to do.

3.3 Timing violation problem

To solve the timing violation problems, I tried to test each IP that I created separately to make sure if the error comes from my IP or the ones provided by Xilinx.

- Vivado project name : test_bram
- Location : E:\Vivado\test\test_bram

PFT packet name	Header	Remarks
A_sync	0x00 00 00 00 00 00 80	Alignment synchronization
I_sync	0x08 @@ @@ @@ @@ IB CC	Instruction synchronization
Atom	0b1xxx xxx0	
Branch address	0bCxxx xxx1	C is 1 if another byte follows, 0 otherwise.
Waypoint update	0x72	
Trigger	0x0C	
Context ID	0x6E	
VMID	0x3C	
TimeStamp	0b0100 0x10	
Exception return	0x76	
Ignore	0x66	

Table 3.1: PFT packet formats

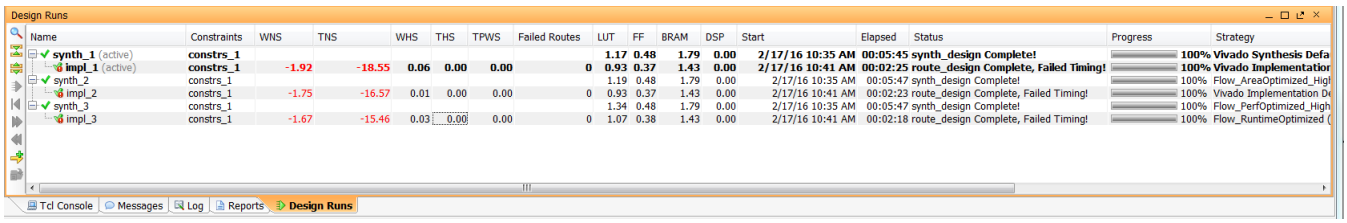

Figure 3.1: TPIU Trace CLOCK

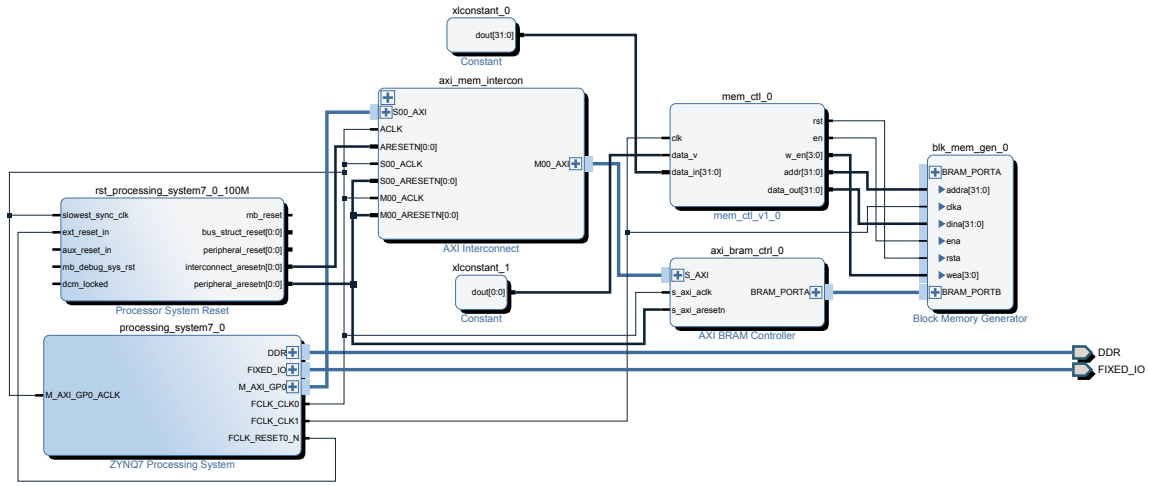

Figure 3.2: Design to test BRAM

The problem appears when we implement the design on the PL of zedboard. The problem stays even when we remove the nested branchs which have the consequence of making larger logic blocks. For example, the "if elsif ..." in the function was replaced by a case in order to check if it was possible to correct this implementation error without redesigning the state machine. This error was obtained regardless of synthesis and implementation strategies (Figure 3.1).

Mem_Ctl

This IP generates the signals to interface correctly to BRAM. It is not very customizable yet but it works. The design to test the IP is presented in figure 3.2.

- The mem_ctl_v_1_0 ip is located in E:\Vivado \test \mem_ctl.
- input data_in value is set to constant value 0xf012340f
- input data_v of this IP is set to constant '1'
- BRAM (Block Memory Generator IP) mode : BRAM controller
- BRAM memory type : True Dual port RAM

First, both the clocks were chosen to work at 250 MHz (because traces are generated at this frequency). Vivado default synthesis and implementation strategies were tested and it gave the "timing requirements did not meet" critical warning.

Then, vivado synthesis settings are changed to Flow_AreaOptimized_High in synthesis settings (as shown in figure 3.3) and Perforamce_retiming strategy is chosen as Implementation strategy (figure 3.4). There was still a critical warning there. By looking at critical paths, the problem was coming from BRAM controller. It looked like the chosen frequency was too high for the BRAM Controller IP. The frequency of reads was changed to 100 MHz.

Then, the clock FCLK_CLK0 is configured at 100 MHz and FCLK_CLK1 at 250 MHz. With default synthesis and Implementation strategies, there was no critical warning on timing violation.

The SDK project is located at E:\Vivado\test\test_bram\test_bram.sdk.

An error was noticed in the generated BSP. Looking at vivado, we notice that the BRAM controller IP is mapped at address 0x40000000 (Figure 3.5). On the other hand, in the generated BSP, the address was not the

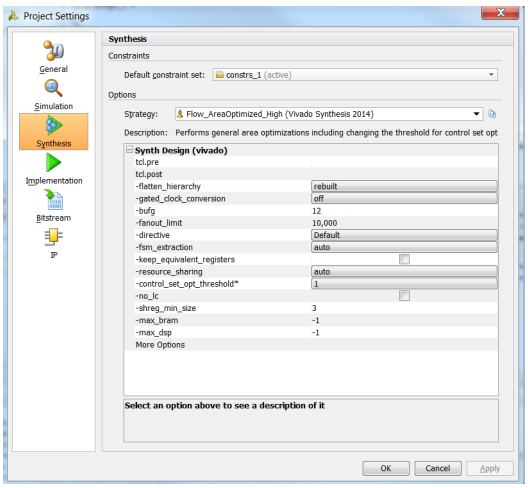

Figure 3.3: Vivado Synthesis settings

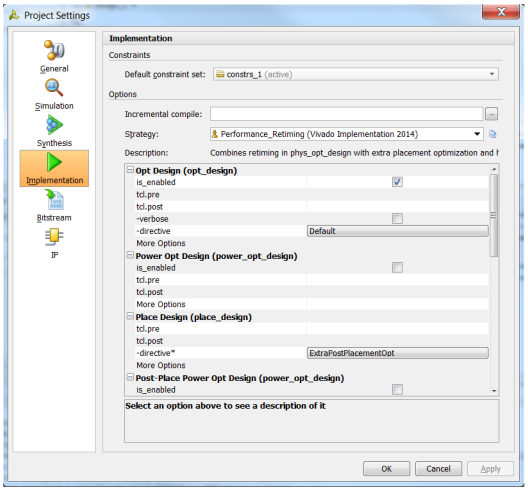

Figure 3.4: Vivado Synthesis settings

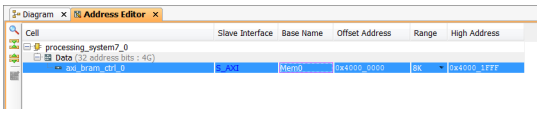

Figure 3.5: Vivado Address editor

same (Figure 3.6). The XPAR_BRAM_0_BASEADDR is 0x00000000 which is not the same as in the Vivado address editor which means that the defines generated by the BSP should be used carefully and checked when we use them. The C code used to read data from BRAM is presented in figure **??**. One strange thing to notice here is that the values read differs slightly sometimes from the original value: this may be due to cache coherency issues but needs to be taken care of once and for all.

```
1  #include "xparameters.h"
   #include <stdio.h>
3  #include "xbram.h"
   #define    BRAM_DEVICE_ID          XPAR_BRAM_0_DEVICE_ID
5
   XBram Bram;
7  int main(void)
   {
9    int Status, i;
     XBram_Config *ConfigPtr;
11
     ConfigPtr = XBram_LookupConfig(BRAM_DEVICE_ID);
13   if (ConfigPtr == (XBram_Config *) NULL) {
       return XST_FAILURE;
15   }
```

```
/******************************************************************/

/* Canonical definitions for peripheral AXI_BRAM_CTRL_0 */
#define XPAR_BRAM_0_DEVICE_ID XPAR_AXI_BRAM_CTRL_0_DEVICE_ID
#define XPAR_BRAM_0_DATA_WIDTH 32
#define XPAR_BRAM_0_ECC 0
#define XPAR_BRAM_0_FAULT_INJECT 0
#define XPAR_BRAM_0_CE_FAILING_REGISTERS 0
#define XPAR_BRAM_0_UE_FAILING_REGISTERS 0
#define XPAR_BRAM_0_ECC_STATUS_REGISTERS 0
#define XPAR_BRAM_0_CE_COUNTER_WIDTH 0
#define XPAR_BRAM_0_ECC_ONOFF_REGISTER 0
#define XPAR_BRAM_0_ECC_ONOFF_RESET_VALUE 0
#define XPAR_BRAM_0_WRITE_ACCESS 0
#define XPAR_BRAM_0_BASEADDR 0x00000000
#define XPAR_BRAM_0_HIGHADDR 0x00000000
```

Figure 3.6: Generated BSP - Xilinx SDK

```
17   Status = XBram_CfgInitialize(&Bram, ConfigPtr,
         ConfigPtr->CtrlBaseAddress);
19   if (Status != XST_SUCCESS) {
       return XST_FAILURE;
21   }

23   //printf("MemBaseAddress : %08x\r\n", ConfigPtr->MemBaseAddress);

25   for (i = 0; i < 200; i++)
     {
27     Status = XBram_ReadReg(0x40000000, i*4);
       printf("%02x ", Status);
29     if (i%20 == 0)
         printf("\r\n");
31   }
     return 0;
33 }
```

Listing 3.1: Descriptive Caption Text

3.4 PFT Decoder V2

The First version of the PFT Decoder did not work although it passed every simulations (Behavioral, Post-synthesis functional, Post-implementation functional). The problem come from the fact that the FSM is too big (40-50 states) for the tool to route the design properly. If we take a look at the delays introduced by routing (See Figures 3.7, 3.8 and 3.9) it represented around 70% of the total delay. The different things recommended by Xilinx in their FSM documentation was done in order to implement successfully the design but without any success.

Simple Solution

An easy and simple solution would be to use a dual clock FIFO in which the data is written at 250 MHz and read at the maximum frequency of the decoder IP. This solution should work but this is not the right way to go before ruling out other possibilities.

Design smaller FSMs

The next approach taken is to break the FSM into multiple FSMs in order to make it modular. There is one global FSM that controlls all the other FSMs. These other FSMs decode each packet: e.g. branch address packets have their own FSM which decodes the packet and so on for each packet. The packets that contains a single header are taken care of in the global FSM.

The new design's schematic is shown in Figure 3.10. The component chemin can be seen in figure 3.11. The simulation showing the actual results of IP are shown in figure 3.12. The global design realized to test the PFT

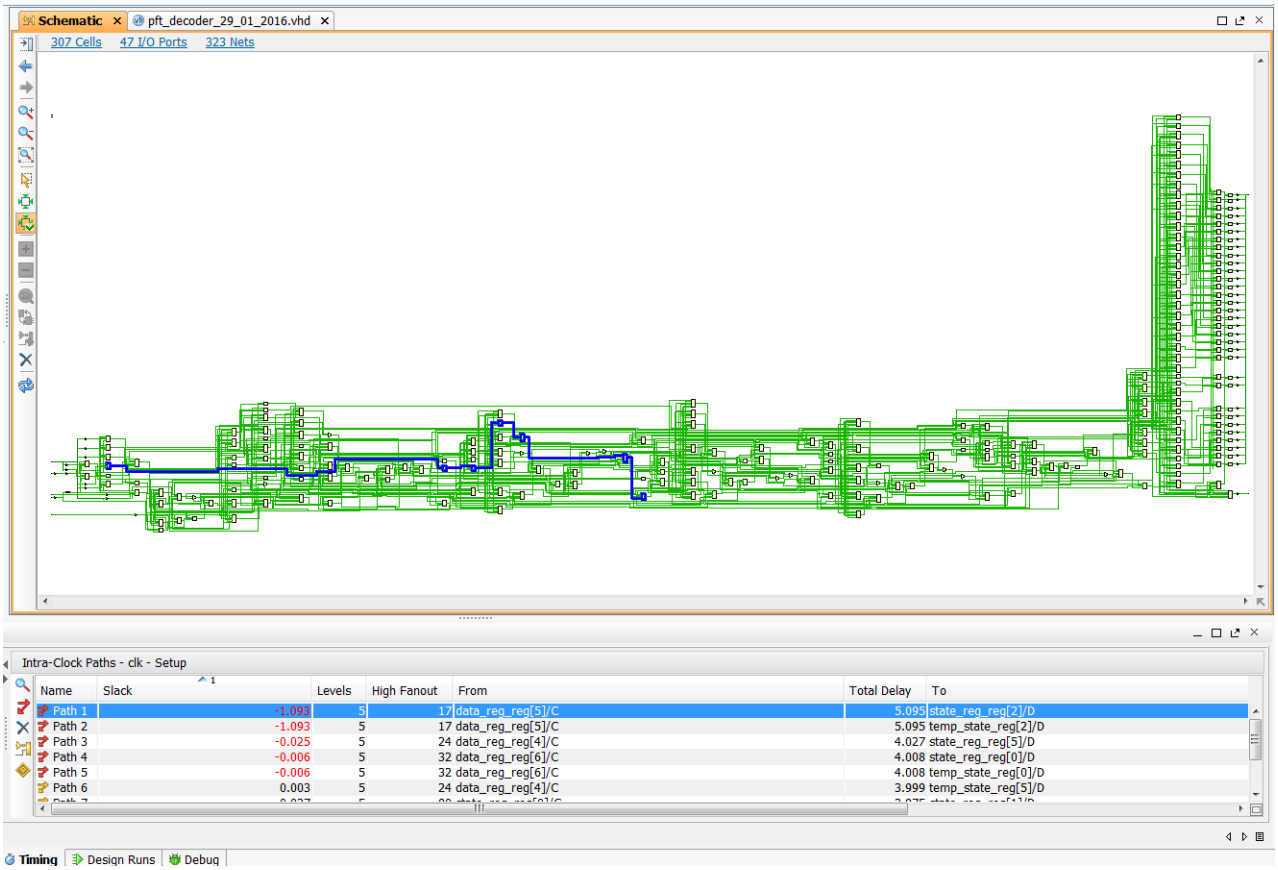

Figure 3.7: Timing violation Path (in Blue) global design

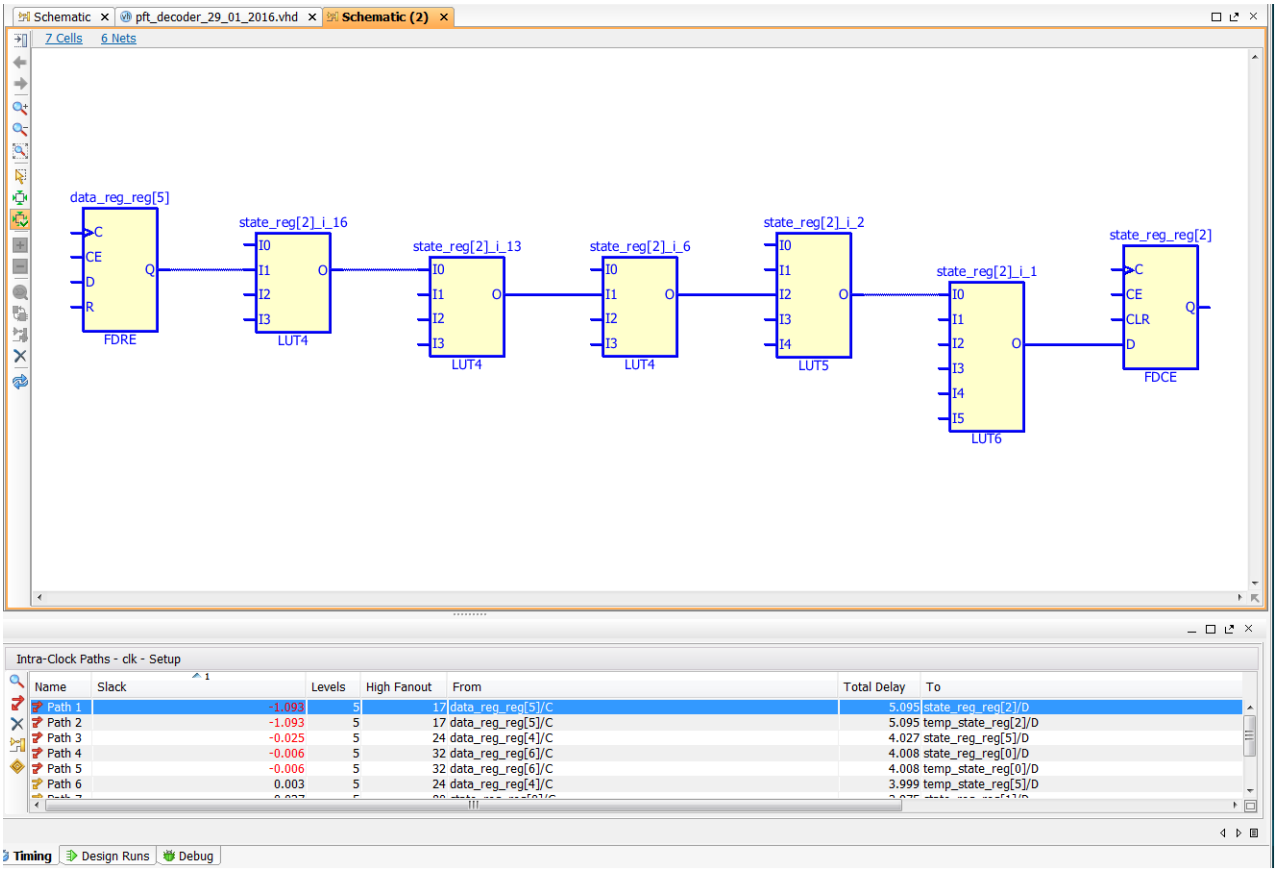

Figure 3.8: Timing violation Path

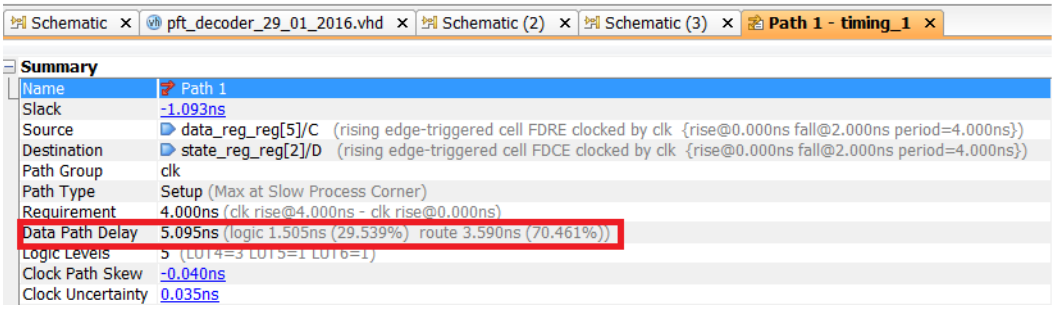

Figure 3.9: Timing violation - Delay

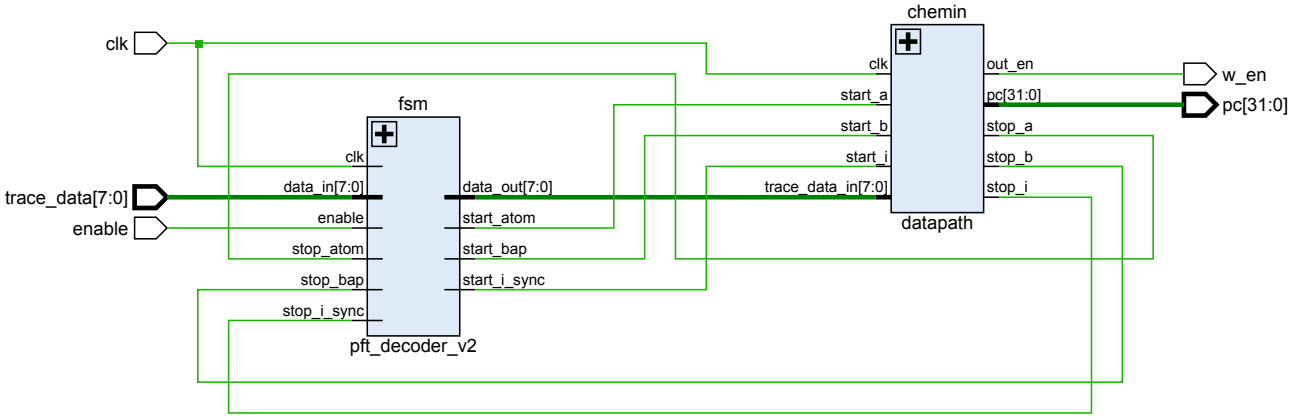

Figure 3.10: PFT Decoder v2 Schematic

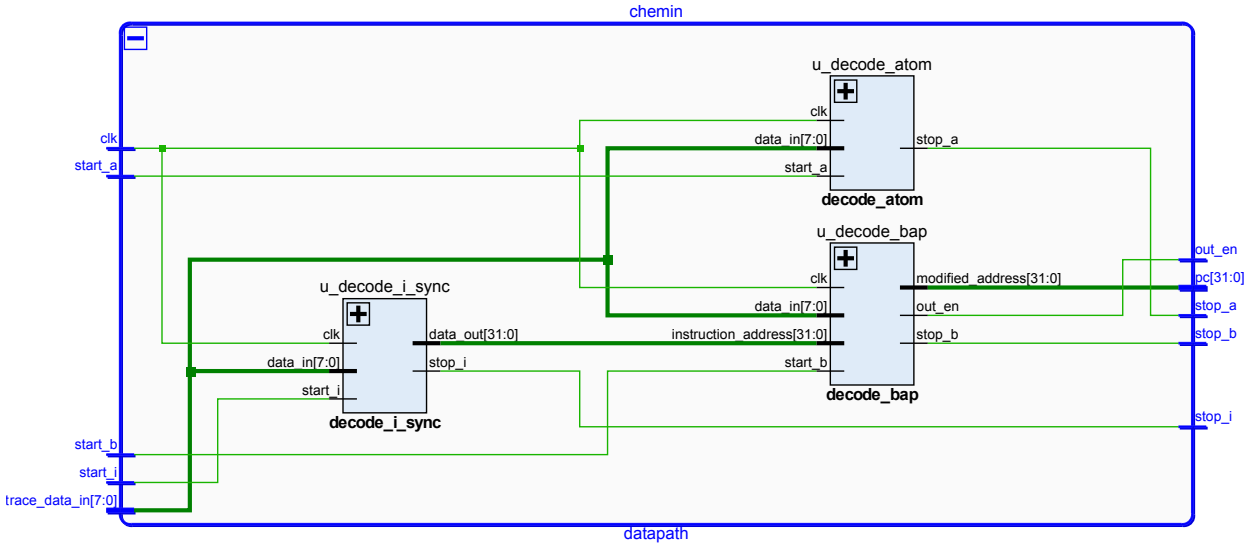

Figure 3.11: PFT Decoder v2 FSMs

Decoder v2 IP is shown in figure 3.13. There are still few things to modify in the actual design to get an IP working at 100%.

The global design realized in order to test the Ip is presented in figure 3.14.

- FCLK_CLK0 = 250 MHz
- FCLK_CLK1 = 125 MHz
- BRAM Memory type : True Dual port
- BRAM Mode : BRAM Controller

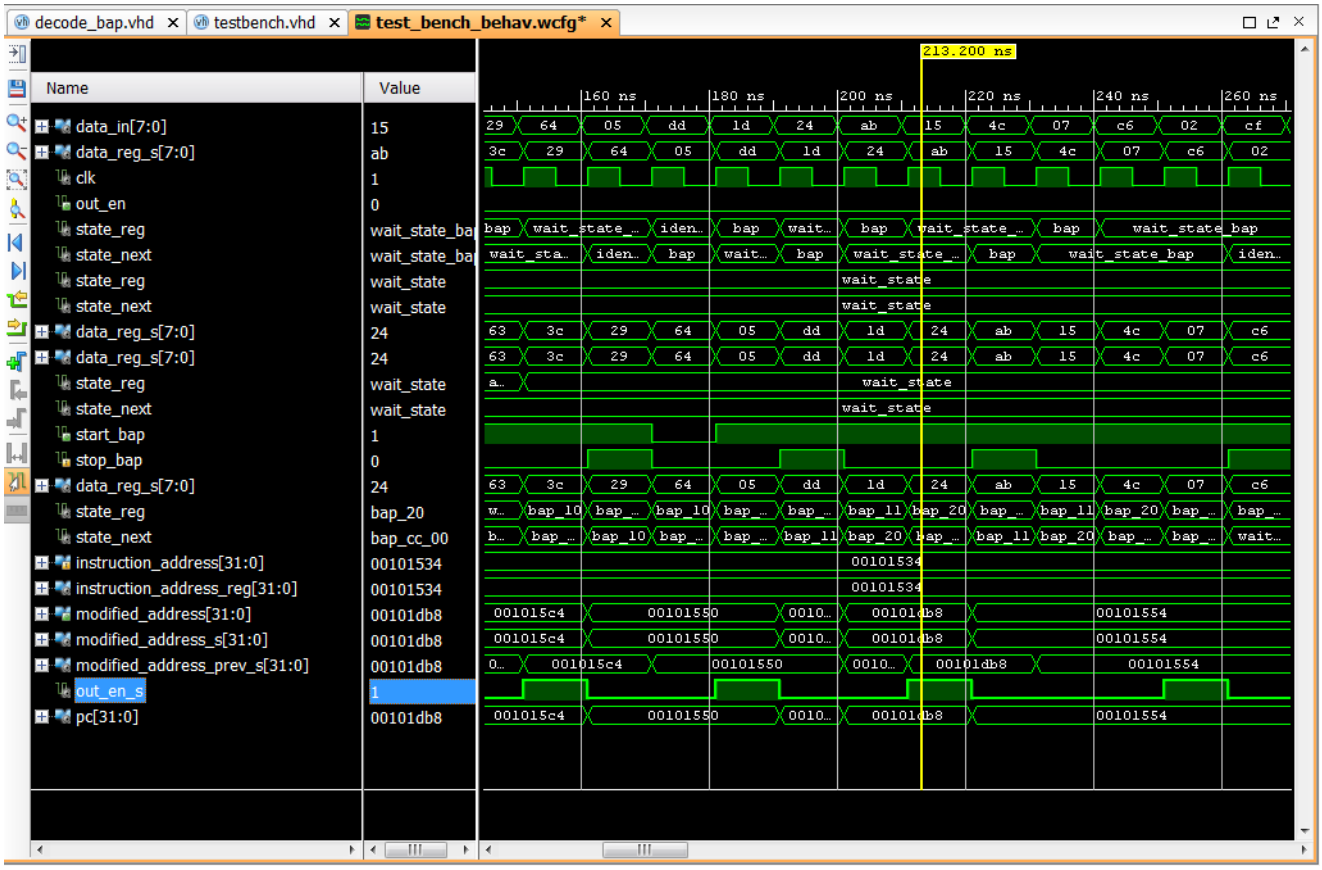

Figure 3.12: PFT Decoder v2 Simulation

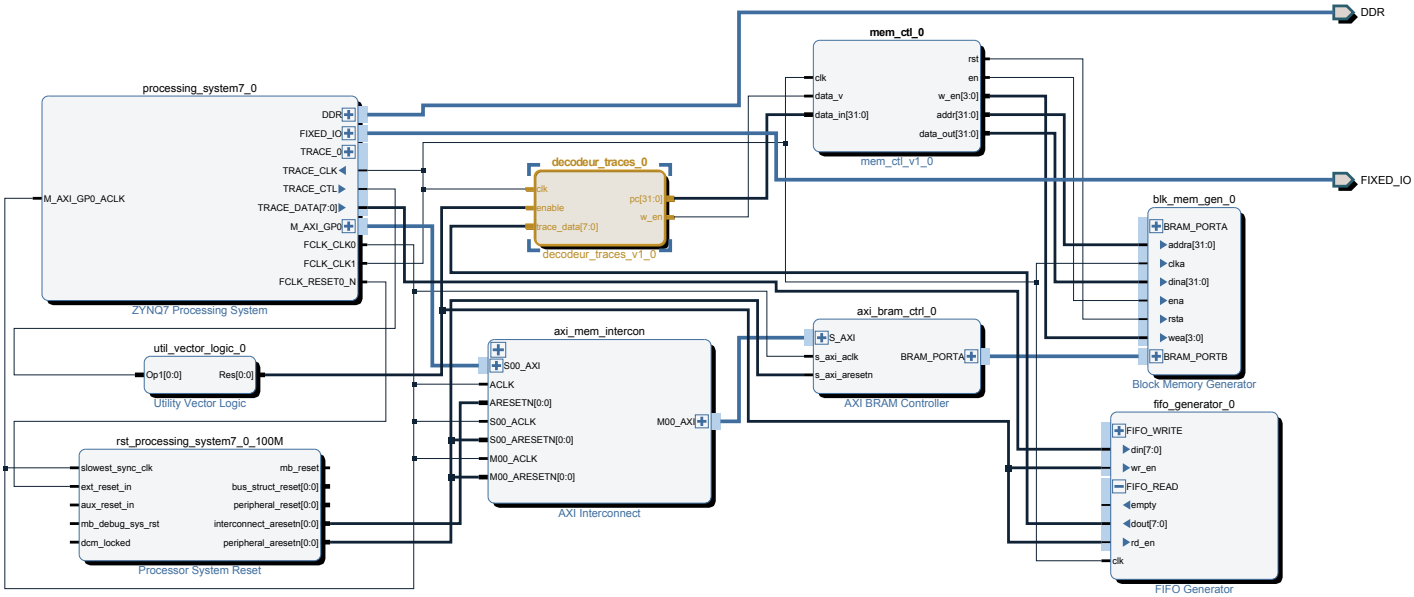

Figure 3.13: PFT Decoder v2 in Global Designs

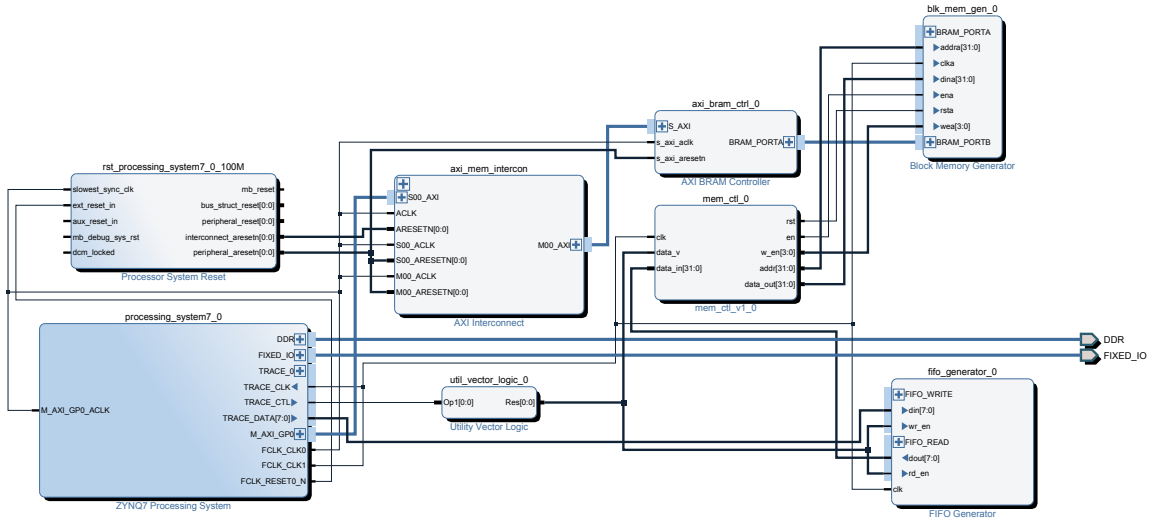

Figure 3.14: TPIU traces PL design

- mem_ctl version is located in E:\Vivado \test \mem_ctl or mem_ctl.zip file on my personal gforge github (redaction_hw)
- utility vector logic is configured to make a not gate
- fifo generator is configured with native interface, common clock BRAM, ports write width of 8 bits

4 Related Work

4.1 Articles read

HARMONI, 2012 [1]

Here is a brief summary of entitled "High-Performance Parallel Accelerator for Flexible and Efficient Run-time monitoring". The architecture proposed in the article was designed to meet frequency requirements. The state of the art articles at the time did not take into consideration this aspect and this is the only article that tries to face this problem. The proposed architecture named HARMONI could work upto 1250 MHz. Furthermore the decoupled co-processor approach is chosen to realize runtime monitroing techniques mainly the following :

1. DIFT
2. UMC (Uninitialized Memory Checking which consists of checking if a write has taken place into the memory location we are reading)
3. BC (memory Bound Checking)
4. RC (Reference Counting)

Three types of tags are defined :

1. Value tag (tag associated to data)
2. Location tag (tag associated to memory location)
3. Object tag (tag for highl-level objects such as structures, arrays, classes,...)

Four types of operations are done on the tags:

1. Read
2. Update
3. Check
4. Write-back

A FIFO is used to forward entries (64) to the coprocessor. The entries are selected based on the opcode and includes :

- opcode
- register indexes of source and destination registers
- the accessed memory address on a load/store
- pointer value for high-level object

The processor and co-processor can run in parallel because of the FIFO. The synchronization is done on system calls.

WHISK: An Uncore Architecture for Dynamic Information Flow tracking in Heterogeneous Embedded SoCs [2]

A quick research on the Internet allowed me to find another interesting article on applying DIFT in SoCs. What's interesting is the background study which shows how different things (such as tag storage, tag management) were done in previous architectures. This explains a little bit more how this was done previously. Most of the things explained were seen before in other articles before but here a summary can be found.

For example, for tag storage there are two types of schemes : coupled and decouples schemes. Coupled schemes consists of physically adding tag bits in the architecture which has the advantage of accessing atomically access both the tag and data in the instruction cycle. The main drawback is the need to modify the existing architecture to carry tag bit(s).

The other approach for tag storage is the decoupled one which consists of storing data and tags separately. The main drawback is the fact that an association algorithm is needed to find the associated tag to the data.

MutekH is used as an operating system (never heard of it) and the implementation of the architecture is done using systemC. The proposed architecture consists of a DIFT wrapper ...

Bibliography

[1] Daniel Y. Deng and G. Edward Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[2] Joël Porquet and Simha Sethumadhavan. Whisk: An uncore architecture for dynamic information flow tracking in heterogeneous embedded socs. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 4:1–4:9, Piscataway, NJ, USA, 2013. IEEE Press.