TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Master Thesis

Hardware Implementation of Preemtable Scheduling Approaches for the Robot Operating System (ROS) via actionlib

Ziyuan Zhang

Born on: 2nd April 1993 in Jinan
Course: Informatik
Matriculation number: 4659711
Matriculation year: 2015

to achieve the academic degree

Master of Science (M.Sc.)

Supervisor
Ariel Podlubne

Supervising professor
Prof. Dr. Diana Göhringer

Submitted on: 7th June 2022

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Task for the preparation of a Master Thesis

Course:	Master Computer Science
Discipline:	Technische Informatik
Name:	Ziyuan Zhang
Matriculation number:	4659711
Matriculation year:	2016
Title:	Hardware Implementation of Preemtable Scheduling Approaches for the Robot Operating System (ROS) via actionlib

Objectives of work

The aim of this research work focuses on the actionlib stack from ROS/ROS2. The main question to answer is whether it is possible to have a reliable scheme for preemtable tasks following the actionlib for Field Programmable Gate Arrays (FPGAs). This would require an analysis of the state-of-the-art on Hardware (HW) schedulers, with a focus on preemtable tasks. The impact of the HW implementation of a client and server on HW must be compared to the standard one on Software (SW) and be quantified to draw conclusions. It is important to evaluate how the exchange of data between HW and SW should be and whether it is benefitial to have the Finite State Machines (FSMs) completley on HW. The benefits must be highlighted. Besides, the impact on the preemtion of HW accelerators must be considered. Finally, the outcome should be a **generalizable** approach, considering that a system will have **multiple** HW-actions (accelerators).

Focus of work

- Research state-of-the-art for similar techniques
  - Real-time and preemtable HW schedulers
  - VHDL implementations of FSMs
- Become familiar with the Robot Operating System (ROS)
  - Understand the differences among **topics**, **services** and **actions**
  - Propose usecases for actionlib as proof of concept for one and multiple actions
- Implement Client and Server FSMs on HW (VHDL)
  - **Proof of concept** for *one action*
  - **Generalize** for *multiple actions* at the same time
    - Are multiple "actionlib" FSMs needed? If so, define the interaction among them
    - Is a *general* FSM needed to coordinate all "child" FSMs?
- Define Performance Metric (e.g., latency, throughput, tasks' and scheduler-FSMs- delays, HW resources) as in the state-of-the-art.
- Evaluate the implementation and compare it using the performance metric with traditional SW actionlib.

Supervisor:	Ariel Podlubne
Issued on:	January 4, 2022
Due date for submission:	June 7, 2022

Prof. Dr. Diana Göhringer
Supervising professor

Statement of authorship

I hereby certify that I have authored this document entitled *Hardware Implementation of Pre-emtable Scheduling Approaches for the Robot Operating System (ROS) via actionlib* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. During the preparation of this document I was only supported by the following persons:

Ariel Podlubne

Additional persons were not involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 7th June 2022

Ziyuan Zhang

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of Adaptive Dynamic Systems

Abstract

We are living in an era where information grows exponentially and creates the need for massive computing power to process that information. At the same time, advances in silicon fabrication technology are approaching theoretical limits, and Moore's Law has run its course. Chip performance improvements no longer keep pace with the needs of cutting-edge, computationally expensive workloads like Artificial Intelligence. To create a faster, more intelligent cloud that keeps up with growing appetites for computing power, data centers need to add other processors distinctly suited for critical workloads. This lead to the need of agile development on hardware architecture. FPGA offer a unique combination of speed and flexibility and greatly shorted the time for hardware development.

Scheduler design is critical in many hardware accelerators. Developers design hardware scheduler component to control the other components of their work independently. This paper explored the possibility of a hardware design method that use a universal hardware switch to substitute scheduler design in each hardware accelerator architecture. This hardware switch tried to fit as much architectures as possible by assuming AXIS Stream connection interface of the to be connected components. In order not to lost the speed gain of hardware accelerator, this architecture should implement most of the component that is possible to be fully combinational, therefore most of the scheduler process can be done within one clock cycle. So the latency need of most scheduler function can be replaced without efficiency loss.

This paper introduced a hardware architecture HWFRA to experiment this idea. HWFRA simulates the behavior of ROS Actionlib to connect independent developed hardware architectures and to make robotic hardware design easier. HWFRA is a communication layer architecture like a switch that lies above all hardware nodes, so these nodes can connect to a ready target node dynamically. This design also adapted ROS actionlib concept so the architectures that implemented the actionlib interface can make a reliable connection. The architecture provide an interface of AXIS Stream and provide a set of scheduler policy to test.

We wrote a test client and a test server and wrapped them with AXIS Stream functions and tested their behavior by analysing the communication of a set of test clients and servers. We experimented the fully combination architecture resource usage scale when client number or server number grows.

Contents

List of Figures III

List of Tables IV

Acronyms V

1 Introduction 1

2 Background 2

3 Related Work 4

4 Design Methodology 7
- 4.1 Design Concept . . . 7
  - 4.1.1 Easy to Parameterize . . . 7
  - 4.1.2 Decouple switch from scheduler . . . 7
  - 4.1.3 Decouple Scheduler Policy From the Scheduler Core . . . 8
  - 4.1.4 3-level Preempting . . . 8
- 4.2 Module design . . . 8
  - 4.2.1 Hardware Architecture . . . 8

5 Implementation 11
- 5.1 Max . . . 11
- 5.2 Min . . . 13
- 5.3 One Hot to Unsigned . . . 14
- 5.4 Generic MUX . . . 14
- 5.5 Generic MUX with switch . . . 15
- 5.6 Signal Register . . . 16
- 5.7 Atomic Client Server Register . . . 16
- 5.8 Priority Table . . . 19
- 5.9 Switch . . . 19
- 5.10 Test Client . . . 21
- 5.11 Test Server . . . 21
- 5.12 Scheduler: Round Robin . . . 22
- 5.13 Scheduler: Least Recent Used . . . 22
- 5.14 Scheduler: Least frequent Used . . . 23

6 Results and Evaluation 24
6.1 Simulation Result 24
6.1.1 Process Explanation 25
6.2 Scaling Result 30
6.2.1 Flip-Flop Usage Scaling Result 30
6.2.2 Lookup Table Usage Scaling Result 35

7 Conclusion and Future Work 40

Bibliography 41

A Appendix 43

List of Figures

4.1 HWFRA-Component Design 9
4.2 Scheduler Design 9
4.3 HWFRA in FPGA 10

5.1 Max Component Design 13
5.2 Min Component Design 13
5.3 One Hot to Unsigned Component Design 14
5.4 Generic MUX Component Design 15
5.5 Generic MUX with switch Component Design 16
5.6 Generic MUX with switch Component Design 16
5.7 Atomic C/S Register Component Design 18
5.8 Switch 20
5.9 Test Client Design 21
5.10 Generic MUX with switch Component Design 21
5.11 Scheduler Design Round Robin 22
5.12 Scheduler Design LRU 22

6.1 Test Bench Architecture 24
6.2 Test Bench Client 25
6.3 Simulation Result on 20ns 26
6.4 Simulation Result on 20ns 26
6.5 Simulation Result on 190ns 27
6.6 Simulation Result on 780ns 27
6.7 Simulation Result on 920 28
6.8 Simulation Result on 1430 29
6.9 Simulation Result on 2280 30
6.10 Flip-Flop Usage Scaling on Client for 4 Servers 32
6.11 Flip-Flop Usage Scaling on Client for 8 Servers 33
6.12 Flip-Flop Usage Scaling on Client for 256 Servers 34
6.13 Flip-Flop Usage Scaling on Server for 8 Clients 35
6.14 Lookup Table Usage Scaling on Client for 4 Servers 36
6.15 Lookup Table Usage Scaling on Client for 8 Servers 37
6.16 Lookup Table Usage Scaling on Client for 256 Servers 38
6.17 Lookup Table Usage Scaling on Server for 8 Clients 39

List of Tables

5.1 Related Index Table for 8 Inputs 12
5.2 Is Bigger Table for 8 Inputs 12
5.3 Example and Explanation 12
5.4 Example and Explanation 13
5.5 Example and Explanation 14
5.6 Example and Explanation 14
5.7 Example and Explanation 15
5.8 Example for Client Register 17
5.9 Example for Server Register 17
5.10 Example for Priority Register 19

6.1 Example for the Input and Output 25
6.2 Flip-Flop Usage Scaling on Client for 4 Servers 31
6.3 Flip-Flop Usage Scaling on Client for 8 Servers 32
6.4 Flip-Flop Usage Scaling on Client for 256 Servers 33
6.5 Flip-Flop Usage Scaling on Server for 8 Clients 34
6.6 Lookup Table Usage Scaling on Clients for 4 Servers 35
6.7 Lookup Table Usage Scaling on Clients for 8 Servers 36
6.8 Lookup Table Usage Scaling on Clients for 256 Servers 37
6.9 Lookup Table Usage Scaling on Servers for 8 Clients 38

Acronyms

DNN	Deep Neural Network	1, 6–8
DSA	Domain-Specific Architecture	2
DSL	Domain-Specific Languages	1, 2, 40
FPGA	Field Programmable Gate Array	1, 2, 4–6, 30, 40, D
HWFRA	Hardware FPGA-ROS-ACTIONLIB	1, 7, 24, 40, D
RTOS	Real-Time Operating System	2, 4, 5

1 Introduction

The growing demand for computationally expensive, state-of-the-art Deep Neural Network (DNN), coupled with diminishing performance gains of general-purpose architectures, has fueled an explosion of Domain-Specific Languages (DSL) and also have to meet the requirements: of flexibility to accommodate evolving state-of-the-art models without costly silicon updates.

This paper introduced a ROS actionlib hardware design framework that based on a scheduler to make robotic hardware design easier. After analyzed many AI hardware accelerators and robotic designs, we concluded the common design of a hardware scheduler to support them independently and designed a communication layer architecture HWFRA above all hardware design to manage data flow. This communication layer architecture adapted ROS actionlib concept and manage data transport between heterogeneous hardware nodes. The framework talk to each independent developers by providing an interface of AXIS Stream and provide a set of scheduler policy.

The remainder of the paper is structured as follows. In Section 2, the background of FPGA and In Section 3, previous research results in FPGA Accelerator, hardware scheduler and robotic systems are reviewed. In Section 4, the design concept and structure of the scheduler is introduced. Section 5 presents the detailed components of hardware scheduler architecture and Section 5 discusses the implementation. In Section 6, its resource utilization and scaling is analyzed.

2 Background

Hennessy, John L. and Patterson, David A. predicted in their report A New Golden Age for Computer Architecture[6] that the new era of computer architecture will expect the same rapid improvement as RISC, the last golden age and this time in terms of cost, security, and energy, as well as in performance. The most important method is high-level, DSL and architectures. The development of DSL freeing architects from the chains of commercial proprietary instruction sets, and require an agile chip development. These changes will lead to a new golden age for computer architects.

AI is perhaps the most important current application space, not only for FPGA, but all segments of the semiconductor industry. Domain-Specific Architecture (DSA) are our hope to feed the huge need on AI compute at the end of Dennard scaling and Moore'DSAs Law. DSAs exploit for the specific domain a more efficient form of parallelism. DSAs can improve performance by making more effective use of the memory hierarchy. Low precision like int8 or int4 are proved adequate for many AI computations and DSAs can use less precision to gain efficiency and save energy when it is adequate. DSAs benefit from targeting software programs written and optimized in DSLs that expose more parallelism.

Agile hardware development start with a software simulator, the easiest and quickest place to find bugs and make changes if a simulator could satisfy an iteration of your project. The next level is FPGA hardware that can run hundreds of times faster than a detailed software simulator. Amazon Web Services offers FPGAs as a service in the cloud, so architects can try FPGAs without even to buy a real hardware and set up a lab.

Real-Time Operating System (RTOS) is an efficient tool to manage the software and make it easy to distribute tasks among developers[10]. Even for small-scale embedded systems, using an software RTOS lets project leaders efficiently partition the entire software design into smaller modular tasks that individual developers can handle. Hardware RTOS enables other developers to develop the hardware drivers and components by providing the abstraction layer and therefore also provides better and safer synchronization.

Robots can be utilized in assembly lines where they interact with human workers in various ways[1]. By completing certain tasks faster and with higher precision, they can have a positive impact on both production cost and output quality. The increasingly complex interaction between human and robot workers intensifies the demand for effective and flexible programming tools. In the past, programming environments and languages for robots were often targeted towards engineers and software developers. As such, they typically favored fine-grained control over all low-level operations performed by the robot over accessibility or intuitive usability. The resulting complexity of the tools causes them to often require years of education and training to use them effectively. Especially in smaller companies, this can slow down the adoption of robotics significantly.

ROS is an open source robot operating system[8]. ROS is not an operating system in the traditional sense that designed for processing management and scheduling; rather, it provides a concept of a structured communications layer above all the host heterogeneous compute cluster with its own operating systems independently. A system built using ROS may consists of a number of processes, potentially in a distribute system on a number of different hosts, connected at run-time dynamically in a peer-to-peer topology.

3 Related Work

The preemptable scheduler problem is a traditional RTOS, and many RTOSs have solved this problem in software and published their solutions in their open source code and documentation. For example, µC/OS-III is a preemptive multitasking kernel[7], so µC/OS-III always runs the most important prepare-to-run tasks. µC/OS-III is a scalable, ROM-capable, preemptive real-time kernel that can manage an unlimited number of tasks. µC/OS-III is a third-generation kernel that provides all the services expected of a modern real-time kernel, such as resource management, synchronization, inter-task communication, and more. However, µC/OS-III offers many unique features not found in other real-time kernels, such as the ability to perform performance measurements at runtime, to signal or message tasks directly, to implement suspend on multiple kernel objects, and more of.

The need for faster response times to external stimuli for fast processing has led to intensive research into processor and RTOS architectures[10]. In this case, most researchers in the field have concluded that certain components (or even the entire HW-RTOS must be embedded in hardware, as it enables increased parallel processing of information, thereby reducing the responsiveness of embedded systems era.

Yi Tang et al. A task queue-based hardware scheduler is introduced for FPGA-based embedded real-time systems[11]. A hardware scheduler is developed to improve the real-time performance of soft-core processor-based computing systems. Hardware schedulers typically accelerate system performance at the expense of increased hardware resources, inflexibility, and integration difficulties. However, the reprogrammability of FPGA-based systems eliminates the problems of inflexibility and integration difficulties. SR-PQ implements a series of shift register blocks that self-order according to their priorities[9]. In addition to priority, the queue block also stores an address field, which is the task identifier (TID) in the task queue. Each node has two parallel entries. The new ingress bus is used to input queue data, while the other input issues control commands (read and write). In addition to this, each block is connected to adjacent blocks. The main disadvantages of hardware schedulers are increased hardware usage and algorithm inflexibility. However, small embedded systems have a limited number of tasks and predefined functions. The hardware scheduler on an FPGA system can be packaged as a customizable IP core. This makes it easy to use and also allows for customization on an application-by-application basis. For example, instead of choosing a fixed number of tasks in the scheduler to support, cores can be tailored to the number of tasks in the target system. This makes the hardware scheduler a more attractive option.

Ionel Zagan et al. introduced Hardware RTOS, a custom scheduler implementation based on multiple pipeline registers and MIPS32 architecture[12]. Task context switching operations, inter-task synchronization and communication mechanisms, and jitter when dealing with aperiodic events are key factors in implementing a real-time operating system (RTOS). In practice and literature, several solutions can be identified to improve the responsiveness and performance of real-time systems[3]. Software implementations of RTOS-specific functions can introduce significant delays that can adversely affect deadlines required by some applications. Their work presents the original implementation of a dedicated processor based on multiple pipeline registers, as well as hardware support for a dynamic scheduler that performs single event management, provides access to architecturally shared resources, prioritizes and executes multiple expected events for the same task[13]. Their work also presents a method for assigning interrupts to tasks. Through dedicated instructions, the integrated hardware scheduler achieves task synchronization with multiple priority events, thereby ensuring the efficient operation of the processor in a real-time control environment.

Davide Conficconi et al. designed a framework for customizable FPGA-based image registration accelerators[2]. The scheduler design is generalized for input fetching of the particular scenario to fit many FPGA devices. At the very beginning of the execution, the architecture acquires two input images (reference and floating). This step depends on the physical memory ports of the device and the available bandwidth. Since not all FPGA-based devices offer multiple memory ports, in order to be as versatile as possible, we consider the case where a single memory port is multiplexed (in the case of having multiple ports, the gas pedal can be replicated depending on the number of ports). However, this situation may impair the performance of the accelerator, especially in the current memory-constrained design. Therefore, it is of utmost importance to design the gas pedal correctly according to the bit width of the memory ports and the memory bandwidth. A solution to alleviate this problem could be to prefetch one or two images on local memory. This is particularly applicable to algorithms like image registration. In fact, in order to register two images, the reference image does not change throughout the optimization process, while the floating image keeps changing. Therefore, if the target FPGA has enough on-chip memory and we apply this feature, our architecture first prefetches the reference image and then reads the floating image as a stream; otherwise, it reads both images simultaneously.

Kaiyuan Guo et al. investigate state-of-the-art CNN models and CNN-based applications and introduced Angel-Eye[5], a complete design flow for mapping CNN onto embedded FPGA. For embedded platforms, CNN-based solutions are too complex to be applied. Various dedicated hardware designs on FPGAs have been carried out to accelerate CNNs, while few of them explore the whole design flow for both fast deployment and high power efficiency. Requirements on memory, computation and the flexibility of the system are summarized for mapping CNN on embedded FPGAs. Based on these requirements, we propose Angel-Eye, a programmable and flexible CNN accelerator architecture, together with data quantization strategy and compilation tool. Data quantization strategy helps reduce the bit-width down to 8-bit with negligible accuracy loss. The compilation tool maps a certain CNN model efficiently onto hardware. Controller part receives, decodes and issues instructions to the other three parts. Controller monitors the work state of each part and checks if the current instruction to this part can be issued. Thus, the host can send the generated instructions to controller through a simple FIFO interface and wait for the work to finish by checking the state registers in controller. This reduces the scheduling overhead for the host at run-time. Other tasks can be done with the host CPU when CNN is running. Parallel execution of instructions may cause data hazard. In hardware, an instruction is executed if: 1) the corresponding hardware is free and 2) the instructions it depends on have finished. Condition 1 is maintained by LOAD Ins FIFO, CALC Ins FIFO and SAVE Ins FIFO as shown in Fig. 10. The instructions in the FIFOs are issued when the corresponding hardware is free. Condition 2 is maintained by checking the dependency code in dep check module.

Michael Cashmore et al. Designed ROSPlan, a Planning in the Robot Operating System[1]. In the general overview of the ROSPlan framework , consisting of the Knowledge Base and Planning System ROS nodes. Sensor data is passed continuously to ROSPlan, used to construct planning problem instances and inform the dispatch of the plan. Actions are dispatched as ROS actions and executed by lower-level controllers which respond reactively to immediate events and provide feedback Architectural overview of the ROSP LAN frame work. Blue boxes are ROS nodes. An example instantiation of the Knowledge Base is illustrated. Sensor data is interpreted for addition to the ontology, and actions are dispatched as ROS actions from the Planning System. The Planning System (PS) and Knowledge Base (KB) communicate during construction of the initial state and plan dispatch. During dispatch the PS will update the KB with the planning filter, and the KB may notify the PS of changes to the environment.

In Microsoft Project Brainwave[4], Jeremy Fowers et al. describe the hardware architecture of a neural processing unit (NPU) designed as a production-scale system for real-time AI and tested on an Intel Stratix 10 280 FPGA. Real-time AI refers to interactive AI-driven services for low-latency evaluation of DNN models. The NPU architecture design for real-time AI services focuses on low latency, high throughput, and high efficiency of DNN model execution. The NPU uses a single-threaded SIMD instruction set architecture (ISA) to achieve good performance in terms of latency and throughput. Brainwave's top-level scheduler must decode each instruction from a scalar control processor into thousands of primitive operations to control operations on many spatially distributed computing resources that scale up to 96,000 multiply-add units and function units for And the vector configuration control signal is based on the arbitration network of the BW ISA instruction chain it receives. The controller architecture is supported by a hierarchical instruction decoder and scheduler and thousands of independently addressable high-bandwidth on-chip memories, and can transparently exploit multiple levels of fine-grained SIMD parallelism. Hierarchical Decoding and Scheduling (HDD) logic extends compound operations into distributed control signals, managing thousands of compute units and dozens of register files and switches. The Brainwave top-level scheduler dispatches to 6 decoders and 4 second-level schedulers, which in turn dispatch to an additional 41 decoders. This scheme, combined with buffering at each stage, keeps the entire compute pipeline running, dispatching a compound instruction from Nios on average every four clock cycles.

4 Design Methodology

4.1 Design Concept

4.1.1 Easy to Parameterize

Hardware HWFRA implementation is designed to serve as a master node for multiple heterogeneous hardware DSA design to communicate like ROS concept. Each architecture design is also called node, DNN can to be parameterized to serve multiple heterogeneous nodes. This design is not re-configurable, that means, after the nodes are selected and the DNN is parameterized, it is fixed on he board and can't change.

Not like traditional Operating System that managing resource like memory or hard disk, the major task for actionlib is to manage data communication, this architecture behave more like a network switch than hardware RTOS implementation like µC/OS. In order to make the design more flexible, the scheduler policy is designed to be configurable, but if a preempt able scheduler policy is chosen, it assumes that the servers must support interrupt management. Like AXIS Stream interface set, if the scheduler policy is fixed, it cannot be reconfigured on the run.

4.1.2 Decouple switch from scheduler

This architecture is designed for heterogeneous client and server nodes to connect. client nodes should be able to connect to any server nodes and vice versa. This architecture separate the switch topology from the scheduler policy by add a register between scheduler and switch. The scheduler can read and write the register and the switch can only read.

The Atomic Client and Server Register hold the information for all nodes. For client nodes, it latches a bit signal indicating if the client is active and a unsigned integer for the server it connected to. For server nodes, the register hold a bit for active and a unsigned for it serving client.

The switch is a full combinational design.

4.1.3 Decouple Scheduler Policy From the Scheduler Core

Full connection for all clients and servers are expensive. We can save the resource usage by change the topology of the switch without affect other designs.

Therefore Scheduler communicate only with the priority table and don't care the topology of the switch component. Priority table can be written only by the scheduler and trigger the change of the atomic client server table before the next clock cycle arrives.

4.1.4 3-level Preempting

The DNN architecture is designed to connect every hardware architecture that communicate with AXIS Stream interface, DNN has been designed to be able to provides 3-level preemption support for preemptable hardware accelerators, DNN regard an architecture as OSLP if the this architecture support interrupt. Traditional RTOS is designed for microprocessors, which always contain a interrupt handling circuit, many large hardware instruction set architectures considered interrupt signal. DNN also designed to be able to support RALP, which means the architecture follows the actionlib framework and implemented the callback interface of actionlib in hardware.

4.2 Module design

This architecture is build for robotic system developers to transport robotic logic seamlessly to a full hardware design. It provides a structured communications layer above the hardware components of a heterogeneous compute cluster like the ROS software implementation did. Hardware ROS application developers need to implement robotic functions with an AXIS-Stream interface to communicate to each other. Then use parametrization tool to generate DNN I/O interface and connect the components together. Every component design is stand alone and heterogeneous so the development work can be easily distributed to multiple engineers and the design can be agilely reused without considering much about interface and synchronization.

This hardware architecture includes xilinx ip cores that synchronize clock rate and unified the AXIS-Stream form, therefore the architecture described following assumes a unified 8-bit wide AXIS-Stream transmission with tvalid, tready and tlast signals. This architecture also assumes a fixed ROS Actionlib signal set and a synchronized clock rate. This Actionlib signal set includes goal, cancel and finish for actionlib client nodes and accepted, rejected, aborted, preempted and succeed for server nodes.

4.2.1 Hardware Architecture

The center of this architecture is a switch. This switch read the information from the scheduler and change the connection of its inputs and outputs.

The I/O signals including AXIS Stream signals, actionlib control signals and actionlib states.

The switch is implemented fully combinational, therefore the re-connection will be done before next clock cycle. If a client or a server is not active, it will be connected to a all '0' dummy target.

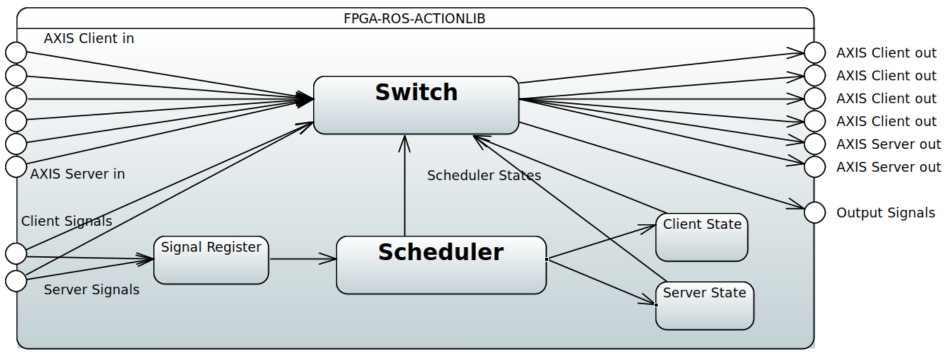

Figure 4.1: HWFRA-Component Design

The scheduler has 3 parts, the center of the architecture lies priority table, a register that holds the priority numbers of all the clients. Only the scheduler core can modify the content and other parts read the register and make a change. The design of the scheduler core is therefore decoupled from other part of HWFRA and easy to change.

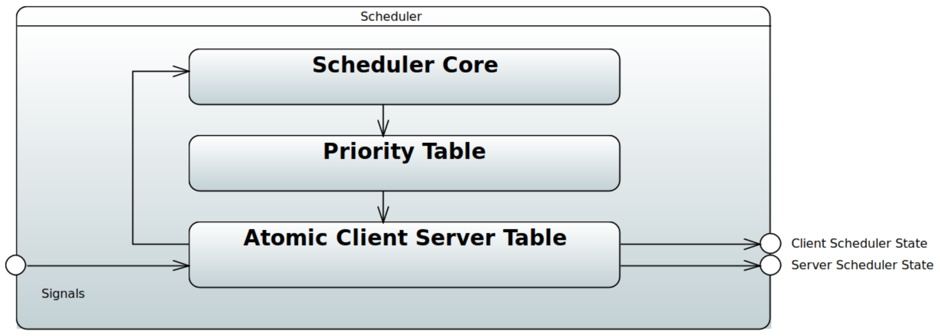

Figure 4.2: Scheduler Design

This architecture lies completely in FPGA, it assumes the clients and servers can be anywhere in the system and communicate via AXIS Stream.

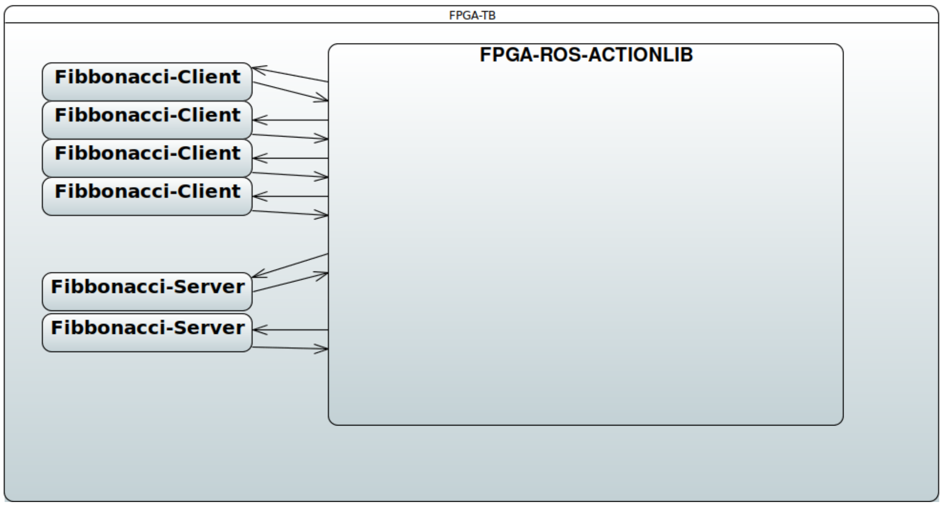

Figure 4.3: HWFRA in FPGA

5 Implementation

Hardware Architecture

The HWFRA architecture is composed by 4 component: Server state machine, Client state machine, Scheduler and Switcher. All AXIS Stream input and output signals from heterogeneous nodes connected to the Switcher and all scheduler control signals and actionlib control signals connect to the Scheduler module.

Max component can find out the maximum number and its index of the given list. The max component output the max number in the input list and the index of the max data in less than 1 clock.

5.1 Max

Max component has a binary tree structure like many other compare circuits of a 2-compare sub-component vector, but each basic compare block output the bigger number and a bit to indicate which side is bigger. '1' for the left number is bigger and '0' means the right one is bigger. We build a compare result table to indicate if each number is bigger in the compare in this round. If a data got '1' in all the compares, than its the biggest number.

Formula 6.1 shows that which output bit in the 2 compare sub-component vector is related to the value of the compare result table. Formula 6.2 compute the bit from index computed in Formula 6.1 should be taken a not function.

let RI, IB functions of a integer, where RI means Related index, and IB = means Is bigger. We define the functions as following:

$IP(i) = \lfloor \frac{\lfloor \frac{i}{r} \rfloor}{2^{i \bmod r+1}} \rfloor + 2^{(r-1-i \bmod r)}$

$IB(i) = \lfloor \frac{i}{r} \rfloor \bmod 2^{i \bmod r+1} < 2^{i \bmod r}$

Here round means the total layer of the compare process and equal to ceil of the data count. The computed tables is a constant to each round number. Therefore the computation is done in compile time and won't add to the hardware architecture complexity. Replace the index of the Related Index Table with the corresponding compare result signal from compare component list, the xor the result with Is bigger Table, the result is one hot vector of the bit-wise-and of r rows of the new table. The basic idea is like finding the winner of a championship match table, you need to check all game of a player, and he is the final winner if all the games is won. Here, the IP

Table 5.1: Related Index Table for 8 Inputs

Input	Round1	Round2	Round3
Input 0	4	2	1
Input 1	4	2	1
Input 2	5	2	1
Input 3	5	2	1
Input 4	6	3	1
Input 5	6	3	1
Input 6	7	3	1
Input 7	7	3	1

Table 5.2: Is Bigger Table for 8 Inputs

Input	Round1	Round2	Round3
Input 0	False	False	False
Input 1	True	False	False
Input 2	False	True	False
Input 3	True	True	False
Input 4	False	False	True
Input 5	True	False	True
Input 6	False	True	True
Input 7	True	True	True

function means to find all the games result of the player, and for compare result ’1’ means the left player won, IB means if the player is on the left side.

Table 5.3: Example and Explanation

I/O	Input	Max	Index
Example	0x34605721	0x7	00000100
Explanation	number list	max input	max number at position 5

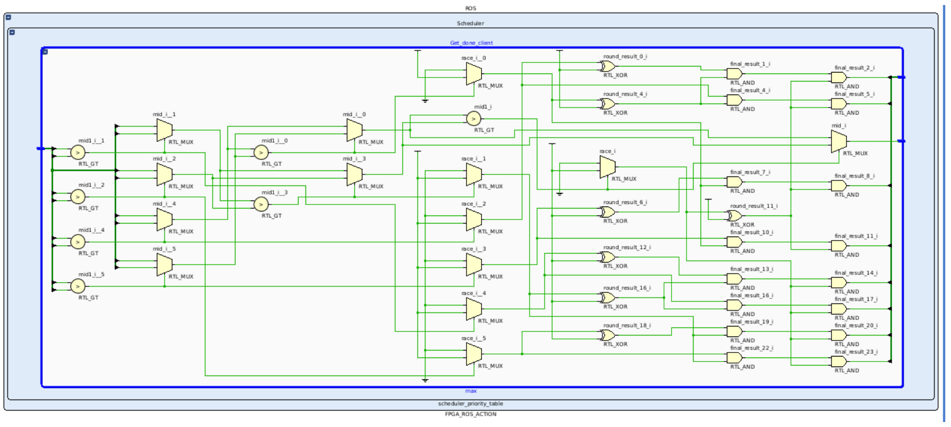

Figure 5.1: Max Component Design

5.2 Min

Min component can find out the minimum number and its index of the given list. Min component is the same like the Max component, except all greater than compare are substituted by less than compares.

Table 5.4: Example and Explanation

I/O	Input	Min	Index
Example	0x34605721	0x0	00010000
Explanation	number list	min input	min number at position 3

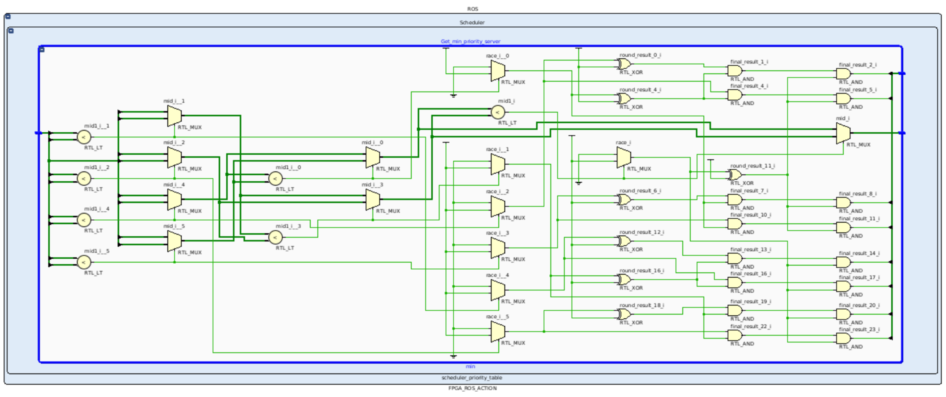

Figure 5.2: Min Component Design

The min component output the min number in the input list and the index of the min data in less than 1s.

5.3 One Hot to Unsigned

This component can convert one hot vector to a compact unsigned integer. One hot vector means a signal vector with exactly one signal is ‘1’ and all other signals are ‘0’.

Table 5.5: Example and Explanation

I/O	One-hot	Index
Example	00010000	3
Explanation	Only the 4th bit is ’1’	Index of ’1’

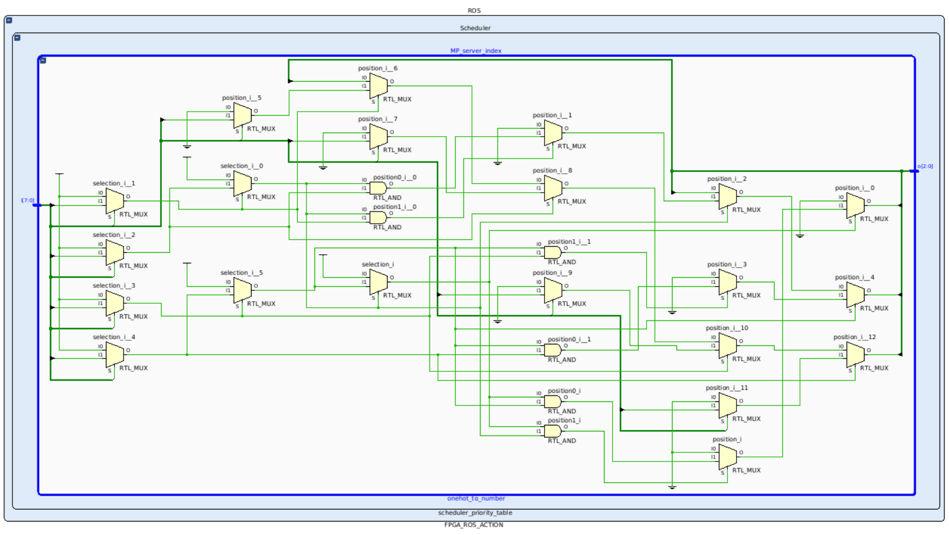

Figure 5.3: One Hot to Unsigned Component Design

5.4 Generic MUX

This component uses the VHDL generic technique and binary tree indexing to generalize a multiplexer for any number of input with any number of data width.

Table 5.6: Example and Explanation

I/O	Input	Selector	Result
Example	0x34605721	3	0
Explanation	8 Input number with 4 bits each		

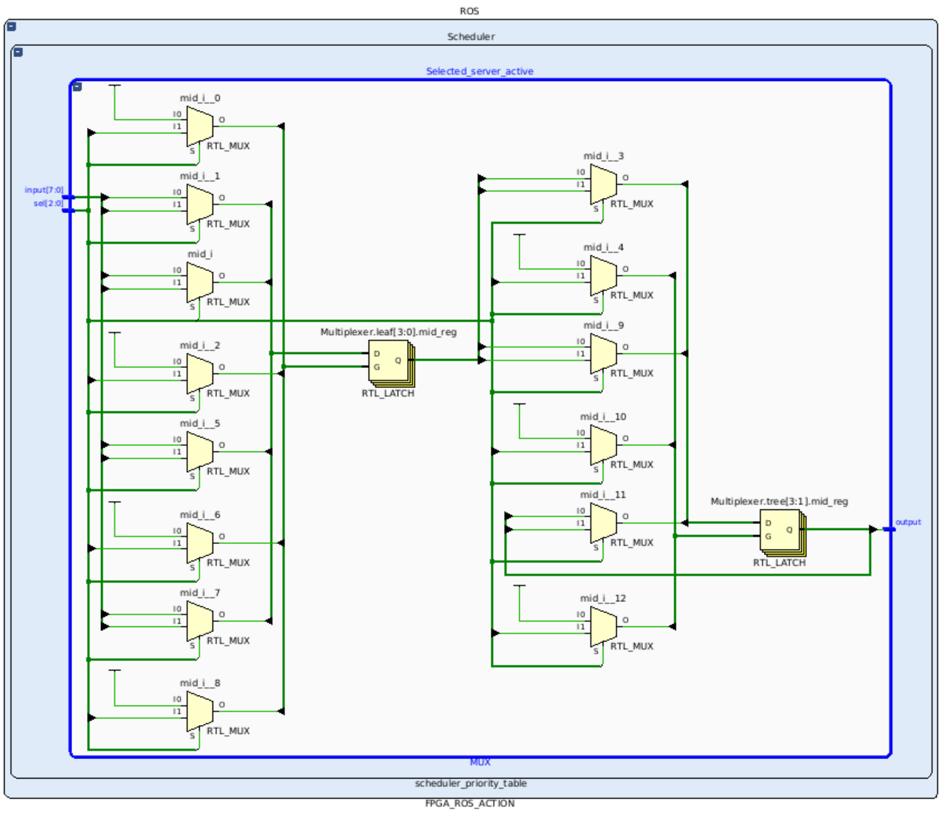

Figure 5.4: Generic MUX Component Design

5.5 Generic MUX with switch

This component add a ready flag bit vector as input to a generic MUX. If the ready bit of the selected data is '0', then it output all zero, else it output the selected data.

Table 5.7: Example and Explanation

I/O	Input	Selector	Switch	Result
Example	0x34605721	3	'0'	0x0
Explanation	Set switch to '0', result will be all zero.			

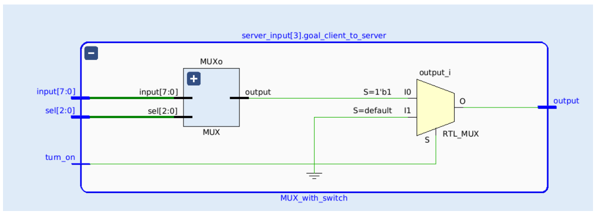

Figure 5.5: Generic MUX with switch Component Design

5.6 Signal Register

This component is designed for save signal that not hold. For example, the AXIS Stream tlast is set only at the last bit of transmission. If we want to use the this signal as a finish control signal to adapt AXIS Stream architectures seamlessly, we have to add a register to hold its value.

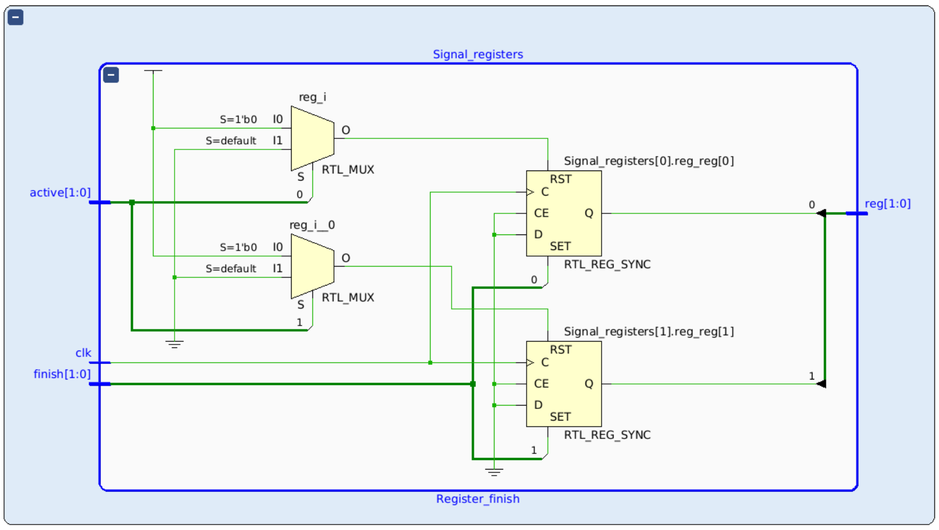

Figure 5.6: Generic MUX with switch Component Design

5.7 Atomic Client Server Register

This register holds the scheduler information for all nodes, including client and server nodes. One server can serve only one client at the same time. The id of its serving client is stored in the register and if the active flag is active, then the id is valid. This the the same for clients and the connection state of a client and a server is added, changed or removed atomically.

Table 5.8: Example for Client Register

I/O	Server	Active
Client0	0x0	’0’
Client1	0x3	’1’
Client2	0x3	’0’
Client3	0x2	’1’

Table 5.9: Example for Server Register

I/O	Client	Active
Server0	0x0	’0’
Server1	0x1	’0’
Server2	0x3	’1’
Server3	0x2	’1’

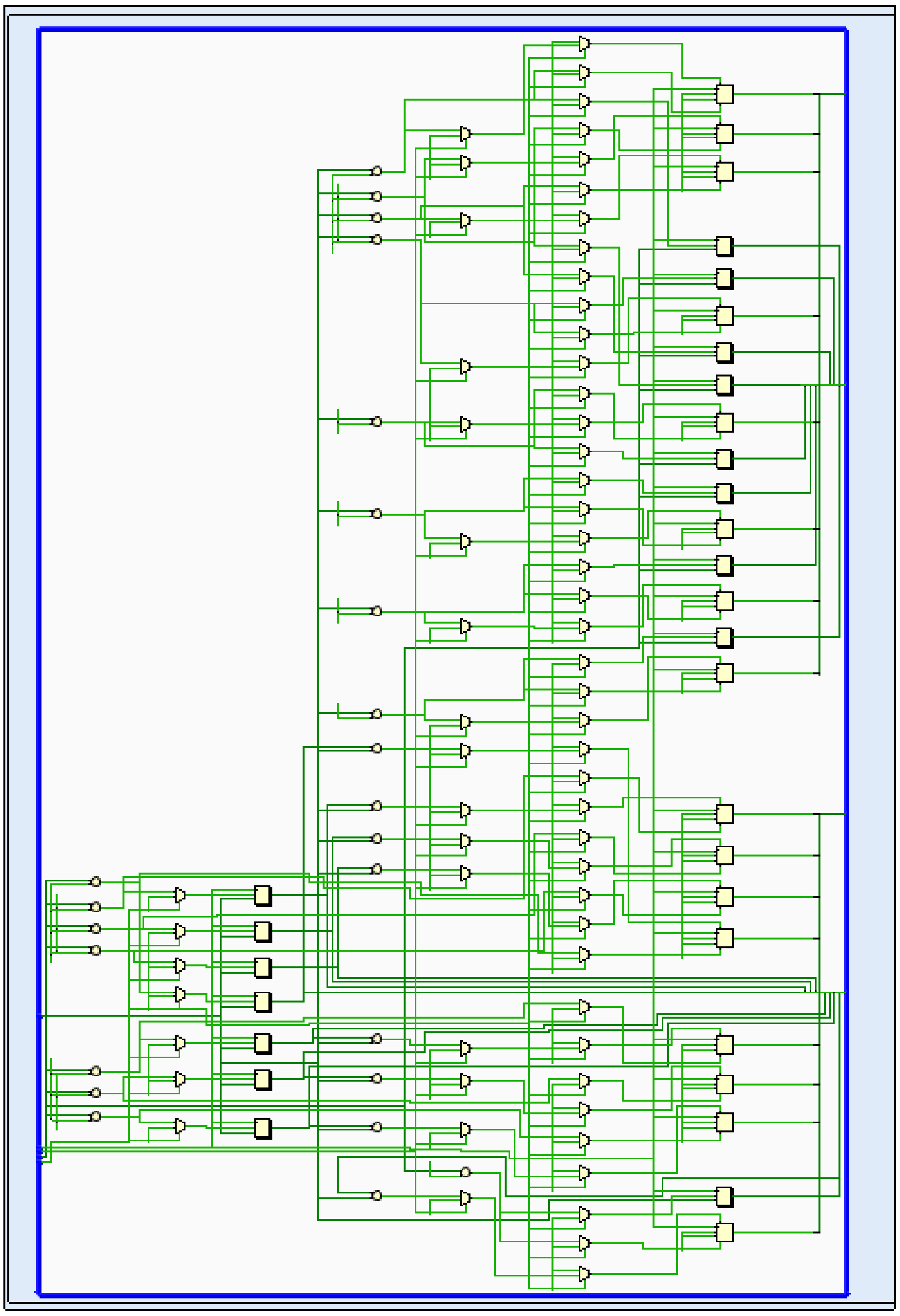

Figure 5.7: Atomic C/S Register Component Design

5.8 Priority Table

This register holds all priority numbers of clients and servers. The max priority number of ready clients and servers and their index will be found and then the client and server will be write into atomic client server register.

Table 5.10: Example for Priority Register

I/O	Priority	Ready
Server0	0x0	’0’
Server1	0x1	’0’
Server2	0x3	’1’
Server3	0x2	’1’
Client0	0x0	’0’
Client1	0x1	’0’
Client2	0x3	’1’
Client3	0x2	’1’

5.9 Switch

This switch design connect each client node to all the server nodes. Each client node has its own circuit to listen to the change of its register in the atomic client server table. This is a simple topology to make sure every client can connected to the correspondent server as soon as possible.

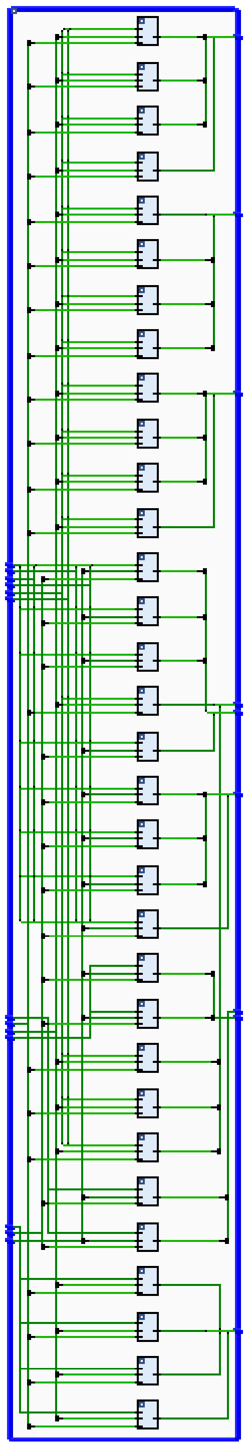

Figure 5.8: Switch

5.10 Test Client

This paper has also implemented a test client that implemented a subset of ROS actionlib framework design. This test client read a input file from simulator host operating system, Linux or Windows, generate a list of goals according to the input and wait to receive a result. When all goals are served, it write a output data and quit.

The goal data structure is an single 32 bit integer.

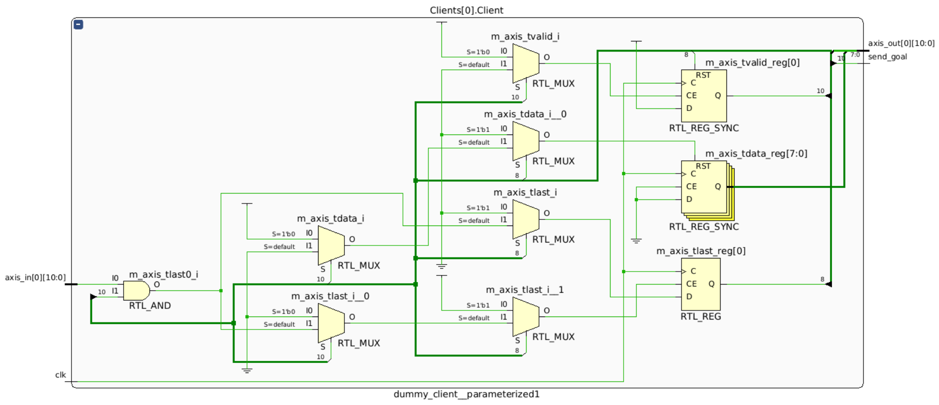

Figure 5.9: Test Client Design

5.11 Test Server

Test server implemented a Fibonacci function to test the functionality of the HWFRA design. After received a 32 bit integer number goal from its input AXIS Stream port, it return the number of the Fibonacci sequence at that position as a 32 bit integer.

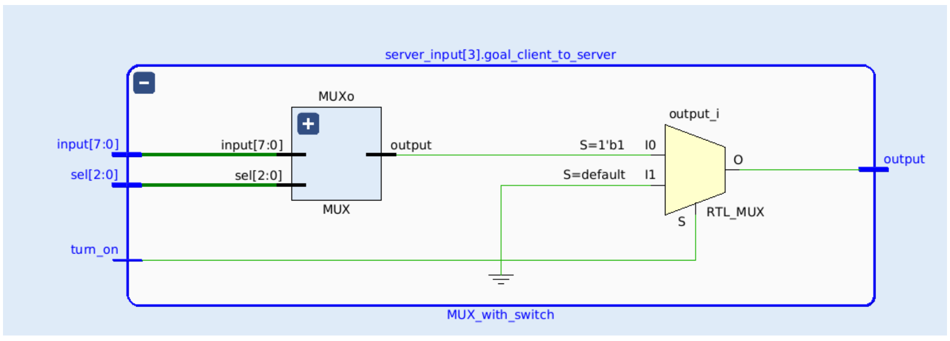

Figure 5.10: Generic MUX with switch Component Design

5.12 Scheduler: Round Robin

This scheduler part implemented the simplest scheduler policy round robin. It checks all the clients if it is ready in turn. If a ready client is ready, then it check all the servers in turn to find a available one.

Note this implementation don't use priority table system described above and is not fully combinational. It takes in worst case number of clients -1 clock cycle to decide next client and number of servers -1 clock cycle to find a server.

It is possible to reduce worst case time to 1 circle by using the priority table system provided above by letting each client/server priority increase by 1 and reset to 0 if it reaches number of client/server.

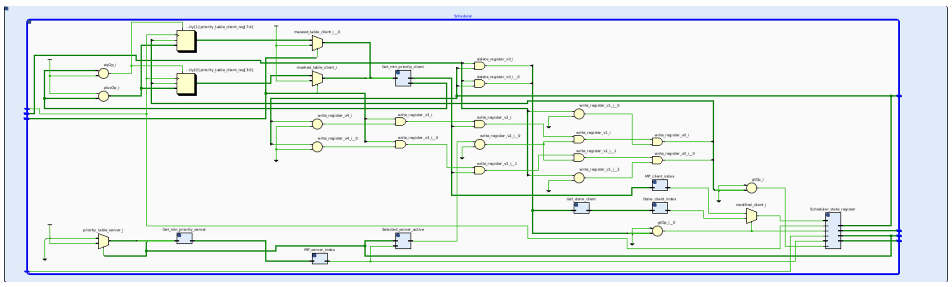
Figure 5.11: Scheduler Design Round Robin

5.13 Scheduler: Least Recent Used

The scheduler read the priority table for the lowest priority number and hold a max signal, if a node is used, then its priority will be set to max priority + 1.

If the max reach the max capacity of the priority number width, which is 255 for width equals 8, reset all priority number to 0.

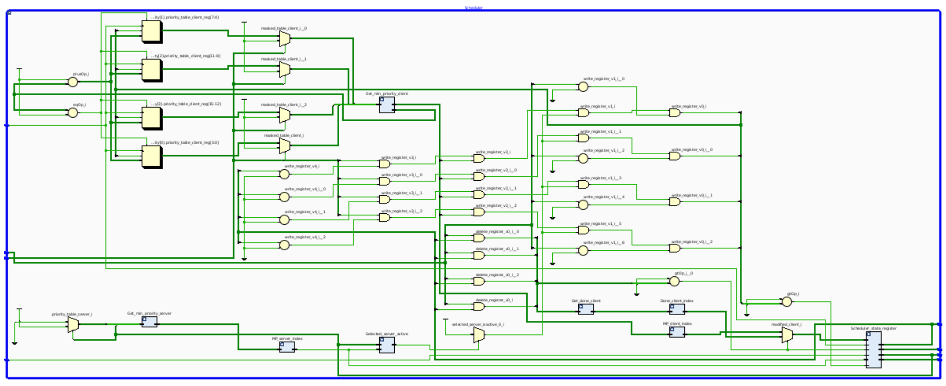
Figure 5.12: Scheduler Design LRU

5.14 Scheduler: Least frequent Used

The scheduler read the priority table for the lowest priority number, if a node is used, then its priority will be set to its priority number + 1.

If the max reach the max capacity of the priority number width, which is 255 for width equals 8, reset all priority number to 0.

6 Results and Evaluation

6.1 Simulation Result

This simulation tested how the architecture works on the test bench. The scheduler is connected to four clients and two servers in this test scenario.

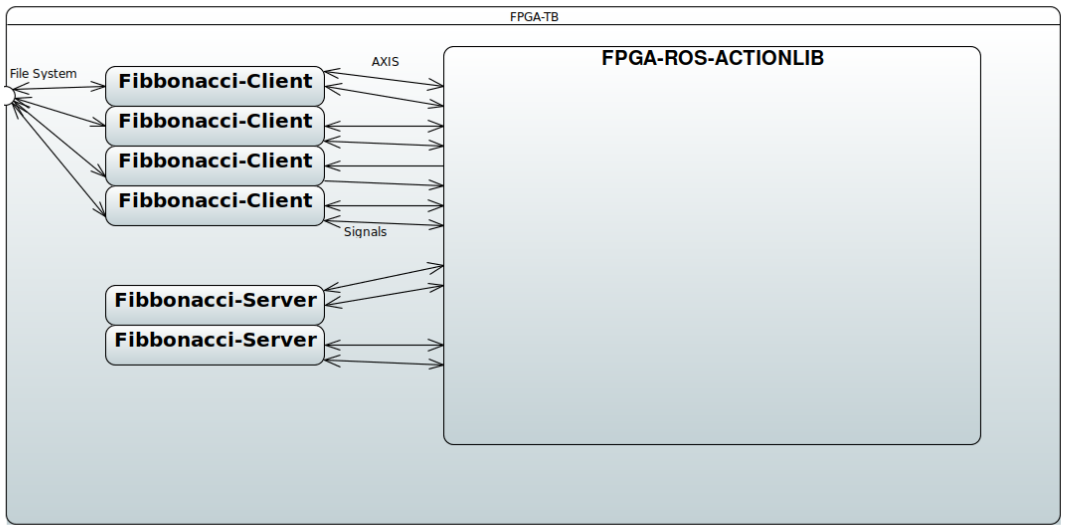

Figure 6.1: Test Bench Architecture

The Clients reads file from its hosting pc and read the content. Then it generates goals and send them out. After that, clients will waiting for the result and write it to a output file.

HWFRA will listen all the clients and servers. If they are ready, then the architecture will link the chosen client and the chosen server together.

Servers wait for goals from the input AXIS Stream after a success receive, it will return the n^{th} number of the Fibonacci sequence. After the last bit of result is transmitted, the Server will set it self to succeed state, and be ready for next goal.

The overall process is showed in a following image. Indicating the general look of the entire process.

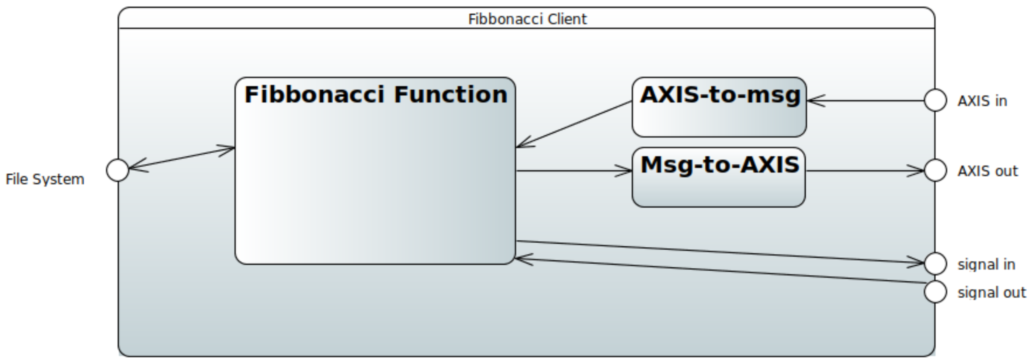

Figure 6.2: Test Bench Client

Table 6.1: Example for the Input and Output

Client Number	Input	Output
4 Servers	4 Clients	116
4 Servers	8 Clients	168
4 Servers	16 Clients	268
4 Servers	32 Clients	464
4 Servers	64 Clients	860
4 Servers	128 Clients	1642
4 Servers	256 Clients	3188

6.1.1 Process Explanation

This is the running log for the test bench process.

Before 70ns, all clients reading its input data via an AXIS Stream interface. Here is process is simulated by reading a file from the hosting operating system.

70ns: All client finish reading their data head

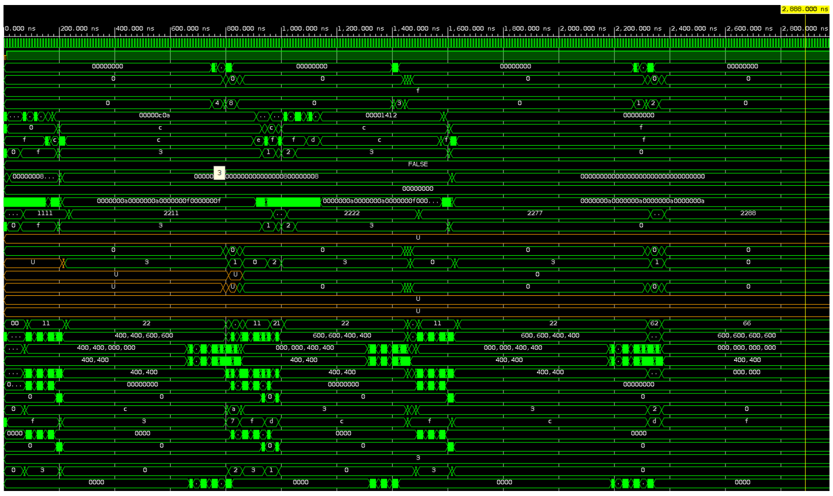
Figure 6.3: Simulation Result on 20ns

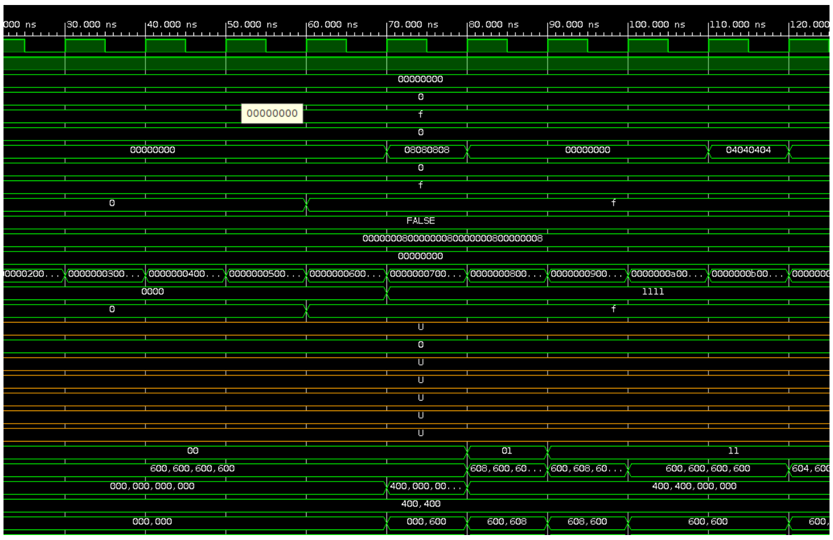
Figure 6.4: Simulation Result on 20ns

80ns: Server 1 linked to Client 3, data transport start ,server state set to 1

90ns: Server 2 linked to Client 2

200ns: Last bit of the data Client 3 and Client 2 , server validating data

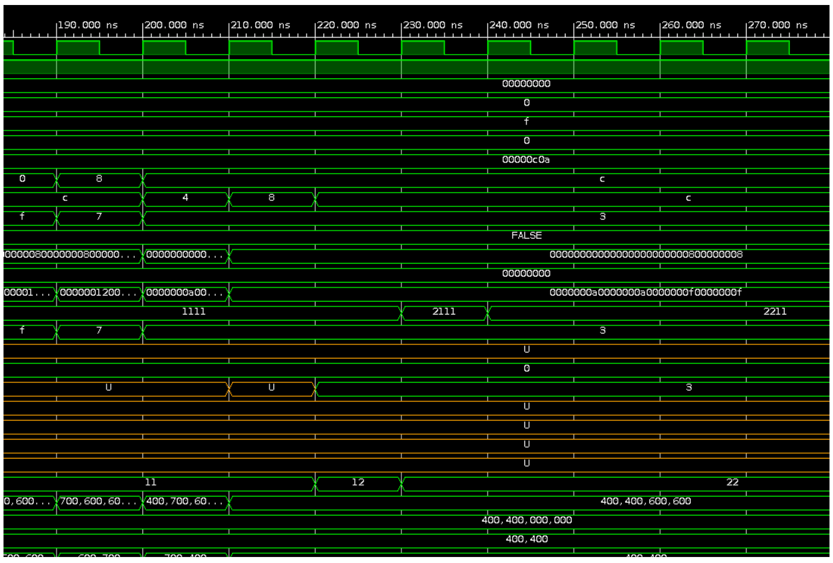

Figure 6.5: Simulation Result on 190ns

220ns: Data verified Server 1 send accept state set to ACTIVE

230ns: Data verified Server 0 send accept state set to ACTIVE , Client 3 set state to ACTIVE

240ns: Client 2 set state to ACTIVE

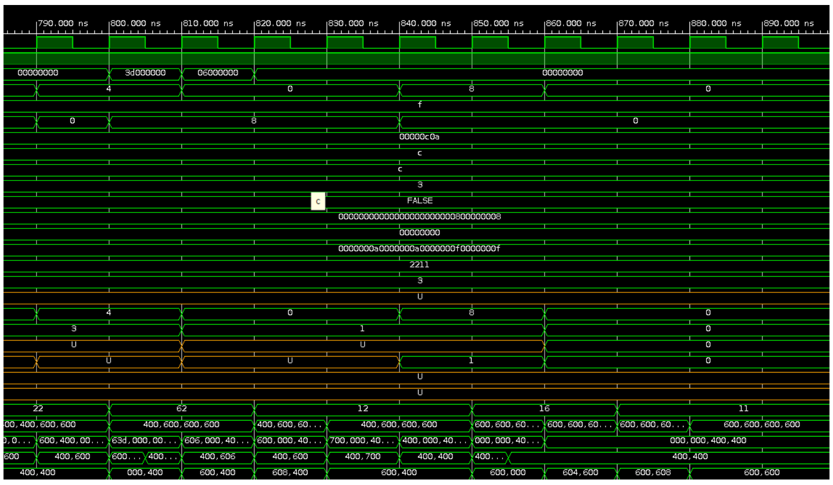

Figure 6.6: Simulation Result on 780ns

800ns: Server 1 state set to SUCCEED after complete the computation send the last bit of result.

820ns: Server 1 state set to ready and receiving data from Client 1

850ns: Server 0 state set to SUCCEED after complete the computation send the last bit of result.

870ns: Server 0 state set to ready and receiving data from Client 1

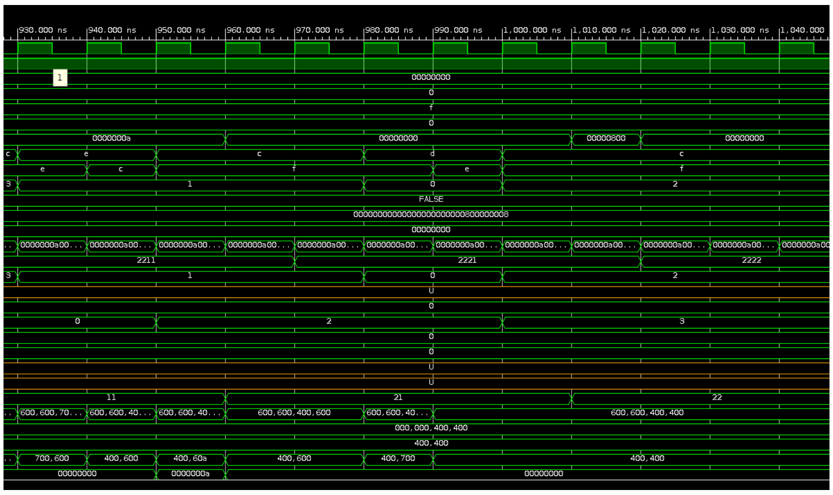

Figure 6.7: Simulation Result on 920

960ns: Data verified Server 1 send accept state set to ACTIVE

970ns: Client 1 set state to ACTIVE

1000ns: Data verified Server 0 send accept state set to ACTIVE

1010ns: Client 0 set state to ACTIVE

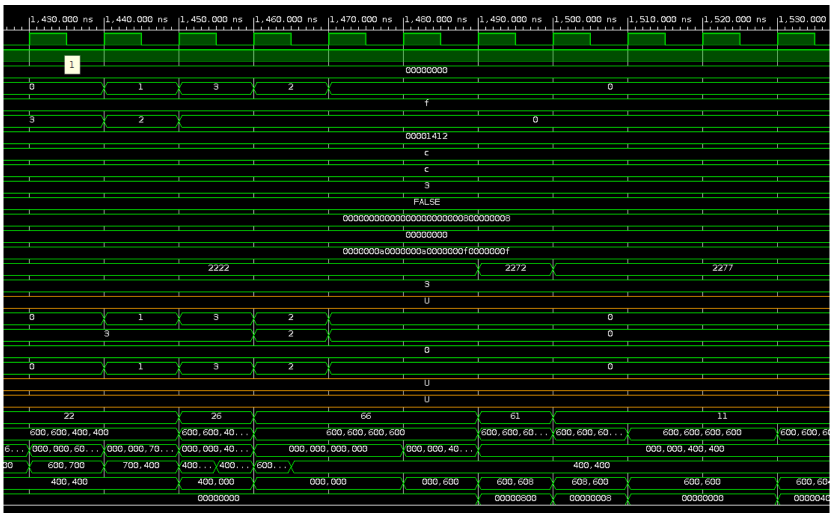

Figure 6.8: Simulation Result on 1430

1440ns: Server 0 set state to SUCCEED

1450ns: Server 1 set state to SUCCEED

1490ns: Client 0 set state to 7, Server 1 set state to 1,

1500ns: Client 1 set state to 7, Server 0 set state to 1,

1630ns: Server 0 set state to ACTIVE

1630ns: Server 1 set state to ACTIVE

2320ns: Server 1 set state to SUCCEED,

2350ns: Server 0 set state to SUCCEED,

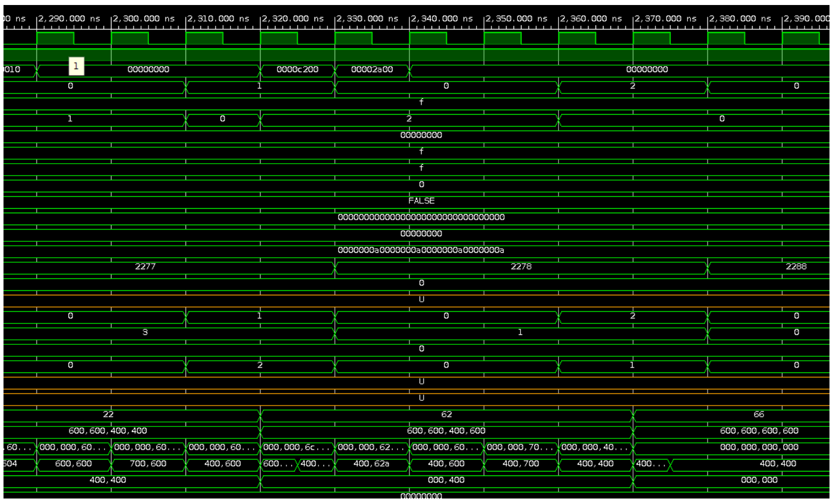

Figure 6.9: Simulation Result on 2280

6.2 Scaling Result

This hardware architecture is designed to be easily scaled to fit different usage scenarios. Thanks to the VHDL generic mechanism, testing a architecture for more client nodes and server nodes requires changing the number in the generic part of the source code.

6.2.1 Flip-Flop Usage Scaling Result

Flip-Flop is one of the most important resource in FPGA design to register data. Flip-Flop array is the core component of the priority table and atomic client/server register design. Flip-flops are also used in the client and server state-machine design to store their status. Scheduler policy implementations also have to use Flip-Flops, but the number is depend on its architecture and hard to predict.

In this section, the usage of Flip-flop will be estimated first theoretically, then a software simulation in amd xilinx vivado™will be posted to prove the conclusion.

Priority Table

Priority table is used to hold the priority number of all client nodes. The width of the priority number should be given by adjusting a parameter. This priority number width should be decided according to the specific scheduler policy. The FF usage is therefor linear to the number of clients if the priority number width is set.

Let F be the number of flip-flops, C the number of clients and W the width of a priority number.

$F = C \times W$

Atomic Client Server Register

The register has two parts, one for clients to hold the server that serving the client, one for servers to hold the client that the server is serving. Each client has also a bit to hold.

Let F be the number of flip-flops, C the number of clients S the number of servers and W the width of a priority number.

$F = C \times \log S + S \times \log C + C + S$

Flip-Flop Usage Scaling on Client for 4 Servers

Flip-Flop Usage Scaling on Client for 4 Servers, the more clients we have, the more Flip-Flop we need to add, the number scaled almost linearly

Table 6.2: Flip-Flop Usage Scaling on Client for 4 Servers

Server Number	Client Number	Flip-Flop
4 Servers	4 Clients	116
4 Servers	8 Clients	168
4 Servers	16 Clients	268
4 Servers	32 Clients	464
4 Servers	64 Clients	860
4 Servers	128 Clients	1642
4 Servers	256 Clients	3188

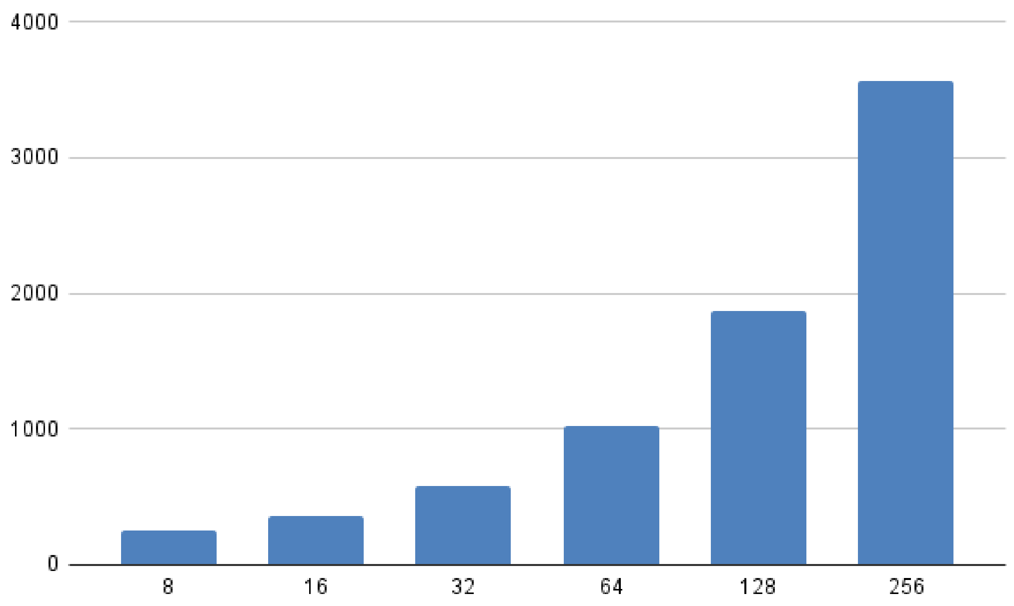

Figure 6.10: Flip-Flop Usage Scaling on Client for 4 Servers

Flip-Flop Usage Scaling on Client for 8 Servers

Table 6.3: Flip-Flop Usage Scaling on Client for 8 Servers

Server Number	Client Number	Flip-Flop
8 Servers	8 Clients	248
8 Servers	16 Clients	360
8 Servers	32 Clients	576
8 Servers	64 Clients	1016
8 Servers	128 Clients	1872
8 Servers	256 Clients	3560

Flip-Flop Usage Scaling on Client for 8 Servers, the usage almost doubled as the Flip-Flop usage than the last simulation.

In this scenario, the Flip-Flop also scaled almost linearly.

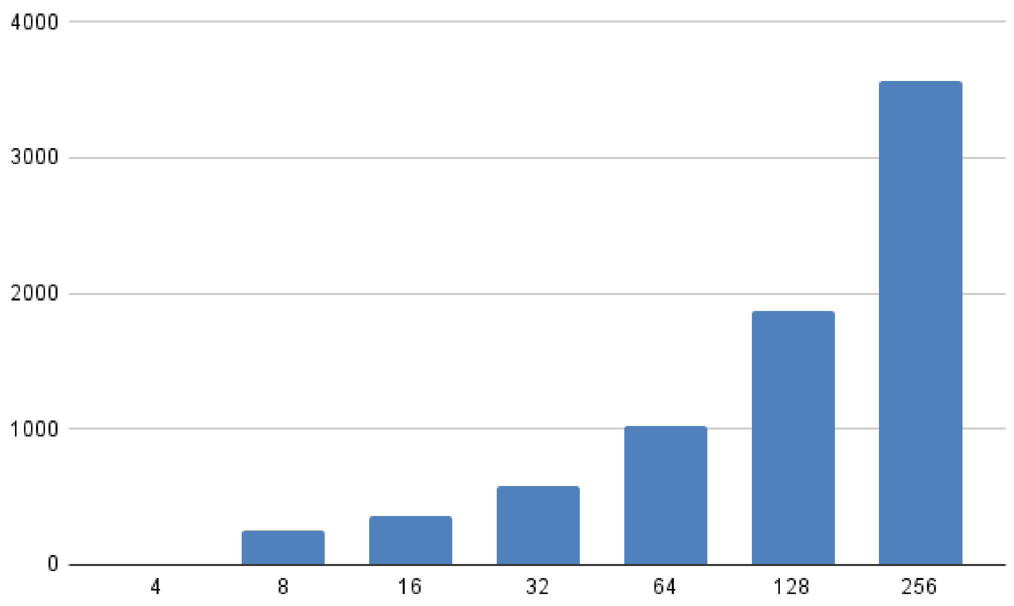

Figure 6.11: Flip-Flop Usage Scaling on Client for 8 Servers

Flip-Flop Usage Scaling on Client for 256 Servers

Table 6.4: Flip-Flop Usage Scaling on Client for 256 Servers

Server Number	Client Number	Flip-Flop
256 Servers	8 Clients	5016
256 Servers	16 Clients	5680
256 Servers	32 Clients	6752
256 Servers	64 Clients	9152

We change the Server number to 256 this time and the resource usage is huge. user should avoid that large number of client and server combination and choose a better topology of nodes.

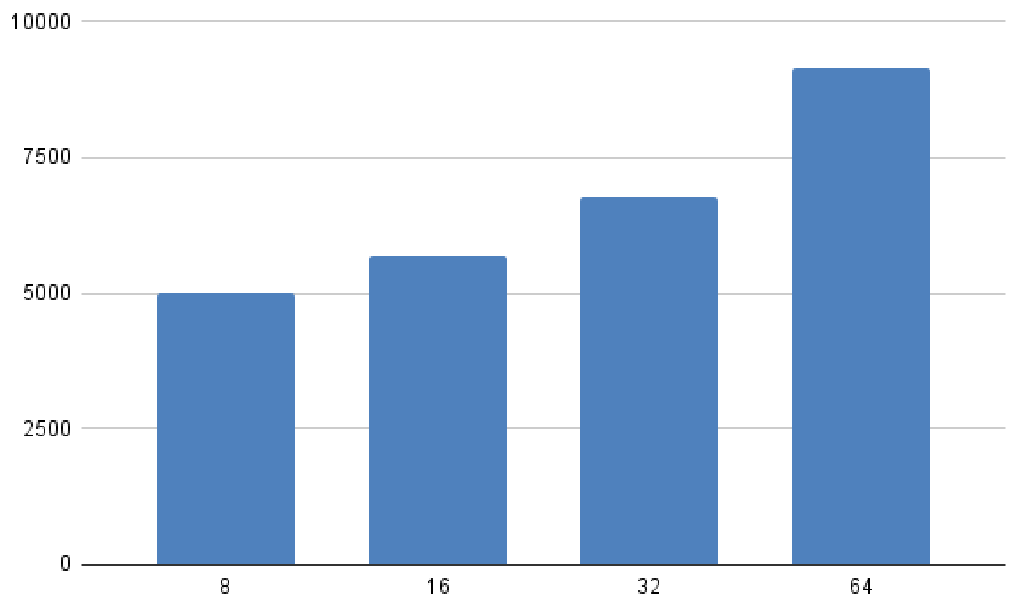

Figure 6.12: Flip-Flop Usage Scaling on Client for 256 Servers

Flip-Flop Usage Scaling on Servers for 8 Clients

Table 6.5: Flip-Flop Usage Scaling on Server for 8 Clients

Client Number	Server Number	Flip-Flop
8 Clients	2 Servers	124
8 Clients	4 Servers	168
8 Clients	8 Servers	248
8 Clients	16 Servers	400
8 Clients	32 Servers	693
8 Clients	64 Servers	1303
8 Clients	128 Servers	2504
8 Clients	256 Servers	5016

This time we fixed the number of client and see how Flip-Flop usage scales on server number. The result is the same, resources usage grows linearly as number of server rises.

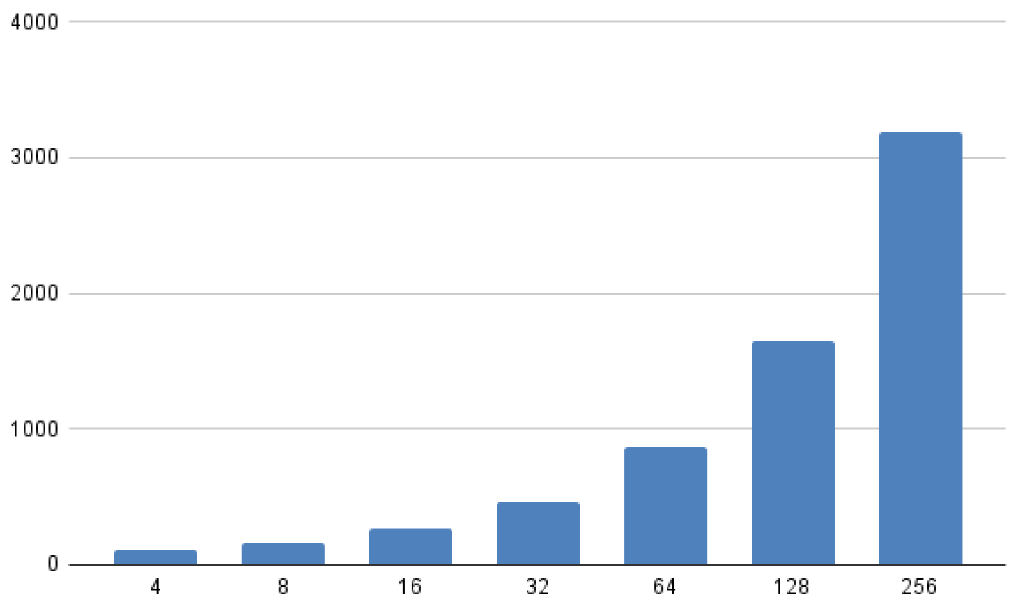

Figure 6.13: Flip-Flop Usage Scaling on Server for 8 Clients

6.2.2 Lookup Table Usage Scaling Result

Lookup Table Usage Scaling on Client for 4 Servers, the more clients we have, the more Lookup Table Usage we need to add, the number scaled almost linearly therefor we know, do not use that big numbers or you will use up all the FPGA resources and have a low overhead code percentage.

Lookup Table Usage Scaling on Clients for 4 Servers

Table 6.6: Lookup Table Usage Scaling on Clients for 4 Servers

Client Number	Server Number	Lookup Table
4 Clients	4 Servers	405
8 Clients	4 Servers	742
16 Clients	4 Servers	1341
32 Clients	4 Servers	2374
64 Clients	4 Servers	4822
128 Clients	4 Servers	9333
256 Clients	4 Servers	18857

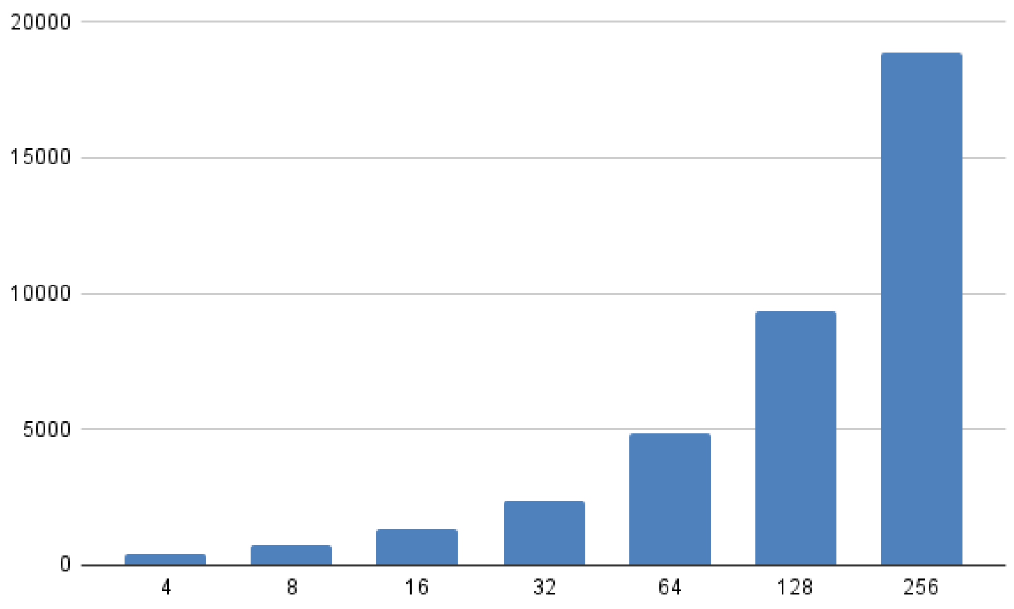

Figure 6.14: Lookup Table Usage Scaling on Client for 4 Servers

Lookup Table Usage Scaling on Clients for 8 Servers

Lookup Table Usage Scaling on Client for 8 Servers, the usage almost doubled as the Lookup Table usage than the last simulation.

In this scenario, the Flip-Flop also scaled almost linearly.

Table 6.7: Lookup Table Usage Scaling on Clients for 8 Servers

Client Number	Server Number	Lookup Table
8 Clients	8 Servers	1143
16 Clients	8 Servers	2044
32 Clients	8 Servers	3709
64 Clients	8 Servers	7342
128 Clients	8 Servers	14314
256 Clients	8 Servers	28542

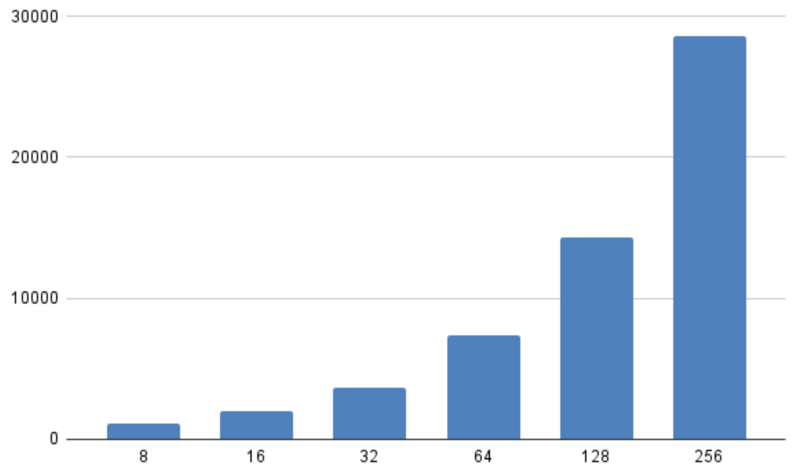

Figure 6.15: Lookup Table Usage Scaling on Client for 8 Servers

Lookup Table Usage Scaling on Clients for 256 Servers

We change the Server number to 256 this time and the resource usage is huge also for lookup tables. User should avoid that large number of client and server combination and choose a better topology of nodes.

Table 6.8: Lookup Table Usage Scaling on Clients for 256 Servers

Client Number	Server Number	Lookup Table
8 Clients	8 Servers	25032
16 Clients	8 Servers	39965
32 Clients	8 Servers	70665
64 Clients	8 Servers	134825

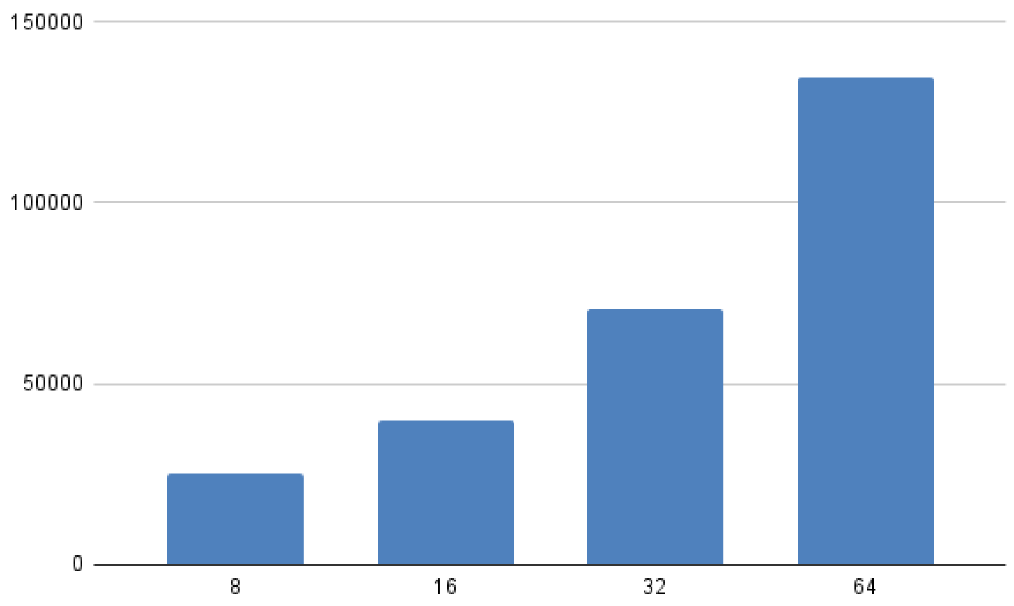

Figure 6.16: Lookup Table Usage Scaling on Client for 256 Servers

Lookup Table Usage Scaling on Servers for 8 Clients

This time we fixed the number of client and see how Lookup Table usage scales on server number. The result is the same, resources usage grows linearly as number of server rises.

Table 6.9: Lookup Table Usage Scaling on Servers for 8 Clients

Server Number	Client Number	Lookup Table
2 Servers	8 Clients	594
4 Servers	8 Clients	742
8 Servers	8 Clients	1143
16 Servers	8 Clients	1871
32 Servers	8 Clients	3310
64 Servers	8 Clients	6321
128 Servers	8 Clients	12686
256 Servers	8 Clients	25032

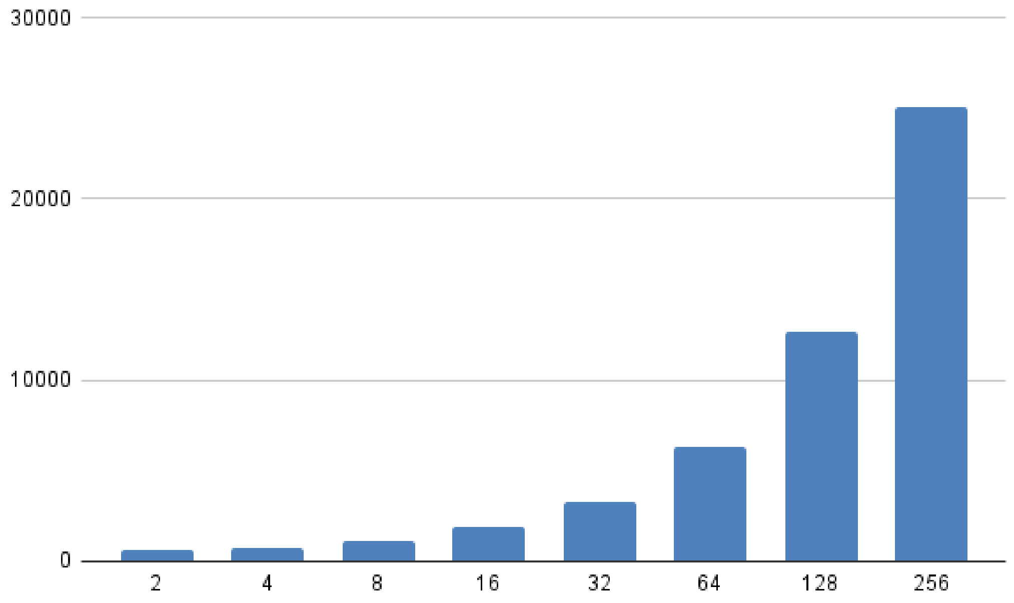

Figure 6.17: Lookup Table Usage Scaling on Server for 8 Clients

7 Conclusion and Future Work

In the era of AI compute at the end of Dennard scaling and Moore's Law, a lot of DSL architectures and codes emerges with help of the agile hardware design empowered by FPGA. This paper provided a general interface design based on AXIS Stream for a heterogeneous distributed hardware design. We experimented the fully combination architecture to choose a max number from a list and complete the switch in two clock cycles. We then analysed the scaling behavior of the design and given suggestions of a complex combined hardware design.

Although the implementation of the hardware scheduler proved the concept of the design, it is in practise not well parameterized and don't fit the purpose of easy to use and flexibility. Although this design separated the scheduler policy from the framework and make the design easier, many popular scheduler policy implementation are not yet available to choose. In the future, this paper will make the HWFRA architecture easier to parameterize and test, the library of scheduler policy will be enriched.

Bibliography

[1] Michael Cashmore et al. “Rosplan: Planning in the robot operating system”. In: *Proceedings International Conference on Automated Planning and Scheduling, ICAPS* 2015-January (2015), pp. 333–341. ISSN: 23340843.

[2] Davide Conficconi et al. “A framework for customizable fpga-based image registration accelerators”. In: *FPGA 2021 - 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2021), pp. 251–261. DOI: 10.1145/3431920.3439291.

[3] Eugen Dodiu, Vasile Gheorghita Gaitan, and Adrian Graur. “Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers-architecture description”. In: *2012 Proceedings of the 35th International Convention MIPRO*. IEEE. 2012, pp. 859–864.

[4] Jeremy Fowers et al. “A Configurable Cloud-Scale DNN Processor for Real-Time AI”. In: (2018). DOI: 10.1109/ISCA.2018.00012.

[5] Kaiyuan Guo et al. “Angel-Eye : A Complete Design Flow for Mapping CNN Onto Embedded FPGA”. In: 37.1 (2018), pp. 35–47.

[6] John L. Hennessy and David A. Patterson. “A new golden age for computer architecture”. In: *Communications of the ACM* 62.2 (2019), pp. 48–60. ISSN: 15577317. DOI: 10.1145/3282307.

[7] *microCOS document*. https://www.silabs.com/developers/micrium. Accessed: 2022-05-30.

[8] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[9] Sergio Saez et al. “Hardware scheduler for complex real-time systems”. In: *IEEE International Symposium on Industrial Electronics* 1 (1999), pp. 43–48. DOI: 10.1109/isie.1999.801754.

[10] William Stallings. “P Ractice P Roblems O Perating S Ystems :” in: *Management* 4.6 (2008), pp. 1–7.

[11] Yi Tang and Neil W. Bergmann. “A hardware scheduler based on task queues for FPGA-based embedded real-time systems”. In: *IEEE Transactions on Computers* 64.5 (2015), pp. 1254–1267. ISSN: 00189340. DOI: 10.1109/TC.2014.2315637.

[12] Ionel Zagan and Vasile Gheorghita Gaitan. “Improving the performance of Real-Time Event Processing based on Preemptive Scheduler FPGA Implementation”. In: *2020 15th International Conference on Development and Application Systems, DAS 2020 - Proceedings* (2020), pp. 49–55. DOI: 10.1109/DAS49615.2020.9108930.

[13] Ionel Zagan and Vasile Gheorghiţă Găitan. “Hardware RTOS: Custom scheduler implementation based on multiple pipeline registers and MIPS32 architecture”. In: *Electronics (Switzerland)* 8.2 (2019). ISSN: 20799292. DOI: 10.3390/electronics8020211.

A Appendix

This project is opensource under `https://phabricator.ads.inf.tu-dresden.de/source/ziyuanzhanghwactionlib/`

It is also available on GitHub. `https://github.com/ziyuan400/FPGA_ROS_ACTIONLIB`