VexRISC-V processor implementation on Intel FPGA PAC Card

Report

Part of Intel Research Fellowship

By

Khyamling

Under the Mentorship of

Rajesh Vivekanandam and Israr Ahmed Sheikh

Intel, Bangalore, Karntaka, India

August, 2021

Contents

1 Introduction

RISC-V is an emerging instruction-set architecture suitable for a wide variety of applications, which ranges from simple microcontrollers to high-performance CPUs. RISC-V is an open ISA, modern, extensible and it has a comprehensive infrastructure of open-source specifications, compilers, libraries, operating systems, and Interface IP. RISC-V processor implementations range from a single processor to heterogeneous multiprocessor architecture. Additionally, soft processor implementation allows the ISA to be customized and extended to suit a specific application. This brings the benefits of making the FPGA easier to program by presenting the view of a conventional multi core CPU. Simultaneously, it also enabling FPGA only custom logic optimizations for the key performance sensitive components of the workloads rather than forcing the entire code base to be ported to HDL. In this report, a 32-bit VexRiscv based on RV32I CPU instruction set has been designed using Intel PAC card. The ISA of VeXRiscv was extended to support multiple additional instructions such as standard Extension and privileged instruction.

Further, we have analyzed various RISC-V soft and hard processors by considering the design parameters, frequency, area, and performance. The VexRISC-V processor has emerged as the state-of-the-art FPGA optimized soft processor. Finally, the design has been examined on the Intel FPGA PAC card such as Arria 10 and Stratix 10. The logic synthesis, placement, and routing were performed incrementally changing the clock constraints. The FPGA area utilization and the maximum operating frequency of VexRISC-V. To enhance the performance of the VexRISC-V processor on the Stratix 10 FPGA board, the FPGA optimization strategies, HyperFlex register, and HyperFlex pipelining techniques were employed. After synthesis, placement, and routing, we observed that the optimized design has a higher operating frequency by utilizing the same area. The VexRISC-V implemented on the Agilex FPGA board with HyperFlex techniques achieves the maximum operating frequency 25% higher than Intel Arria 10 PAC card.

2 Generate the source code of VexRISC-V

We did an extensive survey of overlay and soft processor architectures targeting FPGAs and implementing the RISC-V ISA. Based on the survey, we notice that the VexRISC-V processor design is an FPGA optimized soft processor compared to the state-of-the-art. The VexRISC-V architecture is classified into different CPU instances based on size and performance, which is range from small CPU to CPU with Linux balanced. We choose the VexRISC-V full instance of CPU for this project work, which offers a high performance by consuming less area.

The VexRISC-V full Instance features:

- RV32IM instruction set
- 5-stages
- 4KB-ICache,4KB-Dcache.
- Single cycle barrel shifter.
- Debug module
- Catch exceptions
- Static branch

2.1 Steps for generating RTL code of VexRISC-V

First, Install dependencies software on the Linux platform. The Java JDK 8 and Scala Sbt Software are installed already on the Linux platform, then start with step3. Otherwise, follow steps 1 to 4.
1. JAVA JDK 8

- sudo add-apt-repository -y ppa:openjdk-r/ppa
- sudo apt-get update
- sudo apt-get install openjdk-8-jdk -y
- sudo update-alternatives –config java
- sudo update-alternatives –config javac

2.Install scala SBT -(https://www.scala-sbt.org/)

- sudo apt-get update
- sudo apt-get install sbt

3. Download the VexRISC-V repository using below link

- https://github.com/SpinalHDL/VexRiscv

4. To generate the corresponding RTL as a VexRiscv.v file, run the below commands in the root directory of VexRISC-V repository

- sbt "runMain vexriscv.demo.GenFull"

The generated RTL code of VexRISC-V is used for building the softcore processor on the Intel PAC card. The implementation details of VexRISC-V as an Accelerator Functional Unit(AFU) on the Intel PAC card are explained in section(3).

3 VexRISC-V implementation on Intel PAC card

3.1 Getting Started with Accelerator Functional Unit(VexRISC-V) Development

The design flow of VexRISC-V on Intel FPGA PAC card.

Design Entry Tools

- Intel Quartus® Prime Pro Edition software.
- Open Programmable Acceleration Engine(OPAE) stack.
- OPAE SDK
- Platform Interface Manager-2.0

Platform Interface Manager(PIM) 2.0

The initial version of PIM supports the CCI-P interface for communication between the Host system and AFUrunning on FPGA. The PIM-2.0 is a significant upgrade over the first version. The AXI, Avalon, and CCI-P host interface options are available through PIM 2.0. Accelerator Functional Units(AFUs)continue to work on the new codebase in compatibility mode.
Update the PIM using the following command
update release.sh

3.2 Design of VexRISC-V on Intel PAC card

We implemented a custom VexRISC-V IP module with AXI interface using the Intel Platform designer tool shown in Fig.1. This custom IP module is used to build an SoC system with VexRISC-V core. The Platform Designer(Qsys) tool is used to build an SoC system with VexRISC-V core as shown in Fig. 2.

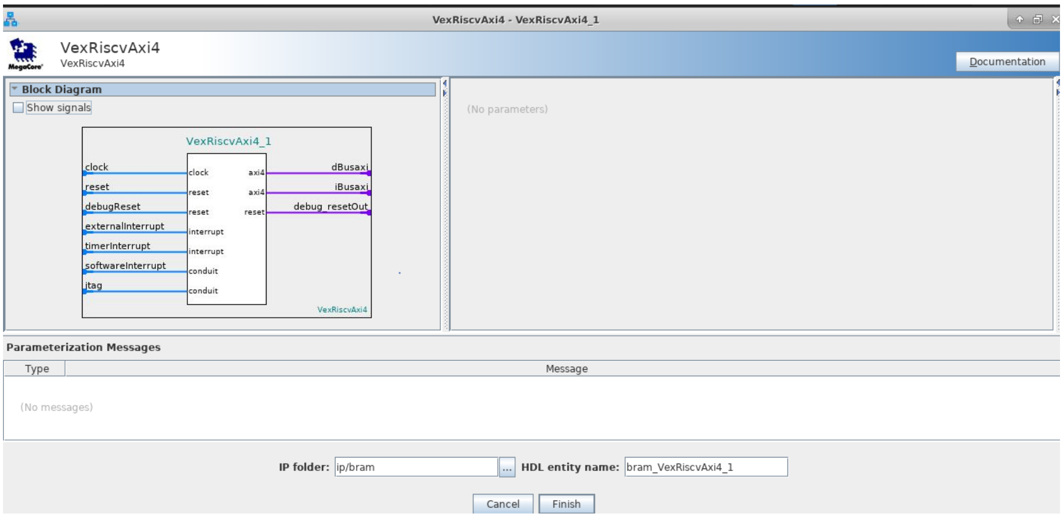

Figure 1: Custom VexRISC-V IP with AXI Interface

The SoC system has following IPs.

- 64Kb Block RAM IP
- JTAG Avalon master bridge IP
- JTAG UART IP
- VexRISC-V IP

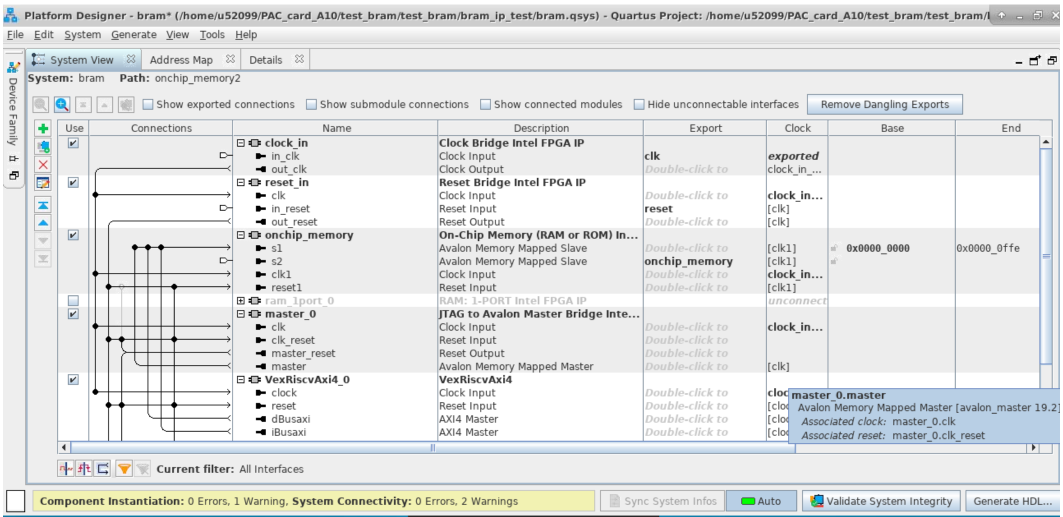

Figure 2: Design of SoC system with VexRISC-V core

Here, AFU is the custom implementation of the SoC system with VexRISC-V core. AFU can be accelerated on the OPAE hardware platform.

An AFU has two main communication paths between the host:

- FPGA to host
- Host to FPGA (MMIO)

FPGA to host

The FPGA accesses host memory using a 512 bit data path. This data path has separate channels for read and write traffic allowing for simultaneous read and write to occur. The read and write channels support bursts of 1, 2, and 4 cache lines.

Host to FPGA (MMIO)

The host can access a 256 KB address space within the FPGA. This address space contains Device Feature Header (DFHs) and the control and status registers of the AFU hardware. DFHs are small ROMs that hold metadata about the hardware that are enumerated by the OPAE SDK.

Further, the MMIO address space has been designed for communication between the Host and SoC system(AFU) with VexRISC-V core on the Intel PAC card.

A SoC system with VexRISC-V(AFU) design includes the following components

- RTL description of the SoC system with VexRISC-V(AFU) to accelerate on Intel FPGA PAC card. Fig. 3 shows all modules are integrated to build AFU.

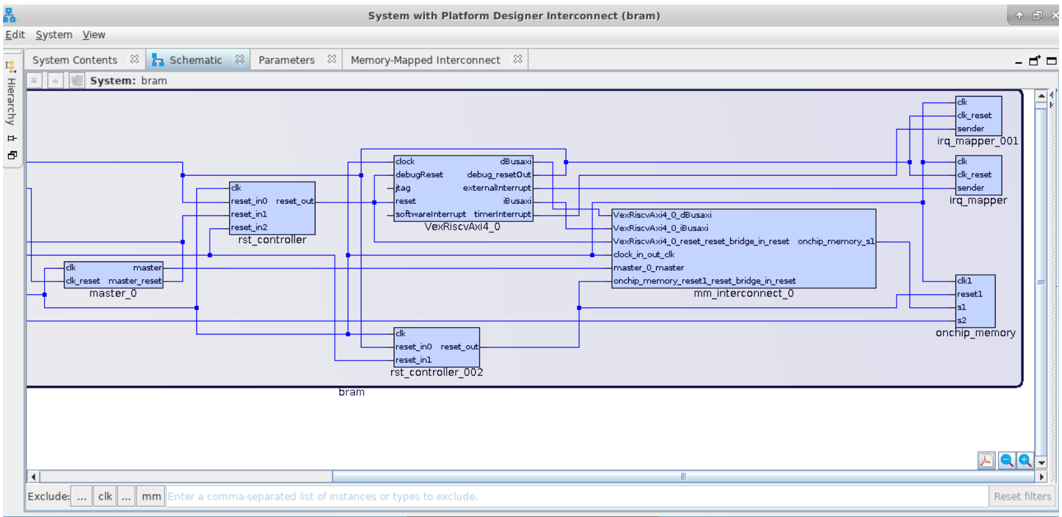

Figure 3: Design of AFU

- RTL description to implement the base requirements placed on AFUs by OPAE (for example DFH, AFU ID in MMIO space) as shown in Fig. 4.
- Fig. 5 shows the logic to map AFU CSRs into MMIO space.
- Memory mastering logic- The RTL code for local FPGA memory as shown in Fig. 6. Both Host and VexRISC-V access this local memory to read and write the data.
- Debug and monitoring using Signal Tap with the Remote Debug feature. Fig.7 shows the remote debugging of VexRISC-V AFU using signal tap.

```
  GNU nano 2.9.3                                              afu.sv


always_ff @(posedge clk)
 //always_comb
  begin
        mmio64_if.readdata <= '0;
        mmio64_if.readdatavalid <= mmio64_if.read && ! mmio64_if.waitrequest;
      case (mmio_address)
  //          case (mmio_address)
         16'h0000: // AFU DFH (device feature header)
                begin
                    // Here we define a trivial feature list.  In this
                    // example, our AFU is the only entry in this list.
                   mmio64_if.readdata  <= 64'b0;
                    // Feature type is AFU
                    mmio64_if.readdata[63:60] <= 4'h1;
                    // End of list (last entry in list)
                     mmio64_if.readdata[40] <= 1'b1;
                end
            // AFU_ID_L
            16'h0001: mmio64_if.readdata <= afu_id[63:0];
            // AFU_ID_H
            16'h0002: mmio64_if.readdata <= afu_id[127:64];

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    M-U Undo      M-A Mark Text
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line M-E Redo      M-6 Copy Text
```

Figure 4: Mandatory AFU CSRs RTL code(DFH, AFU ID)

```
  GNU nano 2.9.3                                              afu.sv

            default:
   // Check to see if the delayed address falls withing the
 if (mmio_address >= BRAM_BASE_MMIO_ADDR && mmio_address <= BRAM_UPPER_MMIO_ADDR)
      begin
             mmio64_if.readdata <= bram_rd_data[31:0];
end
        endcase
    end

    assign mmio64_if.response = '0;

   //assign mmio64_if.readdatavalid = mmio64_if.read && ! mmio64_if.waitrequest;


endmodule



^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    M-U Undo      M-A Mark Text
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line M-E Redo      M-6 Copy Text
```

Figure 5: Logic to map AFU CSRs into MMIO space

```
  GNU nano 2.9.3                                              afu.sv


bram   mmio_bram (
              .clk_clk(clk),
              .onchip_memory_write(bram_wr_en),
              .onchip_memory_address(bram_addr),
//            .onchip_memory_writedata(mmio64_if.writedata[BRAM_DATA_WIDTH-1:0]),
              .onchip_memory_writedata(mmio_writedata),
              .onchip_memory_readdata(bram_rd_data),
       //       .reset_reset_n(reset_n),
             .reset_reset_n(qsys_system_reset),
             .onchip_memory_clken(1'b1),
             .onchip_memory_chipselect(1'b1),
            //.onchip_memory_byteenable(8'b11111111)
              .onchip_memory_byteenable(4'b1111)
              );

   // Misc. combinatonal logic for addressing and control.
always_ff @(posedge clk) begin
//always_comb begin
logic [$size(mmio_address)-1:0] bram_offset_addr;

  bram_offset_addr <= mmio_address - BRAM_BASE_MMIO_ADDR;
  bram_addr <= bram_offset_addr[BRAM_ADDR_WIDTH:1];
 mmio_writedata <= mmio64_if.writedata[BRAM_DATA_WIDTH-1:0];
^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    M-U Undo      M-A Mark Text
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text ^T To Spell   ^_ Go To Line M-E Redo      M-6 Copy Text
```

Figure 6: logic for Local memory access

3.3 Build and compile the SoC system with VexRISC-V(AFU)

Development Environment References

- The OPAE_PLATFORM_ROOT environment variable points to the OPAE SDK installation.

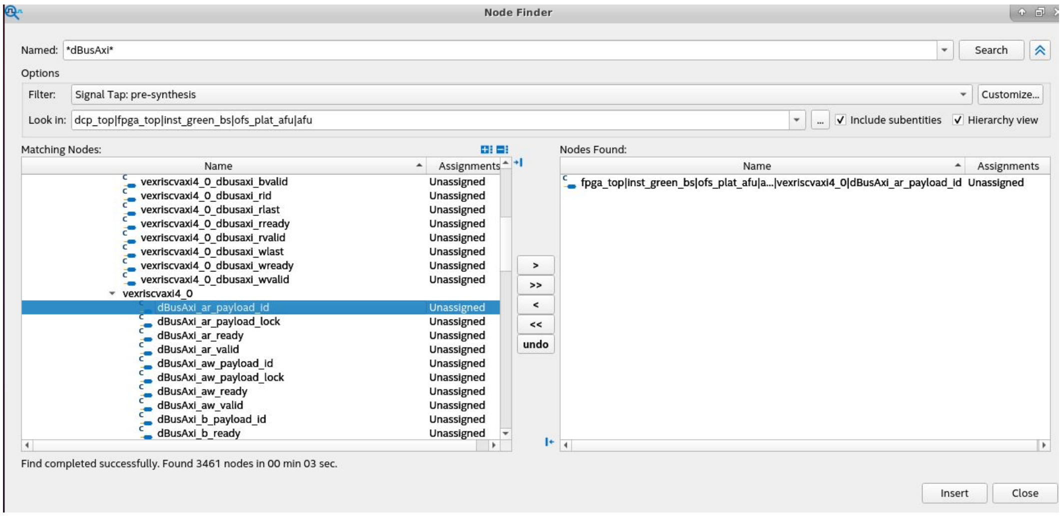

Figure 7: Signal Tap debug

- The QUARTUS_BIN environment variable points to the Intel Quartus installation.

Build and compile the design using the following command

- afu_synth_setup –source filelist.txt vexriscv_test.
  The implemented SoC system with VexRISC-V is build using the command as shown in Fig.8

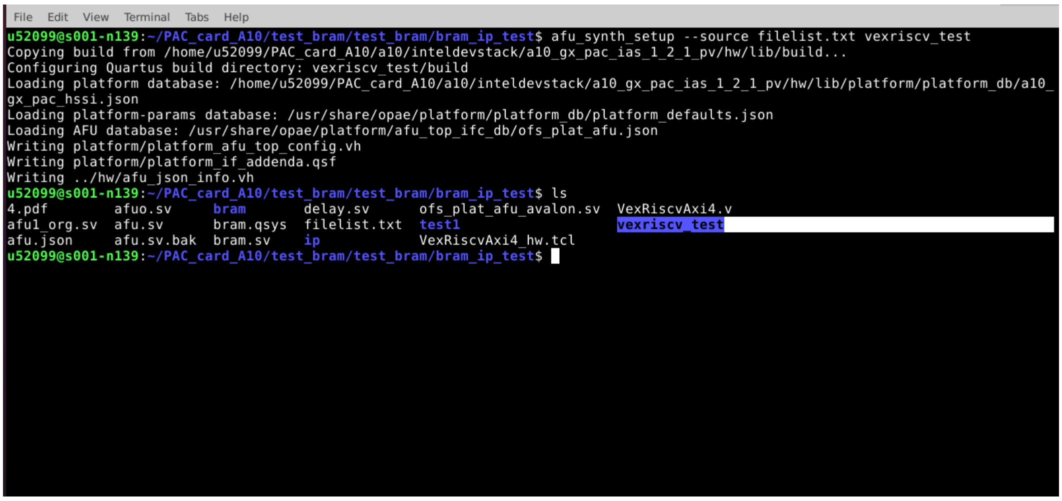

Figure 8: command to build the AFU design

- cd vexriscv_test

- run.sh
  The run.sh command performs the synthesis, design fitter such as place, map, and route, finally the timing analysis of design. Fig. 9 shows the synthesis and fitter of design on the FPGA platform.

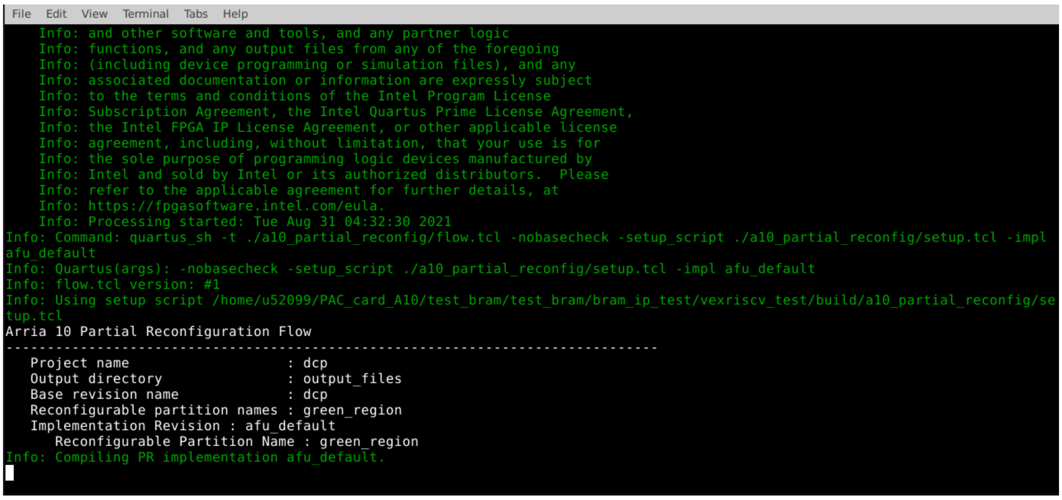

Figure 9: Command to run the AFU design

Authentication of .gbs file to program

The PACSign utility inserts authentication markers into .gbs file targeted for the Intel programmable acceleration cards (PACs) as shown in Fig.10. The PACs will not program .gbs file without proper authentication.

- PACSign SR -t UPDATE -H openssl_manager -i afu.gbs -o unsigned_afu.gbs
  No root key specified. Generate unsigned bitstream? Y = yes, N = no: y
  No CSK specified. Generate unsigned bitstream? Y = yes, N = no: y

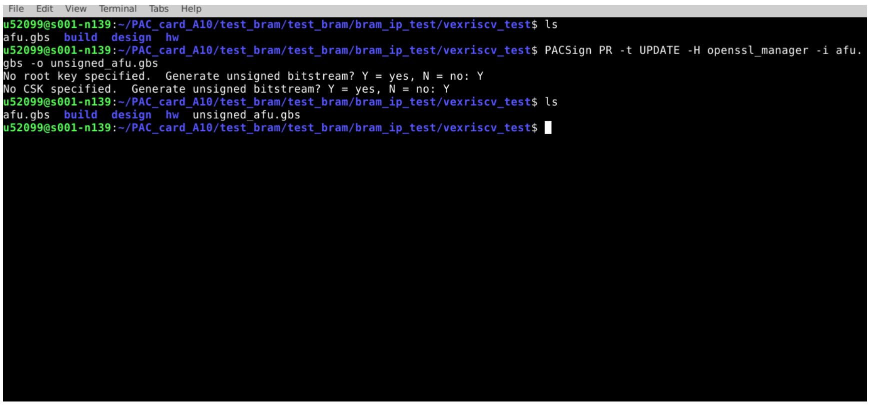

Figure 10: Authentication of AFU

Program Intel PAC card

fpgaconf command configures the FPGA with the accelerator function unit (AFU). Fig.11 shows the programming AFU on the Intel PAC card.

- fpgaconf -B 0x3b unsigned_afu.gbs

Figure 11: Program the AFU on Intel PAC card

3.4 Capturing Signals in SoC system with VexRISC-V(AFU) with signal tap remote debug

The AFU design is debug using a signal tap logic analyzer. The remote debug features of the Intel PAC card is used for debugging the AFU. Here, the Signal Tap GUI running locally on the server with the intel PAC card.

Set up connections for remote debug

- Set up a network connection between the instrumented FPGA AFU and Signal Tap GUI. We can set the network connection based on whether we are using a remote or local debugging configuration.

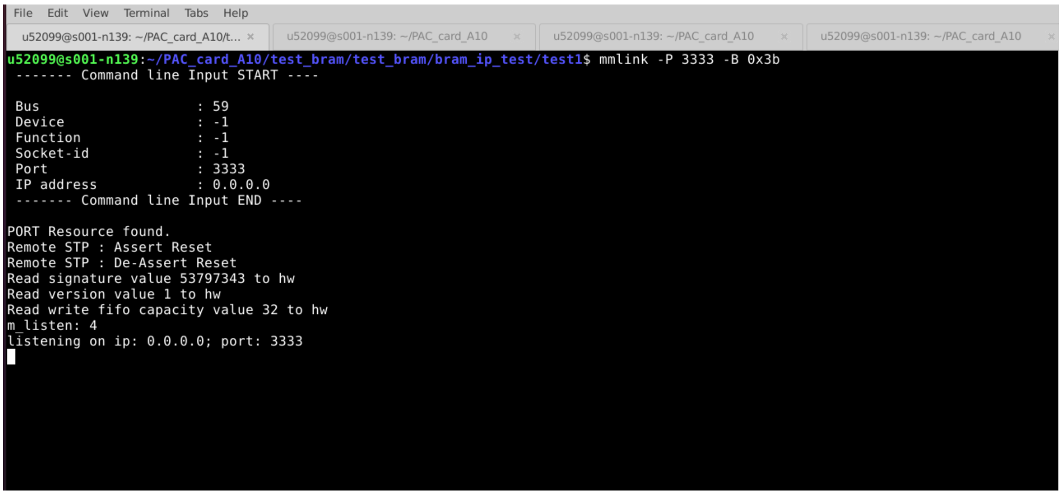

Figure 12: Command for mmlink setup

- Use the OPAE tool mmlink to enable the host system for remote Signal Tap as shown in Fig. 12.

- Open a TCP port to accept incoming connection requests from remote debug hosts.
  - Start System Console for remote debug as shown in Fig.13

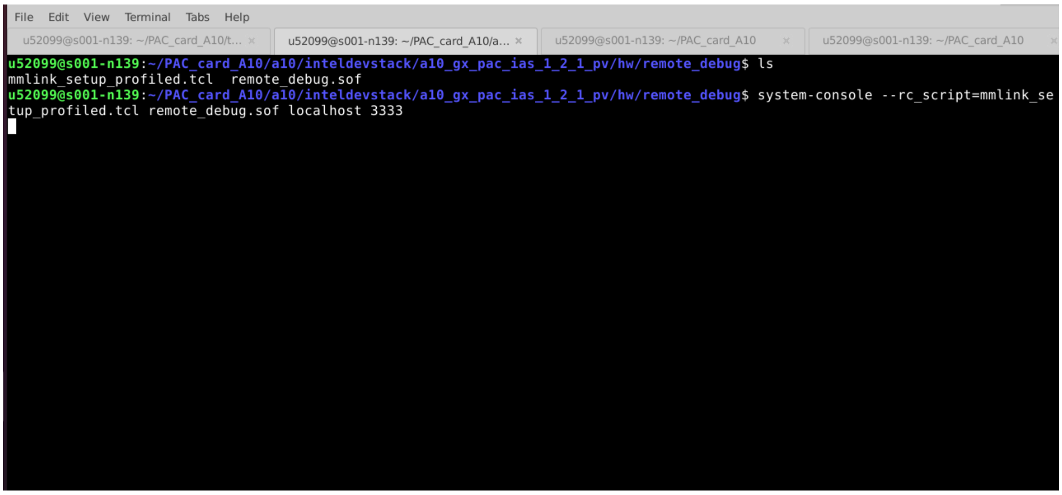

Figure 13: System console

  - Open Intel Quartus Prime GUI on the machine that performs the debug as shown in Fig. 14 and 15.

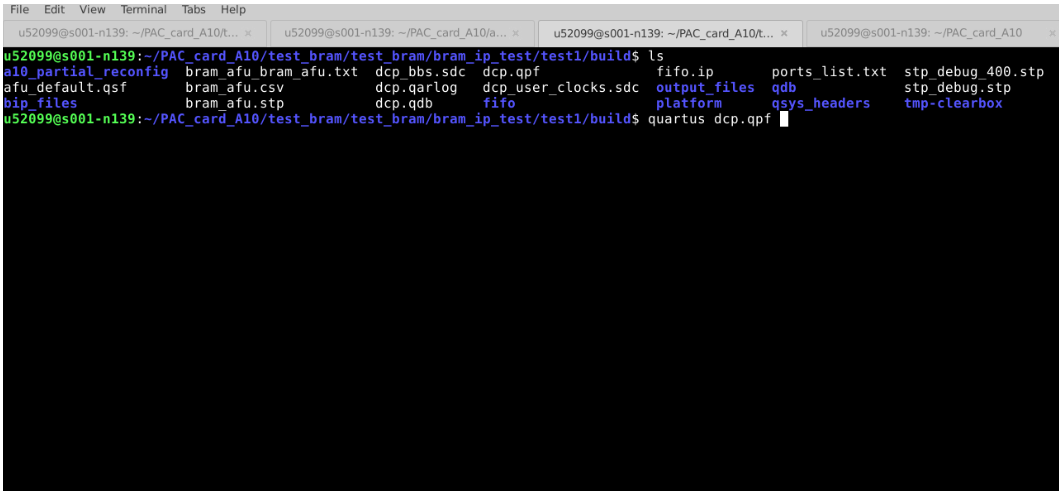

Figure 14: Open AFU design using Quartus software

  - Open Signal Tap GUI. Fig. 16 and 17 show the VexRISC-V core is debug using signal tap. Here, we monitor the VexRISC-V 32-bit register content. When the assembly code is loaded into BRAM, then VexRISC-V starts execution by reading the instruction fromBRAM address space. Once execution is finished, the respective register content is updated.

3.5 Test the VexRISC-V AFU using software code

To test the working of VexRISC-V AFU on Intel FPGA PAC card, the simple load and store assembly program is written as shown below.

addi x2 , x0 , 5
sw x2 , 48(x0)
lw x3 , 48(x0)

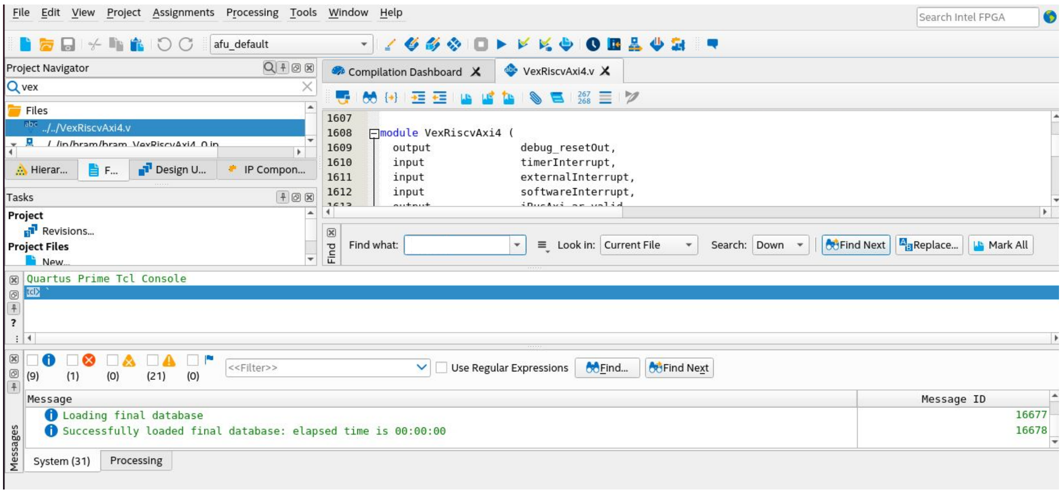

Figure 15: Quartus software

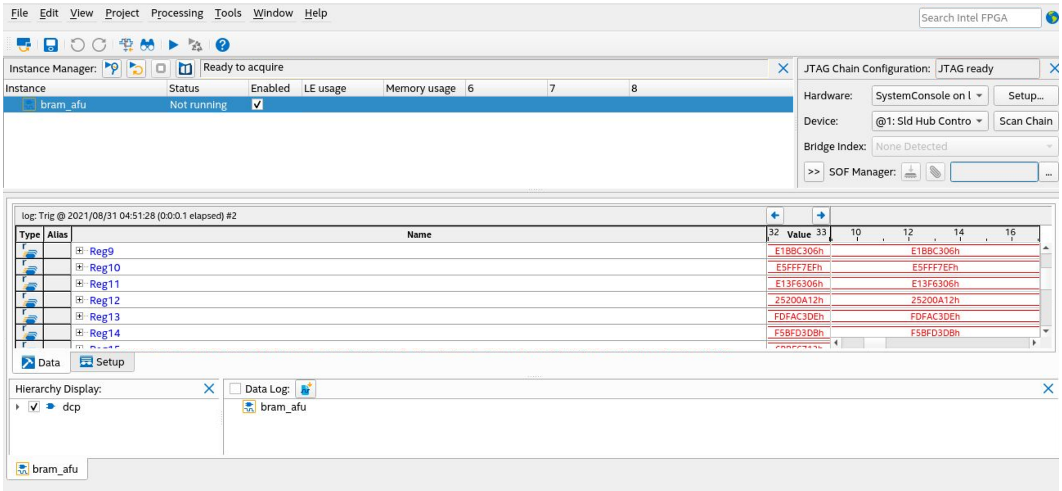

Figure 16: Signal Tap GUI

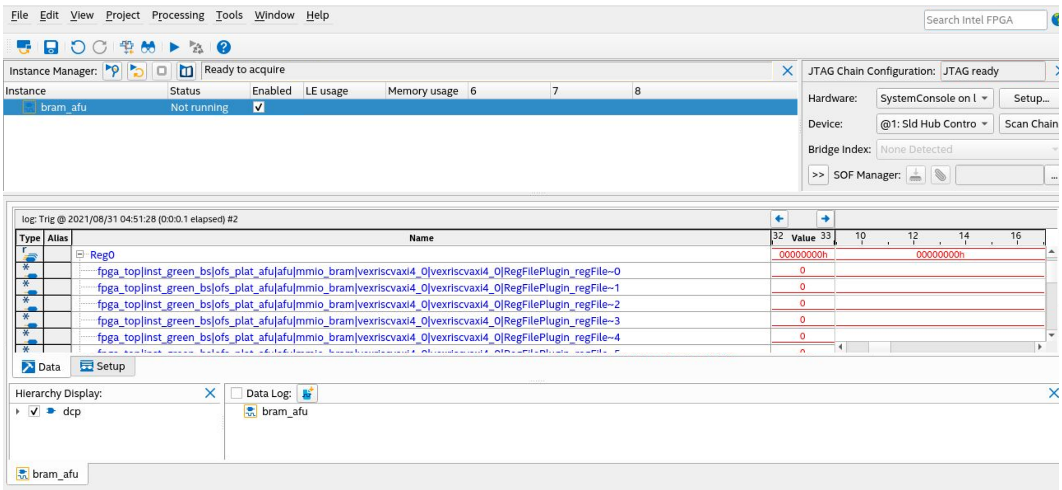

Figure 17: Debug VexRISC-V using signal Tap

Here, addi instruction adds the immediate value 5 and content of register x0 (x0 value is zero), then the result is stored in the x2 register. The instruction SW, store the content of the x2 register into the memory location 48(x0). Last instruction lw, read the content from memory location 48(x0) and load value into x3 register.

The assembly code is compiled and run using the RISC-V toolchain. The .elf file is generated after compilation and run. To know the machine instruction of assembly code by disassembling .elf file as shown in Table .1.

Table 1: Machine code

PC	Machine Code	Basic Code	Original Code
0x20	0x00500113	addi x2 x0 5	addi x2 x0 5
0x24	0x02202823	sw x2 48(x0)	sw x2 48(x0)
0x28	0x03002183	lw x3 , 48(x0)	lw x3 , 48(x0)

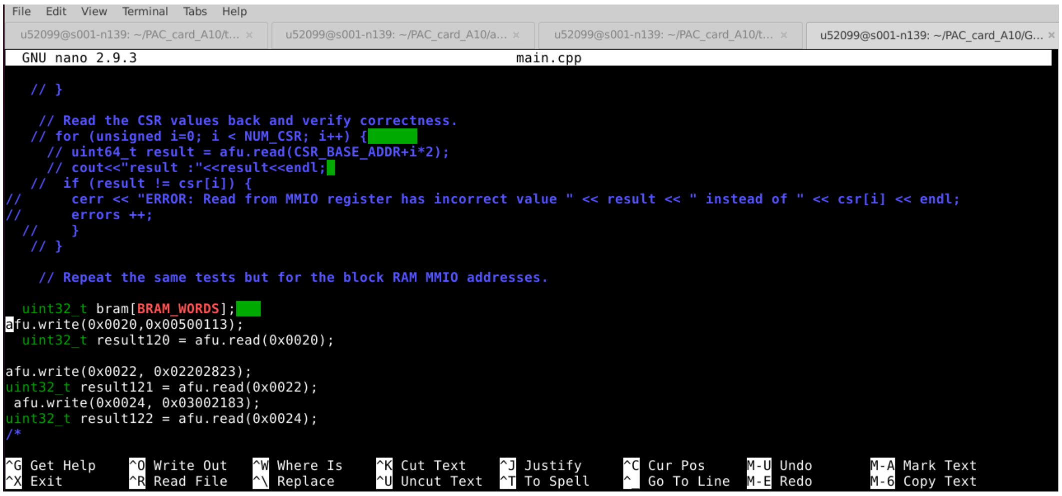

Figure 18: Load the Machine code using OPAE APIs

The machine codes are load into the FPGA block RAM space using the OPAE APIs such as fpgaWriteMMIO64, fpgaReadMMIO64 as shown in Fig. 18 and 19. These APIs feature functions for mapping and accessing control registers through memory-mapped IO. Most FPGA accelerators provide access to control registers through memory-mappable address spaces, commonly referred to as "MMIO spaces".

Write a 64-bit value to MMIO space using the below function.
fpga_result fpgaWriteMMIO64(fpga_handle handle, uint32_t mmio_num, uint64_t offset, uint64_t value)
Read 64-bit value from MMIO space using the below function.
fpga_result fpgaWriteMMIO32(fpga_handle handle, uint32_t mmio_num, uint64_t offset, uint32_t value)

Fig.20 shows the compilation and execute the OPAE software program. The machine code is loaded into the Block RAM, then VexRISC-V starts execution by reading the instruction from Block RAM. Fig. 21 shows the register content of VexRSIC-V is updated after completing the task. Here,x2 and X3 are the Reg2 and Reg3 in signal tap, we can see the Reg2 and Reg3 are updated with value 5.

We analyze the Block RAM content. Fig.22 shows the Block RAM content is updated with value 5 in the memory address location 48.

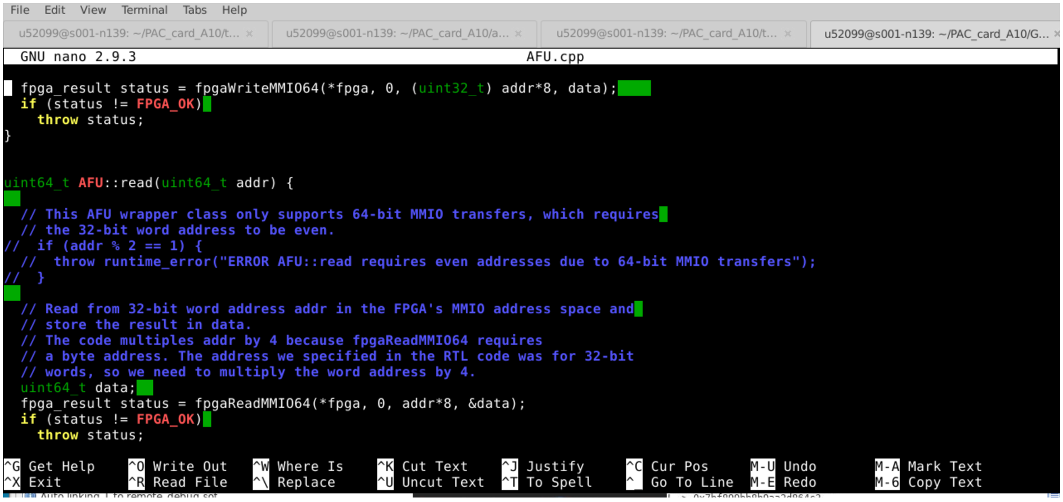

Figure 19: Functions for MMIO space read and write

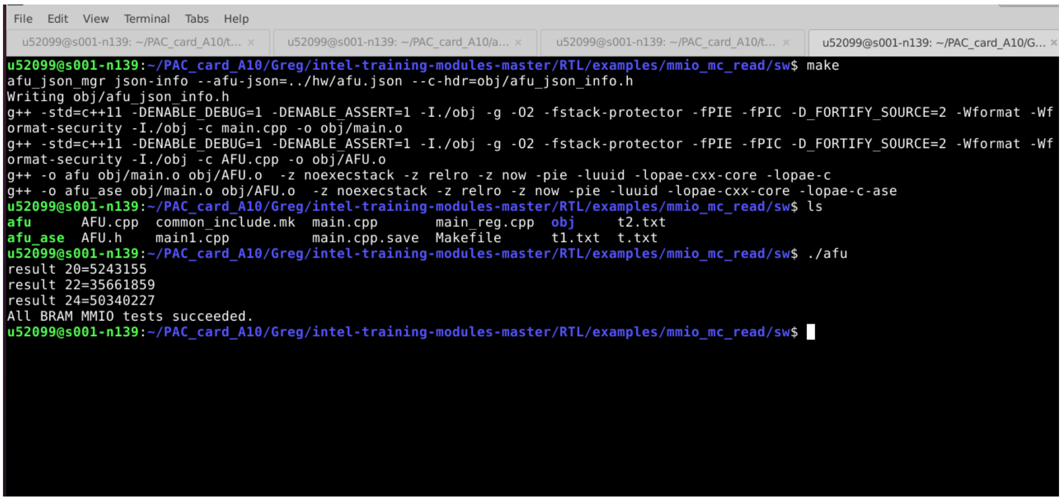

Figure 20: Compile and run the OPAE code

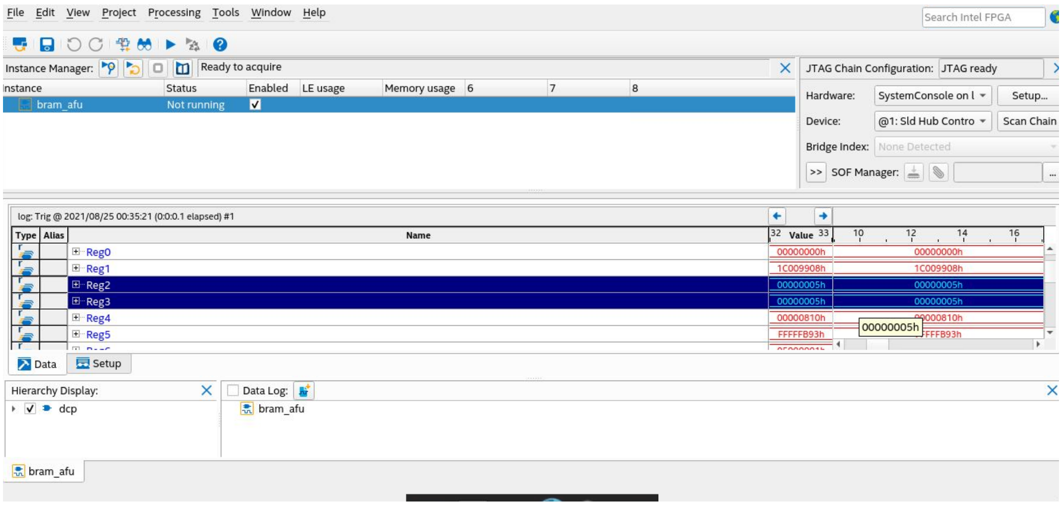

Figure 21: Analyze the register content of VexRISC-V using signa tap

Figure 22: Block RAM memory analysis

4 Source codes generated for the project(Github link) with explanation

https://github.com/khyamling

This repository contains the implementation of VexRISC-V AFU on the Intel FPGA PAC card. We optimize the VexRisc-V processor on Intel FPGA PAC card using optimization strategies and HyperFlex optimizations techniques.
The VexRisc-V has the following features.

- Support the 32-bit RISC-V ISA(RV32IM).
- 5-stage pipeline
- Optimized for Intel FPGA.
- The Instruction cache, Data cache.
- Supports single cycle barrel shifter, debug module, catch exceptions, dynamic branch, memory management unit(MMU).

The AFU with VexRISC-V design, the toplevel and other modules source code is found in ofs_plat_afu_avalon.sv file, the afu.qsf is the Quartus setting file. The synthesis, place, route, and static timing analysis report found in the output_files/ directory. The synthesis result was obtained by synthesizing the VexRisc-V on Intel Quartus Pro 2020.3 software tool with the fastest speed grade and Hyperflex Optimization techniques to get the maximum