NOVA EMULATION PROJECT

Open source RISC V-based Application Class AI SoC

Github: https://github.com/The-Nova-Project
Website: https://the-nova-project.github.io

Team Members
(Hardware Team)
Syeda Rafia Fizza Naveed, Abdul Muheet Ghani , Nameer Iqbal Ansari, Khadija
(Software Team)
Shahzaib Kashif, Muhammad Shahzaib
Team Lead
Sajjad Ahmed, Zeeshan Rafique, Zain Rizwan Khan
Industrial Mentors
Raheel Khan, Hamza Khan, Farhat Abbas, Zainab Khan, Asma Khan
Academic Mentor
Dr. Ali Ahmed

•

TABLE OF CONTENTS

Overview

This Project is to provide undergrad students hands-on experience of creating application class embedded SoC using open source, commercial and custom designed IPs and Booting RTOS (Zephyr) or Linux Kernel on Hydra-SU. Further integration and testing of MATMUL for embedded matrix multiplications which can provide great value in image processing executions.

Features

- AWS Cloud FPGA related logic
  - ➢ AXI-Lite 32-bit master for configuration and control
  - ➢ AXI4 slave for DDR access
  - ➢ AXI4 master 512-bit DMA configuration
- Processor Subsystem (hydra subsystem)
  - ➢ Ariane core.
  - ➢ JTAG, CLINT, debug
  - ➢ Matrix Multiplier (MatMul) Subsystem (In future work extension)
- System Peripherals
  - ➢ BRAM Compute Memory (8MB)
  - ➢ PLIC, AXI-UART Lite
- Runtime Tools(SDK)
  - ➢ Runtime driver for DDR,BRAM and UART
  - ➢ APIs used by C/C++ host applications.
  - ➢ AFI management APIs for loading FPGA image

SoC Design

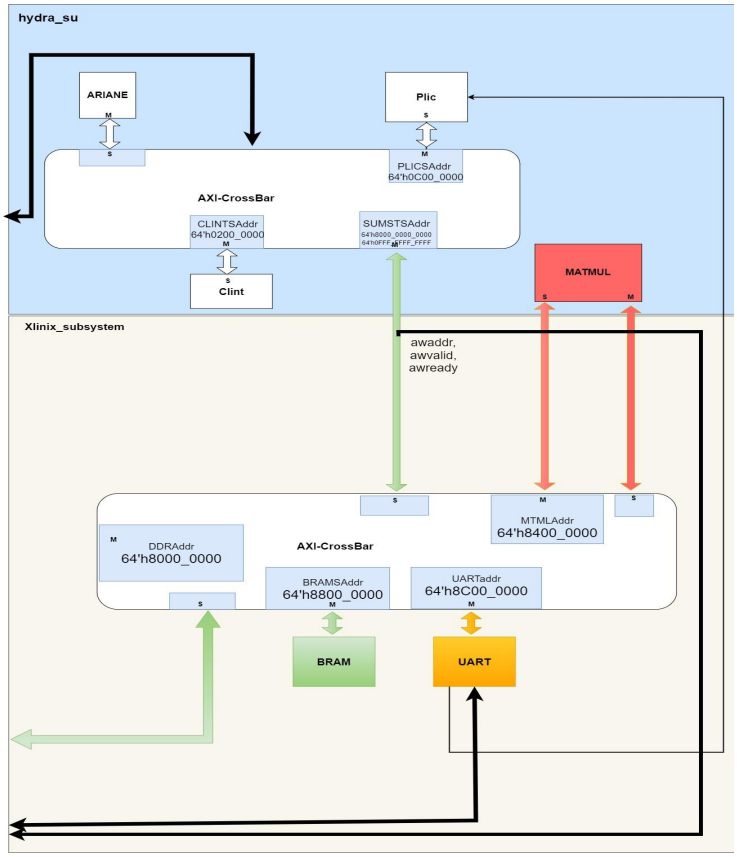

Figure 1: NOVA SoC

The SoC consists of 2 subsystems.

Hydra Subsystem

This subsystem contains Hydra SoC consists of a core Ariane (which is a linux capable, 6-stage, single issue, in-order CPU and it fully implements IMAC extensions) with Jtag interface, PLIC and a CLINT. All the components are connected through a crossbar which has a 64 bit address for each interface. With all these interfaces of the crossbar another master and slave interface is present to provide integration of the hydra Subsystem to Xlinx subsystem.

Xilinx Subsystem

This subsystem contains NOVA's peripherals except DDR and performs communication with HYDRA subsystem. It contains BRAM, UART Lite (which comes with 32 bit data range, AXI lite interface and operates at 9600 baud rate), AXI protocol converters, DATA width converter, Block memory controller and AXI crossbar and they all are Xilinx IP.

AXI crossbar have:

- 3 master interfaces for connecting DDR, Uart Lite (after converting that interface from AXI to AXI-Lite and 64 bit data to 32 bit data width through AXI protocol and data width converter simultaneously) , and BRAM controller which further connects with BRAM.
- 2 slave interfaces for BAR1 (which comes with 32 bit address range), and HYDRA.

An additional interface of crossbar is also tested for MATMUL and can directly be integrated with it in future for performing RIF programming of matrix multiplications through HYDRA.

Components

Core (Ariane)

Ariane is a 64 bit, single-issue, in-order RISC-V core. It has support for hardware multiply/divide, atomic memory operations as well as an IEEE compliant Floating Point Unit (FPU). Furthermore, it supports the compressed instruction set extension as well as the full privileged instruction set extension. It implements a 39 bit, page-based virtual memory scheme (SV39).

To achieve desired performance goals a synthesis-driven design approach leads to a 6-stage pipelined design. To reduce the penalty of branches, the microarchitecture features a branch predictor.

AXI UART Lite

Characters received via the AXI4-Lite interface can be converted from parallel to serial using AXI UART Lite, while characters received from a serial peripheral can be converted from serial to parallel. It sends and receives characters that are 8, 7, 6, or 5 bits long, have one stop bit, and either an odd, even, or no parity bit. The AXI UART Lite has separate transmit and receive capabilities.

When either the transmit FIFO or the receive FIFO stops being empty, UART Lite generates a rising-edge sensitive interrupt. An interrupt enable/disable signal can be used to mask this interrupt. A baud rate generator and separate 16-character deep transmit and receive FIFOs are included in the gadget.

AXI Crossbar

The AXI Interconnect core enables any combination of AXI master and slave devices, each of which can have a different data width, clock domain, and AXI sub-protocol, to be linked to it (AXI4, AXI3, or AXI4-Lite). The required infrastructure cores are automatically inferred and linked within the interconnect to carry out the necessary conversions when the interface characteristics of any connected master or slave device differ from those of the crossbar switch inside the interconnect.

AXI Bram Controller

For integration with the AXI connection and system master devices to interface with local block RAM, it is intended as an AXI Endpoint slave IP. The core is performance-optimized and allows both single and burst transactions to the block RAM. It supports INCR burst sizes up to 256 data transfers, WRAP bursts of 2, 4, 8, and 16 data beats, and AXI narrow and unaligned write burst transfers. It is a low latency memory controller.

Debug module

Debug modules give users access to a core's run-control debugging operations. It is connected at 0x0 offset of the test subsystem's crossbar and extends with OCL interface and on the core side it is connected to jtag interface.

PLIC/CLINT

Platform level interrupt controller and core level interrupt controller have interrupt management jobs from outside and inside of the core simultaneously.
The memory-mapped control and status registers associated with the timer and software interrupts are managed by CLINT. PLIC will manage the external interrupts from the UART, debug interface.
These interrupts are managed by setting some priority scheme which will be done in future.

AWS Cloud FPGA Interfaces

In this project Amazon FPGA is divided into two partition:

- **Shell (SH)** – AWS platform logic implementing the FPGA external peripherals, PCIe, DRAM, and Interrupts.
- **Custom Logic (CL)** – Custom acceleration logic created by an FPGA Developer.

After completing the development process combining shell and CL and creating an amazon FPGA image (AFI) that can be loaded onto the amazon EC2 FPGA instances.

The F1 FPGA platform is divided into two external interfaces the first is PCI and other one is DDR. There are further two PCIe physical functions(PF) which represent the PCI to EC2 Instance.

- ***Management PF:*** This PF is used for management of the FPGA and FPGA management library like fpga-get-virtual-dip-switch fpga-set-virtual-dip-switch for integration with C/C++ applications, as well as runtime tools and does not support any interface with the CL code.
- ***Application PF(AppPF)*** exposes:
    1. BAR0 as 32bit, BAR size as 32Mib. This BAR maps to the OCL AXI-Lite interface.
    2. BAR1 is a 32bit BAR sized as 2MiB. This BAR maps to the BAR1 AXI-Lite interface.

The following Interfaces provides the shell available to develop the CL.

- ***Clock and Reset*** provided by the shell to the CL. All interfaces between the CL and SH are synchronous to clk_main_a0. The maximum frequency on clk_main_a0 is 250MHz.
- ***DDR*** has four interfaces. Each of the four DDR4 Controllers has an AXI-4 interface with a 512 bit data bus. three DDR4 Controllers instantiated in the CL in sh_ddr as DDRA,B,D. However, Shell instantiated DDR-4 controller is DDRC.
- ***DMA_PCIS*** is a 512-bit wide AXI-4 interface.
- ***OCL*** Interface is an AXI-Lite interface associated with AppPF and BAR0.
- ***BAR1*** The BAR1 Interface is an AXI-Lite interface associated with AppPF and BAR1.

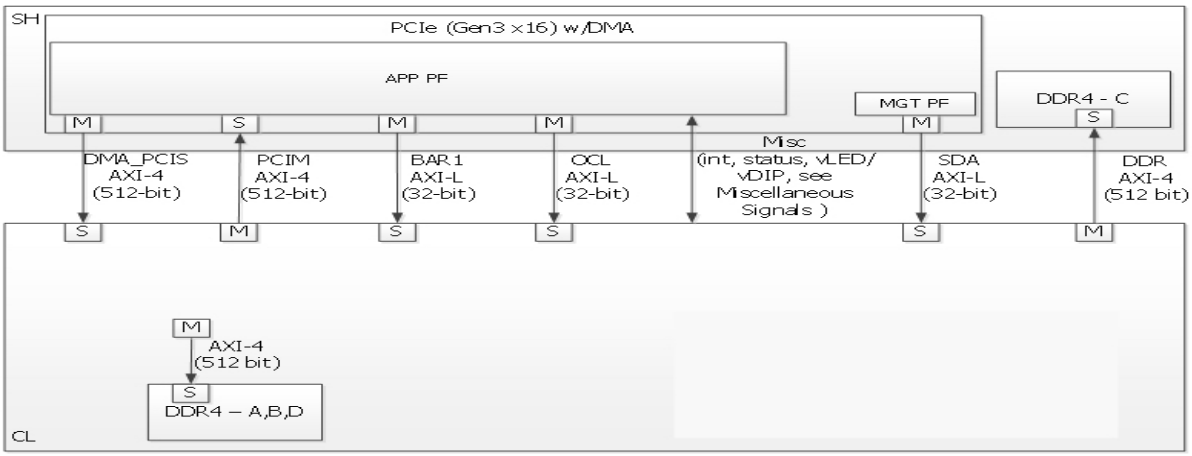

Figure 2: AWS-EC2 Shell

Development Kits

Amazon EC2 comes with 2 type of development kits namely:

- **Hardware development Kit (HDK)**
  The HDK directory contains documentation, examples, simulation, build and AFI creation scripts to start building Amazon FPGA Images (AFI).

- **Software development Kit (HDK)**
  The runtime environment needed to run on EC2 FPGA instances can be found in the SDK directory.It contains the drivers and management tools necessary to manage the AFIs loaded onto the FPGA instance.

Custom Logic

To port the SoC on AWS EC2 it must have to be compatible with AWS-Shell interface which required some additional logic to provide accessing of SoC on AWS-FPGA through PCIe slots. This additional logic is test subsystem and the nova subsystem (all the subsystems combined) is the custom logic for the AWS environment.

Test Subsystem

This is the necessary subsystem for making our actual SoC accessible with the Shell. So, it contains AXI crossbars, data and AXI protocol converters, debug bridge for connecting HYDRA subsystems debug module to shell jtag interface, and UART lite to test the Xilinx subsystem's UART (as xilinx subsystem's UART directly cant be tested on any aws-fpga interface, So this UART receives data from designs UART which can be observed by fetching it from OCL).
DMA interface is connected to DDR using AXI crossbar and BAR1 interface is to xilinx subsystem's crossbar after converting into AXI and 64 bit data bits.

For visible understanding of the whole system, please refer to the mentioned diagram

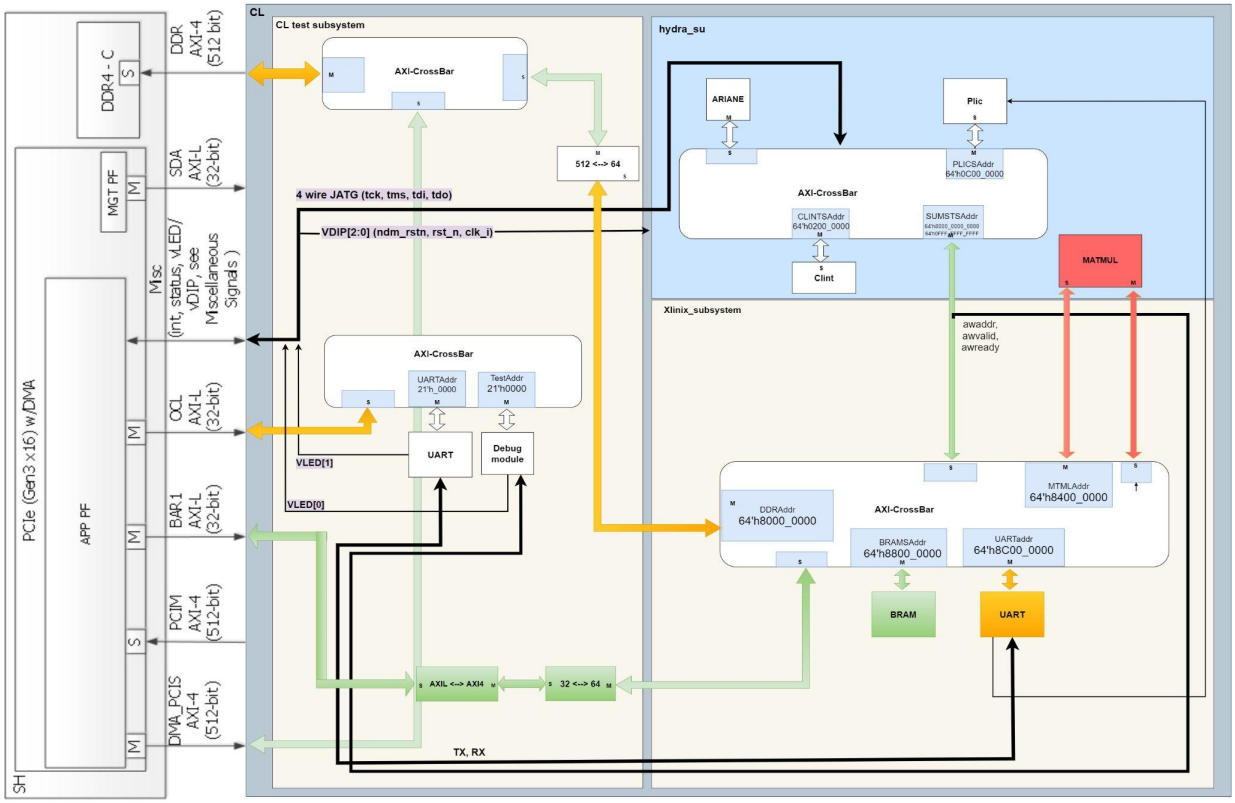

Figure 3: NOVA Subsystem

Data Path Flow

For booting, Hydra fetches the multiple instructions from ddr, 512 bit data from DDR have the current instruction at its 64 bit LSB, So after passing through the AXI crossbar data bit converter forwards those 64 bits to Xilinx subsystem's crossbar which further transfers this instruction to HYDRA.
Hydra can perform operations and use UART and BRAM through Xilinx subsystem's crossbar. Both can also be acceptable through the BAR1 interface. Both DDR and bram are byte addressable memory.

Simulation

Tools for Simulation

AMAZON EC2 supports questa, vcs, ies and vivado environmental tools for simulation. We are primarily using vivado.
The following versions of vivado for simulation and synthesis of the hardware.

- Vivado v2019.1.op (64-bit)
- Vivado v2019.1 (64-bit)
- Vivado v2019.1_AR73068 (64-bit)
- Vivado v2019.1_AR73068_op (64-bit)
- Vivado v2019.1_AR72668 (64-bit)
- Vivado v2019.2 (64-bit)
- Vivado v2019.2_AR73068_op (64-bit)
- Vivado v2019.2_AR73068 (64-bit)
- Vivado v2020.1 (64-bit)
- Vivado v2020.2 (64-bit)
- Vivado v2021.1 (64-bit)
- Vivado v2021.2 (64-bit)

Simulation Steps

Step 1 : Creating the hardware development environment.

$ source hdk_setup.sh

By sourcing the hdk this will setup environment variables, check Vivado, check HDK shell and after all hardware Setup will be passed.

Step 2: Creating the custom logic environment.

Now after that a developer is ready to have custom logic space. For setting up that space in AWS EC2 directory he can adopt 2 approaches:

Approach 1: Setup a new CL directory from scratch

- **mkdir <Your_New_CL_Directory>**
- **cd <Your_New_CL_Directory>**
- **export CL_DIR=$(pwd)**
- **source $HDK_DIR/cl/developer_designs/prepare_new_cl.sh**

Approach 2: Copy one of the example directories and change it with your own custom logic.

- **cd $HDK_DIR/cl/developer_designs**
- **$ cp -r $HDK_DIR/cl/examples/<example> .**
- **$ export CL_DIR=$(pwd)**

After any of the above approach a developer will have an environment containing all the necessary scripts to build, simulate, synthesize, generate AFI and at last emulate your custom logic on aws-fpga whose directory structure will be like that:

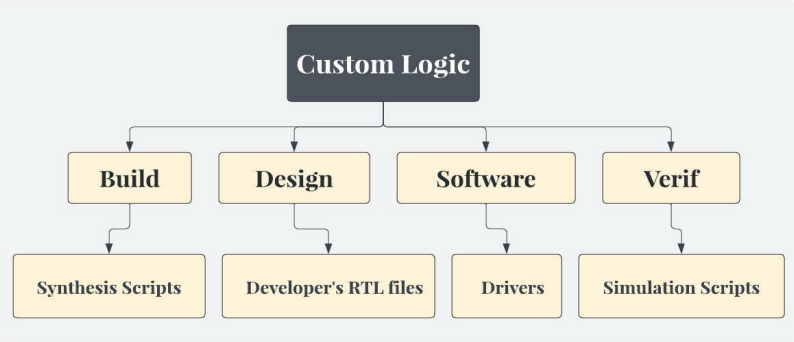

Figure 4: AWS-EC2 CL Directory Structure

Step 3:
Now that a developer have made his custom logic and he want to simulate it he would have to Add all the custom logic file i.e (.v , .sv) and .vhd files in top.vivado.f and top_vhdl.vivado.f file simultaneously.
These both files are present in the scripts of the custom logic and they already contain the shell's RTL files to make our CL compatible with AWS-fpga.
During the rtl compilation, All the files added in top.vivado.f and top_vhdl.vivado.f would be considered as RTL logic files.

Step 4:
At last the design is ready to be simulated which will be done by running the make scrip**t make TEST=test_name present in aws-fpga/hdk/cl/developer_design/custom_logic_design/verif/scripts.**
The test_name is the name of the module of a .sv testbench of the test a developer is running. The tests directory is **aws-fpga/hdk/cl/developer_design/custom_logic_design/verif/tests.**
This script will Compile GCC, Link, Analyze, Elaborate, Optimize, and create Simulation Snapshot to build a given test' simulation environment file.
Now you have a vivado simulation environment file in
aws-fpga/hdk/cl/developer_design/custom_logic_design/verif/sim/vivado/test_name.

xsim -gui tb command will open the simulation file and you can see the simulation of your Custom logic design + Shell.
Note: Custom_logic_design is nova_project in this project.

Important directory Paths

- **aws-fpga/hdk/cl/developer_design/nova_project/verif/tests.** Have all the assembly tests and hex files.
- **aws-fpga/hdk/cl/developer_design/nova_project/verif/sim/test_name/xsim.log** contains the simulation log processes and errors info.
- **aws-fpga/hdk/cl/developer_design/nova_project/verif/scripts**/ contains a Makefile which runs and proceeds the simulation steps. Makefile.vivado, Makefile.questa, Makefile.vcs, Makefile.ies are the optional environments which can be used for simulations. This folder also contains the config file for performing certain configurations on simulation and also other scripts and files some of which are discussed above.
- **aws-fpga/hdk/common/shell_v04261818/design/interfaces/cl_ports.vh** contains all the SHELL's ports logics. So, developers need to integrate the required interfaces of the shell according to this file and must call this file into the top level of the custom logic design file.
  The other SHELL's interfaces which are not used by the developer must be stubbed out from the cl_port.vh file but not manually instead their are some **unused_interface_name.inc** files in the same directory which can be instantiate in custom logic's top level file just like cl_port.vh is instantiated.

AFI Implementation Design Flow

The flow performs full AFI implementation steps from RTL to AFI, which includes: logic synthesis converts high level constructs and behavioral code into logic gates; technology mapping separates the gates into groupings that best match the FPGA available logic; the placement step assigns the groupings to specific logic blocks, and routing determines the interconnect resources that will carry the logic signals; finally, bitstream generation creates a design checkpoints file (.dcp file) that sets all the points to configure the logic block and routing resources appropriately.

1. Synthesis

Synthesis is a process of converting RTL (synthesizable Verilog code) to technology specific gate level netlist (includes nets, sequential and combinational cells and their connectivity).

Logic Synthesis = Translation + Mapping + Optimization

Translation

This is the first step of the logical synthesis. In this step, RTL is converted to a general library netlist. During this step, more emphasis is on whether the intended logic is maintained or it gets changed.

Operations performed in this step:

- Conversion of HDL into functional Boolean equivalent.
- HDL syntax checking
- Optimizes HDL
- Arithmetic, Sequential and Combinational function mapping

Mapping

In this step, the tool will map the generic Boolean netlist into the gates available in the standard cell library. Boolean functions are mapped to technology specific primitives.

Optimization

In this step, the tool modifies the mapping to meet the design goals in the following priority order by default.

Priority: Design rules

Optimization step is constraint-driven. Developers need to provide good constraints for effective optimization results. Optimal design is found as a result of synthesis based on the priorities set by the developer. The tool chooses the combination of library cells that best meet the functional, timing, area and power requirements of the design.

2. FloorPlanning

The floorplanning step consists of the placement of the macros (memory, IPs, etc.) in the desired places to provide the best circuit performance, whether in timing, power, or area. Floorplanning includes macro/block placement, design partitioning, pin placement, power planning, and power grid design. The first rule of thumb for floorplanning is to arrange the hard macros and memories in such a manner that you end up with a core area square in shape.

3. Placement

Placement is the process of placing standard cells in the design. Placement does not just place the standard cells available in the synthesized netlist. It also optimizes the design. Placement also determines the routability of your design.Placement will be driven by different criteria like timing driven, congestion driven, power optimization.

Objective of Placement:

Objective of placement is to optimize the area, timing, power and minimal timing DRCs and minimal cell and pin density.

4. CTS (Clock Tree Synthesis)

Clock Tree Synthesis (CTS) is a process which makes sure that the clock signals are distributed uniformly to all sequential elements in a design.

5. Routing

Routing is the stage after CTS ,it is a process that determines the precise paths for interconnects.After CTS, we have information of all the placed cells, blockages, clock tree buffers/inverters and I/O pins. This information is important for a tool to complete all the connections defined in netlist.

Objective of Routing:

Objective of routing to meet the timing constraints, no DRC errors and minimize the total wire length.

6. Bitstream Generation (.dcp file)

Lastly, the results of mapping, placements, and routing are encoded into a bitstream (.dcp file) that can be used to configure the logic and wires to implement the target design.

7. AFI (Amazon FPGA Image)

After the creation of the .dcp file the last step is to create an AFI. It is basically the compiled FPGA code that is loaded into an FPGA in AWS for performing the Custom Logic (CL) function created by the developer. AFIs are maintained by AWS accordingly and associated with the AWS account that created them. The AFI includes the CL and AWS FPGA Shell. An AFI ID is used to reference a particular AFI from an F1 instance.

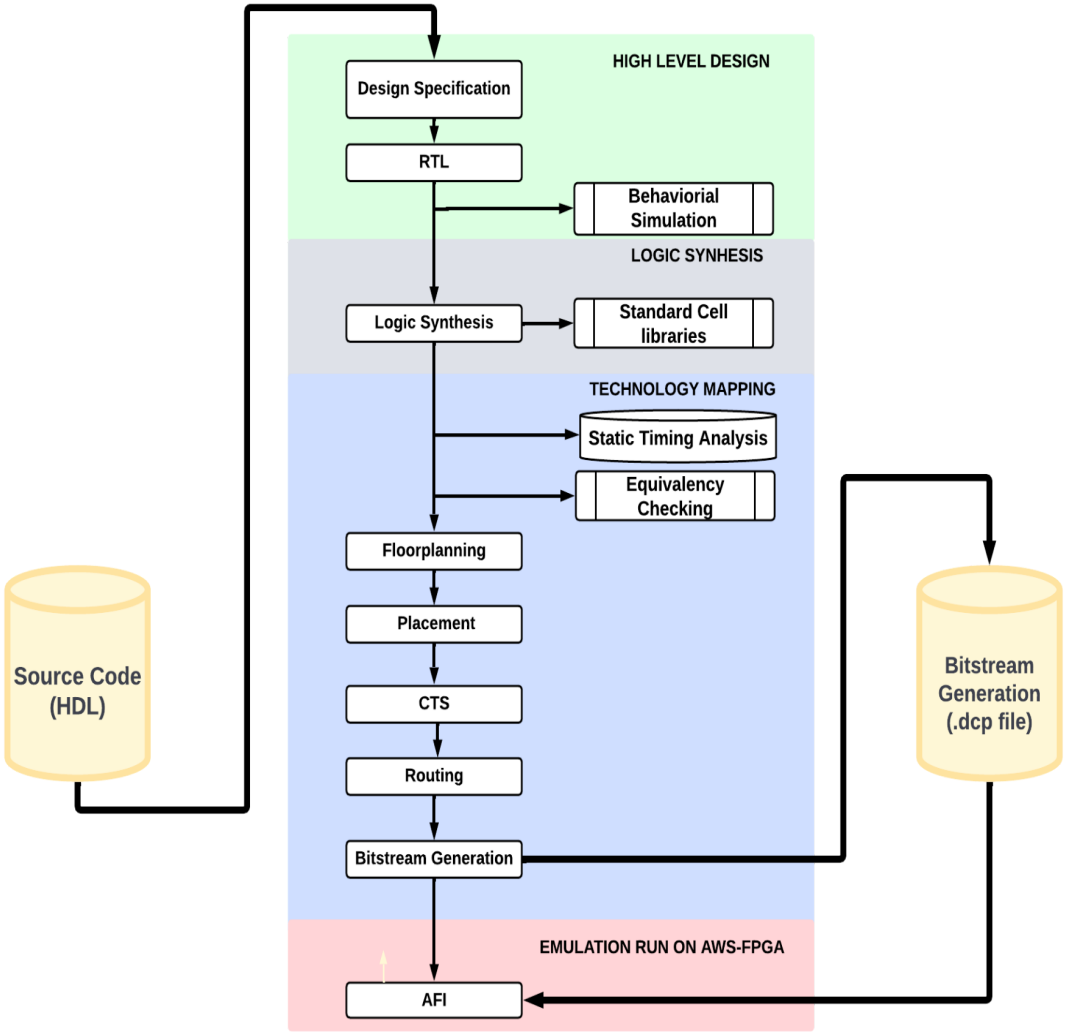

Figure 5: AFI Implementation Design Flow

Prerequisites for AFI

Once the developer has a functional design, the next steps are to: synthesize the design into basic FPGA cells, perform place-and-route, and check that the design meets the timing/frequency constraints. This could be an iterative process.
The DCP includes the complete developer design that meets timing/frequency constraints, placement boundaries within the allocated CL area on the FPGA, and the functional requirements.
To assist in this process, AWS provides a reference DCP that includes the shell (SH) logic with a black-boxed CL under:

$CL_DIR/build/checkpoints/from_aws/SH_CL_BB_routed.dcp

AWS also provides an out-of-the-box generic script called **aws_build_dcp_from_cl.sh** that is used to test and compile a few examples, such as the **cl_nova design**, as if they were developer code. These reference examples can serve as starting points for new designs. The output of the AWS-provided scripts will create a tar file, with both the encrypted placed-and-routed DCP and the corresponding manifest.txt, which AWS will use to generate the final bitstream.

Build Procedure

The following describes the step-by-step procedure to build developer CLs. Some of these steps can be modified or adjusted based on developer experience and design needs.
A developer can execute **$CL_DIR/build/scripts/aws_build_dcp_from_cl.sh** to check the environment, setup the build directory, invoke Xilinx Vivado to create the encrypted placed-and-routed DCP (which include AWS Shell + Developer CL).

Step 1: Install the HDK setup

The AWS FPGA HDK can be cloned to your instance by executing:

$ git clone https://github.com/aws/aws-fpga.git # Fetch the HDK and SDK code
$ cd aws-fpga # Move to the root directory of the repository before running the next script

Step 2: Environment Variables Setup

The environment variable HDK_SHELL_DIR should have been set. This is usually done by executing source hdk_setup.sh from the HDK root directory
Sourcing **hdk_setup.sh** does the following:

- It sets required environment variables that are used throughout the examples in the HDK.
- Downloads DDR simulation models and DCP(s) from S3.

$ source hdk_setup.sh # Build and install the HDK

The CL_DIR environment variable should point to the vivado design folder of the developer design repository

$ export CL_DIR=/home/$USER/aws-fpga/hdk/cl/developer_designs/your_project_name

➔ (e.g., export CL_DIR=/home/$USER/aws-fpga/hdk/cl/developer_designs/nova_project)

Step 3: Encrypt Source Files

CL Encryption is required and AFI creation will fail if your CL source files are not encrypted. As a precursor to the build process, modify the $CL_DIR/build/scripts/encrypt.tcl script to include all the CL source files, so the script can encrypt and copy them to the $CL_DIR/build/src_post_encryption directory.

Step 4: Modify the CL Build

Modify the $CL_DIR/build/scripts/create_dcp_from_cl.tcl script to include:

1. The list of CL encrypted files in $CL_DIR/build/src_post_encryption.
2. The list of CL specific timing and placement constraints in $CL_DIR/build/constraints.
3. The specific constraints and design file for IP any included in your CL (e.g., DDR4).

The following scripts should be modified before starting the build:

- **/build/constraints/***
  - This is to set all the timing, clock and placement constraints.
- **/build/scripts/encrypt.tcl**
  - To enable encryption, include all the header file names of the your designs (e.g. files name having extension .vh)
  - Also add the variables here which you export in the .bashrc file like,
    set CL_DIR $::env(CL_DIR)
- **/build/scripts/synth_cl_nova.tcl**
  - Developers can only add all necessary files in the synth_cl_nova.tcl which is important for synthesis like .sv, .v ,.bd files of your design.
- **/build/scripts/create_dcp_from_cl.tcl**
  - This is to update the final build scripts with right source files and IP.
  - Developer can only change the module name of this file with the top level module name of your design (e.g. cl_nova)

Step 5: Build

Run the build script, aws_build_dcp_from_cl.sh, from the $CL_DIR/build/scripts directory.
$ cd build/scripts

Now the build the script by using this command,
$ ./aws_build_dcp_from_cl.sh -clock_recipe_a A2 -strategy CONGESTION -foreground

The outputs of the build script are:

- **$CL_DIR/build/checkpoints/*:**
  Various checkpoints generated during the build process.
- **$CL_DIR/build/to_aws/SH_CL_routed.dcp:**
  Encrypted placed-and-routed design checkpoint for AWS ingestion.
- **$CL_DIR/build/reports/*:**
  Various build reports (generally, check_timing/report_timing).
- **$CL_DIR/build/src_post_encryption/*:**
  Encrypted developer source.
- **$CL_DIR/build/constraints/*:**
  Implementation constraints.

Aws-Fpga AFI Generation Flow

Steps Of AFI Creation

AFI (Amazon FPGA Image) is a bitstream file used to program FPGA.
Following steps are needed to be run to generate AFI.

Step 1: Sourcing HDK & SDK files

Firstly, Move to the root directory of the repository (aws-fpga) before running the next script

$ cd aws-fpga
$ source hdk_setup.sh
$ source sdk_setp.sh

Step 2: Environment Setup of your CL Directory

The CL_DIR environment variable should point to the vivado design folder of the developer design repository

$ export CL_DIR=/home/$USER/aws-fpga/hdk/cl/developer_designs/your_project_name

Step 3: Setup CLI and Create S3 Bucket

The developer is required to create an S3 bucket for the AFI generation. The bucket will contain a tar file and logs which are generated from the AFI creation service.
First step is to set your AWS-FPGA credentials follow these steps to set the credentials,

$ aws configure # to set your credentials (found in your console.aws.amazon.com page), region (us-east-1) and output (json)

- *AWS Access Key ID [**********2UWA]: **********
- *AWS Secret Access Key [**********YIV8]:**********
- *Default region name [us-east-1]: *********
- *Default output format [json]: ****

Step 4: Create S3 bucket (folder) on aws server

This S3 bucket will be used by the AWS scripts to upload your DCP to AWS for AFI generation which will be packaged into a tar file.

Start by creating a bucket:

$ aws s3 mb s3://<bucket-name> --region <region-name> # Create an S3 bucket. Choose a unique bucket name

➔ (e.g., aws s3 mb s3://my_awsfpga --region us-east-1)

$ aws s3 mb s3://<bucket-name>/<dcp-folder-name> # Create a folder for your tarball files

➔ (e.g., aws s3 mb s3://my_awsfpga/dcp)

Step 5: Copy DCP to S3 bucket

After create a bucket ,then copy it to S3

$ aws s3 cp $CL_DIR/build/checkpoints/to_aws/*.Developer_CL.tar \ # Upload the file to S3 ***s3://<bucket-name>/<dcp-folder-name>/***

NOTE: The trailing '/' is required after <dcp-folder-name>

Step 6: Create a folder for your log files

Now, Create a folder to save your logs by following the steps below

$ aws s3 mb s3://<bucket-name>/<logs-folder-name> # Create a folder to keep your logs
$ touch LOGS_FILES_GO_HERE.txt # Create a temp file
$ aws s3 cp LOGS_FILES_GO_HERE.txt s3://<bucket-name>/<logs-folder-name>/

Step 7: Check Policy

Check your policy (do you have correct rights to create AFI) [optional]

$ check_s3_bucket_policy.py --dcp-bucket <bucket-name> --dcp-key <dcp-folder-name>/<tar-file-name> --logs-bucket <bucket-name> --logs-key <logs-folder-name>

Step 8: Start AFI Creation

Once your policy passes the checks, create the Amazon FPGA image (AFI).

$ aws ec2 create-fpga-image --name <afi-name> --description <afi-description> --input-storage-location Bucket=<dcp-bucket-name>,Key=<path-to-tarball> --logs-storage-location Bucket=<logs-bucket-name>,Key=<path-to-logs>

- ➔ (e.g. aws ec2 create-fpga-image --name first_afi --description "This is my first AFI" --input-storage-location Bucket=my_awsfpga,Key=dcp/tarbar_name.tar --logs-storage-location Bucket=my_awsfpga,Key=logs)

The output of this command includes two identifiers that refer to your AFI:

- **FPGA Image Identifier or AFI ID:** this is the main ID used to manage your AFI through the AWS EC2 CLI commands and AWS SDK APIs. This ID is regional, i.e., if an AFI is copied across multiple regions, it will have a different unique AFI ID in each region. An example AFI ID is afi-0f4ff82e29c6494f9.
- **Global FPGA Image Identifier or AGFI ID:** This is a global ID that is used to refer to an AFI from within an F1 instance. For example, to load or clear an AFI from an FPGA slot, you use the AGFI ID. Since the AGFI IDs is global (by design), it allows you to copy a combination of AFI/AMI to multiple regions, and they will work without requiring any extra setup. An example AGFI ID is agfi-01e33810ff4c9d23c.

Step 9: Provide AFI_ID

Then check if the AFI generation is done. You must provide the FPGA Image Identifier returned by create-fpga-image:

$ aws ec2 describe-fpga-images --fpga-image-ids <AFI ID>

- ➔ (e.g., aws ec2 describe-fpga-images --fpga-image-ids afi-0f4ff82e29c6494f9)

Step 10: Load the AFI

You can now use the FPGA Management tools, from within your F1 instance, to load your AFI onto an FPGA on a specific slot. Make sure you clear any AFI you have previously loaded in your slot:

$ sudo fpga-clear-local-image -S 0

If fpga-describe-local-image API call returns a status 'Busy', the FPGA is still performing the previous operation in the background. Please wait until the status is 'Cleared' as above.

Now, let us try loading your AFI to FPGA slot 0:

$ sudo fpga-load-local-image -S 0 -I agfi-01e33810ff4c9d23c

NOTE: The FPGA Management tools use the AGFI ID (not the AFI ID).

Runtime Tools (SDK)

This section includes the driver and runtime environment required by any EC2 FPGA instance. The drivers and tools are used to interact with the pre-built AFI's that are loaded to EC2 FPGA. The tools come pre-installed in /usr/bin for Amazon Linux, version 2016.09 or later. Alternatively, the tools can be downloaded and installed from AWS SDK/HDK GitHub repository aws-fpga, as follows:

$ cd aws-fpga
$ source sdk_setup.sh
$ source hdk_setup.sh

The sdk_setup.sh script will build the AFI Management Tools and install them in /usr/bin. The SDK is **NOT** used to build or register AFI, rather it is only used for managing and deploying pre-built AFIs. For building and registering AFIs. AFI management tools are used. Once you have the AFI Management Tools installed on your F1 instance, you can display the FPGA slot numbers. *sudo fpga-describe-local-image-slots -H*

To clear the AFI, use the following command will clear the FPGA image, including internal and external memories and expose the default AFI Vendor and Device Id, and display the final state for the given FPGA slot number. *sudo fpga-clear-local-image -S 0 -H*

To load the AFI, use the FPGA slot number and Amazon Global FPGA Image ID parameters; this command will wait for the AFI to transition to the "loaded" state. And expose the unique AFI Vendor and Device Id, and display the final state for the given FPGA slot number. *sudo fpga-load-local-image -S 0 -I agfi-01e33810ff4c9d23c*

Runtime driver for DDR, BRAM and UART

Developers using AMI 1.5.0 or Later Instances that come with pre-installed Xilinx Runtime Environment (XRT). Each CL Example comes with a runtime software under $CL_DIR/software/runtime/ subdirectory. You will need to build the runtime application that matches your loaded AFI.

APIs used by C/C++ host applications

The directory $CL_DIR/software/runtime/ contains compatible Runtime C Drivers that will drive the AWS-FPGA CL Designs to interact with the Cloud FPGA via AFI.

How To Run

1. First Copy the Driver you want to run. (Considering the AFI is loaded in slot 0)
2. Paste it in your design's software/runtime directory.
3. Open Terminal in this directory.
4. First type make driverName (for example if the driver is loader.c then type make loader)
5. Then once the driver is compiled successfully type sudo ./driverName (for example if driver if bramLoader.c then type sudo ./loader *args)
6. All the interactions that the driver has done with the FPGA will be shown on the terminal screen

Drivers

Driver	Command	Purpose
DMA_DDR_Loader	Sudo ./DMA_DDR_Loader	Reads a hex/elf and stores the data into DDR (Byte aligned).
BAR1_Loader	Sudo ./BAR1_Loader	Receive data through BRAM via BAR1 Interface.
uart_recieve	Sudo ./uart_recieve	Receive data through UART via OCL Interface.
uart_socket_thread	Sudo ./uart_socket_thread	MultiThreading of UART by Transmit and Receive data through OCL Interface.
client	Sudo ./client	Receiving transmitted character via socket.
uart_with_socket	Sudo ./uart_with_socket	Transmitting transmitted characters via socket.
uart_to_uart	Sudo ./uart_to_uart	Transmit data through UART via BAR1 interface and Receive data through UART via OCL Interface.

Booting Zephyr RTOS

Zephyr OS is based on a small footprint kernel design for use on resource constraints. The Zephyr kernel supports multiple architectures i.e RISC-V (32- and 64-bit). For platforms without MMU/MPU, it supports combining application specific code with a custom kernel to create a monolithic image. Information from the device tree is used to create the application image.

Every Zephyr application is based on the zephyr kernel. The configurable nature of the kernel allows only those features needed by your application making it ideal for systems with limited amounts of memory (as small as 2 KB!). Zephyr kernel supports a variety of device drivers depending on the board and supports generic type API.

The Zephyr build process can be divided into two main phases: a configuration phase and a build phase. The configuration phase begins when the user generates a build system, specifying a board target and a source application directory.

$ west build -b aws_fpga samples/printk/

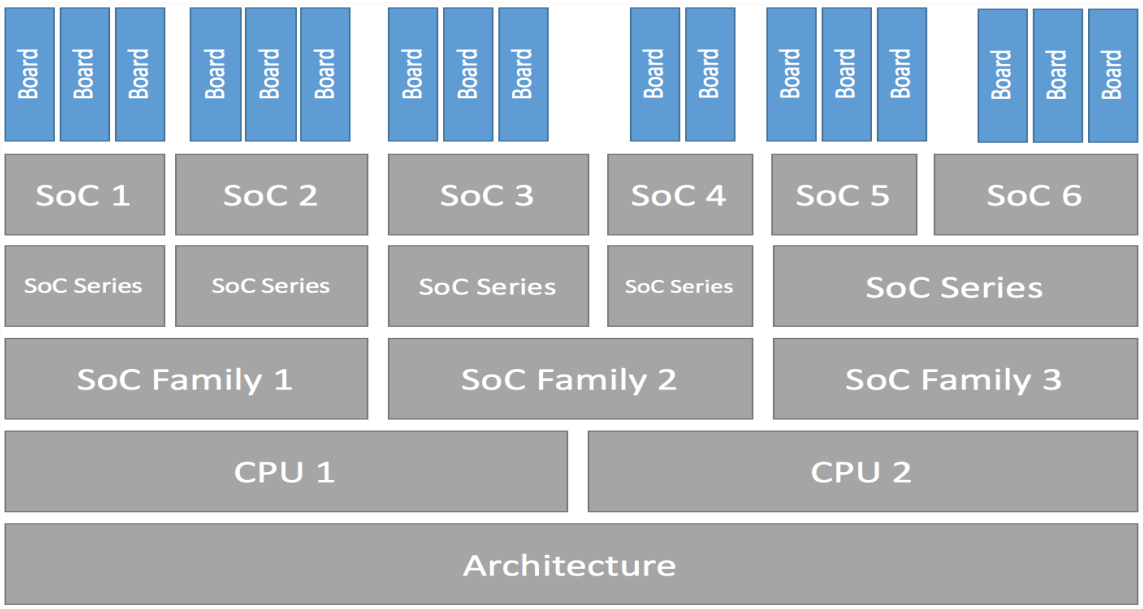

Figure 6: Configuration Hierarchy

The build phase can be divided into three stages: the pre-build, final binary, and post-processing. Pre-build occurs before any source files are compiled, because during this phase header files used by the source files are generated. Final binary from the previous stage is incomplete. The link from the previous stage is repeated, this time with the missing pieces populated. Post processing Finally, if necessary, the completed kernel is converted from ELF to the format expected by the loader and/or flash tool required by the target. This is accomplished in a straightforward manner with objdump.

Create our own application directory in samples/printk. samples/printk/main.c. Create our custom board to initialize SoC by writing own kconfig and dts files. boards/riscv/aws_fpga. Then write the aws_fpga.dtsi(include) file for specifying hardware specification and add it to the on board configuration file. Create an SoC directory on soc/riscv/riscv-privilage/aws/ and write all soc related configuration on it.

For CPU architecture customize .isa and .core files. arch/riscv/*.core *isa in which defining IMA extension for CPU. So in that manner we will define the right architecture for a gcc compiler like -march=rv64ima. Reduce the kernel footprint for applications that do not runtime configuration. I.e driver/serial/Kconfig disable runtime uart configuration.

First build the application with the zephyr kernel. We get zephyr.elf and .lst. By using objcopy extract verilog hex from .elf file. Generated hex will be dumped on DDR-A from the shell DMA port. Hydra starts fetching kernel from DDR. We have two peripheral UART and BRAM to check the expected output from our main application. Writing values of a,b and c in BRAM. Value of e write on the address of UART. Using the runtime driver all peripherals are tested. Uart to uart and hydra to uart testing is done using runtime drivers and assembly.