USER GUIDE

M.Abdul WAHAB

March 28, 2017

Contents

Contents **II**

List of Figures **III**

List of Tables **IV**

1 Introduction **2**
1.1 DIFT . . . 2

2 Retrieving Trace **4**
2.1 Programming CS components . . . 5

3 IP development **9**
3.1 Goal . . . 9
3.2 Dividing the development of IP into multiple components . . . 9
3.3 Timing violation problem . . . 9
3.4 PFT Decoder V2 . . . 12

4 CoreSight components in Linux **17**
4.1 Support for Zedboard . . . 17
4.2 CoreSight PTM features . . . 17
4.3 CoreSight TPIU . . . 17

5 Overall architecture of first prototype **18**
5.1 Design . . . 18
5.2 Implementation . . . 18

6 CFI: Preventing from ROP attacks **20**
6.1 Existing solution(s) and limitations . . . 20
6.2 Our solution . . . 20
6.3 Implementation results . . . 20

7 Related Work **21**
7.1 Articles read . . . 21

Bibliography **23**

List of Figures

2.1 Trace retrieval from ARM Cortex-A9 through Coresight Components (PTM) 4
2.2 PTM registers programming order 5
2.3 Coresight Funnel (Link class) 6
2.4 Coresight Funnel (Link class) 6
2.5 TPIU Traces connected to EMIO 8
2.6 TPIU Trace CLOCK 8

3.1 TPIU Trace CLOCK 10
3.2 Design to test BRAM 10
3.3 Vivado Synthesis settings 11
3.4 Vivado Synthesis settings 11
3.5 Vivado Address editor 11
3.6 Generated BSP - Xilinx SDK 12
3.7 Timing violation Path (in Blue) global design 13
3.8 Timing violation Path 13
3.9 Timing violation - Delay 14
3.10 PFT Decoder v2 Schematic 14
3.11 PFT Decoder v2 FSMs 14
3.12 PFT Decoder v2 Simulation 15
3.13 PFT Decoder v2 in Global Designs 15
3.14 TPIU traces PL design 16

List of Tables

2.1 Coresight Components types and function 4
2.2 PTM Registers and Values 5
2.3 PTM Configuration register 6
2.4 FUNNEL Registers and Values 7
2.5 ETB Registers and Values 7

3.1 PFT packet formats 9

5.1 Example tag dependencies instructions 18

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
M.Abdul WAHAB	2.2	28/03/2017	Changed outline and added few details in chapters 4 and 5

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.

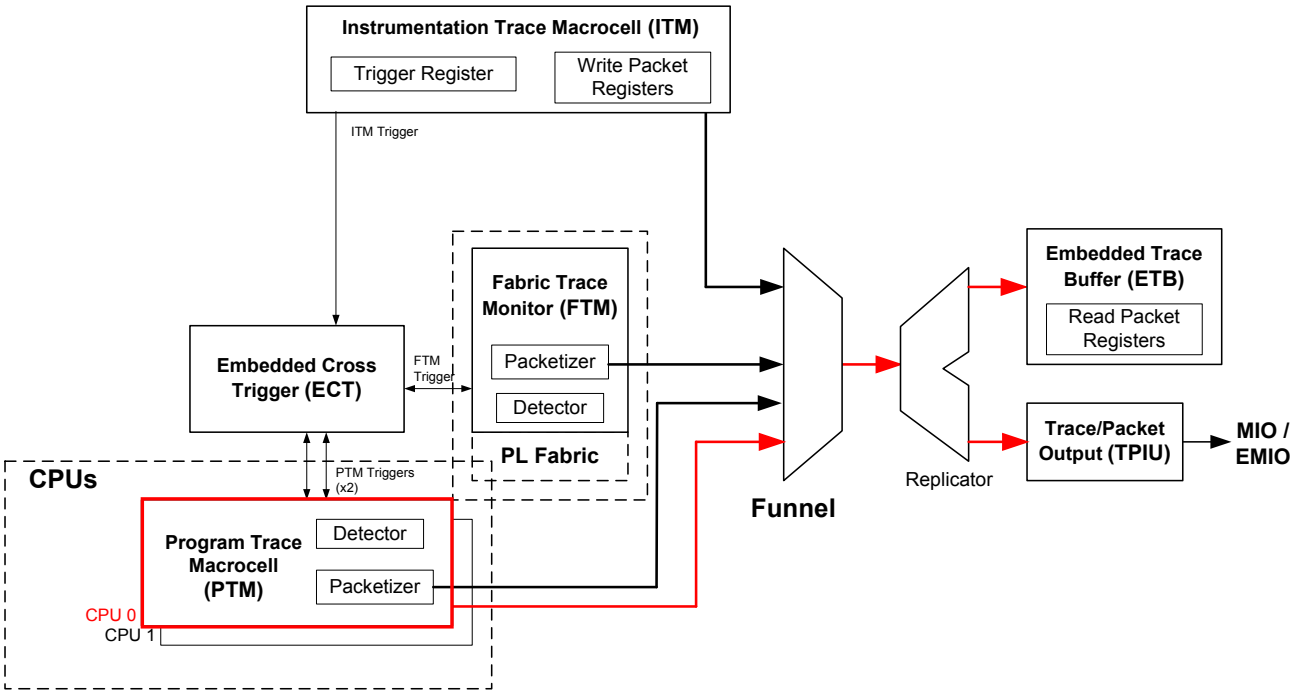

Figure 2.1: Trace retrieval from ARM Cortex-A9 through Coresight Components (PTM)

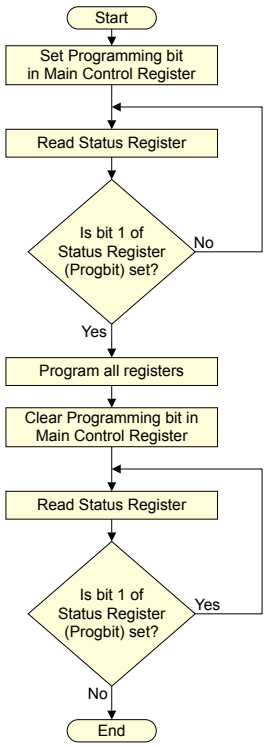

Figure 2.2: PTM registers programming order

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

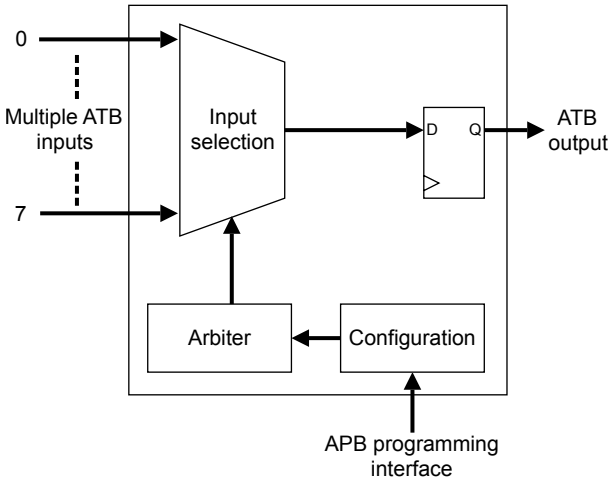

Figure 2.3: Coresight Funnel (Link class)

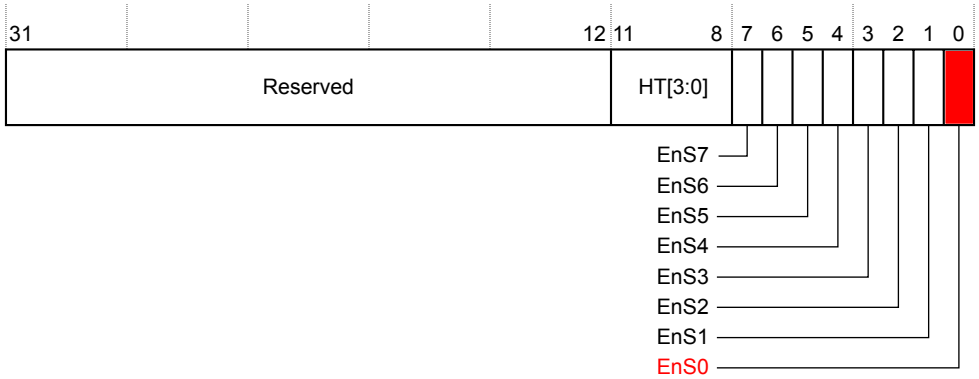

Figure 2.4: Coresight Funnel (Link class)

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 **??**.

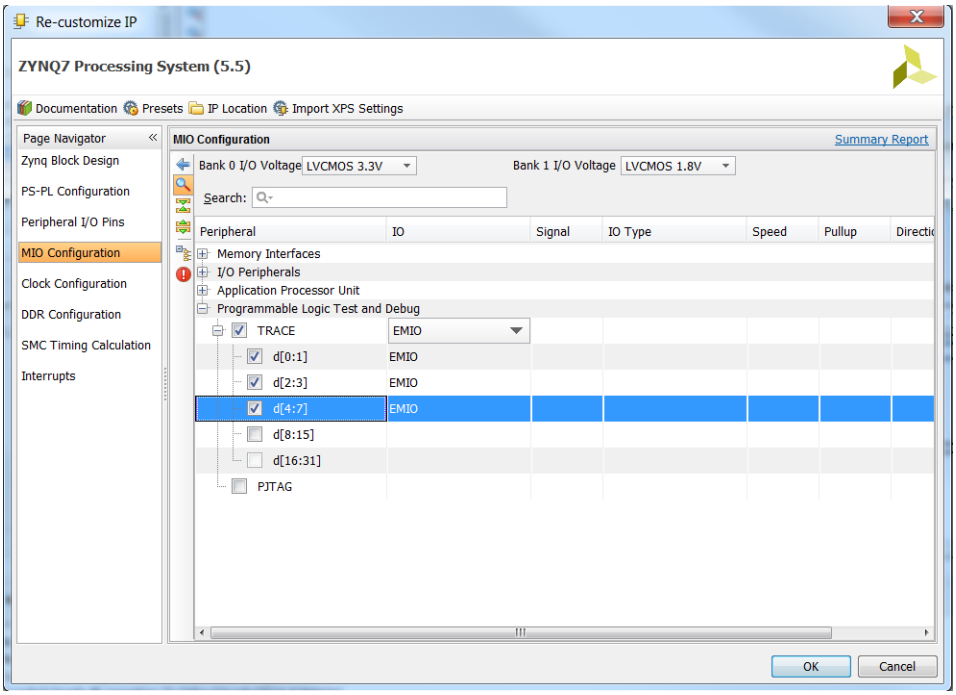

Figure 2.5: TPIU Traces connected to EMIO

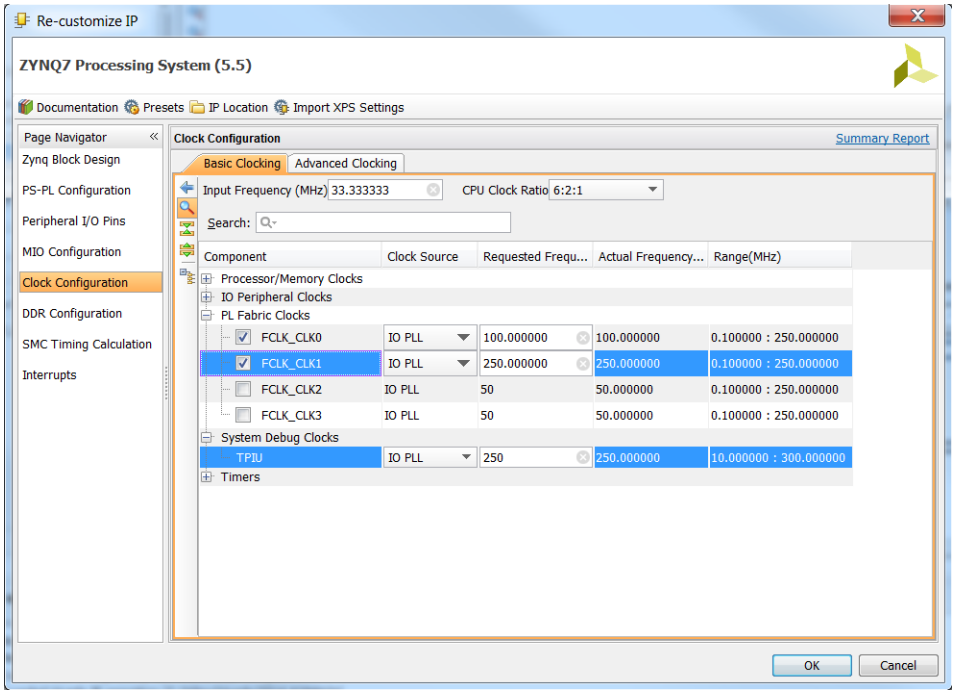

Figure 2.6: TPIU Trace CLOCK

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

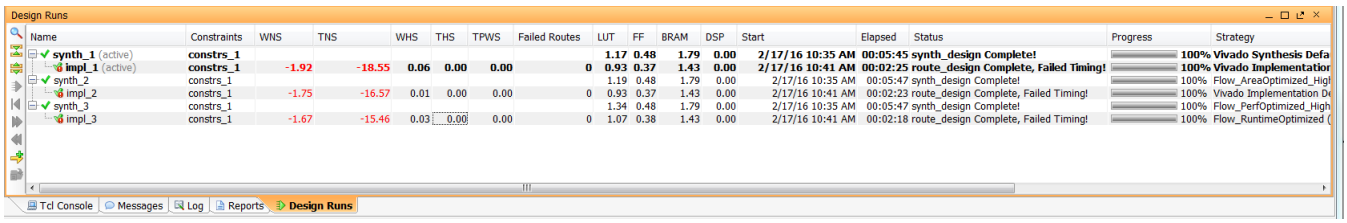

Figure 3.1: TPIU Trace CLOCK

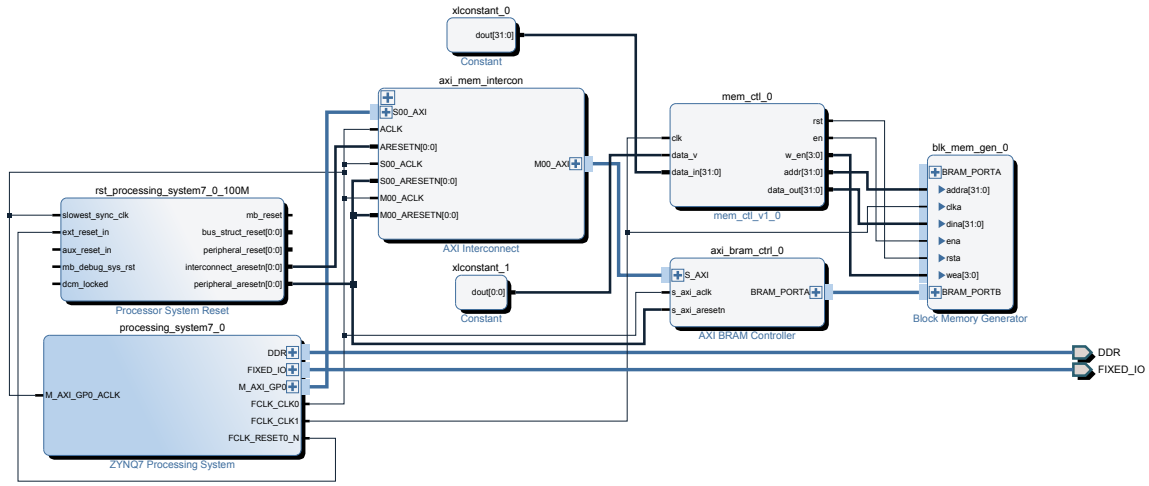

Figure 3.2: Design to test BRAM

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

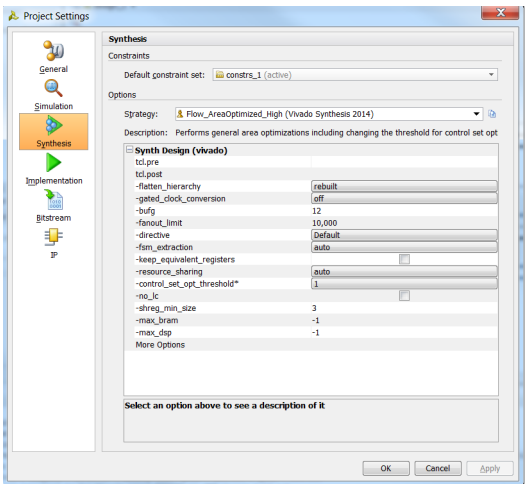

Figure 3.3: Vivado Synthesis settings

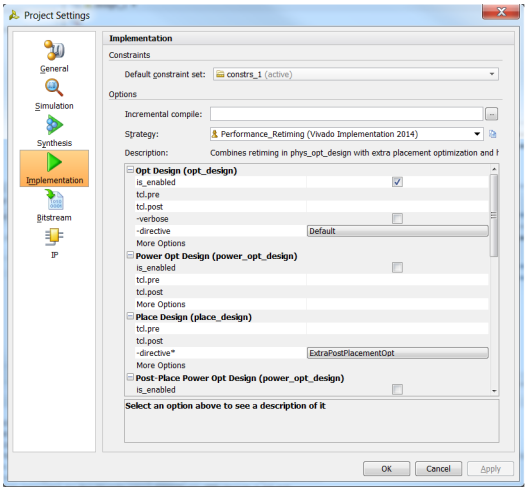

Figure 3.4: Vivado Synthesis settings

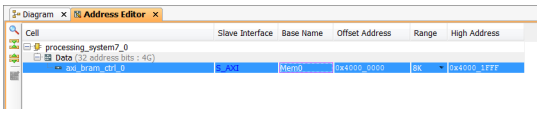

Figure 3.5: Vivado Address editor

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

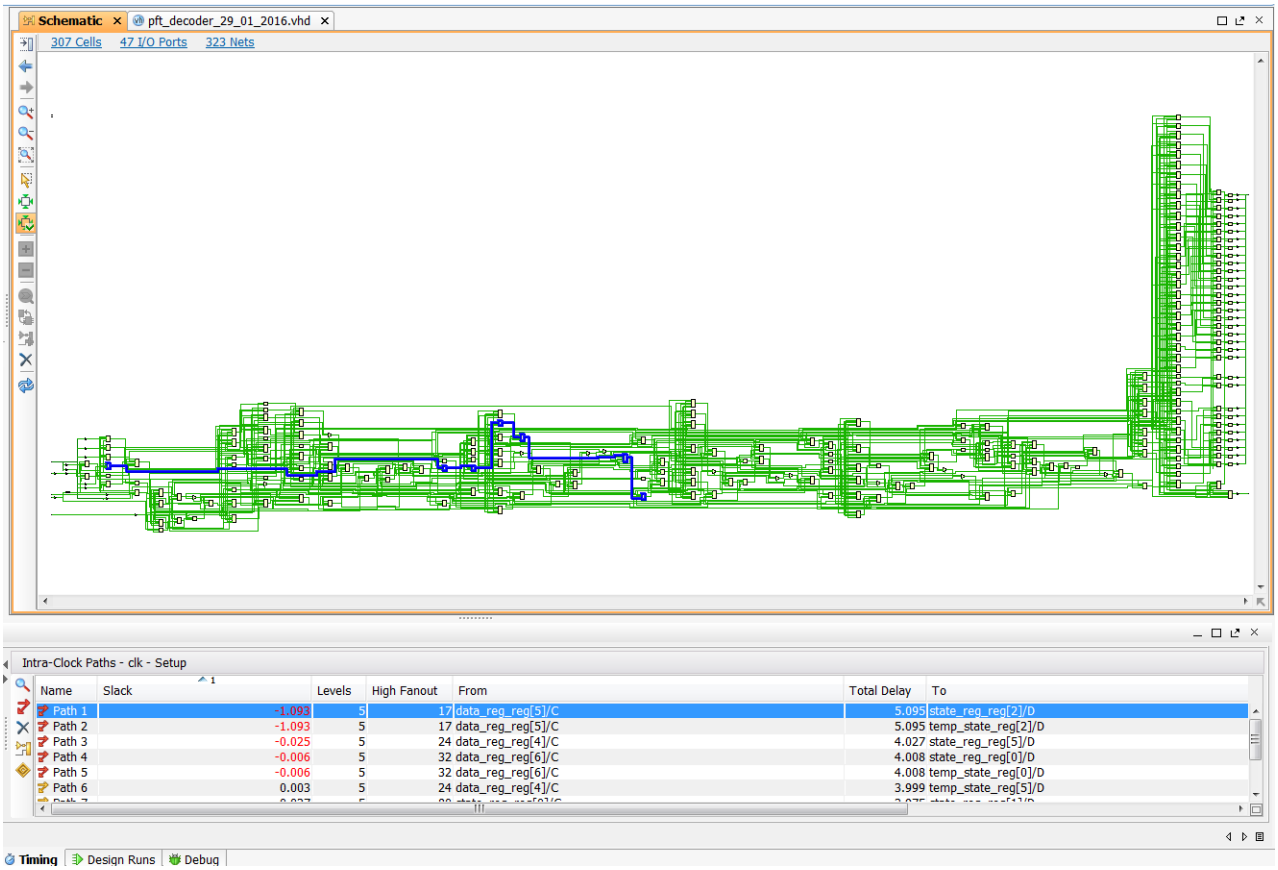

Figure 3.7: Timing violation Path (in Blue) global design

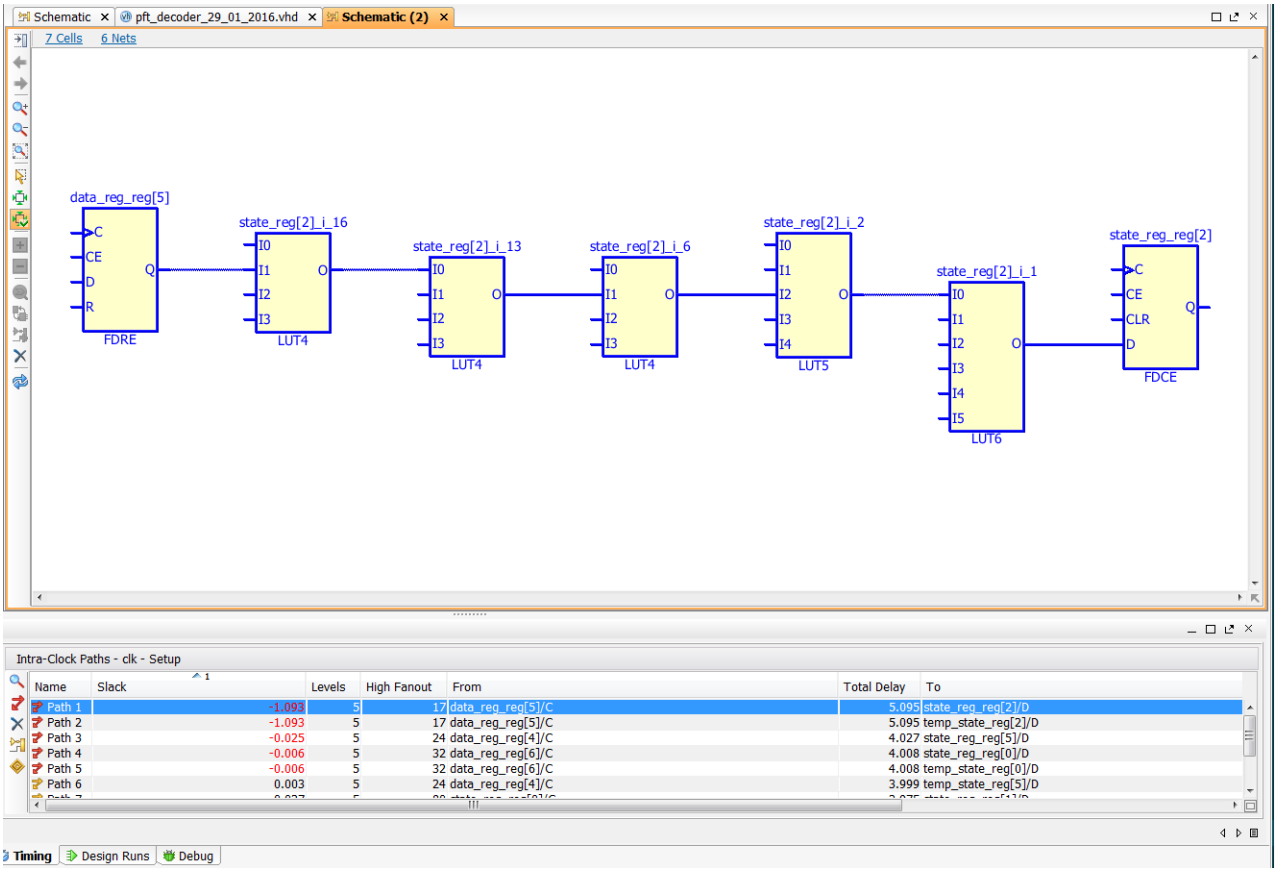

Figure 3.8: Timing violation Path

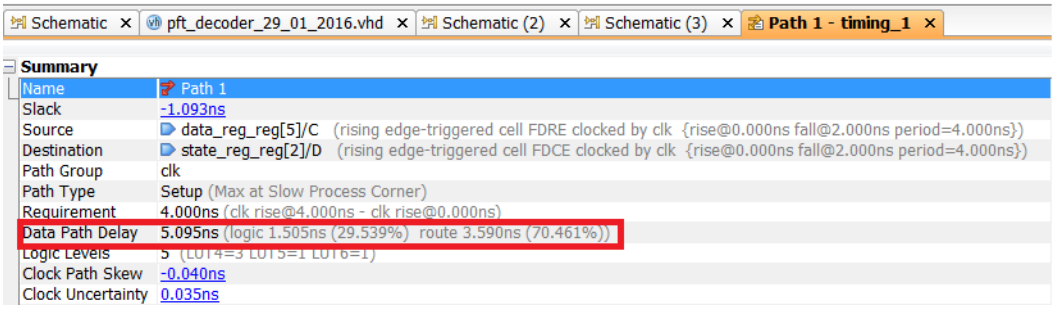

Figure 3.9: Timing violation - Delay

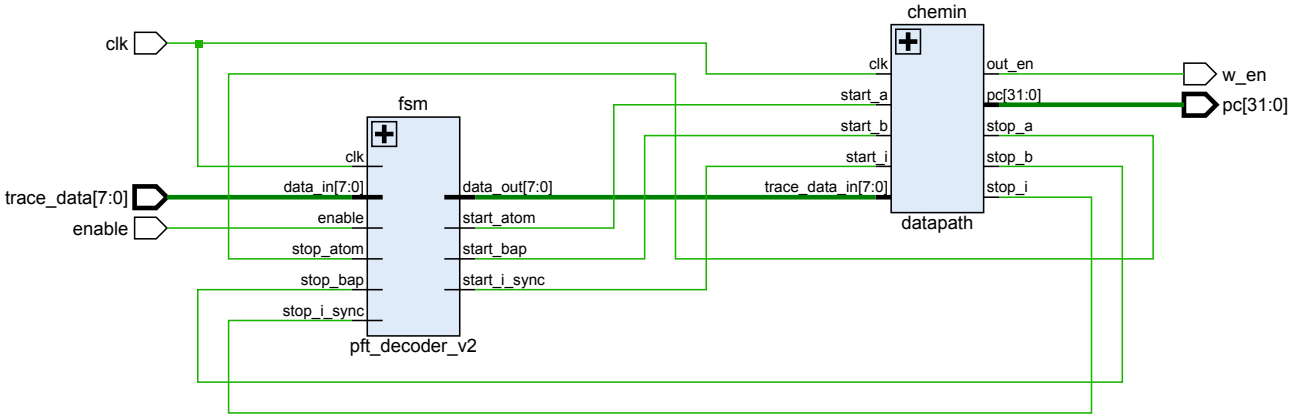

Figure 3.10: PFT Decoder v2 Schematic

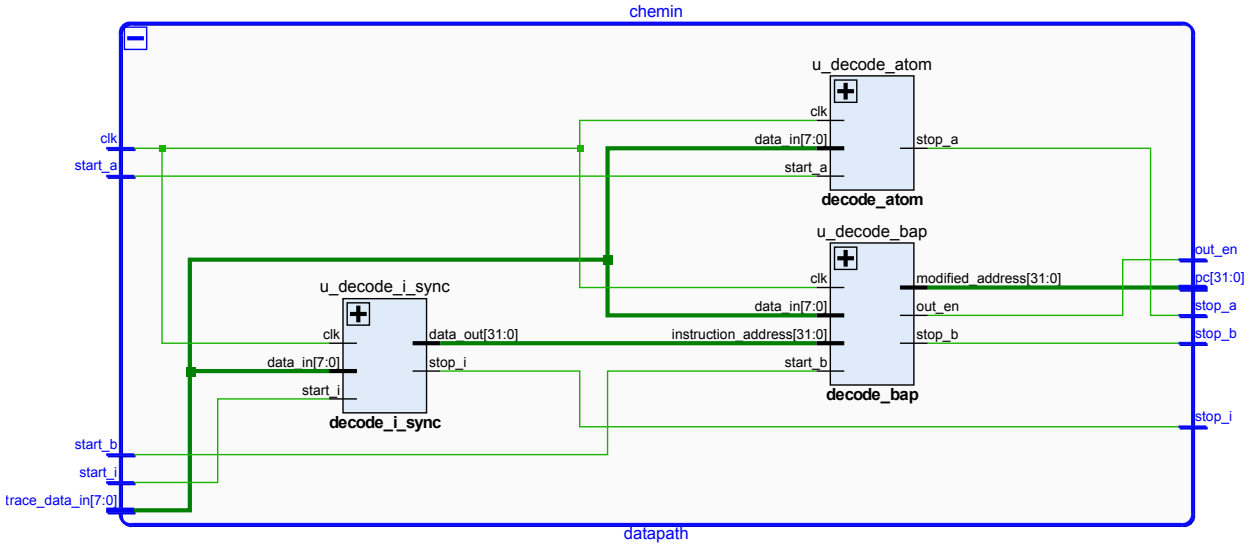

Figure 3.11: PFT Decoder v2 FSMs

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

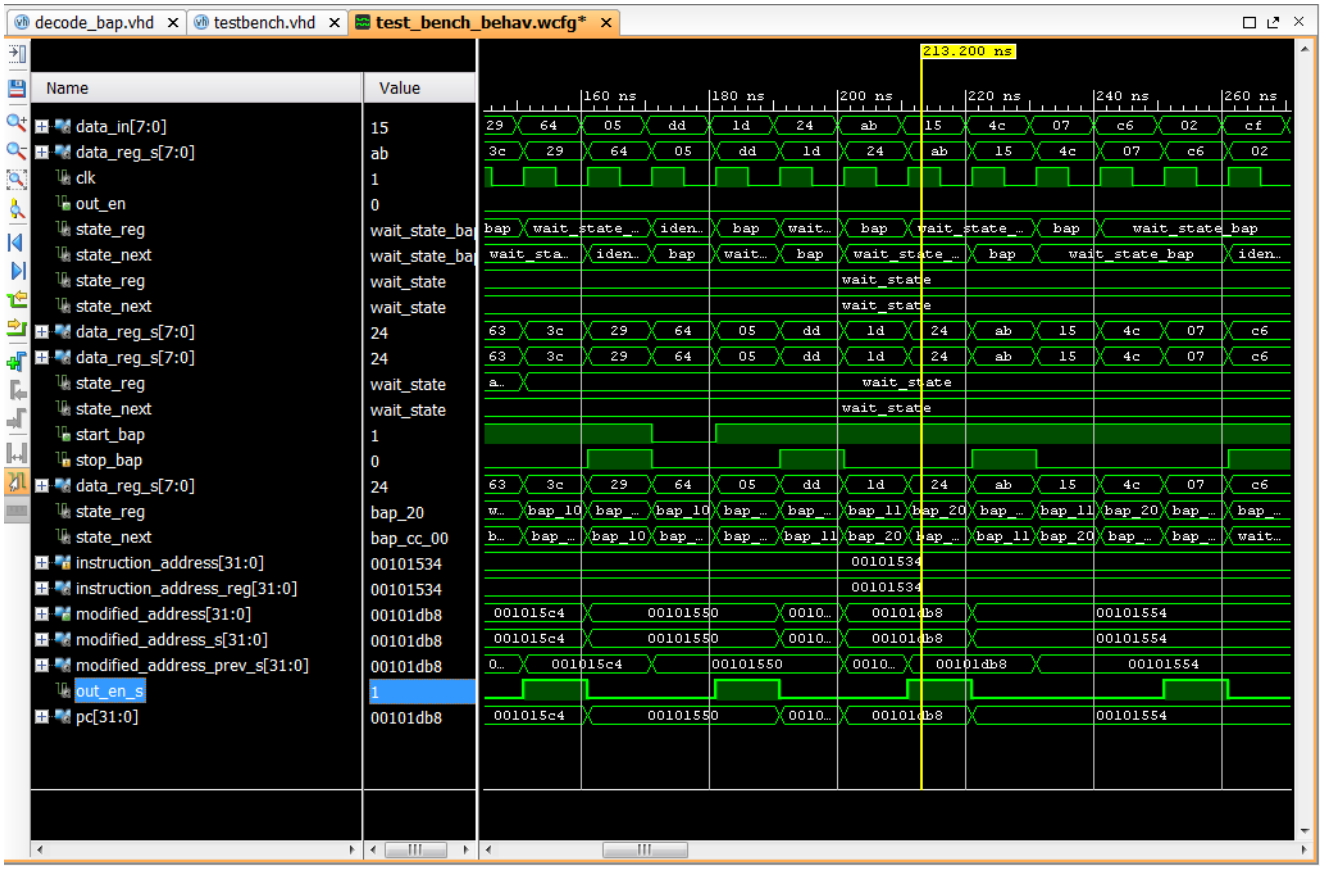

Figure 3.12: PFT Decoder v2 Simulation

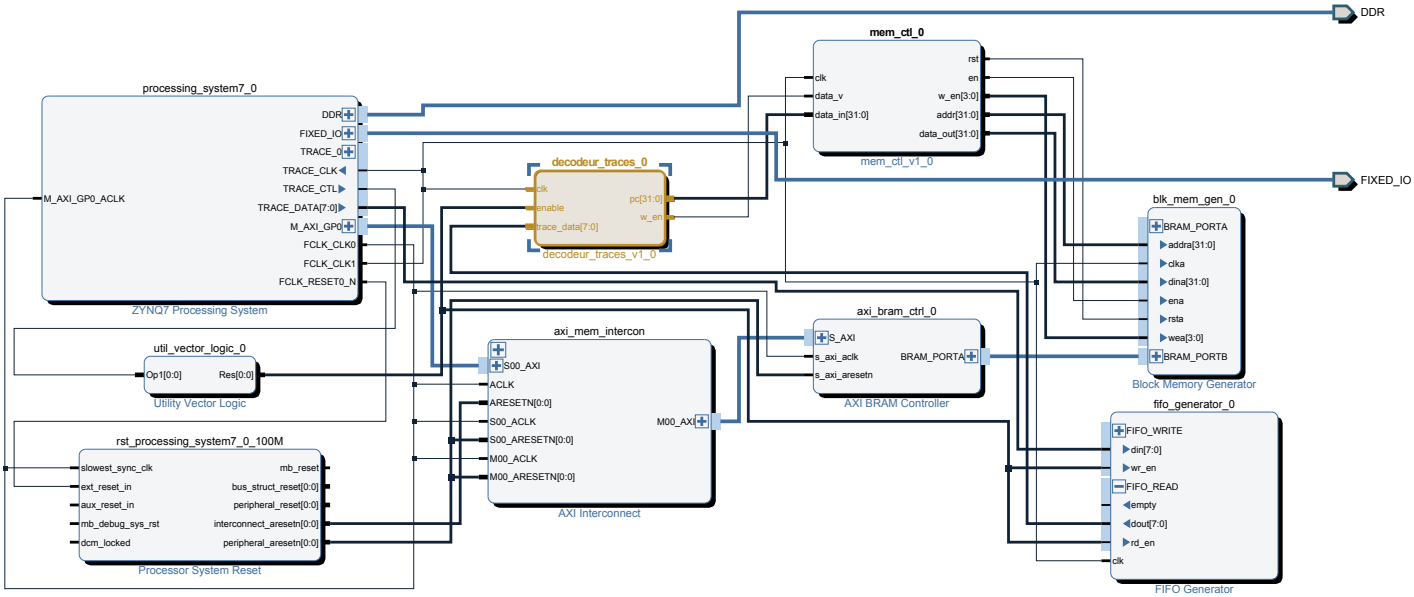

Figure 3.13: PFT Decoder v2 in Global Designs

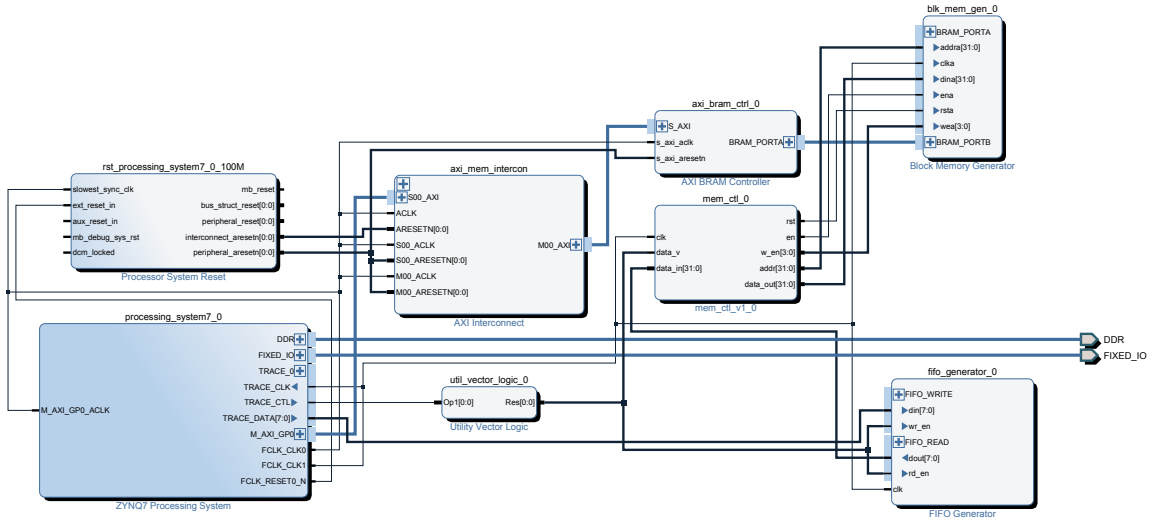

Figure 3.14: TPIU traces PL design

- 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 *CoreSight components in Linux*

The drivers of CoreSight components in Linux did not support our targeted device (Zynq). The first thing done was to add the support for Zynq7000 SoC. Then, the drivers needed to be modified to recover trace with expected features. Besides, we noticed that there were few problems with the TPIU driver as well. It was modified as well to recover trace in reconfigurable logic.

4.1 Support for Zedboard

4.2 CoreSight PTM features

4.3 CoreSight TPIU

5 *Overall architecture of first prototype*

5.1 Design

First DIFT coprocessor: Microblaze

5.2 Implementation

First DIFT coprocessor program

Interrupt

Communication between PS (Linux) and PL

User Space

Kernel Space

Static analysis

Table **??** shows that instructions at address `860c`, `8614`, `8618`, `861c` and `8624` do not produce any trace. To recover information about all other instructions not contained in trace, the source code of program is statically analysed before execution of the program. The static analysis will generate a tag dependencies instruction that needs to be executed by DIFT coprocessor.

Table 5.1: Example tag dependencies instructions

Example Instruction	Tag dependencies instruction
`sub r0, r1, r2`	$\underline{r0} = \underline{r1}$ OR $\underline{r2}$
`mov r3, r0`	$\underline{r3} = \underline{r0}$
`str r1, [SP, #4]`	$\underline{@Mem(SP+4)} = \underline{r1}$
`ldr r3, [SP, #-8]`	$\underline{r3} = \underline{@Mem(SP-8)}$
`str r1, [r3, r2]`	$\underline{@Mem(r3+r2)} = \underline{r1}$
(a)	**(b)**

If the code in Table 5.1(a) is considered, the resulting tag dependencies instruction is shown in Table 5.1(b). The tag dependencies instruction specify the operation to be realized on ARMHEx coprocessor. $\underline{r}$ is used to denote the tag of register `r`. For instance, for the first instruction in Table 5.1, the corresponding dependencies instruction is to associate tags of operands `r1` and `r2` towards the tag of destination register `r0`.

In Algorithm 1, we illustrate our algorithm that generates tag dependencies instructions. The idea is to analyze each instruction's encoding to find operands and determine information flows between these operands. The presented algorithm is applied to all basic blocks of the application to retrieve tag dependencies instructions for each basic block. These instructions are encoded and stored in a memory section.

The `RecoverOperands` function on line 3 determines the operands for each instruction. For each operand, we determine its type (register, immediate or memory) and its value (register number, immediate value or memory address) that are stored in two lists `operandsType` and `operand` respectively. From line 4 to line 21, we determine information flows that takes place between different operands of an instruction. If an instruction has two operands and both of them are registers (case line 6), there is an information flow from the 2nd register towards the 1st register. However, if both registers are the same, there is no information flow. The conditional operation on line 7 allows to eliminate such unnecessary tag dependencies instruction. There are other cases that are not shown for simplicity. For all possible cases, we generate a tag dependencies instruction.

In ARM instruction set, most of the instructions have maximum of four operands except for some instructions. The switch case instructions from line 5 to 15 generate tag dependencies for ARM instructions that have at most four operands. However, other instructions such as `push`, `pop`, `ldm` or `stm` instructions generate multiple information flows. They need to be analyzed separately to generate tag dependencies instructions.

The Algorithm 1 considers only instructions that are part of ARM instruction set. The same algorithm applies to Thumb instruction set as well but minor differences exist and are not explained here to simplify.

Algorithm 1: Algorithm for generating tag dependencies instructions

```
Input  : basic block
Output: tag dependencies instructions (tdi)
i = 0; operandsType = []; operand = [];
foreach instruction instr in basic block do
    operandsType[], operand[] = RecoverOperands (instr);
    if instr != (push OR pop OR ldm OR stm) then
        switch ⋃_i operandsType[i] do
            case [Reg, Reg] do
                if operand[1] != operand[2] then
                    tdi = operand[1] ← operand[2];
                end
            end
            case [Reg, mem] do
                tdi = operand[1] ← operand[2];
            end
            case ... do
        end
    else
        switch instr do
            case push OR stm do
                tdi = generateTDI (instr);
            case pop OR ldm do
                tdi = generateTDI (instr);
        end
    end
    InstructionChangesSP (instr) // Strategy 2 only
end
```

Furthermore, the presented algorithm works for architecture ARM v7-A. It can be easily adapted to any other ARM architecture by taking into consideration minor differences that exist between instruction sets.

Instrumentation

In table 5.1, the `ldr`,`str` instructions contains memory addresses. These addresses need to be known to propagate their associated tags. There are three types of memory instructions (`ldr`, `str`, `ldm`, `stm`, `vldr`, `vstr`): (i) PC relative, (ii) SP relative and (iii) register relative. Some memory addresses cannot be recovered statically. We can define two possible strategies to recover memory addresses.

Strategy 1 For each memory address, an instruction is added, to the source code of program, that sends memory address(es) to ARMHEx coprocessor.

Strategy 2 From all memory instructions, only register relative instructions are instrumented. ARMHEx coprocessor knows the value of the PC thanks to CoreSight components. Therefore, we do not need to instrument memory instructions that are relative to `PC` register. Furthermore, during static analysis, if an instruction changes the SP value directly or indirectly, an instruction is added to keep track of SP value. This is taken care of by the function (`InstructionChangesSP`) on line 34 of Algorithm 1. When the program is launched, the initial value of SP is sent to ARMHEx coprocessor which can then keep track of SP value changes thanks to instructions inserted during static analysis.

6

CFI: Preventing from ROP attacks

6.1 Existing solution(s) and limitations

6.2 Our solution

6.3 Implementation results

7 Related Work

7.1 Articles read

Most recent and closest work

To overcome high runtime overhead of software solutions, some related works propose to filter monitored events. In [8], a general filtering hardware accelerator that filters application events (e.g. instructions, system calls) to improve runtime performance. However, the average slowdown remains important (150%). A similar work [1] identifies common sources of slowdowns in software implementation and proposes a hardware technique to lower time overheads. However, the average slowdown remains important.

Dalton et al. [2] presented one of the first implementations of a DIFT with hardware modifications. Their approach Raksha is implemented on a LEON3, an open-source SPARC v8 softcore processor developed by Gaisler Research [13]. Raksha adds several stages in the pipeline to compute tags: in other words, the architecture itself has been modified. As a consequence, performing DIFT with Raksha requires a custom processor which limits its adoption. Furthermore, Raksha is running at a low frequency of 20MHz while increasing gate count of 4.85% over base LEON. The average time overhead for the most efficient implementation is about 200%.

Flexitaint [14] is an evolution of Raksha. Authors wanted to lower the impact of modifications on the processor pipeline. The Flexitaint accelerator is an in-order component placed at the end of the pipeline. This approach was simulated on a high-end processor, while ARMHEx mainly focuses on embedded system applications with lower requirements regarding processor performances. In terms of performance, Flexitaint shows latency overheads up to 8% when tags propagation is enabled. Regarding in-core approaches, Davi et al. [3] implemented a similar approach with a proprietary architecture based on Intel Siskiyou Peak and SPARC (LEON3 processor).

In [2, 14], both implementations required processor modifications: this is a major drawback as DIFT mechanisms would be hardly portable to other architectures. Offloading approaches suggest to export DIFT computations to another module. In this area, Nagarajan et al. [11] propose to use a dual-core architecture where one core executes the main application and the operating system while the other is in charge of DIFT. This approach still adds up to 48% in terms of computation time. Furthermore, this approach is obviously not a low-power solution as it requires an additional core identical to the main one. More recently, Jee et al. described ShadowReplica [9]: this work showed a slowdown of 20% on Apache web server with an implementation on an Intel Xeon multicore architecture.

Offloading approaches are interesting as they nearly decouple DIFT computations from the main processor pipeline. The main drawback is that it needs another core for DIFT: in most cases, this additional core is oversized for such applications. Therefore, other works had a look at off-core DIFT implementations where tag computations are performed on a dedicated coprocessor.

Kannan et al. [10] proposed an evolution of Raksha [2]. The communication between the coprocessor and the main core is done through a decoupling queue. The main core is configured to transmit information needed for DIFT to the coprocessor (the instruction tuple containing program counters, load/store memory addresses and instruction encoding). The coprocessor is also able to minimize stalls if both components do not run at the same frequency. Raksha as designed by Kannan et al. was fully implemented on a Xilinx Virtex-II FPGA using a LEON3 processor: the area ratio coprocessor/processor is around 8%. Deng et al. [4, 5] proposed Flexcore and Harmoni which are two approaches similar to Raksha [10]. As for Raksha, it is implemented using a LEON3. Harmoni hardware extension [5] can be used for several security procedures; it presents time overheads of less than 15% for DIFT. Dhawan et al. [7] implemented PUMP, a generic hardware support for metadata processing on a simple RISC processor.

HARMONI, 2012 [6]

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 [12]

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] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. *SIGARCH Comput. Archit. News*, 36(3):377–388, June 2008.

[2] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. *SIGARCH Comput. Archit. News*, 35(2):482–493, June 2007.

[3] L. Davi, M. Hanreich, D. Paul, A. R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. Hafix: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[4] Daniel Y. Deng, Daniel Lo, Greg Malysa, Skyler Schneider, and G. Edward Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 137–148, Washington, DC, USA, 2010. IEEE Computer Society.

[5] 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.

[6] 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.

[7] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. *SIGARCH Comput. Archit. News*, 43(1):487–502, March 2015.

[8] Sotiria Fytraki, Evangelos Vlachos, Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. FADE: A programmable filtering accelerator for instruction-grain monitoring. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*.

[9] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 235–246, New York, NY, USA, 2013. ACM.

[10] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 105–114, June 2009.

[11] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. Dynamic information flow tracking on multicores. *Workshop on Interaction between Compilers and Computer Architectures, (colocated with HPCA)*, feb 2008.

[12] 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.

[13] Gaisler research. *LEON3 Processor*, 2010. http://www.gaisler.com/index.php/products/processors/leon3.

[14] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 173–184, Feb 2008.