Towards a New Design and Execution Framework for SystemGALS

Ziming Xie

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Engineering, The University of Auckland, 2022.

Abstracts

Industry 4.0 depicts a vision of intelligent and agile production manufacturing processes that can achieve product customisation rapidly and economically. It encourages the manufacturing systems to comprise a group of interconnected and configurable subsystems rather than a unitary product line with fixed procedures. These agile production systems further require an agile design and deployment methodology that has dramatically facilitated the prosper of software applications. Different from traditional software development, the scope of industrial system design ranges from programming languages that focus on system-level behaviours to execution platforms and physical plant manufacturing. These form an extensive design exploration space for agile system development. This thesis starts with an example of an intelligent Sorter System and leverages SystemGALS, a system-level programming language, to design its control system. In addition, we employ Chipyard, an agile RISC-V SoC framework, to build two execution platforms for the control programs. The main contribution of our research can be summarised in three folds. The first one is to evaluate the pros and cons of current tools for industrial control system design by implementing a smart Sorter System. Secondly, we demonstrate a software and hardware co-design example in SystemGALS. The last one is to present a new potential execution framework for SystemGALS programs that will allow the design of hardware/software customised controllers, including both software/runtime systems and hardware parts of the final design.

Acknowledgements

I would first like to express my special thanks to my supervisor, Professor Zoran Salcic. I have received a great deal of assistance and support from you. Your invaluable expertise and insight feedback help me sharpen my thoughts and lead me out of confusion.

I would also like to acknowledge Hamish Lonergan, my tutor in COMPSYS 704. He nicely organised the learning sources of RISC-V and generously provided them to us.

In addition, I would like to think my parents and my wife for their selfless support, especially during the Covid-19 pandemic. Their help makes it easy for me to get through those tough times.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Nowadays, the increasingly rapid changing of market requirements resulting from the development of globalisation and the prevailing of Consumerism is promoting a new revolution in manufacturing [1]-[5]. It requires not only flexible production of products [6]-[8] but also agile development of product lines that perform production processes [9]-[11]. Compared with pure software development, the development of IoT (Internet of Things) and IIoT (Industrial Internet of Things) applications involves both software and its execution platform. In this chapter, we first discuss the specific requirements in industry 4.0 [12]-[16] and then explore how to achieve the agile development of IIoT applications, especially the industrial control systems.

1.1. The Vision of Industry 4.0

With three industrial revolutions, industrial production is gradually gaining higher productivity and more robust capability to produce complicated products by introducing revolutionary technologies into production processes: the machine and steam power in the first revolution, the electromechanical machine in the second revolution, and the information and communication technology in the third revolution [17], [18]. Currently, there is a new increasing requirement for industrial production, which is neither higher productivity nor higher complexity. That is customisation [6]-[8]. The increasingly rapid change of market requirements has made manufacturers more and more aware that "they can no longer capture market share and gain higher profits by producing large volumes of a standard product for a mass market" [19]. In 2013, the German government published a report that depicts a vision of the fourth industrial revolution (industry 4.0) [13] to satisfy the product customisation in even a small amount and, in the meanwhile, maintain the cost at the mass-production level.

Industry 4.0 describes an intelligent manufacturing paradigm to combine the consumer and the manufacturer much more tightly [20]. A consumer can directly send his/her special requirements for a product to the factory, which then accordingly adjust the manufacturing process and start the manufacturing quickly. The same product line may dynamically employ hundreds of manufacturing processes to produce different customised products. This vision

encourages the manufacturing systems to comprise a group of interconnected and configurable subsystems rather than a unitary product line with fixed procedures [21]. Industry 4.0 also encourages the interconnections among various systems ranging from the custom order system to the logistic system, enabling customers to track the orders during their whole lifecycle [22], [23]. This kind of smart factory introduced in industry 4.0 is more and more like a software application, requiring rapid development, static and dynamic adaptability, flexibility, variety, and fault tolerance.

Undoubtedly, the achievement of the vision of industry 4.0 demands advancement in every part of building up the manufacturing process, especially the design of control systems. Generally, it mainly involves the design paradigm, programming languages, compilers, and execution platforms.

1.2. System-Level Programming Language

The kernel of the smart factory is the control systems, which are more intelligent, more flexible and involve more communication between internal subsystems and between the systems and the environment. With the complexity of the systems significantly increasing, the system designer should put more emphasis on system-level behaviour rather than the behaviour of usual programs, thus requiring system-level design languages [24]-[28]. Concretely, a system-level programming language should promote the modularity of control systems design, facilitate the composition of the components into larger systems, and simplify the concurrent programming design. Furthermore, it should encourage compatibility with existing programming languages due to the difficulty of bringing up all of the facilities for a new language from scratch.

SystemGALS [29], on the basis of its predecessor SystemJ [27], [28], are developed toward this goal. It was promoted Embedded Systems Research Group of the University of Auckland in 2019 and inherited from SystemJ language the Globally Asynchronous Locally Synchronous (GALS) model of computation [30]. And it also keeps the model of time, kernel statements, and the hierarchy of design units, clock domains and reactions in SystemJ. SystemJ programs have JAVA as the host language and must be executed on platforms supporting JVM. In contrast, SystemGALS can target various software programming languages, including C, as well as hardware description languages, which broadly extend the application domain and

potential execution platforms that include the combination of traditional general-purpose processors and application-specific (hardware) processors. Furthermore, SystemGALS completely separates complex data computation from control flow by introducing a new design unit, called the data module.

In this thesis, we leverage SystemGALS to design the controllers for three components in a smart Sorter System. And the controller applications involve an image recognition algorithm to detect the type of an item processed during the system runtime. This algorithm has been implemented in two versions: pure C programs and C programs with two hardware accelerators, promoting the capabilities of SystemGALS to support the specification of systems that are implemented in combinations of hardware and software.

1.3. Execution Architecture

Any control system involves programs and their execution platforms. On the one hand, apparently, with the rapidly increasing diversity of industrial applications, it is almost impossible to design a general-purpose architecture for execution platforms suitable for all applications. On the other hand, accompanied by the gradual invalidation of Moore's law, programmers cannot expect to promote programs execution efficiency by the improvement of the fabrication process of integrated circuits, which also stimulate the exploration of computer systems architecture. John Hennessy and David Patterson, the recipients of the Turing Award (2017), describe the following decades as "a new golden age for computer architecture" [31]. And, the multi-core processor architecture with dedicated accelerators implemented on a single chip tends to be a solution to cope with these problems.

For the controller applications of the Sorter System, we create a multi-core processor architecture with a RISC-V [32] processor, two accelerators and a TDMA-MIN [33] network to demonstrate a potential execution architecture for the software and hardware co-design of SystemGALS specifications.

1.4. Agile Hardware Design

The vigorous exploration of execution architectures further encourages the revolution of hardware design [34], which is inefficient and cumbersome compared to current software application designs. In the last decades, software programming languages have undergone enormous changes by continually adopting new features and new paradigms such as object-orientated programming and functional programming, which significantly facilitate code reuse and thus enable rapid and agile software development [35], [36]. In addition, the prevalence of the philosophy of open-source [37] in software projects broadly promotes the vitality of software development.

In contrast, current dominant hardware description languages (HDL) experience almost no little change from the introduction of VHDL and Verilog in1980s. SystemVerilog, the successor of Verilog, contributes more in hardware verification than in design paradigms. On the other hand, the digital systems tend to be more and more complex, involve more interaction, require a higher degree of customisation and at the same time should provide and lead towards a shorter time of design, test, and fabrication. The pure register-transfer level (RTL) design abstraction in VDHL and Verilog is more and more struggling to cope with all of these requirements.

VHDL and Verilog are powerful to describe a hardware block performing a specific function by declaring registers and using if-then-else and arithmetic statements to describe the combinational logic. However, the procedure-oriented paradigm, which focuses on the flow of signals between registers, is not beneficial for code reuse, which is critical for customisation and agile development. And, the close coupling between the front end and the back end in traditional HDL languages further harms the reuse of existing code. Designers may need to rewrite the programs of the same functions when targeting different FPGAs, ASIC toolchains and VLSI technologies.

Another important factor that hinders the progress of agile hardware design is the scarcity of open-source hardware projects, especially mature commercial projects. Unlike the software designs, the hardware design for an ASIC needs to experience the tape-out and manufacturing process before the products are shipped to the customers. These processes generally cost a large amount of money, ranging from millions to billions of dollars. Moreover, the expected results from experience or simulation experiments in aspects of correctness, performance, and energy efficiency can be very different from physical circuits for the same design. Thus, hardware

designs have a more urgent need than software development for open-source projects that have been successfully fabricated and thoroughly tested to encourage the variety of IoT and IIoT applications and shorten development cycles. However, currently no matter the amount or the degree of openness of open-source hardware projects is far less than those in software designs. For example, the instruction set architecture (ISA) that serves as an application binary interface (ABI) between software and hardware is a decisive factor in system designs. However, any implementation and modification of the currently prevailing ISA may require a large amount of money that can be only affordable for large enterprises.

Fortunately, both academic researchers and the industry have identified the problems [34], [38] and devoted considerable effort to improving the situation, including the emergence of Chisel [39] and RISC-V [32] in the 2010s. Chisel is a Scala-based hardware description language proposed by the University of California, Berkeley (UC Berkeley) in 2012, embedded in a modern software language (Scala) and used to generate synthesisable Verilog. This mechanism enables Chisel to introduce the power of modern software languages, such as object-oriented and functional programming paradigm, into the existing hardware design procedures. Furthermore, by inserting an intermediate language and a hardware compiler framework (FIRRTL) [40] between Chisel and target Verilog programs, Chisel can facilitate the design reuse for ASIC and FPGA designs. RISC-V is an open instruction set architecture, which allows developers to implement a processor and freely modify the instructions without any charge, even for commercial purposes.

Despite its short time being in public view, RISC-V has attracted broad support from academia and industry and encouraged various hardware open-source projects, including Chipyard [41], an open-source SoC development framework. In this thesis, we leverage Chipyard to build two execution platforms for SystemGALS programs and develop two hardware accelerators in Chisel to improve the performance of our programs.

1.5. Thesis Contributions and Outline

Agile hardware design also depends on manufacturing procedures. The fabless-foundry model, a mainstream model in semiconductor manufacturing, is facing the same challenges existing in current mass production. However, this is out of the scope of our research. In this thesis, we will focus on the controller designs of the Sorter System application and their execution

platform. The main contribution of our research can be summarised in three folds. The first one is to evaluate the pros and cons of current tools for industrial control system design by implementing a smart Sorter System. Secondly, we demonstrate a software and hardware co-design example in SystemGALS. The last one is to present a new potential execution framework for SystemGALS programs that will allow the design of hardware/software customised controllers, including both software/runtime system and hardware parts of the final design.

The remainder of this thesis is structured as follows. Chapter 2 gives an introduction about relevant background and tools we will use in this thesis, while SystemGALS designs and an image detection algorithm are explained in Chapter 3. Chapter 4 will illustrate the mechanism behind SystemGALS, including the time model, the execution model and the runtime support system. And, we describe the mapping from SystemGALS programs to C programs in Chapter 5. Chipyard is introduced in Chapter 6 to build an execution platform for SystemGALS that employs hardware/software partitioning and co-design. Chapter 7 will describe the Chisel design of two accelerators for the image detection algorithm and how to integrate these two accelerators into a new execution architecture. Chapter 8 demonstrates the program's execution on the simulators of these two execution platforms. Chapter 9 is the discussion and conclusion.

Chapter 2. Related Work & Background

In this thesis, we use SystemGALS to design the Sort System controller and leverage Chipyard, an open-source SoC framework, to create execution platforms for controller programs. And, Chipyard mainly uses Chisel to design SoC components and employs the RISC-V cores to create SoC variants. In addition, one of our execution platforms involves a network named TDMA-MIN. In this chapter, we briefly introduce the related work and background of these tools.

2.1. SystemJ

SystemJ [27], [28] is a system-level language that introduces formal syntax and leverages the Globally Asynchronous Locally Synchronous (GALS) formal model of computation [30], thus providing an automatic mechanism to validate, verify and debug the system design. Additionally, its incorporation of Java makes it possible to utilize complex data structures and objects to support data-driven operations.

The basic SystemJ design consists of multiple clock domains, with one clock domain corresponding to one physical component in a product manufacturing system. One clock domain can further comprise multiple concurrent reactions to control the component to perform specific behaviour routines or respond to messages from the environment (the external world) and reactions in the same or another clock domain. In addition, each reaction can have child reactions nested in itself up to arbitrary depth, promoting behavioural design hierarchy. Each clock domain is driven by its own logical clock called tick in SystemJ, and all of the reactions in the same clock domain share an identical clock. When executing the SystemJ programs, a clock domain must be waiting for all reactions in it to finish one-tick execution before advancing to the next tick. The execution boundaries are denoted by an explicit pause statement or implicit pause in a derived statement. On the other hand, ticks for different clock domains have no relationship with each other.

SystemJ leverages constructs/objects of channels and signals to enable communication between reactions, clock domains, and environment. The channels are responsible for the data exchange between the clock domains, while the signals are mainly in charge of broadcasting

information within one clock domain or for enabling communication of the clock domain with its environment (physical or artificial). The statuses and values of signals and channels can be changed during the execution in one tick but only updated at the housekeeping stage between two ticks. Thus, it promotes a so-called delayed semantics that guarantees that change of the status of the signal will become visible to potential receivers in the stable state of the clock domain.

In SystemJ, clock domains and reactions are implemented by a mix of SystemJ and JAVA statements. The types of signals and channels can be any type supported in JAVA. The SystemJ compiler will first compile the program into pure JAVA programs that are in turn compiled by the JAVA compiler into Bytecode which can be executed by the JAVA Virtual Machine (JVM).

2.2. SystemGALS

By extending JAVA, SystemJ can support high-level data abstraction and utilize the power of third-party libraries. However, this also limits the execution platform for SystemJ programs due to its demand for JVM, which is too heavy for the resource constraint embedded platforms. SystemGALS [29], a new GLAS system-level language, was proposed in 2019. SystemGALS separates the control flow from data computation in the SystemJ specification by introducing a new construct of data modules to solve this problem. And, the data modules are empowered by other software languages. At the same time, it uses typically one language as the target/host language for control flow of SystemGALS programs. SystemGALS inherits most kernel statements (Table 2-1) and derived constructs from SystemJ. As the control flow statements are designed as plain as possible, they can easily be mapped into any other programming language. A SystemGALS compiler will first compile its statements into the target language programs and let the target language compiler take charge of the rest compilation jobs. In this way, SystemGALS can target a range of host languages (e.g. C, JAVA) and extend the range of options of execution platforms. Furthermore, SystemGALS allows data modules to be implemented in hardware description languages or high-level synthesis (HLS) and thus enables the software and hardware co-design.

Statement	Description
p1;p2	p1 and p2 in sequence
pause	Consumes a logical instant of time (a tick boundary)
[input] [output] [type] signal S	Declaring a pure or valued signal
emit S [(exp)]	Emitting a signal with a possible value
while(true) p	Temporal loop
present(S){p1}else{p2}	If signal S is present do p1 else do p2
[weak] abort ([immediate] S) {p}	Preempt if S is present
[weak] suspend ([immediate] S) {p}	Suspend for 1 tick if S is present
trap (T) {p}	Software exception
exit T	Throw a software execption
p1 ‖ p2	Run p1 and p2 in lock-step parallel
[input] [output] [type] channel C	Declaring input or output channel
send C[(exp)]	Sending data over the channel
receive C	Receiving data over the channel
#C	Retrieving data from a valued signal or channel

Table 2-1. SystemJ Kernel Constructs. [33]

SystemGALS also introduces a new data type of control flow variable (CFV) and interface functions (Table 2-2) to enable the communication between the reaction control flow and data modules. Data modules reside in the reactions and accept signals, channels, and CFVs as parameters. The output of data modules can be stored in typed signals or influence the control flow via CFVs. SystemGALS provides interface functions for host languages to acquire or change the values and statuses of signals and CFVs. Unlike signals and channels, the value of CFV that is either true or false will be changed immediately during the current tick. The control flow can only use the if-else statement to check the value of CFVs and then choose the program branch.

DataModule func(arguments){+ (stmts) +}	Declaration of a data module
dmcall func(arguments)	Call of a data module within a reaction
sgl_GetSigVal	Function that extracts the signal value for the data computation
sgl_SetSigVal	Function that assigns the signal value from the data computation
sgl_GetSigStatus	Function that returns the signal status
sgl_GetChanVal	Function that extracts the channel value for the data computation
sgl_SetChanVal	Function that assigns the channel value from the data computation
sgl_SetCFV	Function that sets the value of CFV to 0, 1 (false, true)
sgl_GetCFV	Function that extracts CFV status

Table 2-2. Data Modules and Interface Functions. [29]

Generally, a typical SystemGALS system can be composed of one or more clock domains, as shown in Figure 2-1. Each CD contains multiple concurrent reactions that can have child reactions nested inside. The reactions employ data modules to perform computation and use CFV as interfaces to branch program execution. Channels and signals are responsible for communication between reactions and clock domains. In addition, signals can enable information exchange between the system and the environment that includes, for example, sensors and devices outside the designed SystemGALS system. Chapter 3 and Chapter 4 will explain the statements and mechanism behind SystemGALS programs in detail.

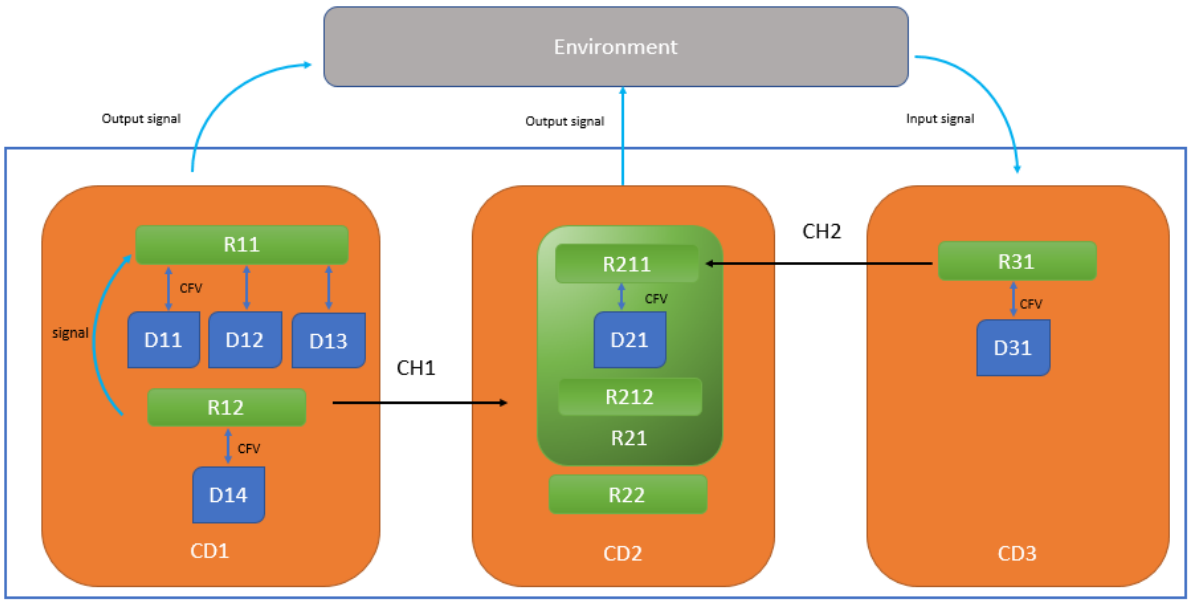


Figure 2-1. An illustration of an example SystemGALS system. [29]

2.3. RISC-V

RISC-V [32] is an open-source and royalty-free instruction set architecture (ISA). It is based on the reduced instruction set computer (RISC) principle developed from the observation that most programs did not use the vast majority of processor instructions. In 1980, David Patterson started the project of Berkeley RISC at the University of California, Berkeley (UC Berkeley). This project with the MIPS project at Stanford University is later hailed as two seminal research projects to develop the RISC concept. RISC-I [42] was the first implementation in the Berkeley RISC project with a technique of register widowing and a smaller set of 32-bit instructions compared with a complex instruction set computer (CISC) architecture. Then RISC-II [43], the second attempt in the project, introduced 16-bit instructions to improve code density. After Berkeley RISC, the SOAR [44] and SPUR [45]projects in 1984 and 1988 were referred to as RISC-III and RISC-IV, respectively. In 2010, motivated by the need for an ISA in parallel computing program and the long negotiation time and high cost to get a license for current successful ISAs, Krste Asanović of the UC Berkeley, together with David Patterson, led a three-month project to develop a new open-source instruction set, currently known as RISC-V. Now, RISC-V already has many contributors outside of UC Berkeley and is under the management of a Swiss non-profit business association, RISC-V International.

RISC-V allows anyone to freely use and customise the ISA [38], thus facilitating the customisation of the execution platform for specific applications. The ISA customisation can be achieved in two aspects. The first one benefits from the modular design of RISC-V. Generally, the integer instruction set (I) serves as the base part. The optional extensions include multiply and divide (M), atomic operation (A), single-precision floating point (F), double-precision floating point (D) and many other extensions. All of these instructions have 32-bit, 64-bit and 128-bit address space variants. This base-plus-extension approach allows designers to choose the proper combination of the base set and extension parts for their design goals. In addition, The RISC-V ISA sticks to the RISC philosophy and puts a significant emphasis on minimising instruction count. The base instruction set contains only 47 instructions. This can save bits in the instruction format and operation codes and give more space to the custom instructions, which may significantly improve the execution performance.

Along with the development of RISC-V architecture, UC Berkeley also implemented a 64-bit in-order scalar processor core (Rocket) [46] and a 64-bit out-of-order core (BOOM) [47]. Furthermore, they introduced the Rocket Chip Generator to serve as a shared code base to generate different SoC variants agilely. And all of these were implemented in the Chisel language proposed by UC Berkeley in 2010 [39].

2.4. Chisel/FIRRTL

Chisel [39] is a hardware design language embedded in the JVM language Scala that has both object-oriented and functional programming features. Chisel serves as a library in Scala, thus can leverage all of the advanced software programming features provided by Scala. The most important concept introduced by Chisel is a generator, which is also the most significant advantage compared to the traditional HDLs (e.g. VHDL, Verilog). Generally, the generator in Chisel is a parameterizable class that can produce a collection of digital circuit instances by manipulating the input parameters. And, based on Scala, generators in Chisel can provide much more flexible parameters than Verilog, thus significantly extending the boundary of instances that one generator can represent. For example, when creating a generator of the finite impulse response (FIR) filters, Chisel allows programmers to pass a window function to the generator as an argument. So the programmer can create a new FIR filter by just defining another window

function and retain the same FIR generator. Furthermore, Chisel provides a range of parameterizable components, such as arbiters and the FIFO queues in its standard library.

Chisel utilizes the meta-programming technique, which means Chisel programs are used to generate programs written in other HDLs. Currently, Chisel only supports Verilog as its target language. Instead of directly generating a Verilog program, a Chisel program will be elaborated into the Flexible Intermediate Representation for RTL (FIRRTL) [40]. FIRRTL has its own syntax to describe the circuit and still contains high-level constructs such as vector types, partial connects, and modules, which Verilog does not support. A FIRRTL compiler will run a chain of lowering transformations written in Scala to gradually translate these high-level constructs into the lowered FIRRTL forms that only contains low-level constructs that then are mapped to equivalent Verilog programs. Programmers can also rewrite these transformations to generate expected Verilog programs for different FPGAs or ASIC technologies.

2.5. Chipyard

Chipyard [41] is an agile and open-source framework based on RISC-V and Chisel ecosystem, allowing programmers and researchers to develop and test their own systems on a chip (SoC) in a singular location. It contains various open-source generators of hardware design to generate SoC components, a convenient parameter system to connect all components, and open-source and commercial simulators to test digital designs. It also incorporates software toolchains to compile software programs into RISC-V binaries and other tools like FireSim [48] and Hammer [49] to facilitate FPGA prototyping and the VLSI flow.

2.5.1 **Components**

The Rocket chip generator [46] in Chipyard serves as a foundation for developing a RISC-V SoC. As shown in Figure 2-2, it contains Rocket cores as default, buses implemented in TileLink [50] protocol, TileLink to AXI converters, and other standard peripherals such as the BootROM, the PLIC (Platform-Level Interrupt Controller), the CLINT (the Core Local Interrupts), and the Debug Unit. A Rocket core is a RISC-V in-order core developed in Chisel. It supports floating-point computation by including a floating-point unit and enables branch prediction and virtual memory by containing a memory management unit (MMU). And a Rocket tile wraps up a Rocket core with L1 caches and a page-table walker (PTW) to

communicate with other components in the SoC via the TileBus. Both the Rocket core and the Rocket tile exit in Chipyard in the form of generators rather than instances, which means one can customise a Rocket core or a tile by changing parameters in the generators, such as the cache size, bit width of the core and even the supporting instruction set. Actually, all of the components developed in Chisel in Chipyard are generators, thus significantly facilitating the design space exploration. Chipyard also contains an out-order RISC-V core (BOOM) [47] and another RISC-V in-order core (CVA6, formerly named Ariane) [51] written in SystemVerilog, which can be alternative cores in the SoC generator or coexist with Rocket cores to create a heterogeneous SoC.

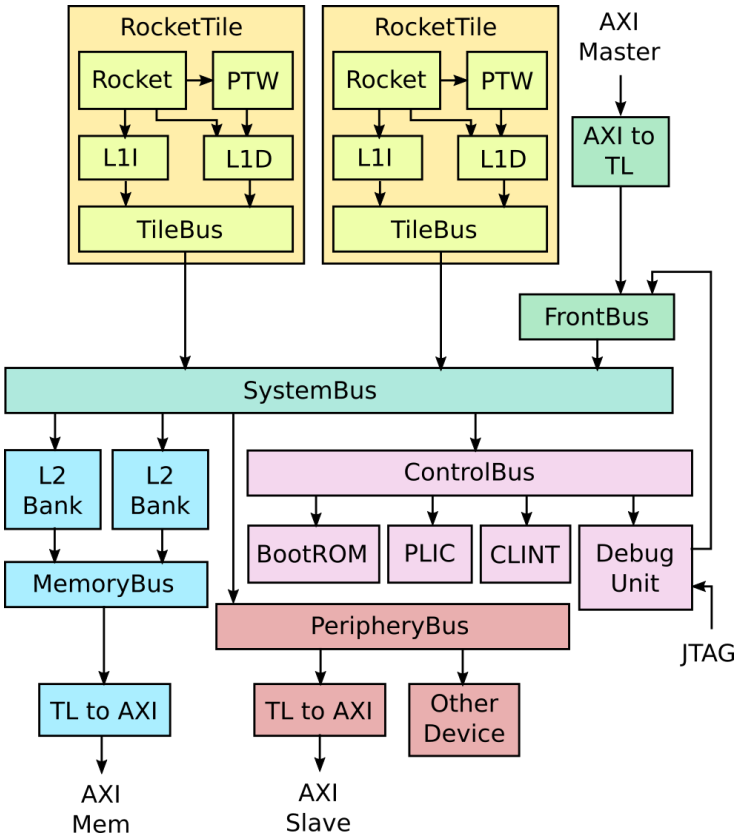


Figure 2-2. Rocket chip generator components. [60]

In addition to the processor cores, Chipyard also provides two main methods to incorporate the accelerators into the SoC: RoCC accelerators and MMIO accelerators. The former ones reside in the tile with a processor core (Figure 2-3) and use custom ISA instructions and the Rocket custom coprocessor interface (RoCC) to communicate with the processor core. One processor which supports RoCC accelerators like Rocket cores and BOOM cores can have up to four

RoCC accelerators in one tile, while there is no such limitation for MMIO accelerators. MMIO (Memory-mapped I/O) accelerators are connected to the peripheral bus and controlled by the processor via resisters in the same address space with main memory. Hwacha [52], Gemmini [53], and SHA3 are three RoCC accelerators provided by Chipyard. Hwacha is a vector architecture coprocessor, Gemmini accelerates the matrix-multiply operations, and SHA3 implements the SHA3 hash function. Chipyard also contains a collection of custom software toolchains for Hwacha to demonstrate how to add custom instruction to Spike (the instruction-level simulator) and the GNU bintuils. Besides processor cores and accelerators, Chipyard incorporates many other system components that can be integrated into the SoC, such as a network interface controller, peripheral components (e.g. UART, SPI, JTAG, I2C, PWM), and the utilities used for chip testing.

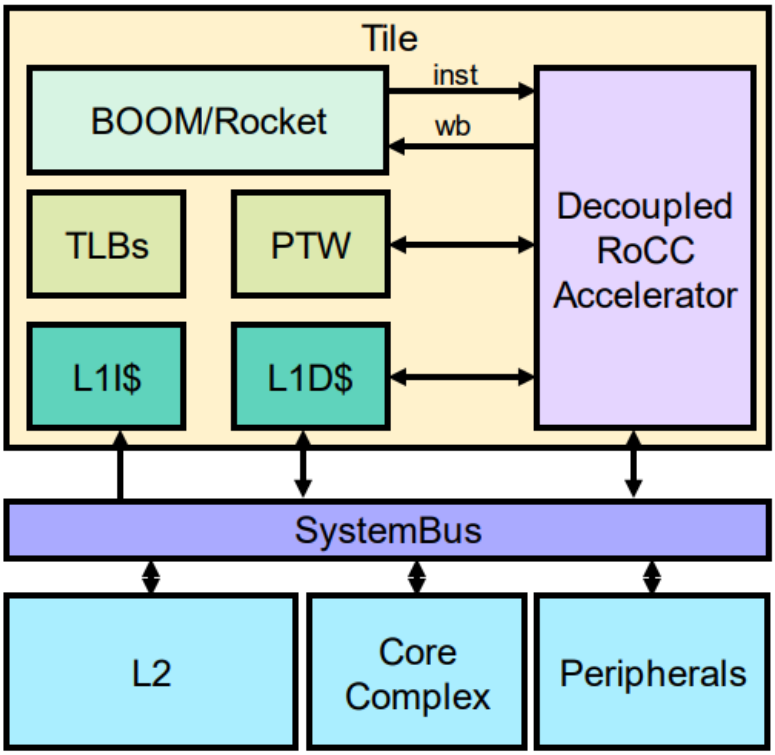


Figure 2-3. RoCC accelerator connection. [60]

2.5.2 **Parameter System**

Chipyard provides a parameter system to facilitate the connections between different components to create a custom SoC. There are three most important problems to solve when using generators to compose a whole system agilely. The first one is that how to configure all of the generators in one location. The parameterised generators of digital designs need to be

appropriately configured, and they tend to be stored in different files. It will be tedious and error-prone to tune the parameters by traversing all of these files. Furthermore, one generator may reference the parameters from another generator when instantiated. For example, the bit width of the components needs to be compatible with the operand width of the processor. The third problem involves the validation of the parameters of generators in the SoC interconnection. That is how to check the compatibility from the whole system's perspective after specifying parameters for all generators.

In Chipyard, the configuration of generators can be accomplished in one configure class, which consists of multiple additive configure fragments with the symbol of '++' between each other. Each configure fragment corresponds to one component and is in charge of specifying parameters for this component. Generally, a fragment is a partial function that maps a component key to a case class. The component key is typically the Option Type in Scala, and the value can be a container that contains the case class in the mapping or the None object when the mapping does not exist. On the other hand, a case class is defined alongside the component generator and gives default values for the generator parameters. And the config fragment can modify all or part of the default values of parameters by passing new parameters to the case class.

In addition, a component generator will be instantiated in a corresponding trait, which also defines how this component will be connected to the SoC. Traits in Scala are similar to interfaces in JAVA, and objects and classes can extend multiple traits simultaneously. Chipyard uses the Rocket chip generator as a basis to extend all of the component traits to create a custom SoC. When instantiating a generator, the trait will make a query using the component key to acquire the concrete values for parameters from the configure class. If the configure class does not contain a specific configure fragment for one component, the query result will be a None object, and the component will not be hooked up to the SoC.

Chipyard also leverages a Context-Dependent-Environment (CDE) parameterisation system to enable the key query. When querying a key, the CDE system provides three "Views" of a configure class: site, here, and up, which designate the query scopes. The site view is a global view of the configure class. Generally, a trait will use a p(somekey) function to make a key query, which means the trait will traverse all of configure fragments in a configure class from the first one (the one closest to the configure class declaration) to the last one until it finds a partial function which maps the key to a parameter case class. In this case, the configure class

is regarded as the global view (site) of this query. The p is a variable of the parameter type that is a subclass of an abstract view class, and it will be bundled together with this concrete configure class when generating the SoC design. In addition, a configure fragment can also map a key to another key query to reference the parameters of the components. For example, in a configure fragment class, a site (somekey) expression will look up the value of somekey in the whole configure class, a here (somekey, site) will just locally search in this fragment class, while an up (somekey, site) will find the value in the fragments behind this fragment in the configure class. So the final result of a key query can be influenced by the order of the fragments in a configure class.

2.5.3 **Diplomacy**

Once all generators' parameters are specified, it is crucial to validate the compatibility between different components. Chipyard introduces a Diplomacy library [49] to extend Chisel for facilitating parameters negotiation and validation. There are two most essential features in Diplomacy. The first one is to enable the two-phase hardware elaboration in Chipyard. In Chisel, the phase of running the Chisel programs to generate FIRRTL representation of a circuit is called elaboration. By declaring the concrete circuit designs of the Chisel code as lazy (a keyword in Scala) modules, Chipyard delays the elaboration of the SoC and leverages Diplomacy to create a graph of edges and nodes demonstrating the interconnection of the SoC first. In the TileLink bus protocol, almost every component is wrapped into a tile, and each tile can contain one or more nodes. In contrast, the edges represent the connections between these nodes. Then, Diplomacy will traverse these nodes and edges to check the mutual compatibility of the system. The other significant feature of Diplomacy is that instead of just validating the parameters, Diplomacy allows the negotiation of the parameters between nodes, which means nodes can specify their parameters based on knowledge of other nodes. Then at the elaboration phase, the generators can leverage these parameters to create concrete hardware circuits.

2.6. TDMA-MIN NoC

TDMA-MIN NoC [33] is a Network-on-Chip (NoC) that integrates the Time Division Multiple Access (TMDA) model with Multistage Interconnect Network (MIN) to provide a time-

predictable and scalable architecture for commutation of multiple cores in one chip. It was first introduced in 2016 [54], [55] and later expanded in 2017 [33] to create a general execution platform for SystemJ programs. It typically consists of multiple 2×2 crossbar switches arranged in k stages and interconnected with each other in a Banyan type. The value of k equals the value of k in the expression of $\boldsymbol{N = 2^k}$, where N represents the total ports in a TDMA-MIN NoC and must be greater than or equal to the number of cores in the chip. Each port is generally connected to a Network Interface (NI) with one input and one output, and a NI is, in turn, ported to one core. Concretely, each crossbar switch has two inputs, two outputs and two states inside: parallel and cross, which control the connection between the inputs and outputs. And the states of all of the switches can be driven by the TMDA slot counter, whose value will cyclically change from 0 to $(\boldsymbol{2^k - 1})$. Thus a full TDMA round has $\boldsymbol{2^k}$ slots with the duration of one slot generally equalling one clock cycle. A straightforward mechanism to drive the switches is to use each bit in the binary code of the slot counter to control the switches in the same stage. The TDMA slot counter also serves as an input for the network interfaces to decide when to send the packets to the NoC.

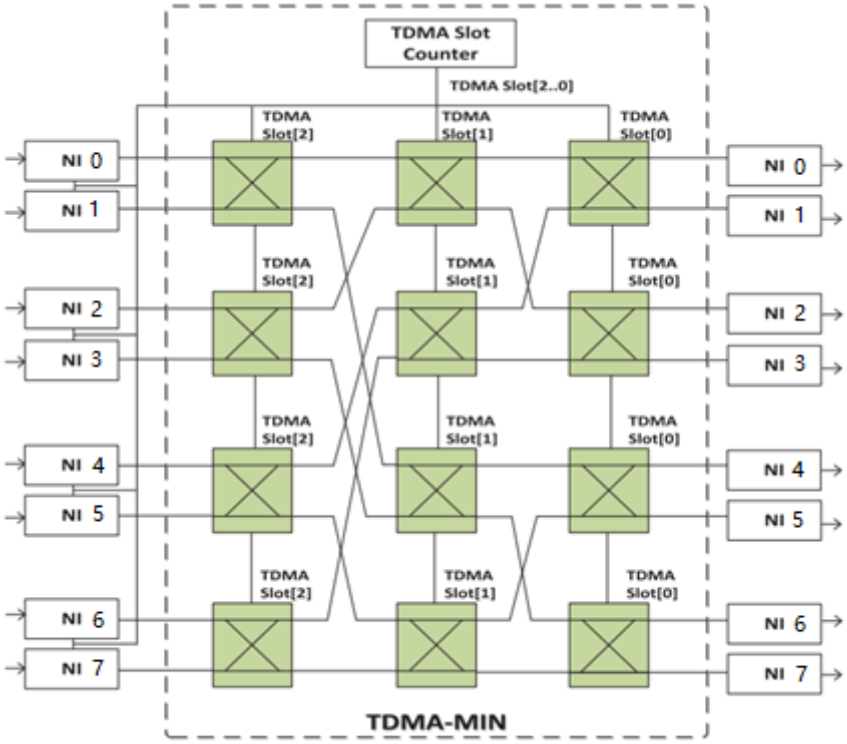


Figure 2-4. 8-port TDMA-MIN NoC with Network Interfaces. [33]

For example, a typical eight ports TDMA-MIN is shown in Figure 2-4. The NoC has three stages of crossbar switches and is driven by a 3-bit slot counter with each bit corresponding to each stage. If 0 represents the state of parallel and 1stands for the state of cross, when the TDMA slot counter is 000 in binary code, states of three stages of the switches will be parallel, parallel and parallel, respectively. Thus, the packets from NI 0 will be sent to NI 0 in this slot. In the next clock cycle, the counter will be 001, changing the state of switches in the first stage (the rightmost stage) from parallel to cross. Then NI 0 will send the packets to NI 1.

Similarly, each of the other seven input ports in the NoC other network interfaces has a different corresponding output port in one TDMA slot. The mapping from input ports to output ports has a relationship in the binary format that can be expressed as in Equation (1), where I is the input port number, D is the destination port number, $\oplus$ denotes the bitwise XOR operation, and S is the current slot value represented by its binary value. Mirror(I) represents the binary mirrored value of I.

$$D = \text{Mirror}(I) \oplus S \qquad \text{Equation (1) [33]}$$

In the 8-port TDMA-MIN NoC, the exact correspondence is shown in Table 2-3. Apparently, each network interface can infer the destination port according to the current slot value. A possible design of the network interface is shown in Figure 2-5.

Input port	Mirror	Destination port in TDMA slots							
		slot 0	slot 1	slot 2	slot 3	slot 4	slot 5	slot 6	slot 7
0	0	0	1	2	3	4	5	6	7
1	4	4	5	6	7	0	1	2	3
2	2	2	3	0	1	6	7	4	5
3	6	6	7	4	5	2	3	0	1
4	1	1	0	3	2	5	4	7	6
5	5	5	4	7	4	1	0	3	2
6	3	3	2	1	0	7	6	5	4
7	7	7	6	5	4	3	2	1	0

Table 2-3. Simultaneous Connections in TDMA Slots. [33]

In this thesis, we implement a 4-port TDMA-NoC in Chisel and use it to connect one Rocket core and two accelerators. Chapter 7 will describe it in detail.

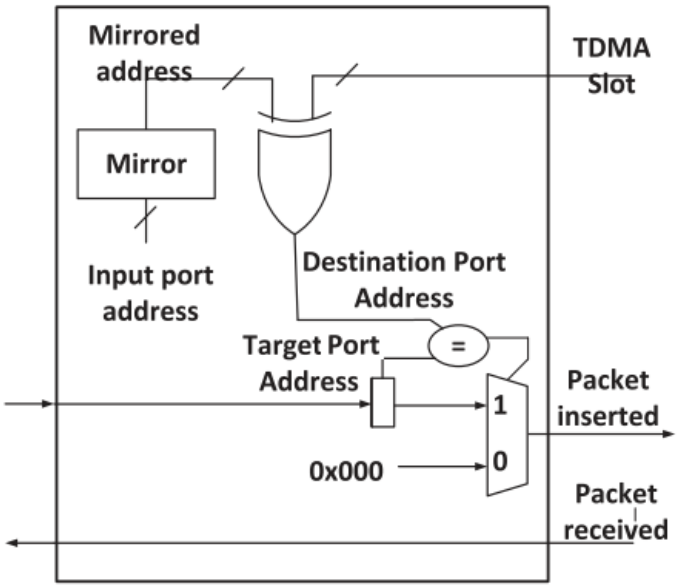


Figure 2-5. Design of Network Interface. [33]

Chapter 3. Motivating Example

In this chapter, we first introduce our motivating example, the Sorter system and describe its functionalities. Then, according to the functionalities, we demonstrate the SystemGALS designs for three controllers within the Sorter System. In addition, one of the controllers also involves an image detection algorithm. Hence, this chapter also illustrates the algorithm and its C implementation. In this thesis, we also implement some functions of this algorithm in hardware, which will be introduced in Chapter 7.

3.1. Sorter System

The primary purpose of this Sorter System [56] is to take items at the input, places items on a conveyor, identify the types of the items while they are transferred towards the output and put them into different bins according to their classes at the output. To simplify the application, we only define two types of objects: cylinders and cones. As shown in Figure 3-1, there are mainly three mechatronic devices in this System: Loader, Conveyor, and Mechanical Arm. The Loader is in charge of grabbing the items one by one and putting them on the input side of the Conveyor. Then Conveyor will take the item to the designated area, where it will be picked up by the Mechanical Arm. While the item is moving, a camera will take a picture of the item, which the Conveyor controller will process to detect the type of the item before the item arrives at the end (output side) of the Conveyor and is lifted by the Mechanical Arm.

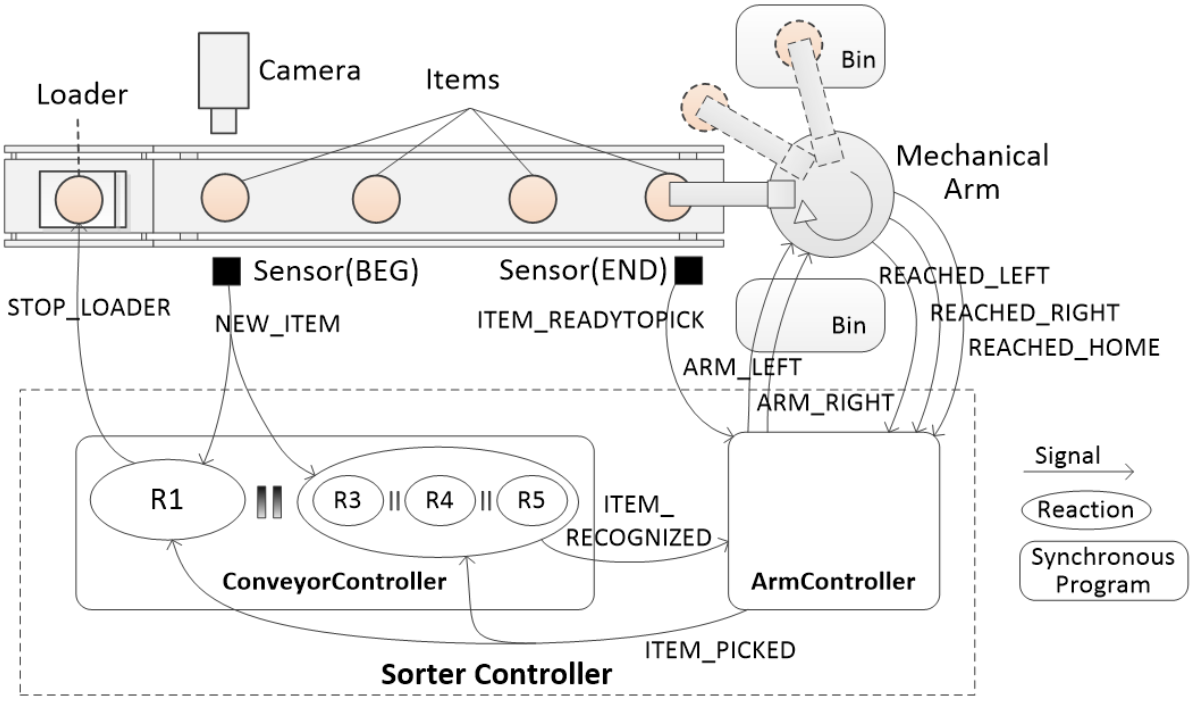


Figure 3-2. Sorter System. [56]

3.2. Model of The System

This Sorter System includes three mechatronic devices and their corresponding controllers. The controllers will issue signals to the devices to perform specific actions and exchange information with each other. And, the devices need to give feedback to the controllers when they accomplish their tasks. For the Loader, it needs to perform four actions: move to a bin that contains the items, grab one item, move its loading arm to the Conveyor, and release the item on the Conveyor. So, we design four pairs of signals between the Loader controller and Loader device, as shown in Figure 3-2. The Mechanical Arm also needs to finish four behaviours: move to Conveyor, grab the item on the Conveyor, move to the right bin if the item is a cylinder or the left bin if it is a cone, and release the item into the bin. So, there are also four pairs of signals between the Arm controller and the Mechanical Arm. As to Conveyor, the Conveyor controller only needs to start the device at the beginning of System operation.

Besides these three devices, the Sorter System also contains two sensors and one camera, as shown in Figure 3-1. The sensor at the beginning position is to generate a NEW_ITEM signal so that the camera can realise when to take a picture of the item. The other sensor at the end

position is to issue an ITEM_READYTOPICK signal to inform the Mechanical Arm to grab the item. In this thesis, we leverage a simulation way to emulate the work process of the Sorter System, which means to define three more SystemGALS programs for devices to generate the corresponding signals. In addition, to simplify the simulation, we integrate the functions of sensors and cameras into the controllers. Specifically, once the Loader puts the item on the Conveyor, the Loader controller is responsible for sending a NEW_ITEM signal to the Conveyor controller, which will get a BMP image from a file to emulate the function of the camera. Then, the Conveyor controller is in charge of sending an ITEM_READYTOPICK signal to the Arm controller after recognising the class of the item. The model discussed above is shown in Figure 3-2.

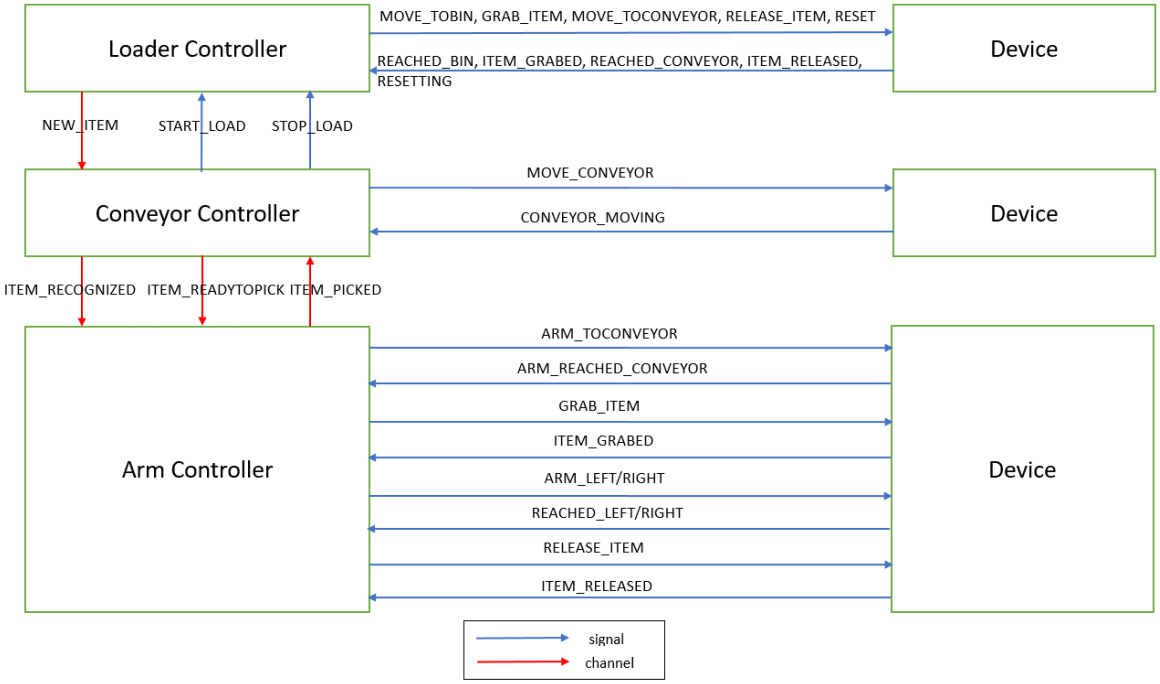


Figure 3-2. Sorter System controller model.

Controllers not only need to take the responsibility of synchronisation and communication but also undertake the computation part. In our case, the Conveyor controller should perform an image recognition algorithm to detect the type of item, which is discussed in detail in section 3.4. Another task for the Conveyor controller is to monitor the load on the Conveyor. For example, if the load reaches Conveyor capacity, the Conveyor controller will inform the Loader controller to stop loading. Once the amount of items (load) on the Conveyor is less than Conveyor capacity, loading will resume. The load will decrease when the Mechanical Arm

picks up one item, so the Arm controller should also notify the Conveyor that an item has been picked up via an ITEM_PICKED signal.

For this model, we made three assumptions in our case: The first one is that the Conveyor controller can successfully detect the type of every item. The second one is that the Conveyor controller can finish the item recognition before the item reaches the picking area. The last one is that the arm has enough time to complete the pick and release procedures. In reality, the program can add an error handler if the type detection fails, for example, by putting the unrecognised item into a particular bin. Also, the system can fulfil the last two assumptions by adjusting the Conveyor moving speed or the loading frequency. These are out of the scope of this thesis.

3.3. SystemGALS Program Design

As discussed above, the Sorter System consists of three global level components or three subsystems: Loader, Conveyor, and Mechanical Arm. Each component can be viewed as a Locally Synchronous system due to the fact that all the operations within the subsystem are driven by the common clock. In SystemGALS, a Locally Synchronous system can be declared as a clock domain (CD). So we design three clock domains for the Loader controller, Conveyor controller and Arm controller.

3.3.1 **Loader Controller**

Generally, a clock domain begins with the name of the clock domain followed by the definition of input and output signals and channels enclosed by a pair of parentheses, as shown between lines 1 to 6 in Figure 3-3. All of the signals and channels that the Loader controller use to communicate with other controllers should be defined in this pair of parentheses. Lines 4 and 5 define four pairs of input and output signals between the Loader controller and the Loader device corresponding to the four actions that the Loader device should perform: move to the bin, grab an item, move to the Conveyor and release the item. Lines 2 and 3 define the signals and channels between the Loader controller and Conveyor controller. The channel definition must designate the type of value that the channel will carry. The type can be any C type, including the struct type. The main body of the clock domain comes after the signal and channel definition with a connection symbol " ->" and is enclosed in a pair of curly braces. Within the main body is the definition of the reactions, which is also encompassed by a pair of curly braces. Multiple reactions can be combined into synchronous parallel composition "||" operator, which

will be demonstrated in the Conveyor controller. In the clock domain of the Loader controller, there is only one reaction (R1) with an infinite loop inside enclosing all of the behaviours of this reaction. It is a typical pattern in SystemGALS to define a reaction using a loop statement because a component in an industrial system tends to perform specific actions repeatedly.

```
1  LoaderController(
2        input signal START_LOADER, STOP_LOADER;
3        output int channel NEW_ITEM;
4        input signal REACHED_BIN, ITEM_GRABBED, REACHED_CONVEYOR, ITEM_RELEASED, RESETTING;
5        output signal MOVE_TOBIN, GRAB_ITEM, MOVE_TOCONVEYOR, RELEASE_ITEM, RESET;
6  )->{
7              {// R1
8                        loop{
9                                 await(START_LOADER);
10                                abort(immediate STOP_LOADER){
11                                          abort(immediate ITEM_RELEASED){
12                                                    sustain MOVE_TOBIN;
13                                          }
14                                          abort(immediate REACHED_BIN){
15                                                    sustain GRAB_ITEM;
16                                          }
17                                          abort(immediate ITEM_GRABBED){
18                                                    sustain MOVE_TOCONVEYOR;
19                                          }
20                                          abort(immediate REACHED_CONVEYOR){
21                                                    sustain RELEASE_ITEM;
22                                          }
23                                          send NEW_ITEM(1);
24                                } do {
25                                          abort(immediateITEM_RELEASED){
26                                                    sustain RESET;
27                                          }
28                                          abort(immediate RESETTING){
29                                                    sustain MOVE_TOBIN;
30                                          }
31                                          abort(immediate REACHED_BIN){
32                                                    sustain RELEASE_ITEM;
33                                          }
34                                }
35                                pause;
36                       }
37             }
38  }
```

Figure 3-3. Loader controller SystemGALS implementation.

Between lines 9 and 36 are the statements that represent the control flow of the Loader controller. The Loader needs to wait for a START_LOADER signal (line 9) to start the loading procedure and stop the processes when the STOP_LOADER signal is present, which means the number of the items on the Conveyor has reached the capacity. The abort statement (line 10) provides a mechanism to pre-empt the control flow (lines 11 to 23) if the STOP_LOADER

signal is present and performs a routine(lines 25 to 33) to reset the Loader device, preparing for the following loading action.

The main functionality is achieved by the statements between line 10 and line 23. The Loader controller sequentially emits the signals of MOVE_TOBIN, GRAB_ITEM, MOVE_TOCONVEYOR, and RELEASE_ITEM to control the device to finish the loading of one item. As the emitted signal only lasts for one tick, the loader device may fail to capture the signal. So the Loader controller needs to continuously emit the signal in a sequence of ticks until it receives the corresponding signal indicating that the device has finished the specific task. The sustain statement (line 12) is a derived statement, syntactic sugar, for programmers to achieve the continuous emission of a signal. For example, the abort statement (line 11) enclosing the sustain statement can guarantee the escaping from the infinite loop once the feedback signal reaches.

After the Loader device puts an item on the Conveyor, the Loader controller will use the send statement (line 23) to inform the Conveyor controller through the NEW_ITEM channel. Once the sending procedure is finished, the control flow will return to the beginning of the loop, waiting for the START_LOADER signal to load another item.

It is worth noting that channels in SystemGALS already have the mechanism to guarantee the information transmission. So it is more common to use channels between clock domains. In this thesis, the way of sustained signals is only used to demonstrate the interaction between controllers and device simulations.

3.3.2 **Conveyor Controller**

The Conveyor controller is the most complicated part of the system. It has three main functions: control the number of items on the Conveyor, detect the type of an item and send the type of items to the Arm controller successively. For the first functionality of limiting the load on the Conveyor, the change of the quantity of the items on the Conveyor derives from two actions: the Loader puts one item on the Conveyor, and the Mechanical Arm lifts one item. And these two actions can be abstracted as two signals: NEW_ITEM and ITEM_PICKED, respectively. As shown in Figure 3-4, the Conveyor controller defines two integer input channels (line 4) to capture these two activities. The NEW_ITEM input channel has the same name as the output channel in the Loader Controller. These two channel definitions with an identical name but opposite directions build up a complete channel between two different clock domains.

Moreover, the Conveyor controller uses two separate reactions to receive the message via the channel concurrently: R8 (lines 47 to 52) for the ITEM_PICKED channel and R9 (lines 55 to 61) for the NEW_ITEM channel. R9 also leverages an internal signal, NEW_ITEM_INTERN (line 58), to notice R1 that there is a new item on the Conveyor. In SystemGALS, signals defined in any position within a clock domain are visible to all Reactions in this clock domain. Similarly, R8 uses an internal signal, ITEM_PICKED_INTERN (line 50), to inform R1 that an item has been picked up.

Once R1 captures these two internal signals, it will change the value of the load variable accordingly, representing the current number of items on the Conveyor. The present statement (line 17) can check the status of a signal and execute the statements in the following block if the signal is present. The operation of load++ (line 20) and load-- (line 26) are C operations and are surrounded by a pair of {+ +}, which SystemGALS uses to introduce host languages in an anonymous data module. In a data module, SystemGALS can leverage any type, statement, and feature of the host language to perform computation operations. In this thesis, we use C as our host language to implement the software parts of data modules. Another data module (lines 29 to 35) in R1 checks if the load reaches the capacity. In our example, we assume that the capacity of the Conveyor is ten items (line 10). If the capacity is reached, the Data Module will access a CFV, isConveyorFull, defined in line 10 and change its value to true via the API of sgl_SetCFV() (line 31). Then the Conveyor controller will emit a STOP_LOADER signal (line 37) to inform the Loader controller to stop loading until the Mechanical Arm lifts one item from the Conveyor. Otherwise, it will emit a START_LOADER signal.

```
1  ConveyorController(
2      output signal START_LOADER, STOP_LOADER, MOVE_CONVRYOR;
3      input signal CONVEYOR_MOVING;
4      input int channel NEW_ITEM, ITEM_PICKED;
5      output int channel ITEM_RECOGNIZED;
6  )->{
7          {// R1
8           cfv isConveyorFull = false;
9           int load = 0;
10          int CAPACITY = 10;
11          signal NEW_ITEM_INTERN;
12          Signal ITEM_PICKED_INTERN;
13          abort(CONVEYOR_MOVING){
14                  sustain MOVE_CONVEYOR;
15          }
16          loop{
17                  present(NEW_ITEM_INTERN)
18                  {
19                          {+
20                                  load++;
```

```
21                                  +}
22                          }
23                          present(ITEM_PICKED_INTERN)
24                          {
25                                  {+
26                                          load--;
27                                  +}
28                          }
29                          {+
30                                  if(load >= CAPACITY){
31                                          sgl_SetCFV(isConveyorFull, true);
32                                  }else{
33                                          sgl_SetCFV(isConveyorFull, false);
34                                  }
35                          +}
36                          if(isConveyorFull){
37                                  emit STOP_LOADER;
38                          }else{
39                                  emit START_LOADER;
40                          }
41                          pause;
42                      }
43                  }
44                  ||
45                  …
46                  ||
47                  {//R8
48                        loop{
49                                  receive ITEM_PICKED;
50                                  emit ITEM_PICKED_INTERN;
51                                  pause;
52                          }
53                  }
54                  ||
55                  {// R9
56                         loop{
57                                  receive NEW_ITEM;
58                                  emit NEW_ITEM_INTERN;
59                                  pause;
60                          }
61                  }
62                  ||
63                  …
64  }
```

Figure 3-4. Conveyor controller SystemGALS implementation (1).

In the Conveyor controller, the reaction of R2 takes responsibility for detecting the type of an item by processing its picture. As shown in Figure 3-5, R2 contains four children reactions: R3, R4, R5, and R6. Each of these children reactions invokes a data module to achieve a subfunction of an image recognition algorithm sequentially. The output of the previous reaction is the input of the next reaction, and the previous reaction will also emit a signal to inform that the result is ready. Concretely, R3 will control the camera to take a picture of an item once the NEW_ITEM_INTERN signal is present. In our case, we call a function (line 11) to get a pointer pointing to an unsigned char array that stores the data of a BMP image to simulate the camera's

action. Then R4 will invoke a Binarization data module (line 17) to convert the colour image to a binary image. Next, the extractCharac data module (line 23) in R5 will extract a character vector from the binary image， which contains five float point numbers to describe features of the image. Finally, R6 will call the itemRecognize data module (line 29) to determine which type the item is and emit an integer signal with a value representing the type.

The statement of "dmcall Binarization (binary pixels, image)" (line 17) is an explicit method to call a named data module. Unlike the anonymous data module, a named data module (lines 35 to 37) can be defined outside the clock domains and with a specific name. The DataModule is a definition keyword, followed by the function name and the parameter list. In addition, R2 uses the types of Image (line 2), PixelArray (line 3), and CharacArray (line 4) defined in C struct to describe the data structure of a BMP image, a binary image, and a characteristic vector, respectively. Section 3.4 will illustrate the algorithm and the data module in detail.

```
1  {// R2
2          Image signal image;
3          PixelArray signal binaryPixels;
4          CharacArray signal characArray;
5          int type;
6          Signal int ITEM_RECOGNIZED_INTERN;
7          loop{
8
9                  await (NEW_ITEM_INTERN);
10                 {//R3
11                         dmcall getImage(image);
12                         emit image;
13                 }
14                 ||
15                 {//R4
16                         await(image);
17                         dmcall binarization(binaryPixels, image);
18                         emit binaryPixels;
19                 }
20                 ||
21                 {//R5
22                         await(binaryPixels);
23                         dmcall extractCharac(characArray, binaryPixels);
24                         emit characArray;
25                 }
26                 ||
27                 {//R6
28                         await(characArray);
29                         dmcall itemRecognize(type, characArray);
30                         emit ITEM_RECOGNIZED_INTERN(type);
31                 }
32                 pause;
33         }
34 }
35 DataModule binarization(signal binaryPixels, signal image){+
36     …
37 +}
```

Figure 3-5. Conveyor controller SystemGALS implementation (2).

Regarding the latest functionality of sending types to the Arm controller, we need to consider the uncertainty of sending and receiving procedures as different clock domains have separate clocks (mutually independent). Channels can guarantee the outcome of send and receive statements; however, they cannot predict when the result comes out. This uncertainty may result in congestion: An item type may not have been sent out when R6 emits a new type value. To solve this problem, as shown in Figure 3-6, we create a soft circular queue (lines 2 to 5) with an int array in R7 and put the value of type into the queue (line 9) when the ITEM_RECOGNIZED_INTERN signal is present. Concurrently, R10 will check whether the queue is empty (lines 19 to 25). If not, it will pop out the first element and send it out via the ITEM_RECOGNIZED channel (line 27). Subsequently, R10 will inform the Arm controller to pick up the item through the ITEM_READYTOPICK channel (line 28).

```
1  {//R7
2          int typeArray[12] = {0};
3          int head = 0;
4          int tail = 0;
5          int maxSize = 12;
6          cfv isQueueEmpty = 0;
7          loop{
8                  present(ITEM_RECOGNIZED_INTERN){
9                      dmcall enQueue(typeArray, &head, &tail, maxSize,  #ITEM_RECOGNIZED_INTERN);
10                 }
11                 pause;
12         }
13 }
14 ||
15 …
16 ||
17 {//R10
18         loop{
19                 {+
20                         if(isEmpty(head, tail)){
21                                 sgl_SetCFV(isQueueEmpty, 1);
22                         }else{
23                                 sgl_SetCFV(isQueueEmpty, 0);
24                         }
25                 +}
26                 if(!isQueueEmpty){
27                         send ITEM_RECOGNIZED(typeArray[head]);
28                         send ITEM_READYTOPICK(1);
29                         deQueue(typeArray, &head, &tail, maxSize);
30                 }
31                 pause;
32         }
33 }
```

Figure 3-6. Conveyor controller SystemGALS implementation (3).

3.3.3 **Arm Controller**

The Arm controller also performs three functions: capture the ITEM_RECOGNIZED and ITEM_READYTOPICK notification via channels, control the device to pick and put the item into a corresponding bin, and notify the Conveyor controller that an item has been removed from the Conveyor. As shown in Figure 3-7, the Arm controller only has one reaction and firstly sustain the ARM_TOCONVEYOR signal (line 14) to move the Mechanical Arm above the Conveyor. Then the controller waits for the type of the coming item (line 16) and sets up two CFVs according to the value (lines 18 to 29). Once the item reaches the designated area (line 30), the controller will sustain the GRAB_ITEM signal (line 32) to control the device to lift the item. The following two parallel blocks perform two concurrent actions: informing the Conveyor controller to decrease the number of items (lines 34 to line 36) and putting the item into a bin (lines 38 to 51). In the second block, depending on the value of two CFVs (lines 39 to 47), the controller will determine the correct bin to which the Mechanical Arm should move.

```
1  ArmController(
2      input int channel ITEM_RECOGNIZED;
3      input int channel ITEM_READYTOPICK;
4      output int channel ITEM_PICKED;
5      output signal ARM_TOCONVEYOR, GRAB_ITEM, ARMLEFT, ARMRIGHT, RELEASE_ITEM;
6      input singal ARM_REACHED_CONVEYOR, ITEM_GRABED, REACHED_LEFT, REACHED_RIGHT, ITEM_RELEASED;
7  )->{
8      {// R1
9                  int itemType = 0;
10                 cfv isTypeA = 0;
11                 cfv isTypeB = 0;
12                 loop{
13                         abort(immediate ITEM_RELEASED){
14                                 sustain ARM_TOCONVEYOR;
15                         }
16                         receive ITEM_RECOGNIZED;
17                         itemType = #ITEM_RECOGNIZED;
18                         {+
19                                 if(itemType==1){
20                                         sgl_SetCFV(isTypeA, 1);
21                                 }else{
22                                         sgl_SetCFV(isTypeA, 0);
23                                 }
24                                 if(type == 2){
25                                         sgl_SetCFV(isTypeB, 1);
26                                 }else{
27                                         sgl_SetCFV(isTypeB, 0);
28                                 }
29                         +}
30                          recieve ITEM_READYTOPICK;
31                         abort(immediate ARM_REACHED_CONVEYOR){
32                                 sustain GRAB_ITEM;
33                         }
34                         {
35                                 send ITEM_PICKED(1);
36                         }
37                         ||
```

```
38                                  {
39                                          if(isTypeA){
40                                                  abort(immediate ITEM_GRABED){
41                                                          sustain ARM_LEFT;
42                                                  }
43                                          }else if (ifTypeB){
44                                                  abort(immediate ITEM_GRABED){
45                                                          sustain ARM_RIGHT;
46                                                  }
47                                          }
48                                          abort(immediate REACHED_LEFT || REACHED_RIGHT ){
49                                                  sustain RELEASE_ITEM;
50                                          }
51                                  }
52                                  pause;
53                          }
54          }
55  }
```

Figure 3-7. Arm Controller SystemGALS implementation.

3.4. Image Recognition Algorithm and C Implementation

3.4.1 **Algorithm Overview**

In the clock domain of the Conveyor controller, we leverage a K-nearest neighbour image recognition algorithm to detect the type of an item. This algorithm consists of two parts: the K-nearest neighbour algorithm (KNN) [57], [58] and an algorithm to characterize an item image [59]. KNN is a classic classification method first developed by Evelyn Fix and Joseph Hodges in 1951 [54]. It determines the category of an under-test item by a plurality vote of its first K neighbours. Generally, neighbours are a set of items whose class membership are known and denoted by abstract features. The features tend to be a character vector such as (x0, x1, x2, x3, x4), facilitating the calculation of the distance between two items. Once the distances between the under-test item and every neighbour are calculated, we can sort the neighbours in ascending order accordingly. The majority class among the first K neighbours is the class membership of the under-test. For example, if K is five and among the first five distances, three neighbours belong to class A, and the other two neighbours belong to class B. Then the type of the under-test item is A. The method to define a feature may vary in different scenarios. In the algorithm to characterize the item image, we use the ratio of valid pixels in five areas to define the features of the item. And to simplify the implementation, we only consider the 24 bit BMP format image with a size of 160 * 160.

Concretely, the entire algorithm contains four steps: binarize the image, detect the item's edge in the picture, extract the characteristic vector, and decide the type. The first step is to binarize

the image, using 0 or 1 to represent a pixel. Generally, in a 24-bit BMP image, one pixel is represented by three bytes, indicating the weight of three colors (i.e. green, red, blue). We use a typical formula (Formula 1) to convert the three bytes of one pixel into a gray value. If the gray value is larger than a threshold, the value will be set to 0. Otherwise, it will be 1. This is contrary to displaying a binary image: 0 for black and 1 for white.

$$Gray = 0.114 * Blue + 0.587 * Green + 0.299 * Red \qquad \text{(Formula 1) [59]}$$

The next step is to detect the item edge. If we consider a 160 * 160 image as a two-dimensional matrix, this matrix will have 160 rows and 160 columns. Finding the top and bottom row and the left and right columns will help cut off the margin part in an image (Figure 3-9), thus improving the accuracy. We describe the area surrounded by the edges as a target area.

The third step is to extract the character vector, which contains five float numbers. In this step, the target area will be evenly divided into four regions. Each region consists of multiple 0 and 1. We will calculate the ratio of 1 in each small area to get four float numbers and the ratio for the whole picture to get the fifth float number.

In the last step, we apply the KNN algorithm to determine the type. Firstly, we prepare six images of cylinders and six images of cones and acquire their character vectors following the first three steps to get twelve character vectors. This procedure is called training. Then extract the character vector of the under-test item and calculate the distance with the twelve character vectors one by one. Finally, we sort these float numbers in ascending order and select the majority type in the first five numbers as the item type.

Section 3.3.2 introduces three data modules in R4, R5, and R6 of the Conveyor Controller to perform the algorithm described above. In SystemGALS, the data module can be implemented by software languages (e.g. C and JAVA) and hardware specifications. At this stage, SystemGALS mainly supports the C programming language. So we use C to implement our image recognition algorithm. The following sections will describe the C implementations in detail. Additionally, we design two hardware accelerators in Chisel, a hardware description language, to improve the execution efficiency of the last two data modules. Chapter 7 will illustrate the hardware implementations.

3.4.2 **Binarization**

The main purpose of the Binarization data module is to convert a colour BMP image to a binary image. In the experiment, we will run the programs in a bare-mental environment on platform simulators. To simplify the execution, we convert an image to an unsigned char array stored in a C file, as shown in Figure 3-8. The compiler will compile this file with programs and load it into the simulator directly, avoiding using file operation functions to retrieve data. Due to the same reason, we will fix the image size to 160 * 160 pixels to avoid using the malloc function to allocate the main memory space dynamically.

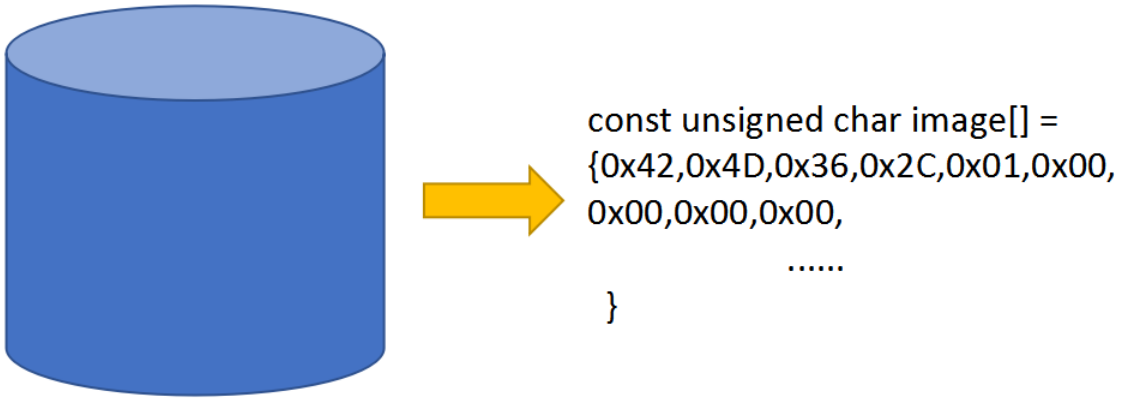


Figure 3-8. Convert the image to an unsigned char array.

A 24-bit BMP image consists of three parts: the file header, information header and data area. The file header and information header have fixed sizes, which are 14 bytes and 54 bytes, respectively. The Binarization function (Figure 3-10) defines two struct types of FileHeader and FileInfo to retrieve essential information from the unsigned char array. And, we can easily target the data area in the array by excluding the file header and information header. As we only deal with images with 160 * 160 pixels, the data array size is also fixed. However, we get the image width and height from the information header to facilitate the processing of general images in future work.

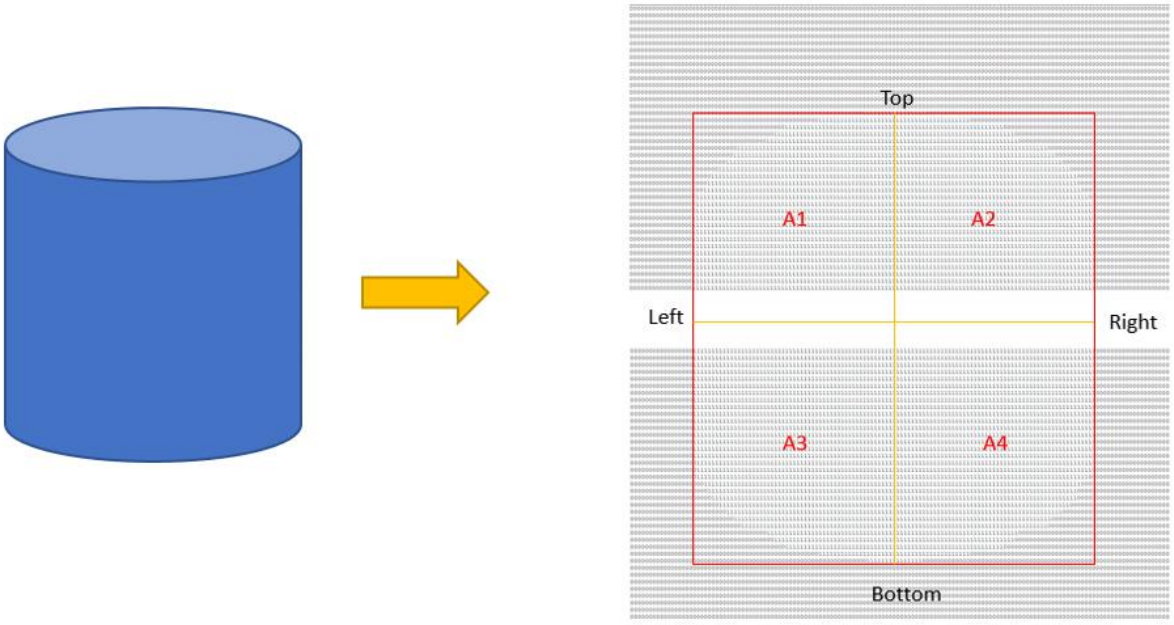


Figure 3-9. Image Binarization.

Once we find the beginning of the data area, we use the formula described in section 3.4.1 to convert the colour pixels into gray ones, reducing three bytes of one pixel into one byte (lines 32 to 33, Figure 3-10). If the gray value is larger than 190, the value will be set to 0. Otherwise, it will be 1 (lines 34 to 35). A sample result is as shown in Figure 3-9. To simplify the conversion procedures, we set the background of all of the images to white. After Binarization, all of the pixel values will be stored in another unsigned char array, each byte for one pixel. It is worth noting that a BMP image saves the pixel bytes in the Little Endian format, which means the first three bytes actually represent the latest pixel. So when storing data, we also convert the data structure into Big Endian format (line 36, Figure 3-10) to facilitate the following processes.

```
1  /*bmp header*/
2  typedef struct tagBITMAPFILEHEADER
3  {
4          unsigned char bfType[2];
5          unsigned int bfSize;
6          unsigned short bfReserved1;
7          unsigned short bfReserved2;
8          unsigned int bfOffBits;
9  }FileHeader;
10
11 typedef struct tagBITMAPINFOHEADER
```

```
12  {
13          unsigned int biSize;
14          int biWidth;
15          int biHeight;
16          unsigned short biPlanes;
17          unsigned short biBitCount;
18          unsigned int biCompression;
19          unsigned int biSizeImage;
20          int biXPixPerMeter;
21          int biYPixPerMeter;
22          unsigned int biClrUsed;
23          unsigned int biClrImportant;
24  }FileInfo;
25
26  unsigned char* binarization(unsigned char* image){
27          ...
28          {
29                  ...
30                  for (j = 0; j < fi->biWidth; j++)
31                  {
32                          gray = (int)(0.114 * (float)tempLine[k] + 0.587 * (float)tempLine[k + 1]
33                                  + 0.299 * (float)tempLine[k + 2]);//k for Blue, k+1 for Green, k+2 for Red
34                          if (190 <= (int)gray) gray = 0;
35                                  else gray = 1;
36                          dst[(imgHeight - i - 1) * imgWidth+j] = gray; // change order from litte endian to big
37  endian
38                          k += 3;
39
40                  }
41
42                  k = 0;
43          }
44          return dst;
45  }
```

Figure 3-10. Binarization C implementation.

3.4.3 **Characteristic Vector Extraction**

The extractCharac date module in R5 consists of the edge detection function and the feature extraction function. For edge detection, we have three assumptions:1. The item in the picture is continued; 2. There is at least a one-pixel distance between the item and the picture edges; 3. This is no noise in the background in the image. As shown in Figure 3-11, we build two projections for the binary image to detect the edge: vertical projection (line 9) and horizontal projection (line 10). These two projections are one dimension arrays containing 160 integers, each integer indicating a row projection or a column projection. A row projection is the sum of all pixel values in one row, while a column projection is the sum of all pixel values in one column. So, the vertical projection is an array of 160 rows projections, and the horizontal projection is an array of 160 columns projections. The item edges will reside in the transition from 0 to non-zero or from non-zero to 0 in these two arrays.

For example, the detectEdge function finds the left and right columns by traversing the vertical projection array (lines 23 to 29). We use “k” to denote the k value in the vertical projection. If k = 0 and k + 1 > 0, the k + 1 column will be the item left edge in the image. And if k > 0 and k + 1= 0, k will be its right edge. Similarly, we can get the top and bottom edges by traversing the horizontal projection array. The output of the edge function is a structure (lines 1 to 6) that contains the number of the top row, bottom row, left column, and right column.

```
1  typedef struct {
2          int top;
3          int bottom;
4          int left;
5          int right;
6  } Edge;
7
8  Edge* detectEdge(unsigned char* binaryImage, int imgWidth, int imgHeight){
9          int vocal_hist[imgWidth];       //vertical projection
10         int horizal_hist[imgHeight];   //horizontal projection
11         ...
12         for (i = 0; i < imgHeight; i++)
13         {
14                 for (j = 0; j < imgWidth; j++)
15                 {
16                         if (binaryImage[i*imgWidth + j] == 1)
17                         {
18                                 vocal_hist[j]++;
19                         }
20                 }
21         }
22         //find left
23         for (k = 0; k < imgWidth-1; k++)
24         {
25                 if (vocal_hist[k] == 0 && vocal_hist[k+1] > 0)
26                 {
27                         edge.left = k + 1;
28                 }
29         }
30         //find right
31         for (k = 0; k < imgWidth - 1; k++)
32         {
33                 if (vocal_hist[k] > 0 && vocal_hist[k + 1] == 0)
34                 {
35                         edge.right = k;
36                 }
37         }
38         ...
39     return &edge;
40 }
```

Figure 3-11. Edge detection C implementation.

In terms of extracting features, we first evenly divide the target area into four regions with the middle row and middle column derived from the middle numbers between top and bottom and between left and right, respectively. Then the function traverses each region to add up all pixel

values and then divides the sum by the number of pixels in this region to get the first four float numbers in the characteristic vector. The last number is the ratio of 1 in the whole target area.

3.4.4 **Item Recognition**

Regarding the item recognition data module in R6, we first define two two-dimensional float arrays to store the neighbours' character vectors acquired in the training stage. The features1 array represents the cylinders, while the features2 array is for the cones. Generally, we use Formula 2 to calculate the distance between two vectors：

$$\text{Distance} = \sqrt{\sum_0^4 \ （x_i - y_i）^2} \qquad \text{(Formula 2)}$$

However, as we only consider the order of the distance rather than the absolute value, the sqrt operation is ignored (line 11 Figure 3-12). Then, we use the bubble sort (line 23) to sort the distances. As we also need to get the corresponding type behind the distance, a struct type (lines 1 to 4) is defined to combine the type value and the distance. After sorting the distances, it will be easy to get the majority type in the first five elements, which is the type of the under-test item.

```
1  typedef struct{
2     int type;
3     float distance;
4  }CharcDistance;
5
6  CharcDistance* caculateCharcDistance(float* charcVector, float (*features1)[5], float (*features2)[5]){
7     static CharcDistance charcDistanceArray[12];
8     for(i = 0; i < 6; i++){
9        for(j=0; j< 5; j++){
10          charcDistanceArray[i+j].type =  1;
11          charcDistanceArray[i+j].distance = ((charcVector[0] - features1[i][0]) * (charcVector[0] - features1[i][0]) +
12                                    (charcVector[1] - features1[i][1]) * (charcVector[1] - features1[i][1]) +
13                                    (charcVector[2] - features1[i][2]) * (charcVector[2] - features1[i][2]) +
14                                    (charcVector[3] - features1[i][3]) * (charcVector[3] - features1[i][3]) +
15                                    (charcVector[4] - features1[i][4]) * (charcVector[4] - features1[i][4]));
16       }
17    }
18    ...
19    return charcDistanceArray;
20
21 }
22
23 void bubble_sort(CharcDistance arr[], int len) {
24             ...
25 }
26
27 int countMajorityType(CharcDistance charcDistanceArray[], int k){
28             ...
29 }
```

Figure 3-12. Item recognition C implementation.

Chapter 4. SystemGALS Execution

Currently, SystemGALS is at the developing stage, and the compiler is not fully functional. So in this research, we manually map the SystemGALS programs into C programs. To achieve this, we must understand the mechanism that supports SystemGALS, which is similar to SystemJ's compilation and execution [28]. This chapter will introduce the essential concepts that support SystemGALS programs execution, including the model of time, the model of execution, communication, and the runtime support system for SystemGALS.

4.1. Model of Time

SystemGALS, as indicated in its name, follow the globally asynchronous (GA) and locally synchronous(LS) paradigm. GA is demonstrated on the clock domain level. Each clock domain is driven by its own logical time called ticks in Systems GALS. So generally, clock domains are mutually asynchronous and operate concurrently except the suspending when two reactions from different clock domains are waiting for the information exchange through a channel. Within one reaction, the statements are executed in successive ticks, and each reaction must be executed for one tick to complete the whole clock domain tick, which reflects locally synchronous (LS).

Ticks are a sequence of discrete instants with different duration. That means the length of a tick does not have to be the same, i.e. ticks are logical clocks. All concurrent reactions within a clock domain need to consume a tick to advance to the next tick; thus, the execution time of one clock domain tick depends on the execution time of concurrent reactions within the clock domain. Furthermore, even for the same clock domain and the same tick, the execution time may vary on the different execution architecture and under various schedule schemes (explained in section 4.2).

During each clock domain tick, every reaction has a chance to execute its statements, and the pause statements in a reaction are used to define the execution boundary. One reaction will keep executing its statements in one tick until it encounters a pause statement. And in the next tick, it continues the execution from this pause statement to the next one. It is worth noting

that a pause statement may hide in a derived statement, for example, the await and sustain statements.

Furthermore, a reaction consists of the control flow part and computation part. In SystemGALS, data modules take charge of all the computation, even the simple add and compare operations, and a data module must be executed in a single tick. Generally, control flow tends to be plain and straightforward, while the complexity of the data module can vary in an extensive range. Thus, the physical execution time of a reaction in one tick mainly relies on its data modules' complexity.

4.2. Model of Execution

The schedule of SystemGALS programs also influences execution efficiency. When running a SystemGALS program on execution platforms, different architectures may result in different execution efficiency. For example, on single-core processor architectures, all clock domains and reactions must be executed one by one on the same core. In contrast, a multi-core processor platform can run different clock domains concurrently. Moreover, even different reactions in one clock domain can be mapped to different cores, thus dramatically improving execution efficiency. Apparently, for a specific architecture, the scheme to map clock domains or reactions to execution resources can also influence efficiency. This mapping arrangement is called schedule in SystemGALS. Thus, the schedule, along with architectures together, determines the execution efficiency.

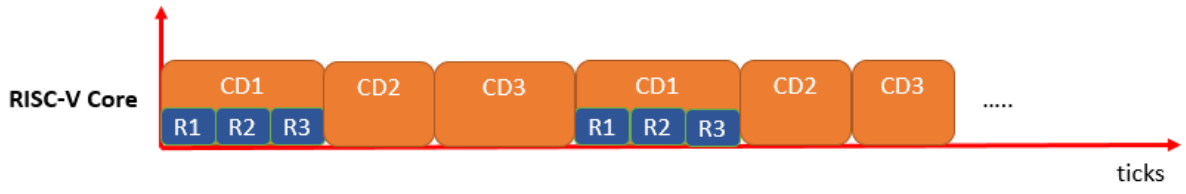
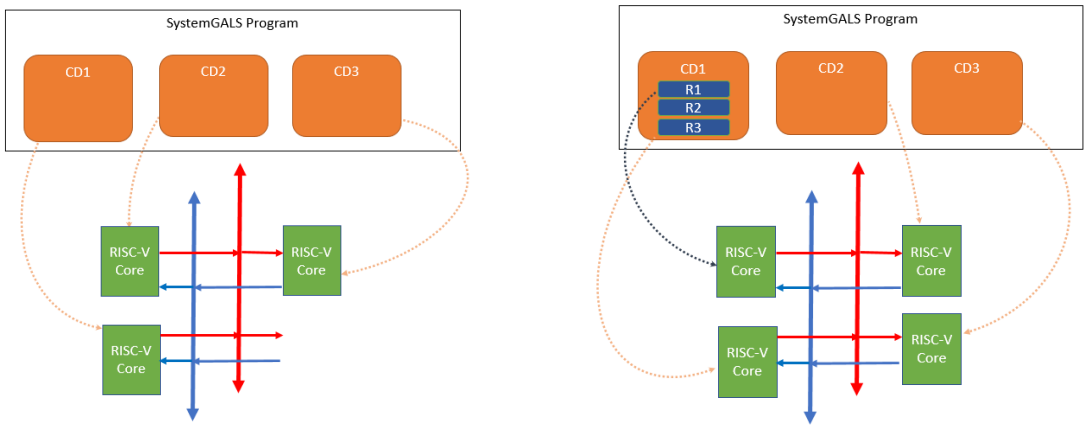


Figure 4-1. Schedule on one RISC-V core.

In this thesis, we perform the experiment on the open-source RISC-V platforms. Figure 4-1 demonstrates the schedule of a SystemGALS system on a single-core architecture. The core executes these clock domains one by one. And the three reactions in the first clock domain are also executed sequentially within the tick. The execution time of one tick in this program is the

sum of the execution time of all of the clock domains. In a three-core architecture, each clock domain can be mapped to an exclusive core (Figure 4-2 a), and the execution time will be the maximum time among the three clock domain. If adding one more core, we can separate the R1 execution from the first clock domain and map it to the additional core (Figure 4-2 b), thus further reducing the execution time.



(a) **(b)**

Figure 4-2. Map clock domains and reactions to multiple cores.

The schedule not only stays on the reaction level but also can reach inside a reaction. SystemGALS divides a reaction into control nodes and data nodes. The control nodes denote the control flow parts in a reaction, while data nodes represent the data modules. These nodes in a reaction can also be mapped to different cores. By predicting or measuring the execution cycles on the processor for every node, we can find suitable schedule schemes for different scenarios.

Furthermore, since data nodes tend to be much more complicated than control nodes, we can introduce two types of cores: control cores and data cores for control nodes and data nodes, respectively. The data cores can also be general processors or accelerators to perform certain tasks, significantly improving the execution efficiency of specific data nodes and breaking the whole system's bottleneck. And the data modules can leverage defined interface functions to call the accelerators (application-specific processor, ASP). This is one way that SystemGALS uses to enable software and hardware co-design, as shown in Figure 4-3. In this thesis, we will run the Sorter System programs on two different platforms: a single RISC-V core platform and a platform with one RISC-V core with two accelerators.

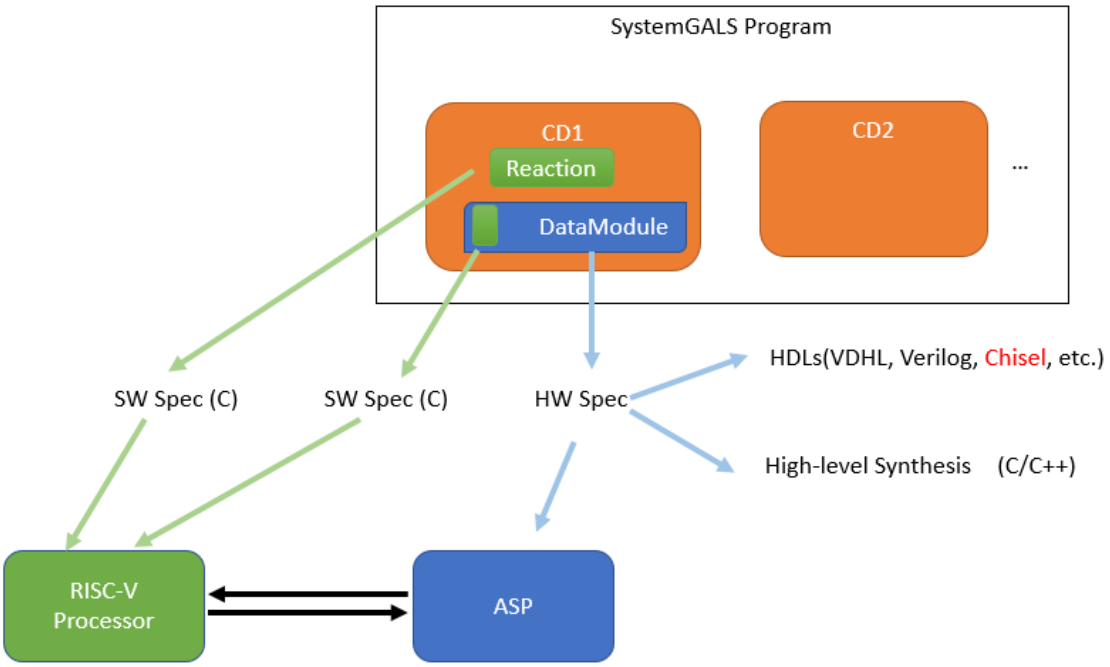


Figure 4-3. Mapping on software and hardware co-design.

Finally, the schedule can be fixed or dynamic. The fixed schedule is determined in the compilation stage, while the runtime support system (RTS) for SystemGALS can provide a dynamic schedule during the program execution. In this thesis, we only consider fixed schedules.

4.3. Communication

There are three main types of communication in SystemGALS: communication between control flow and data modules within one reaction, communication between different reactions within one clock domain, and communication between different reactions in different clock domains.

The communication between control flow and data modules is via the Control Flag Variable (CFV). A CFV is similar to a bool type in C and serves as an interface between a data module and control flow. A data module can take a CFV as a parameter and set the value via an API provided by SystemGALS. Then the control flow can use an if-else statement to check the CFV value and branch the program execution.

Reactions use signals to communicate with each other within one clock domain. The signals defined in one clock domain, no matter in which Reactions, are visible to all of the reactions in this clock domain. One reaction can check the status of one signal in await, present, or abort statement and retrieve the signal value with #<signal name> statement. It is noteworthy that an emitted signal has a limited lifetime of one single clock domain tick. So SystemGALS can only guarantee that reactions can capture the signal state change emitted within the same clock domain. The recipients may fail to capture an emitted signal from another clock domain.

Communication between different clock domains is achieved through channels or signals. A channel is unidirectional and must be defined on both sending and receiving sides. Send and receive statements provide a reliable communication method based on the rendezvous(handshake) protocol to ensure that the recipient can capture the information. Clock domain can also use signals to carry the information by continually emitting the signal until the recipient gives feedback. This can be achieved by the combination of abort and sustain statements.

All information exchange with the exception of CFVs only happens at the housekeeping stage that immediately comes after the end of any tick. So the new status and value of signals and channels will be only valid in the next tick rather than the current tick when the status change statements are executed. The values of CFVs will change during the tick. In SystemGALS, the runtime support system takes responsibility for status and value updating, which will be introduced in the next section.

4.4. Runtime Support System

The runtime support system provides various functions to support the execution of SystemGALS programs. The most relevant one for our research is to update the signals and channels during the housekeeping stage. Concretely, the runtime support system will maintain two copies of the statuses and values of each signal and channel. One copy is the current status and value during the execution tick, while the other is the expected status and value for the next tick. For example, if a reaction emits S1 in tick1, S1’s status will not be changed to present during tick1. Instead, the runtime support system will record the status change and update it after the execution of Tick1. Then during Tick2, the state of S1 will be present.

Chapter 5. Mapping SystemGALS Programs to C Programs

Once we understand the mechanism behind SystemGALS execution, we can map a SystemGALS program to an executable C program. The target C programs must contain the definition of clock domain and reactions, the schedule, signals and channels, control flows, data modules, and the runtime support system. Generally, we define clock domains and reactions as C functions nested in the main function. The schedule can identify with the order to execute these functions. Signals and CFVs tend to be mapped to basic and struct types in C, while channels must include an implementation of a handshake protocol. Furthermore, control flow contains the mapping of SystemGALS statements and a mechanism to guarantee the clock domain and reaction functions can correctly perform the GALS model. And as data modules themselves are implemented by C functions in our SystemGALS programs, it will be convenient to make this transition. So this chapter will not introduce the mapping of data modules. Finally, instead of implementing a whole RTS, we implement separate RTS functions for every clock domain to achieve the signals and channels updating.

5.1. Schedule

In this thesis, we employ a simple fix schedule scheme: successively executing the clock domains and reactions on a RISC-V core. For the single RISC-V core architecture, the RISC-V core will sequentially execute all controllers and plants clock domains. We define clock domains as functions and successively call the clock domain functions in the main function, as shown in Figure 5-1. And every clock domain has one tick execution time, and the reactions within the clock domain also are executed one by one during this tick. So we call the reactions within the corresponding clock domain function (lines 17 to 23). The execution of control flow and data modules will advance along the tick boundaries denoted by pause statements. Once the clock domain finishes one tick execution, RTS will take charge of housekeeping tasks to update the signals and channels. We define an RTS function (line 24) for every clock domain to update the signals and channels and call this function after the execution of all reactions. Moreover, for all controllers and device clock domains in the Sorter System, there is an infinite loop to enclose all of the control flow and data modules in every reaction, which enforces restart after all of the statements complete execution. So in the main function, we also define

an infinite loop (line 4) to surround all of the clock domains. When performing the experiment, we use an integer variable to control the execution. In terms of the schedule on the multi-core architecture, the only difference is that the controller will call the accelerators within the data modules to perform specific actions. Thus, from the reactions perspective, the schedules on both execution platforms are identical.

```
1  int main()
2  {
3     int i = 0;
4     while(i<50){
5        ConveyorController();
6        ConveyorPlant();
7        LoaderController();
8        LoaderPlant();
9        ArmController();
10       ArmPlant();
11       i++;
12    }
13    return 0;
14 }
15
16 void ConveyorController(){
17    ConveyorControllerR1();
18    ConveyorControllerR2();
19    ConveyorControllerR7();
20    ConveyorControllerR8();
21    ConveyorControllerR9();
22    ConveyorControllerR10();
23    RTSForConveyorController();
24 }
```

Figure 5-1. Schedule mapping.

5.2. Signal and CFV

In SystemGALS, there are two types of signals: pure signals, type signals. A pure signal only has the status indicating whether the signal is present or not, while a type signal has an attached value with the status. A signal type can be any basic type in C or complicated types defined in a struct. In the runtime support system (RTS), a pure signal contains two statuses: the current state and the next state. So we define a struct type with two bool variables (lines 1 to 4, Figure 5-2) to represent the pure signal in C programs. When checking the status of a signal, the program is actually accessing the current state of this signal. On the other hand, when we emit a signal, it is the next state that stores the new status. Then at the end of one tick, the RTS will update the current state with the next state and reset the next state to be not present. Thus, the present status of an emitted signal can only last for one tick.

```
1  typedef struct{
2     bool currentState;
3     bool nextState;
4  }Signal;  // pure signal
5
6  typedef struct{
7     bool currentState;
8     bool nextState;
9     int currentValue;
10    int nextValue;
11 }IntSignal; //integer type Signal
12
13 typedef struct{
14    bool currentState;
15    bool nextState;
16    float* currentCharacArray;
17    float* nextCharacArray;
18 }CharacArraySignal; // float array signal
19
20 typedef struct{
21    bool state;
22 }CFV;
```

Figure 5-2. Signal and CFV mapping.

Similarly, a type signal contains two states and two copies of values: current state, next state, current value, and next value. As the type of signals varies, we define different struct types for each type of signal used in our programs. Besides two bool types, the struct of every type signal contains two corresponding variables to store the current value and next value (lines 6 to 11). The current value is used in the current tick, while the next value carries the expecting value in the next tick. And, the RTS will update the status and the value in the housekeeping stage.

Regarding the complicated types like the array or struct types (lines 16 to 17), we define the signal values as pointers pointing to the concrete variables. If there is no change for the variable, these two pointers point to the same variable. Otherwise, the program needs first to make a copy of the variable and assign the address to the current pointer and then make changes on the variable to which the next pointer points.

CFV is a special signal that only contains the current status because the CFV serves as the interface between the data module and the following control flow, which are commonly executed within the same tick. So in SystemGALS, the updating of a CFV status happens immediately. We simply use a struct type (lines 20 to 22) that contains a bool variable to define a CFV construct.

5.3. Channel

Channels provide a mechanism that can ensure reliable communication between clock domains. The send and receive statements upon the same channel serves as rendezvous for information transmission. The SystemGALS pseudocodes for the concrete procedures are shown in Figures 5-3 and 5-4.

```
1  // the send statement, e.g. send C(val)
2
3  trap(T){
4
5          while(true){
6
7                  abort(immediate (!ReceiverPresent || ReceiverPreempted)){
8
9                          abort(immediate ACK) {halt;}
10
11                         abort(immediate !ACK) {sustain REQ(val);}
12
13                         exit(T);
14
15                 }
16
17                 pause;
18
19         }
20
21 }
```

Figure 5-3. Pseudocodes for send statement. [56]

```
1  // the receive statement e.g. receive C
2
3  trap(T){
4
5          while(true){
6
7                  abort(immediate (!SenderPresent || SenderPreempted)){
8
9                          abort(immediate !REQ) {halt;}
10
11                         abort(immediate REQ) {sustain ACK;}
12
13                         var = #REQ;
14
15                         exit(T);
16
17                 }
18
19         pause;
20
21         }
22
23 }
```

Figure 5-4. Pseudocodes for receive statement. [56]

Generally, the sender or the receiver will keep performing a specific action until the signal from the other side release it from the loop. And for every channel, RTS maintains a set of parameters to assist these procedures. The parameters involved are ReceiverPresent, ReceiverPreempted, ACK, SenderPresent, SenderPreempted, and REQ. REQ is a type signal whose type is consistent with the type of the channel, while the other signals are pure signals. Furthermore, the signals of ReceiverPresent, ReceiverPreempted and ACK will be checked by the sender side, and it is the responsibility of the receiver to update these signals. Similarly, the value and statuses of SenderPresent, SenderPreempted, and REQ are maintained by the sender and are used to control the receive procedures.

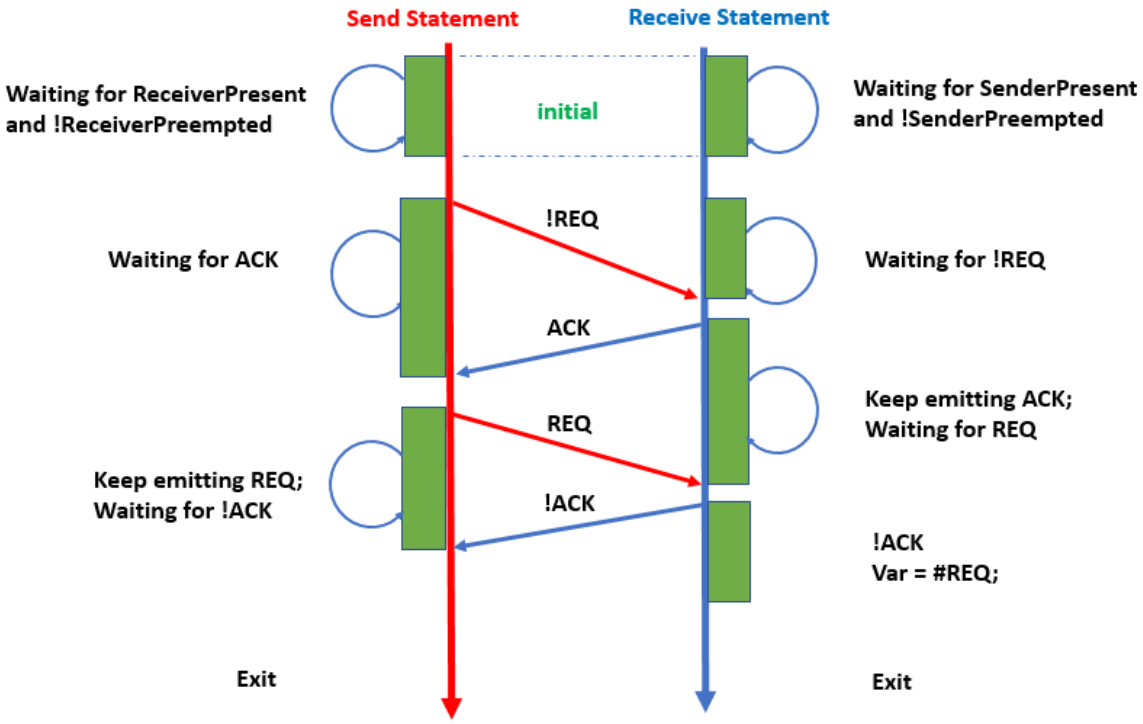


Figure 5-5. Channel handshake flow.

The handshake flow is shown in Figure 5-5. The complete transmission procedures consist of six stages: the initial stage, !REQ stage, ACK stage, REQ stage, !ACK and exit stage. At the initial stage, the sender and receiver need to guarantee that the other side is at the correct status for data transmission by checking a pair of signals. Concretely, the sender clock domain will check the signals of ReceiverPresent and ReceiverPreempted, while the receiver clock domain will check SenderPresent and SenderPreempted. Generally, these signals are not present by default and will be changed by RTS under specific conditions. For example, when the control flow in the sender clock domain hits the send statement, RTS will change the SenderPresent to

be present and change it back to be not present after the transmission is finished. And, SenderPreemted indicates if there are higher priority statements in control flow preempting the sending procedures. For example, if an abort statement encloses a send statement, and the abort condition is satisfied, the abort statement will interrupt the sending procedures, and the RTS will change the SendPreempted to be present. Only if SenderPresent is present, and SenderPreempted is not present, the receiver will continue the receiving procedure.

After the initial stage, the receiver will be waiting for the !REQ signal from sender and then send ACK signal to the sender once it receives the !REQ signal. Then the sender will send the REQ signal that conveys the required value and be waiting for the !ACK signal. Once the receiver receives the REQ signal, it will issue the !ACK signal that can release the sender from the loop and exit the sending procedures. Then the receiver saves the value from the REQ signal and finishes the receiving processes. So we can consider the ACK and REQ signals as tokens that control the preceding of the handshake protocol.

It is noteworthy that the sending of REQ and ACK is not directly from one side to the other side but through the parameters maintained by RTS. For example, when the receiver sends the ACK signal to Sender, it changes the ACK status to be present. Then, the RTS will update the ACK at the housekeeping stage. And the receiver will continually emit the ACK until the sender checks it and issue the signal that the receiver needs. One other thing worth noting is that the conditions in the initial stage need to be checked at the beginning of every tick during the handshake process. The sending and receiving procedures tend to last for multiple ticks for both sides. Control flow can change the statuses of SenderPresent, SenderPreempted, ReceiverPresent, and ReceiverPreempted at the end of any tick during the procedures. So at the beginning of every tick, the conditions need to be rechecked.

As discussed above, the send and receive statements themselves can be treated as small SystemGALS programs. So we design these two statements as two functions in C, and the send function is shown in Figure 5-6. We define a struct type (lines 1 to 6) to contain the parameters required in the procedures. We only use int channels in our case, so the REQ type is defined as the int type. The first statement in the function is to change the status of SenderPresent (line 14), informing the receiver that the sender is ready. Then we use the combination of a switch-case statement (line 15) and if-else statements to define the execution flow. And as a C function cannot record the execution position in the next tick (i.e. the next function call), we define a control flag (line 9) to choose the case branch in every send function call. In addition, a finish

flag (line 10) is used to indicate if the sending procedure is completed, which can trigger the reset of the relevant parameters. The receive function has a similar implementation.

```
1  typedef struct{
2     Signal SenderPresent, SenderPreempted;
3     Signal ReceiverPresent, ReceiverPreempted;
4     Signal ACK;
5     IntSignal REQ;
6  }CHIntParams;
7
8  typedef struct{
9     int cHControlFlag;
10    bool cHFinishFlag;
11 }CHInterFlags;
12
13 void sendInt(int val, CHIntParams* cHIntParams, CHInterFlags* cHInterFlags){
14    cHIntParams->SenderPresent.nextState = 1;
15    switch(cHInterFlags->cHControlFlag){
16       case 0: {
17             if(!(cHIntParams->SenderPresent.currentState) || (cHIntParams->SenderPreempted.currentState))
18             {
19                cHInterFlags->cHControlFlag = 0;
20                break;
21             }else if (cHIntParams->ACK.currentState){
22                if(cHIntParams->ACK.currentState){
23                   cHIntParams->REQ.nextValue = val;
24                   cHIntParams->REQ.nextState = 1;
25                   cHInterFlags->cHControlFlag = 1;
26                   break;
27                }else {
28                   cHInterFlags->cHControlFlag = 0;
29                   cHInterFlags->cHFinishFlag = 1;
30                   break;
31                }
32             } else {
33                cHInterFlags->cHControlFlag = 0;
34                break;
35             }
36          }
37
38       case 1: {
39             if(!(cHIntParams->ReceiverPresent.currentState) || (cHIntParams->ReceiverPreempted.currentState))
40             {
41              ...
42             }else if(cHIntParams->ACK.currentState){
43                                    ...
44                }else { //Finish send precedures
45                   cHInterFlags->cHControlFlag = 0;
46                   cHInterFlags->cHFinishFlag = 1;
47                   break;
48                }
49             }
50       }
51 }
```

Figure 5-6. Send function.

5.4. Control Flow

The control flow mapping contains two parts: statements mapping and execution logic mapping. In our case, we mainly use these statements: if-else, await, emit, sustain, abort, and channel operations: send and receive statements, which has been illustrated in section 5.3.

5.4.1 **Statements**

The if-else statement is similar to the one in C except that the if-else statement only accepts CFV expressions as its condition. The other expressions, including comparison operations, are not eligible. This design is to keep the control flow part as simple as possible so that it can target various host languages. So the if-else statement in SystemGALS is actually a subset of the one in C and can be replaced directly by the if-else statement in C.

The emit statement is to change the status and the value of a signal. As discussed in section 5.1, we use an assignment statement in C to change the next state and the next value of a signal, as shown on line 23 and line 24 in Figure 5.6. The sustain statement is syntactic sugar for continually emitting a signal, which is equivalent to while {true} {emit S; pause}. It is an infinite loop that repeatedly executes the emit operation in every tick. So it can be treated as an assignment statement inside a while(true) loop.

Generally, the sustain statement is used with the abort statement to escape from the infinite loop. The statement of abort([immediate] Sexp) {p} [do {q}] is similar to if-else statement, utilizing a signal expression (Sexp) as a condition to branch the execution. Nevertheless, the condition of the signal expression in the abort statement need to be checked at every tick during the execution of the block of {p}. So we can also use the if-else statement to substitute the abort statement and carefully design the execution logic to guarantee that the control flow performs the condition checking properly. Section 5.4.2 will discuss the concrete scheme in detail. Finally, the abort statement can also include an optional immediate keyword before the condition to check the condition immediately once the control flow hits the abort statement. Otherwise, the control flow will start to check the condition after the first pause statement, which means the block of {p} at least has a one-tick execution time.

5.4.2 **Execution Logic**

The most important part of control flow is the execution logic, which determines the order of statements' execution. In this thesis, we implement all SystemGALS reactions in C functions.

However, a C program function has three significant differences with a reaction in SystemGALS. The first one is that a C function will not record the parameters' value and release all of the parameters after the execution. However, signals and channels in a reaction relate to their previous states or values. Secondly, a C function executes all of the logic during one function call, while a reaction only executes the statements between the start point and the first pause or between two sequential pause statements. The last one is that a C function always executes the statements from the beginning, but a reaction will start from the previous pause.

The first problem can be solved by defining all of the signals and channels as global variables in C programs. For the next two, we can leverage the switch-case statement to wrap up all possible statements between two pauses into a case branch and define a global control flag for each reaction to choose the branch for every reaction call. And the break statement in the switch-case statement can serve as a pause in SystemGALS,

Generally, this way can work but may result in serious statements redundancy in specific scenarios, especially when there are nested abort statements in the program. In this situation, the number of possible statements combination within one tick will dramatically increase. For example, there is a nested abort statement (Figure 3-3, lines 10 to 22) in the Loader controller, where there is one external abort statement and four internal ones. The control flow needs to first check the conditions of the external abort at the beginning for every tick during the block execution. Then the internal abort condition will be checked. We assume that after the first (external) abort condition is checked, control flow enters the block. Then it will check the second abort condition and then choose to advance to the third abort or enter the execution body to end this tick due to the hidden pause in sustain statement. If proceeding to the next abort, a similar situation happens again. So it will be tedious and redundant to enumerate all of the possibilities in a switch-case statement.

So instead of just encapsulating the logic between two pauses into one case branch, we also include abort condition statements as breakpoints for case branches. If the next statement is an abort statement, the case branch will stop before the abort statement. This mechanism will significantly reduce the redundant code but cause another program: mistakenly dividing one-tick logic execution into two or more ticks. Because in C programming language, the switch-case statement will finish its execution once it encounters the break, resulting in postponing the next abort condition checking into the next tick.

```
1  void LoaderControllerR1(){
2     switch(LCR1ControlFlag){
3        case 0: {
4           if(startLoader.currentState){
5              LCR1ControlFlag = 1;
6              break;
7           }else{
8              LCR1ControlFlag = 0;
9              break;
10          }
11       }
12    switch(LCR1ControlFlag){
13       case 1:{
14          if(stopLoader.currentState){
15             LCR1ControlFlag = 11;
16             break;
17          }else {
18             LCR1ControlFlag = 2;
19             Break;
20          }
21       }
22    …
23 }
```

Figure 5-7. Loader controller execution logic.

We use a round-about way to solve this problem, which includes two principles. The first one is to insert another switch-case statement when a case branch stops before an abort statement, while the other one is that the case constants will be continuous across these switch-case statements. As shown in Figure 5-7, the await statement in the Loader controller is mapped into the case 0 branch (lines 3 to 11). If the startLoader signal is present, the next statement is an abort statement. We use a new switch-case statement to map this abort statement, but the case constant start with 1, the number after 0. Even though the first switch-case finishes its execution, it will check the control flag again in the next switch-case statement to make sure the abort condition can be checked in the same tick.

5.5. Runtime Support System

In terms of the runtime support system, in our case, we only implement the most relevant parts: the functions to update the signals and channels. Regarding pure signals, the RTS simply assigns the next state to the current state after every tick and set the next state to false at the same time. If the control flow emitted a pure signal during the tick, the next state is true, thus updating the current state to be true and resulting in the presence of the signal in the next tick. Otherwise, the values of the current and next state remain false. We employ the same scheme to the state part in type signals.

On the other hand, the RTS function only updates the current value with the next value when the next value is true, demonstrating the value of the signal may change. If the control flow does not change the signal's value and only change the state, the next value is consistent with the current value. Thus the update will not change the current value.

In terms of channels, as we discussed in section 5.3, the sending clock domain should update the parameters of SenderPreempted, SenderPresent, and REQ signals. In contrast, the receive clock domain should maintain ReceiverPreempted, ReceiverPresent, and ACK signals. The RTS functions for the sender and receiver clock domains should perform corresponding actions. In addition, if the sending and receiving procedures have finished (i.e. the value of a finish flag is true), the RTS functions also need to reset the SenderPresent and ReceiverPresent signals and the control flags.

Finally, It is worth noting that the schemes we use to map the SystemGALS programs apply only to this research. The developing SystemGALS compiler may adopt different mechanisms, and the runtime support system is much more powerful.

Chapter 6. Single-core Execution Platform

When finishing the mapping from SystemGALS programs to C programs, we leverage Chipyard to prepare the execution platforms for the target C programs. In this chapter, we modify the default SoC configuration in Chipyard to create a single-core platform. In contrast, chapter 7 will illustrate the Chisel design of two accelerators or application-specific processors (ASPs) and connect them with the single-core platform via a 4-port TDMA-MIN to create a multiple-core execution platform. This chapter will first introduce the general structure of a Chipyard SoC and then explain the SoC bring-up method. Next, we explain how to build up our Single-core execution SoC platform by modifying a default configure class. The last section will explain the building processes of a software SoC simulator.

6.1. Chipyard SoC Structure

Chipyard is an open-source project, and anyone can access the code via GitHub. The community also provides an online document to facilitate the usage of Chipyard with conception explanation, component description, and concrete examples. Both the project and the document are constantly evolving and updating. In this thesis, we use the Chiyard 1.3 version on Ubuntu to perform our research. A Chipyard SoC project in Chipyard mainly contains three components: ChipTop, Testharness, and TestDriver. The ChipTop is a concrete Verilog digital design of an SoC, while Testharnesess and TesDriver are used to facilitate the software simulation of the SoC.

6.1.1 **BaseSubsystem**

The ChipTop Verilog file is generated by Chisel generators, and Chipyard leverages multiple layers of Scala classes and traits to organise these generators. The bottom class, BaseSubsystem, resides in the Rocket chip project, and it takes charge of instantiating all top-level buses generators, including the system bus, front bus, interrupt bus and periphery bus. However, it does not specify connections between these buses. In addition, the BaseSubsystem class is used to generate the device tree string and the diplomacy graph

visualization file that is used to validate and negotiate connection parameters between different components.

6.1.2 **ChipyardSubsystem**

At the second level, The ChipyardSubsystem class (Figure 6-1) extends the BaseSubsystem class, thus inheriting the ability to initiate the buses. It also mixes in a HasTiles (line 2) trait that defines and instantiates Rocket or Boom tiles (lines 5 to 8) according to specified parameters. Each tile mainly contains one Rocket or Boom core and L1 caches. And it also has a CanHaveHTIF (line 3) trait to set up the SoC bring-up methods that will be discussed in Section 6.2 in detail. Traits in Scala are designed to enable multiple inheritance. That means a class in Scala can only have one superclass but many traits. In contrast to an interface in Java, a trait can contain not only abstract members but also concrete members. Generally, Chipyard instantiates different component generators in traits and composes these components with the lower layer class to gradually build up a complete SoC. In this case, ChipyardSubsystem uses BaseSubsytem as a base class and extends it with the HasBoomAndRocketTiles trait to integrate cores into the Soc. It is worth noting that Scala uses a rule to avoid the diamond problem in multiple inheritance, which is that if there are multiple implementations of a given member, the furthest one to the right wins. So the order of composing traits in a class can influcence the final result.

```
1  class ChipyardSubsystem(implicit p: Parameters) extends BaseSubsystem
2    with HasTiles
3    with CanHaveHTIF
4  {
5    def coreMonitorBundles = tiles.map {
6      case r: RocketTile => r.module.core.rocketImpl.coreMonitorBundle
7      case b: BoomTile => b.module.core.coreMonitorBundle
8    }.toList
9    ...
10   }
11   override lazy val module = new ChipyardSubsystemModuleImp(this)
12 }
13
14 class ChipyardSubsystemModuleImp[+L <: ChipyardSubsystem](_outer: L) extends
15 BaseSubsystemModuleImp(_outer)
16   with HasTilesModuleImp
17 {
18   ...
19 }
```

Figure 6-1. ChipyardSubsystem. [60]

This pattern that uses traits or mixins to compose a class is called the cake pattern, while a class that mixes in traits is called a “cake” or mixin. In fact, there are two mixins in every SoC class layer: one for the lazy module (line 1) and one for the lazy module implementation (line 14). And the traits that are mixed in a class can also be either a lazy module trait or a lazy module implementation trait. These two types of constructs are defined in Diplomacy (another library embedded in Scala) to split Chisel module elaboration into two phases.

The modules in Chisel, similar to Verilog modules, are the basic constructs to design circuits. An SoC component generator tend to be a Scala class that inherits Chisel Module construct, and it mainly contains two parts: parameters passed into the class, the concrete RTL design that contains an interface wrapped in the IO() method and the wiring part. Generally, after the parameters are specified, the interface and the wiring part are together elaborated into FIRRTL specifications and in turn Verilog designs. However, Diplomacy leverages the LazyModule and LazyModuleImp constructs to delay the elaboration by inserting a process that collects and validate all components parameters.

On the one hand, lazy module classes and traits specify all the logical connections and communicate configuration information with each other. This generally involves creating TileLink nodes for SoC components, constructing a Directed Acyclic Graph (DAG) containing all nodes, and negotiating and validating the annotated parameters along the DAG. On the other hand, lazy module implementation classes and traits perform the real RTL elaboration (i.e. evaluating the Chisel specification to generate circuit designs for the components). In addition, the lazy module class instantiates the corresponding lazy module implementation class with a lazy keyword within the class body to combine these two phases. For example, ChipyardSubsystem defines a lazy val to instantiate ChipyardSubststemModuleImp (line 11). In Scala, the lazy keyword can offer a val an advantage that it can only get initialized on the first access. In Chipyard name convention, the name of the lazy module implementation class is the name of the corresponding lazy module class with a “Module” or “ModuleImp” suffix, while the name of the lazy module implementation trait is the lazy module trait’s name with an “Imp” suffix that refers to implementation. For example, BaseSubsystem is a lazy module class, and BaseSubsystemModuleImp is the corresponding lazy module implementation class. Similarly, ChipyardSubsystem and ChipyardSubsystemModuleImp are the lazy module class and the lazy module implementation, respectively. And HasTiles is a lazy module trait, while HasTielsModuleImp is the lazy module trait.

6.1.3 **ChipyardSystem**

Coming back to the SoC structure, ChipyardSystem and ChipyardSystemModule (Figure 6-2) extend the ChipyardSubsystem and ChipyardSubsystemModuleImp classes with some other traits on the next level to complete a fully-featured SoC definition. The HasAsyncExtInterrupts and HasExtInterruptsModuleImp traits integrate IOs for external interrupts and connect them to core tiles, and the CanHaveMasterTLMemPort and CanHaveMasterAXI4MemPort trait expose a TileLink port and an AXI port for the outer memory, respectively. In addition, ChipyardSystem adds a BootROM device into the SoC.

```
1  class ChipyardSystem(implicit p: Parameters) extends ChipyardSubsystem
2    with HasAsyncExtInterrupts
3    with CanHaveMasterTLMemPort // export TL port for outer memory
4    with CanHaveMasterAXI4MemPort // expose AXI port for outer mem
5    with CanHaveMasterAXI4MMIOPort
6    with CanHaveSlaveAXI4Port
7  {
8    ...
9    override lazy val module = new ChipyardSystemModule(this)
10 }
11
12 class ChipyardSystemModule[+L <: ChipyardSystem](_outer: L) extends ChipyardSubsystemModuleImp(_outer)
13   with HasRTCModuleImp
14   with HasExtInterruptsModuleImp
15   with DontTouch
```

Figure 6-2. ChipyardSystem. [60]

6.1.4 **DigitalTop and ChipTop**

At last, DigitalTop and DigitalTopModule (Figure 6-3) compose this SoC with many optional components by extending ChipyadSystem with corresponding components traits on the top level. It is worth noting that the lazy module implementation trait is not necessary for a specific component when it does not need to instantiate another concrete module or be physically connected into IOs or wires. Instead, it can put the Chisel specification directly into a Diplomacy LazyModuleImp class, initiate this class in a LazyModule class and then wrap the LazyModule class into a LazyModule trait that can be mixed in the DigitalTop class. In the next chapter, we will create a new trait and add it to DigitalTop to build up our second execution platform. The DigitalTop level classes offer possibilities to create various SoC variants. However, it needs concrete parameters to instantiate a specific SoC. In Chipyard, a separate ChipTop class is used to input parameters to DigitalTop. We have briefly introduced the parameter system in Section 2.5.2 and will discuss it in detail in Section 6.3.

```
1  class DigitalTop(implicit p: Parameters) extends ChipyardSystem
2    with testchipip.CanHavePeripheryCustomBootPin
3    with testchipip.HasPeripheryBootAddrReg
4    with testchipip.CanHaveTraceIO
5    with testchipip.CanHaveBackingScratchpad
6    with testchipip.CanHavePeripheryBlockDevice
7    with testchipip.CanHavePeripheryTLSerial
8    with sifive.blocks.devices.i2c.HasPeripheryI2C
9    with sifive.blocks.devices.pwm.HasPeripheryPWM
10   with sifive.blocks.devices.uart.HasPeripheryUART
11   with sifive.blocks.devices.gpio.HasPeripheryGPIO
12   with sifive.blocks.devices.spi.HasPeripherySPIFlash
13   with sifive.blocks.devices.spi.HasPeripherySPI
14   with icenet.CanHavePeripheryIceNIC
15   with chipyard.example.CanHavePeripheryInitZero
16   with chipyard.example.CanHavePeripheryGCD
17   with chipyard.example.CanHavePeripheryStreamingFIR
18   with chipyard.example.CanHavePeripheryStreamingPassthrough
19   with nvidia.blocks.dla.CanHavePeripheryNVDLA
20 {
21   override lazy val module = new DigitalTopModule(this)
22 }
23
24 class DigitalTopModule[+L <: DigitalTop](l: L) extends ChipyardSystemModule(l)
25   with testchipip.CanHaveTraceIOModuleImp
26   with sifive.blocks.devices.i2c.HasPeripheryI2CModuleImp
27   with sifive.blocks.devices.pwm.HasPeripheryPWMModuleImp
28   with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
29   with sifive.blocks.devices.gpio.HasPeripheryGPIOModuleImp
30   with sifive.blocks.devices.spi.HasPeripherySPIFlashModuleImp
31   with sifive.blocks.devices.spi.HasPeripherySPIModuleImp
32   with chipyard.example.CanHavePeripheryGCDModuleImp
33   with freechips.rocketchip.util.DontTouch
```

Figure 6-3. DigitalTop. [60]

6.1.5 **Testharness and Testdriver**

Besides the ChipTop that contains the real SoC design, Chipyard also has a Testharness and a Testdriver to facilitate the SoC design testing. The Testharness is also designed in Chisel and Diplomacy specification. There are two main functions of the Testharness. Firstly, it takes responsibility to initiate the ChipTop class. This means all SoC components are instantiated in the Testharness class scope, and the SoC parameters are passed to the ChipTop from the Testharness. Secondly, it integrates various software-simulated devices into the SoC. For example, it leverages a SimAXIMem module to simulate an off-chip DRAM, which simply connects a single-cycle SRAM for each memory channel. In addition, Testhaness employs IOBinder functions to initiate IO ports and cells on the ChipTop side and pass the generated IO port to HarnessBinder functions to connect ChipTop with the simulated devices. It is worth noting that IOBinders and HarnessBinders in Chipyard are designed to decouple the SoC digital design from different simulation methods or testing purposes. In this way,

Chipyard can use these two types of functions to attach the ChipTop to an FPGA harness without changing the ChipTop class.

Although the ChipTpop's instantiation happens within the Testharness scope, the project, after the Chisel module elaboration, produces two separate Verilog files for these two classes: Top.v and Testharness.v. And the Testdriver itself is a Verilog file in Chipyard. These three Verilog files will be compiled together to generate a software simulator. The Testdriver Verilog module is used to instantiate the Testharness during the software simulation and drive the clock and reset signals for the SoC.

6.2. SoC Bring-up Method

In addition to the Testharness and Testdriver, Chipyard provides two ways to bring up the SoC simulation. The bring-up procedures mainly involve program loading from somewhere to the SoC memory space, program execution on the SoC, and shutdown of the SoC. Chipyard provides two methods to achieve the SoC bring-up. The first one is to use specific interfaces to enable communication between the host (e.g. the PC that runs the simulator) and the SoC software simulation. In this case, the SoC is called a tethered DUT. The second way is to build up a standalone DUT that can load programs from an SD card and have its own BootROM.

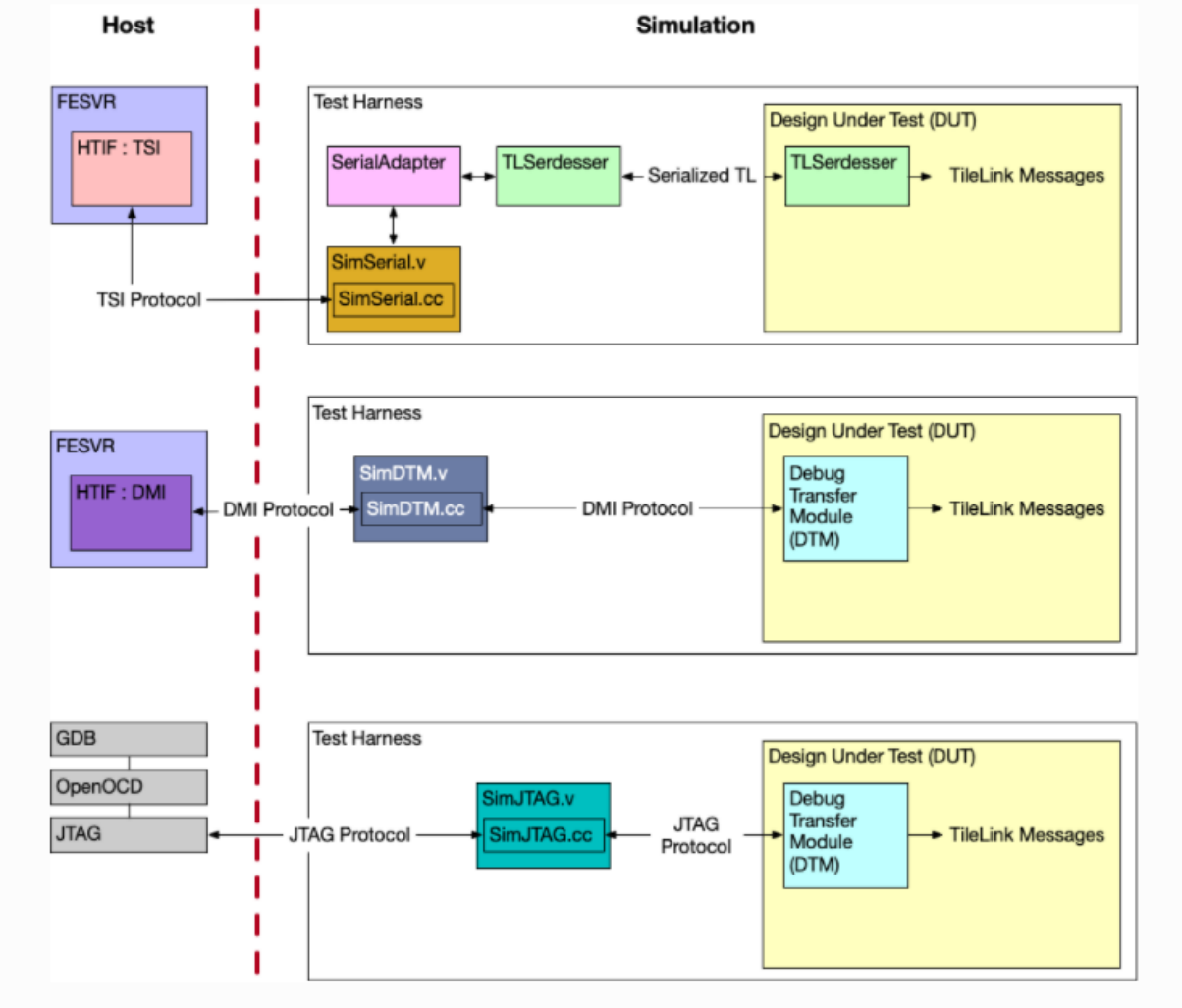


Figure 6-4. Communication between the Host and the Simulation. [60]

A tethered DUT can leverage two types of interfaces (Figure 6-4) to enable the communication between the host and the simulator in Chipyard: the Host Target Interface (HTIF) and the JTAG interface. And for both interfaces, it requires specific facilities on the host side and the simulation side. JTAG is an industry-standard protocol, and it uses OpenOCD and GDB on the host side, and it employs a SimJTAG module and Debug Transfer Module (DTM) residing in TestHarness and DUT, respectively. In contrast, HTIF is a non-standard interface that is developed by UC Berkeley. And it has two types of implementation in Chipyard: the Tethered Serial Interface (TSI) and the Debug Module Interface (DMI). The CanHaveHTIF trait that is mixed in the ChipyardSubsystem provides these two implementation options. Then, the Testharness and ChipTop will integrate corresponding modules into the simulator. For DMI protocol, Testharness contains a SimDTM module, while DUT has the DTM. In contrast, the Testhaness leverage a serial port that involves a SimSerial module that simulates a standard

serial port, a TLSerdesser that issues serialized TileLink messages, and a SerialAdater that bridges these two types of messages. At the same time, DUT integrates another TLSerdesser module to communicate with Testharness. Although the TSI is more complicated than the DMI, Chipyard uses the TSI as a default interface due to its higher transmission speed compared to the DMI. In this thesis, we also employ the TSI to load programs to the SoC simulator.

In addition, both TSI and DMI leverages a Front-End Server (FESVR) on the host side. FESVR is a C++ library that provides an API to use Read/Write commands to access the main memory of the DUT. Specifically, it sends and receives messages via a non-blocking FIFO interface provided by the HTIF to release the DUT out of reset signal, load programs from the host to the DUT memory, and run the programs on the DUT. It is worth noting that although FESVR is responsible for communication between the host and the Testharness, it is complied with the Testdriver, Testharness, and ChipTop together to produce the software simulator.

6.3. SoC Configuration

For a Chipyard SoC, the Top-level class, DigitalTop, provides and organizes many optional components by mixing corresponding traits. In order to make use of and change these components, we must configure these components in Chipyard's parameter system, which is briefly introduced in Section 2.5.2. This section will use a concrete example to demonstrate how to configure a Chipyard SoC (i.e. the DigitalTop class). Then, we customize the default configuration class to build up our single-core execution platform.

6.3.1 **Configuration Example**

Similar to the SoC design classes, the SoC configuration also has multiple layers that consist of a configuration class, config fragments, and case classes from top to bottom. Every component generator should have a case class that defines the parameters that can be customized for the generator. The case class can also assign default values for these parameters. In addition, a component key is also created along with the case class to identify this component. Then a config fragment builds up a mapping from this component key to the case class. This config fragment can also accept parameters that can be used to initiate the case class and thus change the default parameters of the component. Furtherly, a config fragment is essentially the

same as a mixin. So these config fragments for different components can be composed in a config class. Hence, a Chipyard SoC can use a single config class to complete the Digital class configuration. Chipyard has pre-defined various default configuration classes in the project config fold. The config class name can be assigned to a CONFIG variable in the make command to make a custom SoC. In Figure 6-5, a default configuration class (lines 39 to 48) is used to incorporate an InitZero component into the SoC.

```
1  case class InitZeroConfig(base: BigInt, size: BigInt)
2  case object InitZeroKey extends Field[Option[InitZeroConfig]](None)
3
4  class WithInitZero(base: BigInt, size: BigInt) extends Config((site, here, up) => {
5    case InitZeroKey => Some(InitZeroConfig(base, size))
6  })
7
8  class InitZero(implicit p: Parameters) extends LazyModule {
9    val node = TLHelper.makeClientNode(
10     name = "init-zero", sourceId = IdRange(0, 1))
11
12   lazy val module = new InitZeroModuleImp(this)
13 }
14
15 class InitZeroModuleImp(outer: InitZero) extends LazyModuleImp(outer) {
16   val config = p(InitZeroKey).get
17 …
18   val addr = Reg(UInt(addrBits.W))
19   val bytesLeft = Reg(UInt(log2Ceil(config.size+1).W))
20 ..
21   when (state === s_init) {
22     addr := config.base.U
23     bytesLeft := config.size.U
24     state := s_write
25   }
26
27 …
28 }
29
30 trait CanHavePeripheryInitZero { this: BaseSubsystem =>
31   implicit val p: Parameters
32
33   p(InitZeroKey) .map { k =>
34     val initZero = LazyModule(new InitZero()(p))
35     fbus.fromPort(Some("init-zero"))() := initZero.node
36   }
37 }
38
39 class InitZeroRocketConfig extends Config(
40               …
41               new testchipip.WithTSI ++
42               ..
43               new chipyard.example.WithInitZero(0x88000000L, 0x1000L) ++
44               new freechips.rocketchip.subsystem.WithNoMMIOPort ++
45               …
46               new freechips.rocketchip.subsystem.WithNBigCores(1) ++
47               new freechips.rocketchip.subsystem.WithCoherentBusTopology ++
48               new freechips.rocketchip.system.BaseConfig)
```

Figure 6-5. InitZero configuration. [60]

The DigitalTop (line 15, Figure 6-3) has integrated an InitZero component by mixing a CanHavePeipheryInitZero lazy module trait. However, we still need a config fragment (line 43) in the configuration class to instantiate this component. Otherwise, the DigitalTop will ignore this component. Then, the InitZero trait that is mixed in DigitalTop can use a component key to get the concrete configuration from the config fragment in the configuration class. Specifically, this component simply initializes the values in a specific memory segment to Zero. The InitZeroConfig case class (line 1, Figure 6-5) defines two parameters: base and size, which stands for the start address and the length of the memory segment. Then, a component key, InitZeroKey (line 2), is defined as an Option type that can contain an object created from the InitZeroConfig class. An Option type is a container or an encapsulation of an optional value of a specific type. The value of an Option object can be a meaningful value that is donated as Some(X), where X represents the concrete object. The Option object's value can also be None that means no meaningful object or value is returned. In this case, the value of the InitZeroKey can be Some (InitZeroConfig Object) or None. Next, a config fragment, WithInitZero (line 4), maps this key to the InitZeroConfig class that is initialized in the config fragment (line 43).

Then the CanHavePeripheryInitZero trait uses the p (InitZeroKey) (line 33) to form a query that traverses every fragment to find the mapping containing the InitZeroKey. In this case, the query result is Some(InitZeroConfig(0x88000000L, 0x1000L)). It is worth noting that the InitZeroConfig object is wrapped by the Some and cannot be directly accessed. So the trait uses a get method (line 16) to get the object inside the container. The map after p (InitZeroKey) is a high order function that can be used to simplify the pattern match. If the query result is not None, the map function will perform the operations within the following block. Otherwise, it will do nothing. In this case, the map function instantiates the InitZero Lazy module class with the query result and connect its node to the front bus. This is a typical pattern about how the configuration class instantiate a component.

The definition of p: Parameter (lines 8 and 31) appears in many class definitions in Chipyard. This parameter can be viewed as a variable that passes a site view of a configuration class. In section 2.5.2, we have mentioned that the Chipyard parameter system provides three views for a key query: site, here, and up. The site view stands for the whole configuration class. Hence, the query of p(InitZeroKey) will check every config fragment within the InitZeroRocketConfig class from bottom to up. And this view is passed from the Testharness and then down to the ChipTop, DigitalTop, CanHavePeripheryInitZero trait, InitZero Lazymodule class, and finally

the InitZeroModuleImp class. So the InitZeroModuleImp can use p(InitZeroKey) and the get method to retrieve the object and access the concrete parameters. In terms of the up view, if we change the size value assignment to an up view query (line 2, Figure 6-6), then the WithInitZero fragment will find the size value in the next level fragment (WithNoMMIOPort, line 44, Figure 6-5) with the RocketTile key. If we change it to a here view query (line 10, Figure 6-6), the searching scope is only within the WithInitZero fragment. And the size value will be 256.

```
1  class WithInitZero(base: BigInt) extends Config((site, here, up) => {
2    case InitZeroKey => Some(InitZeroConfig(base, size = up (RocketTileKey, site).length)
3  })
4
5  class WithInitZero(base: BigInt) extends Config((site, here, up) => {
6   case  SizeKey => 256
10  case InitZeroKey => Some(InitZeroConfig(base, size = here(SizeKey, site)))
11 })
```

Figure 6-6. Up and here view. [60]

6.3.2 **Single-core Execution Platform Configuration**

In this section, we will define a custom configuration class for our single-core execution platform. Chipyard has pre-defined config fragments for exiting component that resides in the file of ConfigFragments.scala. The single-core platform does not contain any new components, so we simply use pre-defined config fragments to set up the basic configuration for the SoC, as shown in Figure 6-7.

```
1  class MySingleCorePlatform extends Config(
2    new chipyard.iobinders.WithUARTAdapter ++
3    new chipyard.iobinders.WithTieOffInterrupts ++
4    new chipyard.iobinders.WithBlackBoxSimMem ++
5    new chipyard.iobinders.WithTiedOffDebug ++
6    new chipyard.iobinders.WithSimSerial ++
7    new testchipip.WithTSI ++
8    new chipyard.config.WithBootROM ++
9    new chipyard.config.WithUART ++
10   new chipyard.config.WithL2TLBs(1024) ++
11   new freechips.rocketchip.subsystem.WithNoMMIOPort ++
12   new freechips.rocketchip.subsystem.WithNoSlavePort ++
13   new freechips.rocketchip.subsystem.WithInclusiveCache ++
14   new freechips.rocketchip.subsystem.WithoutFPU++
15   new freechips.rocketchip.subsystem.WithRV32 ++
16   new freechips.rocketchip.subsystem.WithNExtTopInterrupts(0) ++
17   new freechips.rocketchip.subsystem.WithNBigCores(1) ++
18   new freechips.rocketchip.subsystem.WithCoherentBusTopology ++
19   new freechips.rocketchip.system.BaseConfig)
```

Figure 6-7. Single-core platform configuration. [60]

In the 1.3 version of Chipyard, the BaseConfig (line 19) is required for every SoC, and the WithCoherentBusTopology (line 18) specifies the top-level buses connection as a coherent bus topology. It can be changed to a ring topology with a WithRingSystemBus fragment. Next, WithNBigCores(1) (line 17) adds a Rocket tile with a Rocket core, and one can integrate multiple rocket cores with the specified number. And as we target an embedded application, the configuration class has a WithRV32 trait (line 15) that changes the bit width from 64 to 32. Due to the same reason, we also remove the Floating Point Unit (FPU) from the Rocket with a WithoutFPU fragment (line 14). The WithTSI (line 7), WithSimSerial (line 6), and WithTiedOffDebug (line 5) are used to integrate the necessary modules to enable the communication between the host and the simulation. In addition, WithNoMMIOPort (line 11) and WithBlackBoxSimMem (line 4) will override the default memory port on the Rocket chip and add a DRAM simulation module.

6.4. Simulator Building Process

Once we create our configuration class, it is easy to produce a software simulator. After executing the command of make CONFIG = MySingleCorePlatform, Chipyard will generate a set of makefiles that take charge of the following procedures. The other make variables are listed in the variables.mk file under the Chipyard root folder. One can leverage these variables to create a custom SoC project rather than only modifying the default project. For example, we can use the TB, MODEL, and TOP variables to replace the default TestDriver, Testharness, and ChipTop, respectively.

Chipyard contains two simulation software to generate an executable simulator for a Chipyard SoC: Verilator and Synopsys VCS. Verilator is open-source software and can generate a cycle-accurate simulator, so we leverage Verilator to perform the experiment. Verilator is basically a compiler, and it takes SystemVerilog/ Verilog codes as input and in turn compile them into C++ codes. Then these C++ codes, along with the header files and libraries provided by Verilator, are compiled with a C++ compiler to generate an executable simulator.

Although the command is simple, it takes a long journey to produce a generator from the project source code to a simulator. Firstly, the project uses the specified configuration class to configure all parameterized generators. Then, the Diplomacy creates a DAG according to the logical connections defined in the lazy module traits and negotiates the parameters along the

DAG edges. Next, the Scala compiler compiles the source code into Byte codes and run the Bytecodes to generate the FIRRTL specification. Then, the FIRRTL transformations convert the FIRRTL presentation into Verilog files. Next, Verilator “verilates” all Verilog files into C++ programs. Finally, the C++ compiler compiles all C++ files into an executable binary simulator.

Chapter 7. Multi-core Execution Platform

This chapter will demonstrate how to build up our multiple-core execution platform that is used to promote the performance of the execution of our programs by adding two accelerators or application-specific processors (ASP) into the single-core platform. The new platform also involves a 4-port TDMA-NoC and a DMA device. We will first introduce the architecture of this multi-core platform and then illustrate the concrete digital design of the TDMA-NoC and ASPs. The last section will demonstrate how we integrate these modules into the SoC.

7.1. Architecture Overview

Compared with the single-core platform, the multi-core platform contains some new components: a 4-port TDMA-MIN NoC, a DMA device, and two ASPs. On the one hand, from the Rocket core's perspective, these components are wrapped up into an MMIO peripheral and connected with the front bus and peripheral bus, as shown in Figure 7-1. The peripheral exposes a number of memory-mapped registers whose addresses are in the same address space as the main memory. Thus, it can directly receive the parameters or commands from the Rocket core. In particular, these parameters define the beginning address and size of the input data in the main memory. The DMA device will retrieve data from the main memory according to these parameters, and then the peripheral transfers the data to the first port of the 4-port TDMA-MINNoC through a buffer and a Network interface sequentially. Then the TMDA-MIN NoC delivers the data to the specific port and to the corresponding ASP. Once the ASP finishes the computation, it will return the result from the port where it receives the input data. After that, the DMA device writes the result back to the main memory. Thus, the parameters received from the Rocket core also involve the port number of the ASP and the result address and size. In addition, a status register is used to show the state of the peripheral. During the data transmission and computation, the Rocket core will poll the status register to check if the result has been returned.

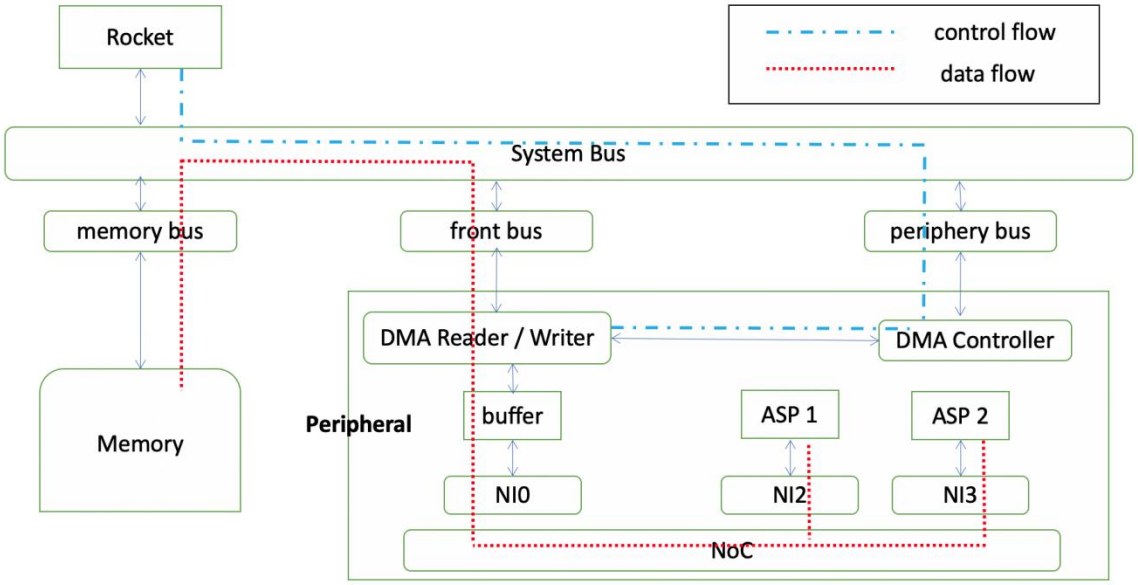


Figure 7-1. Multi-core execution platform architecture overview.

On the other hand, from the TDMA-MIN NoC perspective, the Rocket core and two ASPs are three execution sources connected to different ports (Figure 7-2). The Rocket core is responsible for executing all control flow and parts of the computation, while the two ASPs take charge of two specific functions. In section 3.4, we have introduced that our image recognition algorithm consists of three parts: binarization, characteristic vector (CV) extraction, and item recognition, which has been implemented with C in data modules in the conveyor controller. For the programs executed on the multi-core platform, we will change these data modules implementation to leverage the ASPs. Concretely, the binarization data module remains unchanged, while all software functions in the CV extraction recognition data module are replaced with a C API (Application Programming Interface) function that can call ASP1. In terms of item recognition, the subfunction of calculating the majority type stays the same, but the distance calculation and bubble sort functions are achieved by ASP2, which is also called through a C API function. When the Rocket core is calling a specific ASP, it inputs data from the main memory to a corresponding port on TDMA-MIN NoC and accesses the results after the ASP returns them to the main memory through TDMA-MIN NoC.

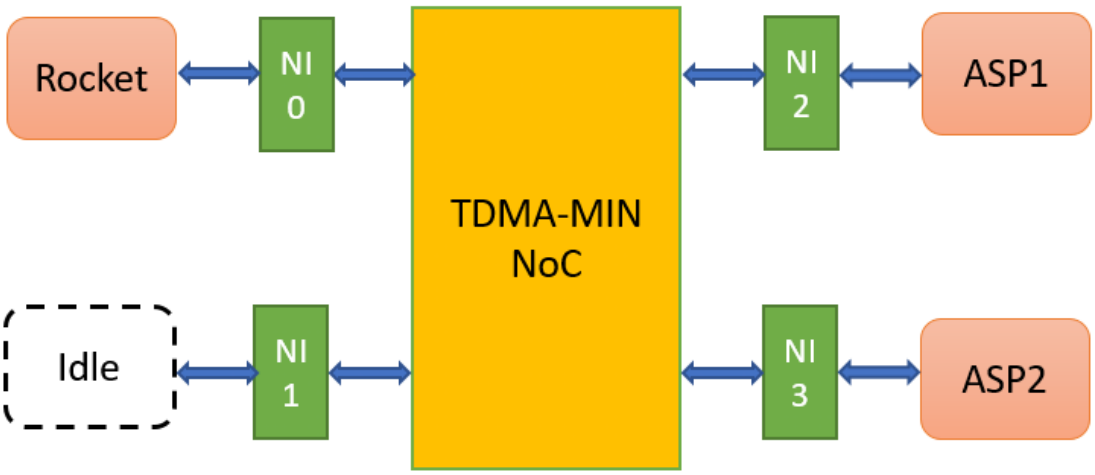


Figure 7-2. TDMA-MIN NoC connections.

7.2. 4-Port TDMA-MIN NoC and Network Interface

The TDMA-MIN NoC serves as an information exchanging network between the Rocket core and ASPs, while a network interface (NI) is used to bridge one core and one NoC port. In our case, we introduce a 4-port TMDA-MIN NoC and adopt the design of the NI demonstrated in Section 2.6. Specifically, the 4-port TDMA-MIN NoC consists of four 2×2 crossbar switches that are interconnected with each other in a Banyan type, as shown in Figure 7-3. Thus this NoC has four inputs and four outputs, and the connections between inputs and outputs are cyclically changed in different TDMA slots that are driven by a slot counter. In addition, each port of the NoC contains one input and one output and is generally connected with a network interface (NI). This section will introduce the Chisel implementation of the 4-port TDMA-MIN NoC and network interfaces.

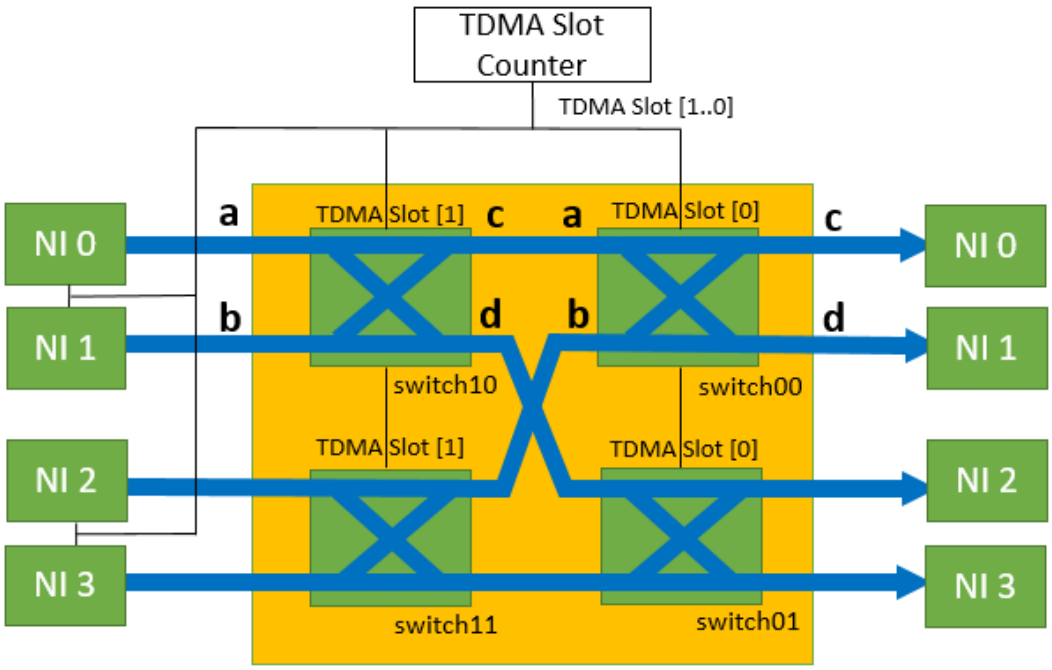


Figure 7-3. 4-port TDMA-MIN NoC with Network Interfaces.

7.2.1 **NoC Implementation**

The TDMA-MIN NoC comprises the crossbar switches and a TDMA slot counter. The slot counter is basically a 2-bit number that is cyclically changed from 00 to 11. Each bit of the counter serves as a slot input of a crossbar switch to control its status. A crossbar has two inputs (i.e., a and b) and two outputs (i.e. c and d). If the slot input is 0, the status of the crossbar switch is parallel, which means that the output c and d are connected with the input a and b, respectively. Otherwise, the crossbar switch is on the cross status resulting in the interconnections between a and d and between b and c. In addition, the four crossbar switches are arranged in two stages. And the switch00 and swith01 are in the first stage and use the first bit (furthest to the right) of the slot counter as their slot input, while switch10 and swich11 are in the second stage and receive the value of the second bit in the slot counter.

In Chisel, modules are basic constructs to design hierarchical structures for digital circuits. A user-defined module is basically a class that inherits the Module class. And each module generally consists of two parts: an interface that contains the module's input and output ports and is wrapped in a Module's IO() method, and the construction part that defines combinational and sequential logic and wires them together. And, the Module in Chisel provides an implicit

clock and reset. One can also define explicit clock and reset signals by extending a RawModule class instead of the Module one.

```
1  class CrossbarSwitch(val bitwidth : Int) extends Module {
2    val io = IO(new Bundle {
3      val in_s  = Input(UInt(1.W))
4      val in_a = Flipped(Valid (UInt(bitwidth.W)))
5      val in_b = Flipped(Valid (UInt(bitwidth.W)))
6      val out_c = Valid (UInt(bitwidth.W))
7      val out_d = Valid (UInt(bitwidth.W))
8    })
9
10   when (io.in_s === 0.U) {
11     io.out_c <> io.in_a
12     io.out_d <> io.in_b
13   }.otherwise {
14     io.out_c <> io.in_b
15     io.out_d <> io.in_a
16   }
17 }
18
19 class TMDAMIN44 (val bitwidth: Int)extends MultiIOModule {
20     val io = IO(new Bundle {
21                 val out_TDMASlot = Output(UInt(2.W))
22                 val in0  = Flipped(Valid (UInt(bitwidth.W)))
23                 val in1  = Flipped(Valid (UInt(bitwidth.W)))
24                 val in2  = Flipped(Valid (UInt(bitwidth.W)))
25                 val in3  = Flipped(Valid (UInt(bitwidth.W)))
26                 val out0 = Valid (UInt(bitwidth.W))
27                 val out1 = Valid (UInt(bitwidth.W))
28                 val out2 = Valid (UInt(bitwidth.W))
29                 val out3 = Valid (UInt(bitwidth.W))
30   })
31     val CrossbarSwitch00 = Module(new CrossbarSwitch(bitwidth))
32     val CrossbarSwitch01 = Module(new CrossbarSwitch(bitwidth))
33     val CrossbarSwitch10 = Module(new CrossbarSwitch(bitwidth))
34     val CrossbarSwitch11 = Module(new CrossbarSwitch(bitwidth))
35
36     val counter = RegInit(0.U(2.W))
37     when(counter === 3.U){
38      counter := 0.U
39      }.otherwise{
40      counter := counter + 1.U
41      }
42
43     io.out_TDMASlot := counter
44
45     CrossbarSwitch10.io.in_a <> io.in0
46     CrossbarSwitch10.io.in_b <> io.in1
47     CrossbarSwitch11.io.in_a <> io.in2
48     CrossbarSwitch11.io.in_b <> io.in3
49     CrossbarSwitch10.io.in_s := counter(1)
50     CrossbarSwitch11.io.in_s := counter(1)
51
52     CrossbarSwitch00.in_a <> CrossbarSwitch10.out_c
53     CrossbarSwitch00.in_b <> CrossbarSwitch11.out_c
54     CrossbarSwitch01.in_a <> CrossbarSwitch10.out_d
55     CrossbarSwitch01.in_b <> CrossbarSwitch11.out_d
56     CrossbarSwitch00.io.in_s := counter(0)
57     CrossbarSwitch01.io.in_s := counter(0)
58
59     io.out0 <> CrossbarSwitch00.io.out_c
60     io.out1 <> CrossbarSwitch00.io.out_d
61     io.out2 <> CrossbarSwitch01.io.out_c
62     io.out3 <> CrossbarSwitch01.io.out_d
63 }
```

Figure 7-4. 4-port TDMA-MIN NoC Chisel implementation.

For our NoC implementation (Figure 7-4), we first define a module to construct the crossbar switch (lines 1-17). The module's interface (lines 2-7) defines five ports (i.e. in_s, in_a, in_b, out_c, and out_d) wrapped in a bundle and in turn wrapped in the IO() method. The Input and Output keywords represent primitive port constructors that define the directions of the ports. Within the Input and Output constructors are the port type and the bit width. For example, the in_s port (line 3), corresponding to the slot input, is unsigned Int (UInt) type with one-bit width (1.W where W denotes width). The Valid constructor in the out_c and out_d port definitions is a built-in bundle containing one output bool signal and one bits signal whose type and width can be specified. The additional bool signal can be used to demonstrate if the data on the bits signal is valid. Furthermore, the CrossbarSwitch module uses the bitwidth parameter that can be determined when initializing the module to parameterize the input and output bit width. It is worth noting that the default direction of the ports in the Valid constructor is output. So we use the Flipped function (lines 4-5) to change the ports' direction to input.

The construction part (lines 10-15) in the CrossbarSwitch module simply connects in_a with out_c and connects in_b with out_d when the slot input equals 0. Otherwise, the switch uses the cross-connection mode. In addition, Chisel uses the val keyword to define a variable and utilizes the compiler to infer the variable type. And the interface variable can use .XX to access the port filed. Furthermore, Chisel generally uses the := operator to connect two ports or wires with the input on the right and the output on the left, while the <> operator provides a syntax-sugar to connect two bundles when they have identical port members but different directions. For example, on line 11, the bool and bits signals in in_a will be connected to the bool and bits signals of in out_c, respectively.

Next, we define a TDMAMIN44 module to represent the 4-port TDMA-MIN NoC. The module's interface contains four input ports and four output ports (lines 22 to 29) that are connected to the crossbar switches. In addition, the TDMA-MIN NoC needs a slot count output (line 21) to inform the network interfaces when to send the data. In the construction part, we first define a 2-bit register with an initial value of 0 (0.U where U denotes unsigned Int) for the TMDA slot counter. The register's value will increase by one in every cycle and return to 0 when it reaches 3 (lines 37 to 41).

The counter register outputs its value through the out_TDMASlot port. The counter's two bits also separately serve as the slot input for the four Crossbar modules initialized as sub-modules (lines 31 to 34)within the TDMAMIN44 module. Chisel can extract specific bits of the register

by attaching the bit number as a suffix (line 49). At last, the TDMA-MIN NoC module inputs are connected with the inputs of the CrossbarSwitch10 and CrossbarSwitch11 (lines 45-48), while the outputs of the CrossbarSwitch00 and CrossbarSwitch00 serves as the NoC outputs (lines 59-62). And these four switches are also interconnected with each other in a Banyan type.

7.2.2 **Network Interface Implementation**

The network interface (NI), as shown in Figure 2-5, has two main functions. On the one hand, the NI needs to decide when to send the data according to the data's destination port and the current connectivity of the TDMA-MIN NoC. On the other hand, it serves as a buffer between the TDMA-MIN NoC and cores. The core sends a single word every time to the NI and informs the NI to which port the data should be sent. Then the NI sends the data to one input port of the NoC at the correct TDMA slot and tells the core that it is ready for the next word. In the opposite direction, the NI receives data from one output port of the NoC and pass-through the data to the core.

In terms of the first function, we leverage the equation of D = Mirror(I) $\oplus$ S described in Section 2.6 to calculate the current destination port for a specific NI in different TDMA slots. The NI module (Figure 7-5) uses a variable of destPortAddr (line 19) to represent the calculation result. The Reverse() function is a Chisel built-in function that can output a "bit-mirrored" value of the input. For example, it can take "1011" as the input and output "1101". In our case, the input of Reverse() is the number (PortAddr) of the NoC port to which the NI is connected. And, the PortAddr serves as a parameter (line 1) of the module and is specified when the module is initialized. The denotation of 2.W converts the PortAddr to a 2-bit number as we only have four ports in the NoC. In addition, the ^ operator denotes the XOR operation, and the io.in_TDMASlot is the current value of the slot counter received from the NoC.

Regarding the data transmission function, the NI module first defines two inputs (lines 5 to 6) at the core side: in_data and in_tagertPort. The in_targertPort input is the data's target port, while the in_data utilizes a built-in Decoupled bundle that contains three wires: an input bool signal named valid, an output bool signal (ready) used to denote if the receiver is receiving the data, and an output bits port to convey the data. As we want to input data to the NI module, a Flipped function is used to change the direction of the port members within the Decoupled bundle. The NI module also defines three registers (lines 15 to 17) to store the input data and

indicate the module's status. In particular, the targetPort and regData registers store the target port number and the input data in the in_data.bits signal, respectively. And, if the inDataFlag register's value equals 1, it represents that the NI has unsent data and cannot receive the new data.

```
1  class NI(val PortAddr :Int, val bitwidth: Int) extends Module {
2     val io = IO(new Bundle {
3     val in_TDMASlot  = Input(UInt(2.W))
4
5     val in_targetport = Input(UInt(bitwidth.W))
6     val in_data  = Flipped(Decoupled (UInt(bitwidth.W)))
7     val out_data = Valid (UInt(bitwidth.W))
8     val out_sourceport = Output(UInt(bitwidth.W))
9
10    val out_payload = Valid (UInt(bitwidth.W))
11    val in_recieve = Flipped(Valid (UInt(bitwidth.W)))
12
13   })
14
15    val targetPort = RegInit(0.U(bitwidth.W))
16    val inDataFlag = RegInit(0.U(1.W))
17    val regData = RegInit(0.U(bitwidth.W))
18
19    val destPortAddr = Reverse(PortAddr.U(2.W)) ^ io.in_TDMASlot
20
21    when(inDataFlag === 0.U && in_data.valid === true.B ){
22              inDataFlag := 1.U
23              regData := io.in_data.bits
24              targetPort := io.in_targetport
25              in_data.ready := true.B
26    }.otherwise{
27       in_data.ready := false.B
28    }
29
30   when(inDataFlag === 1.U && destPortAddr === targetPort){
31              io.out_payload.valid := true.B
32              io.out_payload.bits := regData
33   }.otherwise{
34              io.out_payload.valid := false.B
35              io.out_payload.bits := 0.U
36   }
37
38   when(out_payload.valid === true.B){
39              inDataFlag := 0.U
40   }
41
42   out_data <> in_recieve
43
44   when(in_recieve.valid === true.B){
45      io.out_sourceport :=  Cat(0.U(30.W), destPortAddr)
46   }.otherwise{
47      io.out_sourceport := 0.U
48   }
49
50 }
```

Figure 7-5. Network Interface Chisel implementation.

When a core is sending the data, the in_data.valid signal is set. If the inDataFlag's value is 0 at the same time (line 21), then the NI module will fill the regData register and the targetPort register (lines 23 to 24). In addition, the module will also change the inDataFlag's value to 1 (line 22) and the value of the io.in_data.ready to be true (line 25) to inform the core that the data is being received. In the next cycle, when the value change of the registers takes effect, if the destination port is the same as the data's target port (line 30), the module will set the valid signal of the io.out_payload to be true and send the regData's value to the NoC via the bits port (lines 31 to 32). And, the inDataFlag's value will be changed back to 0 in the next cycle.

For data receiving procedures, the NI module simply defines two Valid bundles on both the NoC and core sides and connect them together (line 42). When the valid signal of the in_receive bundle from NoC is asserted, the module also needs to inform the core from which port the data comes (lines 44 to 45). The built-in Cat() function concatenates the 2-bit destination port with a 30-bit Zero to fulfil the bit width requirement.

7.3. Application-Specific Processors

The two application-specific processors (ASPs) are responsible for accelerating the computation in the image detection algorithm. The ASP1 corresponds to the characteristic vector extraction module and generate a characteristic vector (CV) from the binary image, while the ASP2 calculates the distances between the CV of the under-test image and the CVs of neighbours and sorts these distances in ascending order. The last step of the item recognition data module is to calculate the majority type in the first k items. The C implementation of the last step remains unchanged to give the flexibility to choose the k value. The next sections will introduce the Chisel implementation of these two ASPs. Compared to the software implementation, we make slight changes to the logic to facilitate the hardware implementation.

7.3.1 **Characteristic Vector Extraction ASP Implementation**

Once the binarization data module converts an item image into a binary image, it generates a one-dimensional char array with each element's value being either 0 or 1. The main task of the characteristic vector extraction ASP (ASP1) is to process this array and get a characteristic vector with five features. When the Rocket core calls this ASP, the DMA device will transport

the array from the main memory to the ASP1. The main data processing procedures in ASP1 are shown in Figure 7-6. The first step of ASP1 is to receive and store these data. Concretely, as we use the 32-bit Rocket core, the bit width of the NoC is also 32-bit. This means the ASP1 will receive from the NoC a 32-bit payload from the NoC at a time, and the payload consists of four bytes, with only the first bit in every byte indicating the value. So when the ASP1 receives the first packet, it first extracts the 0^{th}, 8^{th}, 16^{th}, and 24^{th} bits and stores these bits in the first four bits of a 160-bit register. After 40 rounds of receiving, the register will be fully filled, and its value will be in turn stored in a piece of local memory that has 160 elements with a 160-bit width. Thus, the fully filled local memory can represent an entire 160 * 160 binary image.

Furthermore, during the first step, ASP1 also generates the values of the vertical and horizontal projection of the binary image. As discussed in Section 3.4.3, these two values are stored in two 160-bit registers and used to calculate the edges of the item within the image. In terms of the horizontal projection, once the register keeping the pixel values of one row is fully filled, ASP1 applies a bit-wire OR function on the row register's value to check if there is at least one valid pixel (whose value equals 1) on this row. This function will first perform the logical OR on the first two bits of the value and then OR the result with the third bit. It will repeat this process until the last bit. Thus, the result is either 1 or 0 and can be stored in one of the bits in the horizontal projection register. At the same, ASP1 also applies the bitwise OR operation on every bit in the row register and the same bit in the vertical projection register. And, the result is stored back in the vertical projection register. Thus, after all bytes of the binary image received by ASP1, we can get the vertical projection.

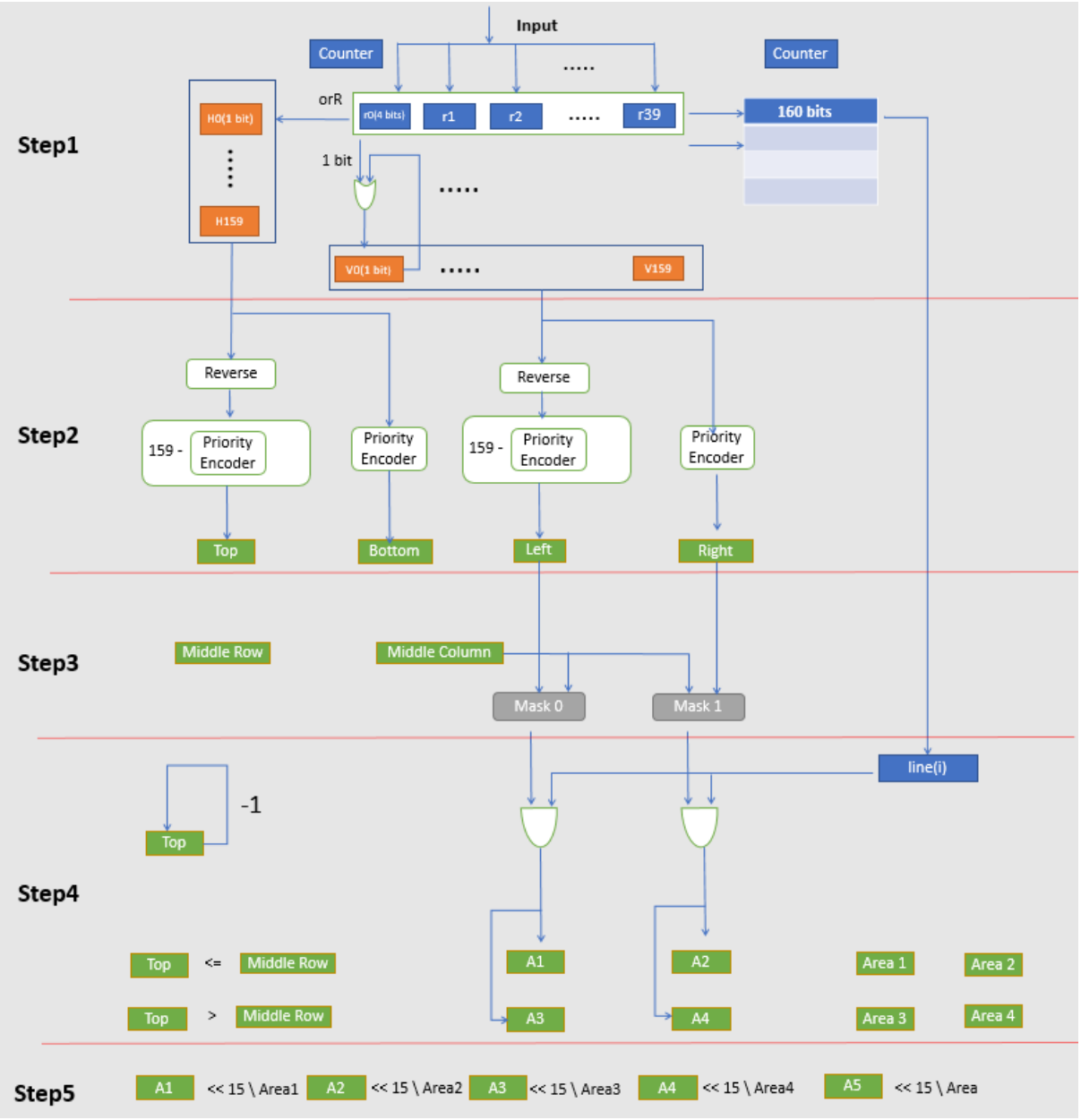


Figure 7-6. Main data processing procedures in ASP1.

In the second step, ASP1 leverages the values in horizontal and vertical projection registers to find out the edge positions of the item (i.e. the values of the Top, Bottom, Left and Right). As discussed in section 3.4.3, we make three assumptions about the image: (1) The item in the picture is continuous; (2) There is at least a one-pixel distance between the item and the picture edges; (3) There is no noise in the background in the image. Thus, from the least significant bit (LSB) to the most significant bit (MSB), the position of the first non-zero value in the horizontal vector is the Top value. In Chisel, the PriorityEncoder function can return the position of the least significant 1 of the input. So ASP1 can easily get the Top value by applying

this function to the value of the horizontal projection register. Similarly, the Bottom value is the first non-zero value from MSB to LSB in the register. ASP1 also uses the PriorityEncoder function to get this position, but the input is the reversed (mirrored) value of the horizontal projection register. And if the Bottom position in the reversed value is x and the position in the original value is y, then we have the relationship of x + y = 159. So ASP1 uses 159 to subtract the result getting from the PriorityEncoder function to get the Bottom value. The same process can be performed on the value of the vertical projection registers to get the Left and Right positions. At last, these four positions will be stored in four 8-bit registers.

The third step makes preparations for step 4 that divides the target area (i.e. the area surrounded by the edges) into four subareas and calculates the sum of the valid pixels in each area. We leverage two masks to achieve the functions in step 4. The first mask (mask 0) is a 160-bit value whose bits only between the Left and middle-column are set. In step 3, ASP1 first calculates the Middle-row and Middle-column based on the values of Top, Bottom, Left, and Right. Then, it leverages a GenerateMask module with the Left and Middle-column as input to generate the mask0. Next, in step 4, ASP1 can perform the OR operation on this mask and every row between the Top and bottom to filter out the valid pixels on the left side in the target area. And then, a built-in function, Popcount (line 55, Figure 7-7), is applied to the result to get the number of hot (=1) bits. And during this process, the Top register value is decreased by one and serves as a counter to distinguish the upper and lower half zones. In particular, when it is larger or equals the middle-row, the results calculated with mask0 and mask0 are accumulated in the registers of A1 and A2, respectively. Otherwise, the A3 and A4 registers are responsible for the accumulation. In addition, in step 3, ASP1 also keeps in two registers the widths between the Right and the middle column and between the middle column and the Left. And during the accumulation of A1, A2, A3 and A4, four registers of Area1, Area2, Area3, and Area4 will also increase by the corresponding width for each row in the target area to get the total number of pixels in each subarea.

```
1  class GenerateMask extends Module {
2     val io = IO(new Bundle {
3     val in0 = Input(UInt(8.W))
4     val in1 = Input(UInt(8.W))
5     val out = Output(UInt(160.W))
6     })
7              val SetBound = (1.U << (io.in1 - io.in0 + 1.U)) - 1.U
8              io.out := SetBound  << (159.U-io.in1)
9
10 }
11 class extractCharac(val bitwidth: Int) extends Module {
12    val io = IO(new Bundle {
```

```
13                          val in_sourceport = Input(UInt(bitwidth.W))
14                          val out_targetport = Output(UInt(bitwidth.W))
15                          val in_fromNI  = Flipped(Valid (UInt(bitwidth.W)))
16                          val out_toNI = Decoupled (UInt(bitwidth.W))
17    })
18
19  val s_idle :: s_receiving :: s_edgeDetecting :: s_maskGenerating :: s_caculating :: s_extracting :: s_sending :: Nil =
20  Enum(7)
21  …
22    val wordCount = Counter(40)
23    val lineCount = Counter(160)
24    val Receive = RegInit(VecInit(Seq.fill(40)(0.U(4.W))))
25    ...
26
27    when(state === s_idle || state === s_receiving ){
28                          ...
29                          when(wordCount.value === 40.U){
30                                  mem(lineCount.value) := receiveWireConnection
31                                  horizontalProjection(lineCount.value) := receiveWireConnection.orR
32                                  for(i <- (0 to 159)){
33                                    verticalProjection(i) :=  receiveWireConnection(i) | verticalProjection(i)
34                                  }
35                                  lineCount.inc()
36                                  wordCount.inc()
37                          }
38    }
39
40    when(state === s_edgeDetecting){
41              top := 159.U - PriorityEncoder(Reverse(horizontalWire))
42              bottom := PriorityEncoder(horizontalWire)
43              left := 159.U - PriorityEncoder(Reverse(verticalWire))
44              right := PriorityEncoder(verticalWire)
45    }
46
47    val generateMask0 = Module(new GenerateMask)
48    val generateMask1 = Module(new GenerateMask
49    generateMask0.io.in0 := left
50    generateMask0.io.in1 := midCol
51    mask0 := generateMask0.io.out
52    ...
53    when(state === s_caculating){
54                          when(top >= midRow){
55                                  A1 := Popcount(mem(top) & mask0) + A1
56                                  A3 := Popcount(mem(top) & mask1) + A3
57                                  Area1 := Area1 + A1Width
58                                  Area3 := Area3 + A1Width
59                                  top := top - 1.U
60                                  lineCount.inc()
61                          }
62  …
63  }
64
65    val feature = Reg(VecInit(Seq.fill(5)(0.U(32.W))))
66              when(state === s_extracting){
67                 feature(0) := (A1 << 15.U) / Area1
68                 ...
69              }
70    ...
71  }
```

Figure 7-7. ASP1 Chisel implementation.

Next, we calculate the ratios of the valid pixels in each area to get the features in the characteristic vector in step 5. Unlike the software implementation, ASP1 will first left shift the dividends by 15 bits to avoid the floating-point operation. In ASP2, we will also perform this operation on the neighbours, so the final order of distances between the under-test item and the neighbours will remain unchanged. In addition, as an integer in C occupies 4 bytes and the largest number of the dividends, A5 (the number of valid pixels in the whole target area), is less than 2^{15}, so this operation will not cause overflow. After the calculation, we can get a characteristic vector with five integers. The last step is to send these features back to the source port from where the inputs come.

In the Chisel implementation (Figure 7-7), we leverage many Chisel built-in functions to facilitate the implementation. In the interface part (lines 12 to 16), the extractCharac module defines four input and output ports and bundles that can be connected with the NI. Next, the module defines different states (line 19) corresponding to different steps. And the changing of the states is based on the specific counters and conditions. In the first step, the module utilizes the bit-reduction method, .orR (line 31), to get the one-bit horizontal projection for each row's pixels. Then, step 2 uses the PriorityEncoder and Reverse functions (line 41) to conveniently acquire the positions of the target area's edges. Next, in step 3, we create a separate module, GenerateMask (lines 1-10), to generate the masks. Concretely, this module takes the boundaries (in0 and in1) of the consecutive hot (=1) bits as inputs. Then it right shifts '1' with (in1 – in0) bits and next minus 1 to build up the consecutive hot bits (line 7). At last, it right shifts the result (line 8) to generate the mask. In the next step, ASP1 uses the PopCounter function (line 55) to get the number of valid pixels. These built-in functions largely simply the design on the RTL level.

7.3.2 **Item Recognition ASP Implementation**

The item recognition data module consists of three subfunctions: distance calculation, distance sorting, and majority type selection. In the item recognition ASP (ASP2), we only implement the first two functions in hardware because the software implementation of the last function can allow users to freely choose the value of k that specifies the selection scope.

```
1  class ItemRecogition(val bitwidth: Int) extends Module {
2      val io = IO(new Bundle{...})
3
4     val s_idle :: s_receiving :: s_caculating :: s_sorting :: s_sending :: Nil = Enum(5)
5     val feature0 = RegInit(VecInit(Array(18929.U, 19744.U, 19763.U, 19054.U, 19372.U)))
6     ...
7     val characVector =  RegInit(VecInit(Seq.fill(5)(0.U(32.W))))
8     val distance =  RegInit(VecInit(Seq.fill(12)(0.U(32.W))))
9     val distanceResult =  RegInit(VecInit(Seq.fill(12)(0.U(32.W))))
10
11
12    when(state === s_caculating){
13              distance(0) = (characVector zip feature0).map{(a:UInt, b:UInt) => (a.asSInt - b.asSInt)*(a.asSInt -
14 b.asSInt)}.reduce(_+_)
15                ...
16     }
17
18    val itemType = RegInit(VecInitSeq.fill(12)(0.U(8.W)))
19    ..
20    val typeSwapStep0 = Wire(VecInit(Seq.fill(12)(0.U(8.W))))
21    ...
22    val distanceStep0 = Wire(VecInit(Seq.fill(12)(0.S(32.W))))
23    ...
24    var even = Array(0, 2, 4, 6, 8, 10)
25    var odd = Array( 1, 3, 5, 7, 9)
26
27   when(state === s_sorting){
28             when(stageCounter === 0.U && stageCounter === 2.U){
29               //step0
30                       for(i <- even){
31                                 when(distance(i) < distance(i+1)){
32                                           distanceStep0(i) := distance(i)
33                                           distanceStep0(i+1) := distance(i+1)
34                                           typeSwapStep0(i) := itemType(i)
35                                           typeSwapStep0(i+1) := itemType(i+1)
36
37                                 }.otherwise{
38                                           distanceStep0(i+1) := distance(i)
39                                           distanceStep0(i) := distance(i+1)
40                                           typeSwapStep0(i+1) := itemType(i)
41                                           typeSwapStep0(i) := itemType(i+1)
42                                 }
43                           ...
44                         }
45                         ...
46             }
47    }
48 }
```

Figure 7-8. ASP2 Chisel implementation.

The first function, distance calculation, involves 12 characteristic vectors of the neighbours with six CVs in each category, which are acquired in the training stage. ASP2 presets these values in the registers (line 5, Figure 7-8) and will start the calculation (lines 12 to 16) after receiving the CV of the under-test item from the NoC. Compared to the software implementation, all CVs stored in the registers are acquired by left shifting the dividends by 15 bits. And it is worth noting that the features in the CVs involved in the distance calculation

should be represented in signed integers due to the possible negative result when doing the minus operation. Thus, ASP2 uses the .asSInt method (line 13) to convert all the features into signed integers. In addition, AP2 employs a series of functions to implement the distance calculation (lines 13 to 14). Firstly, the zip function can merge two collections and return a new collection of 2-tuple elements from both collections. AP2 applies this function on two CVs and then perform a map method on the new collection. The map method takes a function as the parameter and applies this function to the input to generate another collection. Concretely, if each element of the input (the new collection) is a pair of unsigned integers, the map method will return a collection of the squares of the difference between these two integers. At last, the reduce function reduces the result (the collection) into a number. And, the reducing action can also be defined by a function. In this case, we simply calculate the sum of all elements in the collection to get the distance between the under-test item and the neighbour.

The other function of ASP2 involves a hardware bubble sort implementation (lines 27 to 46) to sort the twelve distances in ascending order. In order to improve the performance, we leverage a parallel variant of the bubble sort algorithm, the odd-even transposition sort [61]. It contains two phases: the even phase and the odd phase, as shown in Figure 7-9. In the even phase, every even indexed item (0 based indexing) is compared with the adjacent element, and if a pair is not in the ascending order, these two items are switched. Similarly, the odd phase compares every odd indexed item with the next one and adjust their order accordingly. The even and odd phases are alternate during the sorting process, and twelve steps are required to get the correct order of the distances. In the Chisel implementation, these twelve steps are divided into three stages, with each stage containing two even phases and two odd phases. At the first stage, the distances in the registers (distance(i)) follow the rules to swap their positions. The intermediate results are saved in the wire constructs (line 22), while the final results are stored in another series of registers (distanceResult(i)). Then, the next stage starts from distanceResult(i) to distances(i), and the last stage from distances(i) to distanceResult(i). Finally, the sorted distances are stored in the distanceResult(i) register.

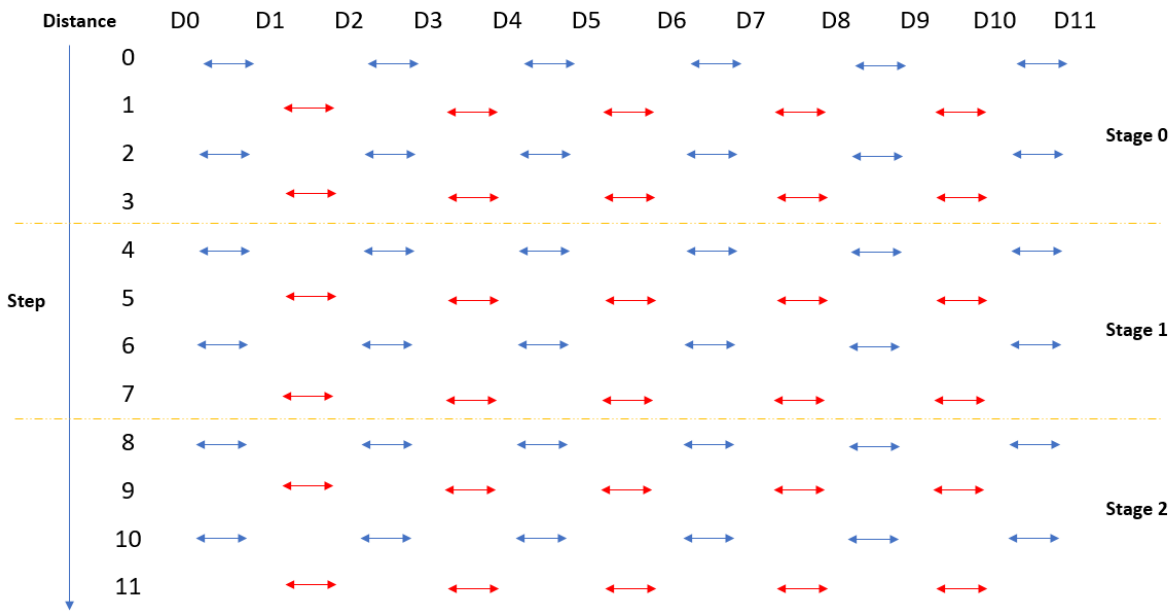


Figure 7-9. Sorting distances with Odd-Even Transposition Sort.

During the distance sorting procedures, a series of numbers (line 20, Figure 7-8) containing the type indexes (0 for cylinder, 1 for cone) is also sorted accordingly. Thus, once the distances are sorted, the type indexes can represent the types of the neighbours whose distances with the under-test item are in ascending order (lines 34 to 35, lines 40 to 41). ASP2 only returns these indexes back to the NoC, and in turn, to the main memory through the DMA device.

7.4. DMA Device

The DMA device is based on the TileLink bus protocol, which is designed for a System-on-Chip to provide both high-through and low-latency transfers. This section will first give a brief introduction of the TileLink protocol and its relationship with Diplomacy in Chipyard. Then, we illustrate the Chisel implementation of the DMA device that contains a DMA controller, a DMA reader, and a DMA writer.

7.4.1 **TileLink Overview**

TileLink enables the chip-scale interconnections between different system components such as general-purpose processors, coprocessors, accelerators, DMA engines, and MMIO peripherals. An example of a TileLink topology [62] is shown in Figure 7-10.

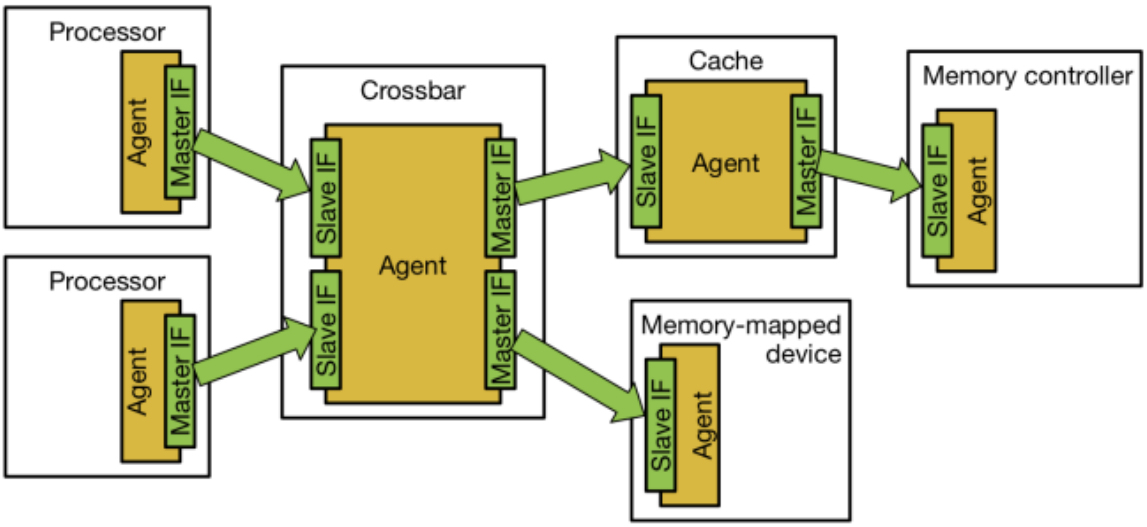


Figure 7-10. Example of a TileLink network topology.[62]

Concretely, every two hardware modules are connected over a link. Residing on the two ends of one link is a pair of TileLink agents containing one master interface and one slave interface. Within a link, there are two or five directional channels (as shown in Figure 7-11) depending on which subset of the protocol the agents support. In particular, TileLink has three subsets: TileLink Uncached Lightweight (TL-UL), TileLink Uncached Heavyweight (TL-UH), and TileLink Cached (TL-C). TL-UL contains two channels (i.e. channel A and D) and only support memory read and write operations of single words. In particular, the master interface sends a request message on channel A to the slave interface, and the slave interface gives a response through channel D. TL-UH also only has channel A and D, but it supports the multi-beats message, atomic operations, and hint operations. At last, TL-C contains all five channels and supports additional coherent cache block transfers compared to TL-UH. Furthermore, these channels have strict priorities (in order of increasing priority from A to E) to avoid deadlock. In our DMA device, we only use channels A and D to read and write burst data to the main memory.

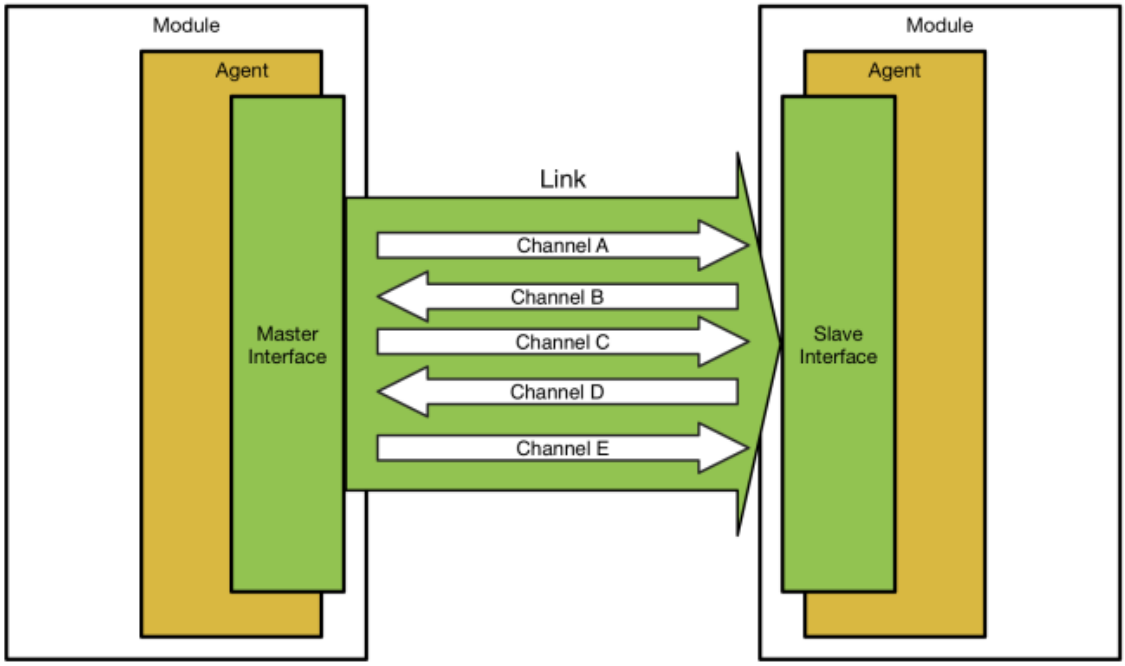


Figure 7-11. The five channels that comprise a TileLink link. [62]

In Chipyard, Diplomacy divides the elaboration of Chisel designs into two stages. At the first stage, Diplomacy builds up a directed acyclic graph for the SoC with nodes representing agents insides SoC components and edges standing for links between agents. And, these nodes come in different types. For example, client nodes contain a master interface that can initiate a TileLink transaction on channel A and receive a response from channel D. Manager nodes support slave interfaces to receive requests and give replies back. Register nodes can also handle requests from clients nodes, but they are specially designed for MMIO devices to facilitate the construction of control and status registers. It is worth noting that an agent in the TileLink protocol can only have one interface, while a hardware module can have multiple agents. Thus, a system component may contain multiple nodes. In our DMA device, we create a register node for the DMA controller and two clients nodes for the DMA reader and writer. In addition, Diplomacy also provides many useful methods to construct TileLink messages and check the channel status, which will be introduced in the following sections.

7.4.2 **DMA Controller**

Our DMA device contains three modules: a DMA controller, a DMA reader, and a DMA writer. The DMA controller is responsible for receiving control parameters from the Rocket core and indicating the device status through the MMIO registers. Furthermore, the DMA controller will pass the parameters to the DMA reader, DMA writer, and the peripheral glue logic. Thus, the DMA controller also serves as the peripheral interface and should receive the addresses and sizes of both the input and output data of the peripheral. In addition, the Rocket core needs to specify which ASP (or which port on the NoC) it calls.

In the Chisel implementation (Figure 7-12), we first define the input and output bundles (lines 1-8) for the controller module. On the one hand, the output ports are used to transfer the parameters into the DMA reader and writer and the peripheral glue logic. In particular, the wires of dataAddress and dataSize (lines 2-3) connect the DMA reader to pass the address and size of input data, while the resultAddress and resultSize (lines 4-5) specify where the DMA writer should write back the computation results. The value of the port (line 6) is stored in a port register in the peripheral to indicate which ASP will be used. On the other hand, the input bundles of ifComplete (line 7) receive signals from the DMA writer to tell if the result writing is finished. Furthermore, the valid and ready signals inside the ifComplete and dataSize bundles are used to set up the status of the entire peripheral.

Next, the DMAControllerModule (lines 10-46) trait that extends the HasRegMap class is created to set up the memory-mapped registers. The module simply defines a bunch of wires and general registers (lines 15-21) to connect the controller input and output port. Then, it leverages the regmap method (line 32) to convert these wires and general registers into memory-mapped registers. For a specific register, it first assigns the offset address and uses the Regfield function to specify the access attribute and the bit width. For example, Regfield.w(param.width, dataAddress) (line 42) defines the dataAddress register as a plain write-only register (.w) with the assigned bit width. And, the decoupled interface signal dataSize (line 44) is also associated with a write-only MMIO register, causing dataSize.valid to be set when the register is written. This valid signal is in turn transferred into the peripheral glue logic and DMA reader to change the peripheral state and trigger the data reading, respectively. Similarly, when the read-only register (line 46) is read, the ifComplete.ready signal is asserted to indicate that the Rocket core has released that the data writing is finished. And, this asserted signal also triggers the state of the peripheral to be idle. At last, the signals

of ifComplete.valid and dataSize.ready (line 30) compose the status register. When the peripheral's state is idle, the dataSize.ready is high, and the ifComplete.valid is low. However, if the peripheral state is done, the ifComplete.valid is high, and the dataSize.ready is low. In other states, both these signals are low.

```
1
2   trait DMAControlBundle extends Bundle{
3               val dataAddress = Output(UInt(32.W))
4               val dataSize = Decoupled(UInt(32.W))
5               val resultAddress  = Output(UInt(32.W))
6               val resultSize = Output(UInt(32.W))
7               val port = Output(UInt(32.W))
8               val ifComplete = Flipped(Decoupled(Bool()))
9   }
10
11  trait DMAControlModule extends HasRegMap{
12              val io: DMAControlBundle
13              implicit val p: Parameters
14              def params: DMAControlParams
15
16              val port = Reg(UInt(params.width.W))
17              val dataAddress = Reg(UInt(params.width.W))
18              val dataSize = Wire(new DecoupledIO(UInt(params.width.W)))
19              val resultAddress  = Reg(UInt(params.width.W))
20              val resultSize = Reg(UInt(params.width.W))
21              val ifComplete = Wire(new DecoupledIO(UInt(params.width.W)))
22              val status = Wire(UInt(2.W))
23
24              io.port := port
25              io.dataAddress := dataAddress
26              io.dataSize <> dataSize
27              ifCompelete <> io.ifCompelete
28              io.resultAddress := resultAddress
29              io.resultSize := resultSize
30
31              status := Cat(io.dataSize.ready, io.ifCompelete.valid)
32
33             regmap(
34               0x00 -> Seq(
35               RegField.r(2, status)),
36             0x04 -> Seq(
37               RegField.w(params.width, port)),
38             0x08 -> Seq(
39               RegField.w(params.width, resultAddress)),
40             0x0C -> Seq(
41               RegField.w(params.width, resultSize)),
42             0x10 -> Seq(
43               RegField.w(params.width, dataAddress)),
44             0x14 -> Seq(
45               RegField.w(params.width, dataSize)),
46             0x18 -> Seq(
47               RegField.r(params.width, ifComplete)))
48  }
49
50  class DMAController(params: myPeripherialKey, beatBytes: Int)(implicit p: Parameters) extends TLRegisterRouter(
51              params.address, "dmacontroller", Seq("UoA,dmacontroller"), beatBytes=beatBytes)(
52              new TLRegBundle(params,_) with DMAControlBundle)(
53              new TLRegModule(params,_,_) with DMAControlModule)}
```

Figure 7-12. DMA controller Chisel implementation.

The last step is to create a register node (lines 49-53) for the DMA controller. Diplomacy provides a TLRouter class to facilitate this procedure. We can simply create a DMAController class by extending TLRouter and pass three sets of parameters to complete the construction of the MMIO DMA controller. The first set of parameters specifies the base address of the memory-mapped register, the information of the device tree entry, and the beat bytes, while the last two sets of parameters are the IO bundle constructor and module constructor, respectively. These constructors take the concrete IO bundle and module design as input and handle the interconnect protocols.

7.4.3 **DMA Reader**

The DMA reader is in charge of retrieving data from the main memory and sending the data to a buffer in the peripheral. The implementation of the DMA Reader (Figure 7-13) follows the typical two-stage elaboration pattern. It first creates a logical TileLink node in a lazy module (line 1) and then initiates a lazymoduleImp module (line 3) that achieves the physical design. Specifically, in the DMAReader lazy module, the TLhelper object (line 2) provided by Diplomacy is used to create a client node for the DMA reader. The sourceId argument defines source identifiers that this client will use to send requests. If the value is (0, 4), it means the client can send up to four requests with distinct values at a time. In this case, the client can only send one request in flight.

For the concrete implementation (lines 3 to 45), the DMAReader first creates the interfaces (lines 4 to 8) to receive parameters from the DMA controller, like the address and size of the target data. In addition, the data decoupled bundle is used to transfer data to the buffer, while the output of the comp bundle indicates if the DMA reader finishes the reading process.

Next, the lazymoduleImp module calls the node.out method (line 10) to get a pair that contains a list of bundles (mem) in a TileLink link and an edge object (edge) that represents the edge in the Diplomacy graph. On the one hand, the list of bundles consists of hardware decoupled bundles for channels from A to E. For TL-UL and TL-UW, it only comprises the bundles of channel A and channel D. In particular, channel A signals (as shown in Table 7-1) contains a code signal to specify the request operations. The basic types are the get and put operations to read and write data, respectively. The size and address signal defines the target data's size and

start address, while the data signal conveys the payload if the operation is the put. At last, channel A use valid and ready signals to control the transaction progress. Channel D has similar signals (Table 7-2), except that these signals are used to respond to the requests from channel A. On the other hand, the edge object can provide some convenient functions to construct TileLink transactions and check the status of each channel.

```
1  class DMAReader(implicit p:Parameters, width:Int) extends LazyModule{
2          val node = TLHelper.makeClientNode(name="dmareader", sourceId=IdRange(0,1))
3          lazy val module = new LazyModuleImp(this){
4                  val io = IO(new Bundle{
5                          val dataAddress = Input(UInt(width.W))
6                          val dataSize = Flipped(Valid(UInt(width.W)))
7                          val data = Decoupled(UInt(32.W))
8                          val comp = Decoupled(Bool())
9                  })
10                 val (mem, edge) = node.out(0)
11                 val blockBytes = p(CacheBlockBytes)
12                 ...
13
14                 when (state === s_idle && io.dataSize.valid ) {
15                         addr := io.dataAddress
16                         bytesLeft := io.dataSize.bits
17                         state := s_read
18                 }
19
20                 mem.a.bits := edge.Get(
21                         fromSource = 0.U,
22                         toAddress = address,
23                         lgSize = log2Ceil(blockBytes).U)._2
24                 mem.a.valid := state === s_read
25
26                 when (edge.done(mem.a)) {
27                         addr := addr + blockBytes.U
28                         bytesLeft := bytesLeft - blockBytes.U
29                         state := s_resp
30                 }
31
32                 io.data.bits := mem.d.bits.data
33                 io.data.valid := mem.d.bits.valid && state === s_resp
34                 mem.d.ready := io.data.ready
35
36                 when(edge.done(mem.d)){
37                         state := Mux(bytesLeft === 0.U, s_done, s_read)
38                 }
39
40                 io.comp.valid := (state===s_done)
41                 io.comp.bits := (state===s_done)
42
43                 when (io.comp.ready && state===s_done){
44                         state===s_idle
45                 }
46         }
47 }
```

Figure 7-13. DMA reader Chisel implementation.

Signal	Type	Width	Description
a_code	C	3	Operation code. Identifies the type of message carried by the channel.
a_param	C	3	Parameter code. Meaning depends on a_opcode; specifies a transfer of caching permissions or a sub-opcode.
a_size	C	z	Logarithm of the operation size: 2^n bytes.
a_source	C	*o*	Pre-link master source identifier.
a_address	C	*a*	Target byte address of the operation. Must be aligned to a_size.
a_mask	D	*w*	Byte lane select for messages with data.
a_data	D	*8w*	Data payload for messages with data.
a_corrupt	D	1	The data in this beat is corrupt.
a_valid	V	1	The sender is offering progress on an operation.
a_ready	R	1	The receiver accepted the offered progress.

Table 7-1. Channel A signal description. [62]

Signal	Type	Width	Description
d_code	C	3	Operation code. Identifies the type of message carried by the channel.
d_param	C	2	Parameter code. Meaning depends on d_opcode; specifies permissions to transfer or a sub-opcode.
d_size	C	z	Logarithm of the operation size: 2^n bytes.
d_source	C	*o*	Pre-link master source identifier.
d_sink	C	*i*	Pre-link slave sink identifier.
d_denied	C	1	The slave was unable to service the request.
d_data	D	*8w*	Data payload for messages with data.
d_corrupt	D	1	Corruption was detected in the data payload.
d_valid	V	1	The sender is offering progress on an operation.
d_ready	R	1	The receiver accepted the offered progress.

Table 7-2. Channel D signal description. [62]

At last, in the construction part, the lazymoduleImp module defines the control flow of the DMA reader. Once the module receives the address and the size, it will store these two parameters in the addr and bytesLeft registers (lines 14 to 18), respectively, and the valid signal will trigger the reader to change its state from s_idle to s_read. Then, the module starts the reading procedures with a data request transaction. The request data in one transaction is the same size as the cacheblockBytes, acquired from a parameter query (line 11). And, the edge.get() method (lines 20 to 24) is used to initiate the channel A signal fields: fromSource for the identifier, toAddress for the target byte address, and lgSize for the logarithm of the data size. The method will return a tuple with two elements, and the ._2 method (line 23) is used to

choose the second element, which is a channel A bundle. When the module state is s_read, the valid signal of the channel A bundle is set to indicate that there is a request on Channel A. If the result of edge.done(mem.a) (line 26) is true, it means the opposite side has received the request. And then, the module will accordingly adjust the parameters of the addr and bytesLeft and set the module state to s_resp.

At the s_resp stage, the module is waiting for response data from channel D. The payload signal of channel D is connected to the io.data.bits port to send the response data to the buffer. In addition, Channel D can only send a word every beat. Thus, the response message may last for multiple beats. The edge.done() function (line 36) can also check if the current beat is the last beat. If the response message is finished, the module can choose to send another read request or change the module state to s_done according to the value of bytesLeft. After the DMA reader retrieves all data, the comp signal is set to inform the peripheral to move to the next state.

7.4.4 **DMA Writer**

The DMA writer takes responsibility to write back the computation results to the main memory. Once the peripheral receives the ASPs, it will set the DMA writer's input valid signal to trigger the data writing back procedures. The target address and data size are also accepted from the DMA controller. In addition, the implementation of the DMA writer has a similar structure with the DMA reader, except that we use the edge.put method instead of the get method and replace the output data bundle with an input one receive data from the buffer. And, the signal of io.data.bits is directly used to initiate the payload of channel A in each put request.

7.5. System Integration

Now, as we get all submodules prepared, in this section, we will show how to use the submodules to create the entire peripheral module and how to integrate this peripheral to the single-core platform to generate the multi-core platform.

7.5.1 **Peripheral Module**

The implementation of the peripheral Module (Figure 7-14) also follows the two-stage elaboration design pattern. In its lazy module, the peripheral first initiates the DMA Controller, DMAReader, and DMAWriter (lines 2 to 4). And then, it creates a TileLink identity node (line 6) to converge the client nodes of DMAReader and DMAWriter into one signal node (lines 7-8). Thus, we only need to connect the DMAController's register node to the peripheral bus and this identity node to the front bus.

```
1  class MyPeripherial(params: MyPeripherialParams, beatBytes: Int)(implicit p:Parameters) extends LazyModule{
2            val control = LazyModule(new DMAController(params, beatBytes))
3            val reader = LazyModule(new DMAReader)
4            val writer = LazyModule(new DMAWriter)
5
6            val dmanode = TLIdentityNode()
7            dmanode := reader.node
8            dmanode := writer.node
9            lazy val module = new LazyModuleImp(this){
10
11             val s_idle :: s_read :: s_transfer :: s_resp :: s_write :: s_done :: Nil = Enum(6)
12             ...
13             val myNoC = Module(new TDMAMIN_NI((params.width)))
14             val ASP1 = Module(new extractCharac((params.width)))
15             val ASP2 = Module(new extractCharac((params.width)))
16
17             ASP1.io.in_fromNI <> myNoC.io.out_NI2_data
18             myNoC.io.in_NI2_data <> ASP1.io.out_toNI
19             myNoC.io.in_NI2_targetport := ASP1.io.out_targetport
20             ASP1.io.in_sourceport := myNoC.io.out_NI2_sourceport
21
22             ASP2.io.in_fromNI <> myNoC.io.out_NI3_data
23             myNoC.io.in_NI3_data <> ASP2.io.out_toNI
24            myNoC.io.in_NI3_targetport := ASP2.io.out_targetport
25            ASP2.io.in_sourceport := myNoC.io.out_NI3_sourceport
26             ...
27             val buffer = Queue(Flipped(Decoupled(UInt(params.width.W))), 64)
28             when (state === s_resp || state === s_write ){
29               buffer.io.enq.bits := myNoC.io.out_NI0_data.bits
30               buffer.io.enq.valid := myNoC.io.out_NI0_data.valid
31               bufferEnqState := buffer.io.enq.ready
32               myNoC.io.out_NI0_data.bits writer.module.io.data <> buffer.io.deq
33             }.otherwise{
34               buffer.io.enq <> reader.module.io.data
35               myNoC.io.in_NI0_data <> buffer.io.deq
36             }
37           ..
38           }
39 }
```

Figure 7-14. Peripheral module Chisel implementation.

In the lazymoduleImp module, the peripheral leverages a built-in Queue module (line 27) in Chisel standard library to create a FIFO buffer. It also initiates all other modules, including the 4-port TDMA-MIN NoC with four NIs, ASP1, and ASP2 (lines 13 -15). Then, it connects these

modules properly (lines 17-25). The Queue has 64 entries with a 32-bit width for each entry. Its interfaces contain one flipped decoupled source, one decoupled sink, and one output count port indicating the number of elements in the queue. When the DMA reader is reading data, the Queue's source and sink are connected to the DMA reader's data output and the NI's data input on port 0 of the NoC (lines 34 to 35), respectively. However, when the peripheral is waiting for the result from the ASPs, the data output of the port 0 NI will be the Queue source (lines 29 to 30), and the data input of the DMA writer will be the sink (line 32). As the data output of an NI does not have the ready signal, the peripheral will connect the ready signal of the Queue source to a register (line 31). In addition, ASP1 and ASP2 are connected to port 2 and port 3 on the NoC, the output of the DMA controller are connected to the input of the DMA reader and writer.

The peripheral module has six states (line 11), with each state triggering specific actions. Once the last input register, the dataSize, is written by the Rocket core, the valid signal of the dataSize is set, which in turn trigger the DMA reader to start the data reading. At the same time, the peripheral changes its state from s_idle to s_read. When the DMA reader generates the complete signal through the comp bundle, the peripheral will assert the ready signal of the com bundle to reset the state of the DMA reader, and itself moves to the s_transfer state to allow the remaining data in the buffer to be transferred to the specific ASP. The port register in the module controls to which port the data will be transferred. Then, the zero value of the Queue counter signal triggers the peripheral to be the s_resp state, which makes the Queue adjust its source and sink connection to get ready for receiving computation results from the ASP. The asserted valid signal in the data output bundle of the NI indicates that the result data has arrived. Next, the peripheral will move to the s_write state and set the valid input signal of the DMA writer to start the data writing procedures. Then, the complete signal from the DMA writer set the peripheral state to be s_done, which will change the status register value of the DMA control to inform the Rocket core that the computation result is already in the main memory. At last, the Rocket core reads the com register to assert the ready signal of the ifComplete bundle that triggers the peripheral state to be s_idle.

7.5.2 **System Composition**

Once the peripheral module is ready, we need to create the top-level lazy module trait, lazy module implementation trait, and config fragment for the peripheral to complete the system

composition. Since the peripheral does not need to initiate hardware wires or components on the top level, the lazy module implementation trait is not required.

```
1  trait CanHavePeripheryNoCAndASP { this: BaseSubsystem =>
2    implicit val p: Parameters
3
4    p(NoCKey) .map { k =>
5      val MyNoCAndASP = LazyModule(new MyPeripherial(k.get, pbus.beatBytes)(p))
6      pbus.toVariableWidthSlave(Some("MyPeripherial")) { NoCAndASP.control.node }
7      fbus.fromPort(Some("MyPeripherial"))() :=* MyNoCAndASP.dmanode
8    }
9  }
10
11 case class MyPeripherialParams (
12   address: BigInt = 0x2000,
13   width: Int = 32)
14
15 case object MyPeripherialKey extends Field[OptionNoCParams(None)
16
17 class With MyPeripherial(width: Int) extends Config((site, here, up) => {
18   case MyPeripherialKey => Some(MyPeripherialParams (width = width))
19 })
```

Figure 7-15. Peripheral Lazy module trait and config fragment.

The lazy module trait (lines 1-9, Figure 7-15) tends to contain a parameter query (line 4) that searches the peripheral key in the SoC configuration class. If the key can be found, the trait will initiate the peripheral module (line 5) and connect the peripheral nodes to the buses (lines 6-7). In particular, the DMA Controller register node is connected to the peripheral bus with the port name of "MyPeripherial" (line 6), while the front bus links to the identity node containing the DMA reader and writer client nodes (line 7). At last, this trait should be added into the DigitalTop module to enable the integration of the peripheral and the SoC.

If the lazy module trait cannot find the peripheral key in the SoC configuration class, the SoC still cannot initiate the peripheral. Generally, the key resides in the config fragment (lines 17 to 19) and is mapped to a parameter class. Thus, we need to define a case object as the peripheral key (line 15) and define a case class (lines 11 to 13) with the default values for the base address of memory-mapped registers and the bits width. Next, the config fragment mapping the key to the parameter class is added to the SoC configuration class to complete the configuration. At last, we pass the name of the new configuration class to the make command to generate the simulator of our multi-core execution platform.

Chapter 8. Experiment & Results

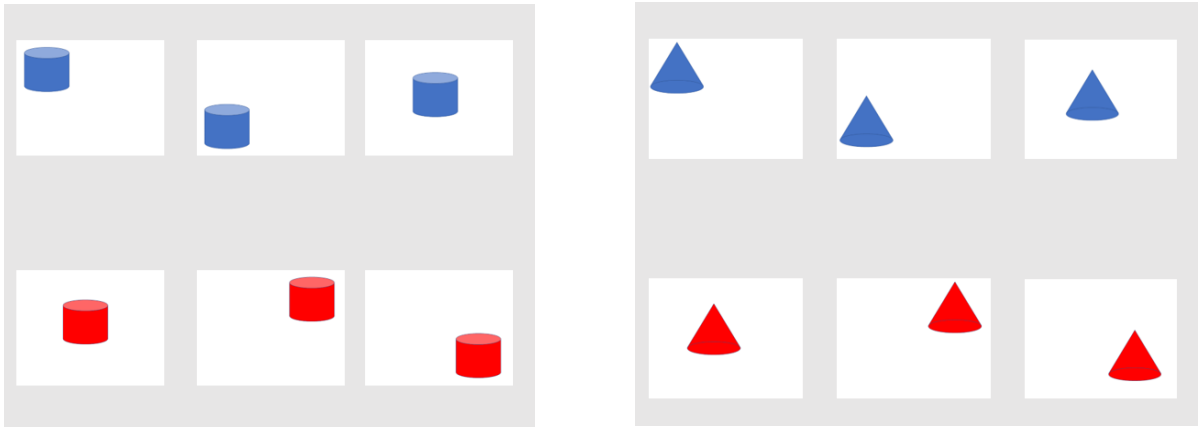
Now, we have built up the simulators of two platforms. In this chapter, we will perform an experiment to compare the performance of two execution platforms. The experiment will compare clock cycles of the Rocket core when executing programs for the same given ticks of SystemGALS. We also introduce preparations for the experiment, including neighbours training, the peripheral software interface implementation, and C programs compilation. In addition, the simulator boot process is explained to demonstrate the interaction mechanism between the host and the simulators.

8.1. Training

The image detection algorithm described in Section 3.4 is based on the neighbours whose types are known. So we select a batch of sample images as neighbours, and the training stage is to extract CVs from these images in two steps. The first step involves converting image files to image char arrays, while the next step leverages the binarization and CV extraction functions to get the neighbours' CVs. And, since we expand the original features into integers in the ASP2, one additional set of integer CVs is generated in the training stage.

8.1.1 **Image Conversion**

The implementation of the image detection algorithm only deals with the corresponding char array of an image instead of the original image file. Thus, we first convert the images into char arrays. The sample item images or neighbours involved in the training stage consist of 12 images with six cylinders and six cones, as shown in Figure 8-1. Each image contains one item, and all images are with white background to eliminate the image noise. And, the items in the image differ in colours and the positions located in the images. As the item colour is not the determining factor, we only choose two colours to simplify the training procedures.



(a) Cylinders **(b) Cones**

Figure 8-1. Sample item images.

In addition, all these images are in the 24-bit bitmap image file (BMP) format, where one pixel is stored in three bytes, with each byte standing for the blue, green, and red components of the pixel. These pixel values are stored in the pixel array of the BMP file. Besides the pixel array, a 24-bit BMP file also contains a bitmap file header and an information header. The file header has a fixed size of 14 bytes and includes the fields like the file size and the start address of the pixel array. The information header contains 54 bytes indicating information, like the width and height of the image. In this thesis, the images of both neighbours and under-test items are with 160*160 pixels.

We leverage a separate C program to convert a 24-bit BMP image into a char array. The C program employs the File Open function to open a BMP image file and use the File Read function to read the entire file byte by byte. And, each byte of the file is stored in a char array without any processing. Therefore, this char array can completely represent the BMP image, including the file header, information header, and pixel array.

8.1.2 **Neighbours Characteristic Vector**

The char array will be then passed to the binarization function to get the reduced binary values of the pixels, as described in section 3.4.2. Next, The CV extraction function follows the steps described in section 3.4.3 to figure out the neighbours' CVs according to the binary pixels. The original set of neighbours' CVs contains 12 vectors, with all five features within each vector being less than 1. At last, we multiply all features by a factor of 2^{15} to get an expanded set with all integer features.

8.2. Peripheral Software Interface

Data modules leverage a software API to call specific ASPs instead of the software implementation (Figure 8-2) when running programs on the multiple-core execution platform. This API consists of two parts: the Marco definition of the addresses of the memory-mapped registers and a C function to write and read these registers.

```
1  #include "mmio.h"
2  #include <stdio.h>
3
4  #define STATUS 0x2000
5  #define PORT 0x2004
6  #define RESULTADDR 0x2008
7  #define RESULTSIZE 0x200C
8  #define DATAADDR 0x2010
9  #define DATASIZE 0x2014
10 #define IFCOMPLETE 0x2018
11
12 void peripherialInterface(uint32_t port, uint32_t resultAddr, uint32_t resultSize, uint32_t dataAddr,uint32_t
13 dataSize)
14 {
15   while ((reg_read8(STATUS) & 0x2) == 0) ;
16   reg_write32(PORT, port);
17   reg_write32(RESULTADDR, resultAddr);
18   reg_write32(RESULTSIZE, resultSize);
19   reg_write32(DATAADDR, dataAddr);
20   reg_write32(DATASIZE, resultSize);
21
22   while ((reg_read8(GCD_STATUS) & 0x1) == 0) ;
23   reg_read32(IFCOMPLETE);
24  }
```

Figure 8-2. Peripheral software interface implementation.

A register address comprises a base address and an offset address. The base address serves as a parameter in the case class and is set to 0x2000 in the experiment, while the offset address is defined in the DMA controller. In Figure 8-2, the API defines the physical address Macros for all the peripheral's registers like the status, port, dataAddress and dataSize registers (lines 4-10).

The C function named peripherialInterface (line 12) accepts concrete values for the registers in its parameter list and executes the register writing and reading logic in the function body. Specifically, the function leverages the inline functions provided by the header file of "mmio.h" to perform the writing and reading actions. These inline functions simply assign a register address to a pointer pointing to a specific memory segment and change or read the value in the segment.

When the status register shows that the peripheral is in an idle state (line 15), the function will start to write data to the registers (lines 16-20). It is worth noting that the dataSize register must be the last register to be written. Because it will cause the valid signal connected to the dataSize register to be asserted and in turn trigger the peripheral's following operations. After the status register indicates that the peripheral has finished all procedures (line 22), the function will read the ifComplete register (line 23). The value of ifComplete itself is irrelevant, while the action of reading the register can set the ready signal to change the peripheral state from s_done to s_idle.

8.3. Program Compilation

The execution platform simulators can only run RISC-V binary programs. Thus, we need to use the RISC-V GNU compiler toolchain to compile the C programs. Chipyard also incorporates this toolchain in its framework. However, it provides the 64-bit version as default. Therefore, in this section, we will introduce how to build up a RISC-V compiler and then how to use this compiler to compile the C programs of the Sorter System.

```
1  ./configure --prefix=/opt/riscv --with-arch=rv32gc --with-abi=ilp32
2  Make
3
4  riscv32-unknown-elf-gcc -I./../env -I./common -DPREALLOCATE=1 -mcmodel=medany -static -std=gnu99 -O2 -ffast-
5  math -fno-common -fno-builtin-printf -o example.riscv .example/example.c  ./common/syscalls.c ./common/crt.S -
6  static -nostdlib -nostartfiles -lm -lgcc -T ./common/test.ld
```

Figure 8-3. Program compilation.

The compiler toolchain is also an open-source project. After retrieving the project from GitHub, there are two installation options: the Newlib cross-compiler and the Linux cross-compiler. We choose the first one to build a compiler that is suitable for a bare-metal environment, as shown in Figure 8-3. The command on line 1 specifies the installation path, the supported architecture, and the compatible Application Binary Interface (ABI). The keyword rv32gc (line 1) defines the 32-bit architecture with the Integer (I), Atomics (A), Multiplication and Division (M), Float (F), Double (D), and Compressed (C) instruction sets, while the ilp32 indicates that the compiler only supports soft-float instructions. After the execution of the make command, a compiler named riscv32-unknown-elf-gcc will be generated.

The command between lines 4 and 6 demonstrates how to use this compiler to compile a C program. Apart from the specific C program, the compiler also requires three additional files

to generate the corresponding bare-metal binary. Concretely, the syscalls.c (line 5) provides necessary functions implementation like printf, strcpy functions, while the crt.s (line 5) contains crt0 code that prepares the C runtime environment like initializing the stack and global pointers and clearing Block Stated by Symbol (BSS). At last, the test.ld (line 6) is a linker script tailored to the physical address space. In the Chipyard SoC, the default program segment located in DRAM starts at 0x80000000. And, these three files can be found in the Riscv-Tests project, another open-source project that provides benchmarks for the RISC-V processors. In the experiment, we use a makefile to compile the programs. After the compilation, two binary programs named sorterSystemS.riscv and sorterSystemM.riscv will be generated for the single-core and multi-core execution platforms, respectively.

8.4. Boot Process

When passing the binary programs to the platform simulators, we leverage the FrontEnd-Server (FESVR) and Tethered Serial Interface (TSI) to load the programs, as described in section 6.2. In this section, we will describe the boot process in detail.

A Chipyard-based SoC contains a default assembly bootloader program (bootrom.S) for BootROM, which simply loops on a wait-for-interrupt(WFI) instruction. Once the simulated SoC is "powered on", the processor will keep executing this instruction, and meanwhile, the FESVR begins to load the binary program into the main memory of the SoC via the TSI interface. Once the loading procedure is finished, FESRV will write to the software interrupt register for the RISC-V core and bring it out of its WFI loop. Then the processor will jump to the beginning of DRAM to execute the program loaded before.

The binary program should designate two memory positions named as the tohost and fromhost. FESVR uses these memory locations to communicate with the binary program once it is running. The binary program uses tohost to send commands to FESVR for things like printing to the console, proxying system calls and shutting down the SoC. The fromhost is used to send back responses for tohost commands and for sending console input. The addresses of the tohost and fromhost are defined in the crt.s file, while the function for the simulator to read and write them resides in the syscall.c file.

8.5. Experiment Result

In order to evaluate the performance of two execution platforms, we make all controllers of the Sorter System execute 50 SystemGALS ticks on both execution platforms, which means to set the i in the main function to 50. Since the program is executed cyclically, the execution of 50 ticks is representative to compare the performance of two platforms. In addition, the platform simulator are the cycle-accurate ones that require long simulation time, so the selection of 50 ticks can also make the execution time reasonable to perform the experiment.

During the program execution, the same under-test item images will be processed on two platforms. Then, we compare the execution cycles of the Rocket core. Since the programs executed on two execution platforms only differ in the data modules implementation, the results actually demonstrate the performance of the software and hardware implementation of the CV extraction and item recognition data modules. When passing the binary programs to the simulator, the keyword of +verbose can allow the host to print out the cycle count. The experiment shows that the single-core platform spends 3.274 billion clock cycles to finish the execution. In comparison, it takes the multi-core platform 2.973 billion clock cycles to complete the program running, with a 10.3 per cent improvement compared with the single-core platform.

Chapter 9. Discussion and Future Work

The performance comparison result is obvious. However, the procedures and tools to build up the entire Sorter System application are worthy of being discussed. The Sorter System described in this thesis is a small but typical IIoT application, which employs a highly modular design, involves various communications, and requires massive computation. In the era of industry 4.0, similar applications will be diverse in the manufacturing industry and should be implemented rapidly. The critical role of agile development in the success of the software industry gives a clue that the same pattern may be instrumental in IIoT application implementation. Generally, agile development, to the best of our knowledge, relies on reusability and open source, and reusability further depends on modularization and compatibility. In this chapter, we will first evaluate SystemGALS language and Chipyard framework from these aspects. Then the architecture used in the multi-core execution platform is discussed to demonstrate its suitability toward a general execution framework for SystemGALS programs. At last, we put forward expectations for the future IIoT application development.

9.1. Evaluation

Generally, an IIoT application control system consists of both software and hardware parts, which in the Sorter System correspond to the SystemGALS controllers and the Chipyard-based SoC, respectively. In this section, we will discuss the pros and cons of both SystemGALS and Chipyard based on the Sorter System application.

9.1.1 **SystemGALS Evaluation**

SystemGALS underlines system-level behaviour rather than the behaviour of usual programs，and it promotes how the modularity of design can facilitate the composition of the components into larger systems. These features naturally conform to the design pattern of IIoT applications. Programmers can easily define a clock domain for a specific component and encapsulate its behaviours within multiple reactions within the clock domain. In addition, channels provide

clear interfaces between components to communicate with each other and a mechanism to synchronize components' behaviours.

SystemGALS's introduction of the data module construct also enhances its modularity. On the one hand, data modules separate control flow and data computation. This isolation not only makes it easier to share data module implementation between SystemGALS programs but also largely increases SystemGALS's ability to reuse open-source code. On the other hand, with the detachment of the complicated computation from simple control flow, execution architecture designers can accordingly integrate and assign execution resources that optimize and accelerate the computation.

Data modules also enhance SystemGALS's ability by expanding the scope of host language. SystemGALS is still under development, and it currently targets C and JAVA. However, it can theoretically use any programming language as its host language. Data modules can be directly implemented by the host language, while the control flow statements are easy to be mapped to the host language. Furthermore, High-Level Synthesis (HLS) and Hardware Description Languages (HDL) can also be used to design data modules, like the use of Chisel in this thesis.

SystemGALS also has its disadvantages and limitations. For example, SystemGALS programs are driven by logical ticks, and the boundary of ticks is set by pause statements. However, programmers cannot place pause within data modules. If the data modules are very complicated, the duration of one tick may be unacceptable. Thus, it may require extra effort to design and decompose the implementation into a series of data modules to shorten the tick time.

9.1.2 **Chipyard Evaluation**

Chipyard leverages object-oriented and functional programming features provided by modern software languages to create a modular hardware SoC framework. Designers can manipulate SoC components like constructing LEGO sets. In addition, various open-source projects incorporated in the framework provide revisable instances to start the design space exploration. Furthermore, the parameter system and Diplomacy facilitate the entire system integration. Hence, Chipyard has a good performance in terms of modularization and open-source.

The modularization of the Chipyard-based SoC firstly benefits from the Scala programming language in which Chisel is embedded. Traits or mixins in Scala are initially designed for scalable component abstraction. By employing meta-programming technique and FIRRTL compile framework, Chisel bridges the gap between high-level software programming

languages and traditional HDLs and naturally enables Chipyard to use this Scala feature for the SoC design pattern. Next, the parameter system also uses this ‘cake pattern’ to provide flexibility and convenience for the SoC’s modularization. Designers can easily initialize the parameters for all components in one class and make cross-reference between components’ parameters. Then, Diplomacy guarantees the correctness of the integration of different component modules by creating an acyclic graph for the entire SoC and checking parameter compatibility along the edges between component nodes. At last, the optional subsets in TileLink and RISC-V also reflect the modularization principle.

Regarding openness, Chipyard has improved the open-source level of hardware design to a new stage. On the one hand, the RISC-V ISA can be freely used and modified for both research and commercial purposes, largely reducing the cost of processor customization and promoting the prosperity of the RISC-V ecosystem. On the other hand, Chipyard provides many open-source projects that have gained commercial success. For example, UC Berkeley have tape-out the Rocket core successfully more than ten times. And, the Si-Five company has launched the processors based on Rocket and BOOM cores for various commercial applications.

Chipyard also makes a lot of effort to increase its compatibility. For example, the block box construct can directly contain Verilog codes in a Chisel module. And, as the output of Chisel programs are Verilog codes, the following procedures and tools that are compatible with Verilog are also suitable for the Chipyard-based SoC. In addition, the FIRRTL compiler framework largely detaches the front end and back end in the hardware design, thus improving the design reusability for different FPGAs, ASIC toolchains and VLSI technologies.

Unfortunately, Chipyard does not currently support VHDL, another popular HDL in the industry. This may limit its scope of application. Furthermore, Chipyard has higher requirements for practitioners as it involves many new concepts, languages, and tools that are not well known for the traditional SoC designers. A Chisel module design may mix specifications from Scala, Chisel, Diplomacy, and Verilog. And, for industrial-grade products, designers must be very familiar with FIRRTL and write custom transforms for specific applications. Thus, both software and hardware knowledge are necessary for Chipyard-based SoC designs.

9.2. SystemGALS Execution Framework

Generally, the ASPs can serve as two separate MMIO devices in the multi-core platform and be connected to the TileLink bus. However, the introduction of TDMA-MIN NoC is towards a new design and execution framework for SystemGALS. The scalability and time predictability of TDMA-MIN NoC in a heterogeneous multi-core processor system for SystemJ has been demonstrated in previous research [33]. In this thesis, we create a prototype of a similar structure on the RISC-V platform for SystemGALS. During the process, we also leverage many new tools in Chipyard that may facilitate the design space exploration in the future. The work in this thesis still has a long distance from our final goal. In this section, we will briefly discuss the potential ways that can improve this prototype.

Firstly, the execution framework targets a fully time-predictable structure. However, the TileLink bus between the Rocket core and the NI in the multi-core platform does not have this attribute. One way to solve this problem is to customize the TileLink protocol to make a time predictable variant. The other way is to eliminate TileLink and directly connect RISC-V cores to the NIs. And then, a TDMA controller can be integrated to give an equal chance to each core to access the memory, thus making memory access time predictable.

In addition, since control flow in SystemGALS is relatively simple and stable, a custom core can be proposed to be dedicated to executing control flow. The Rocket core is too complicated to achieve this customization. In the latest version, Chipyard includes another RISC-V core, Sodor, which is initially designed for educational purposes and only support the RISC-V base set (Integer). Sodor also provides multiple pipeline options ranging from one stage to five stages. Its simple structure may be suitable for customization.

9.3. Expectation

Complete agile development for IIoT applications still has a long way to go. It requires continuous innovation and revolution in different fields as it involves different disciplines and multiple layers. For example, once the amount of IIoT applications dramatically increases, how to rapidly manufacturing the machines that produce custom products will also be a problem. In this section, we brainstorm some ideas to depict visions for future agile development in Industry 4.0.

Apart from control systems, IIoT applications also involve physical equipment like the mechanical arm in the Sorter System. Currently, mechanical device design largely depends on visual Computer Aided Design (CAD) software like Solidworks and engages many details. We expect the emergence of a programming language for mechanical design, which can allow designers to focus on the functions rather than the mathematics and mechanics details. The programming language may also provide a library containing commonly used components that can be assembled into the desired equipment. Furthermore, once design drawings are generated by the programs, a machine that integrates 3D printers can accept these designs as input and directly produce the mechanical equipment that then is integrated with actuators. This, at least, can rapidly build a prototype of the IIoT manufacturing applications that can be tested in a close to real environment.

In addition, compilers may play a more critical role in the future to bridge agile development tools with existing design procedures like the scenario described in the last paragraph. Meanwhile, in order to cope with new requirements, compilers should develop new features like reusability and intelligence. On the one hand, the trend of reusability can be found in both the famous LLVM (Low Level Virtual Machine) compiler infrastructure and the FIRRTL compiler framework in Chipyard. System-level languages, like SystemGALS, can also leverage compilers with similar structures to facilitate the incorporation of different host languages. And, the design with the multiple layers and detachment between the front end and back end can allow different System-level languages to share one compiler framework. It is possible that these are multiple System-level languages coexisting in the future. On the other hand, we can see the requirement of complier intelligence in the execution resources mapping of SystemGALS programs. The compiler may need to learn from the previous experience to provide an optimal mapping for a specific execution architecture. Furthermore, it may even give suggestions of execution architectures based on scenario specifications.

Finally, current chip manufacturing processes also limit the agile development of IIoT applications. We expect these steps, including photolithography, etching, ion implantation, and chemical vapour deposition, to be encapsulated into one machine with the wardrobe size, which can accept integrated circuit layouts and directly produce chips. With the continuous development of carbon-based semiconductor materials [60] and Chiplet technology [61], this may be achieved in the future. After all, fifty years ago, a computer with simple functions also used to occupy many rooms.

References

[1] M. R¨ußmann, M. Lorenz, P. Gerbert, M. Waldner, J. Justus, P. Engel, and M. Harnisch, "Industry 4.0: The future of productivity and growth in manufacturing industries," in *2nd International Conference on Materials Manufacturing and Design Engineering*, 2015, pp. 1–14.

[2] F. Almada-Lobo, "The industry 4.0 revolution and the future of manufacturing execution systems (mes)," *Journal of Innovation Management*, vol. 3, no. 4, pp. 16–21, 2015.

[3] B. Bagheri, S.Yang, H.A. Kao, and J. Lee, "Cyber-physical systems architecture for self-aware machines in industry 4.0 environment," *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 1622–1627, 2015.

[4] S. Erol, A. J¨ager, P. Hold, K. Ott, and W. Sihn, "Tangible industry4.0: a scenario-based approach to learning for the future of production," *Procedia CIRP*, vol. 54, pp. 13–18, 2016.

[5] M. Landherr, U. Schneider, and T. Bauernhansl, "The application center industrie 4.0-industry-driven manufacturing, research and development," *Procedia CIRP*, vol. 57, pp. 26–31, 2016.

[6] X. Du, J. Jiao, and M. M. Tseng, "Understanding customer satisfaction in product customization," *The International Journal of AdvancedManufacturing Technology*, vol. 31, no. 3, pp. 396–406, 2006.

[7] R. Lachmayer, P. C. Gembarski, P. Gottwald, and R. B. Lippert, "The potential of product customization using technologies of additive manufacturing," in *Managing Complexity*. Springer, 2017, pp. 71–81.

[8] J. Pallant, S. Sands, and I. Karpen, "Product customization: A profile of consumer demand," *Journal of Retailing and Consumer Services*, vol. 54, p. 102030, 2020.

[9] C. Scheuermann, S. Verclas, and B. Bruegge, "Agile factory-an example of an industry 4.0 manufacturing process," in *2015 IEEE 3rd International Conference on Cyber-*

Physical Systems, Networks, and Applications. IEEE, 2015, pp. 43–47.

[10] S. Wang, J. Wan, D. Li, and C. Zhang, "Implementing smart factory of industrie 4.0: an outlook," *International Journal of Distributed Sensor Networks*, vol. 12, no. 1, p. 3159805, 2016.

[11] K. Thoben, S. Wiesner, and T. Wuest, ""Industrie 4.0" and smart manufacturing-a review of research issues and application examples," *International Journal of Automation Technology*, vol. 11, no. 1, pp. 4–16, 2017.

[12] E. Hozdi´c, "Smart factory for industry 4.0: A review," *International Journal of Modern Manufacturing Technologies*, vol. 7, no. 1, pp. 28–35, 2015.

[13] Germany Industrie 4.0 Working Group, "Recommendations for implementing the strategic initiative industrie 4.0," Germany. Apr. 08, 2013. [Online]. Available: https://en.acatech.de/publication/recommendations-for-implementing-the-strategic-initiative-industrie-4-0-final-report-of-the-industrie-4-0-working-group/

[14] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann, "Industry4.0," *Business & Information Systems Engineering*, vol. 6, no. 4, pp.239–242, 2014.

[15] A. Rojko, "Industry 4.0 concept: Background and overview." *International Journal of Interactive Mobile Technologies*, vol. 11, no. 5, 2017.

[16] S. Vaidya, P. Ambad, and S. Bhosle, "Industry 4.0–a glimpse," *Procedia Manufacturing*, vol. 20, pp. 233–238, 2018.

[17] L. H. Melnyk, O. V. Kubatko, I. B. Dehtyarova, I. B. Dehtiarova, O. M.Matsenko, and O. D. Rozhko, "The effect of industrial revolutions on the transformation of social and economic systems," 2019.

[18] E. G. Popkova, Y. V. Ragulina, and A. V. Bogoviz, "Fundamental differences of transition to industry 4.0 from previous industrial revolutions," in *Industry 4.0: Industrial Revolution of the 21st Century*. Springer, 2019, pp. 21–29.

[19] G. Qiao, R. F. Lu, and C. McLean, "Flexible manufacturing systems for mass customisation manufacturing," *International Journal of MassCustomisation*, vol. 1, no. 2-3, pp. 374–393, 2006.

[20] D. d. S. Dutra and J. R. Silva, "Product-service architecture (psa): Toward a service engineering perspective in industry 4.0," *IFAC-PapersOnLine*, vol. 49, no. 31, pp. 91–96, 2016.

[21] A. Schumacher, S. Erol, and W. Sihn, "A maturity model for assessing industry 4.0 readiness and maturity of manufacturing enterprises," *Procedia CIRP*, vol. 52, pp. 161–166, 2016.

[22] G. Schuh, T. Potente, C. Wesch-Potente, A. R. Weber, and J.-P. Prote, "Collaboration mechanisms to increase productivity in the context of industrie 4.0," *Procedia CIRP*, vol. 19, pp. 51–56, 2014

[23] K. Witkowski, "Internet of things, big data, industry 4.0–innovative solutions in logistics and supply chains management," *Procedia Engineering*, vol. 182, pp. 763–769, 2017.

[24] V. Berman, "System-level design language standard needed," *IEEE Design & Test of Computers*, vol. 21, no. 6, pp. 592–593, 2004.

[25] S. Ramesh, "Communicating reactive state machines: Design, model and implementation," *IFAC Proceedings Volumes*, vol. 31, no. 32, pp.105–110, 1998.

[26] [26] O. Tardieu and S. A. Edwards, "Scheduling-independent threads and exceptions in shim," in *Proceedings of the 6th ACM & IEEE International Conference on Embedded software*, 2006, pp. 142–151.

[27] F. Gruian, P. Roop, Z. Salcic, and I. Radojevic, "The SystemJ approach to system-level design," in *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings. IEEE, 2006*, pp. 149–158.

[28] A. Malik, Z. Salcic, P. S. Roop, and A. Girault, "SystemJ: A GALS language for system level design," *Computer Languages, Systems &Structures*, vol. 36, no. 4, pp. 317–344, 2010.

[29] Z. Salcic, P. HeeJong, B. Morteza, and T. Juergen, "SystemGALS – a language for the design of gals software systems," unpublished.

[30] A. Girault, "A survey of automatic distribution method for synchronous programs," in

International workshop on synchronous languages, applications and programs, SLAP, vol. 5, 2005.

[31] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, p. 48–60, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3282307

[32] A. S. Waterman, *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[33] Z. Salcic, H. Park, J. Teich, A. Malik, and M. Nadeem, "Noc-HMP: A heterogeneous multicore processor for embedded systems designed in SystemJ," *ACM Transactions on Design Automation of ElectronicSystems (TODAES)*, vol. 22, no. 4, pp. 1–25, 2017.

[34] Y. Lee, A. Waterman, H. Cook, B. Zimmer, B. Keller, A. Puggelli, J. Kwak, R. Jevtic, S. Bailey, M. Blagojevicet al., "An agile approach to building RISC-V microprocessors," *IEEE Micro*, vol. 36, no. 2, pp.8–20, 2016.

[35] B. Kent, B. Mike, B. Van, C. Alistair, C. Ward, F. Martin, G. James, H. Jim, H. Andrew, J. Ron, and K. Jon, "Manifesto for agile software development," 2001.

[36] K. Conboy, "Agility from first principles: Reconstructing the concept of agility in information systems development," *Information systems research*, vol. 20, no. 3, pp. 329–354, 2009.

[37] A. Bonaccorsi and C. Rossi, "Why open source software can succeed," *Research Policy*, vol. 32, no. 7, pp. 1243–1258, 2003.

[38] K. Asanovi´c and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[39] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avi˘zenis,J. Wawrzynek, and K. Asanovi´c, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[40] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim,C. Schmidt, C. Markley, J. Lawsonet al., "Reusability is firrtl ground: Hardware construction

languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-AidedDesign (ICCAD)*. IEEE, 2017, pp. 209–216.

[41] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew,A. Magyar, H. Mao, A. Ou, N. Pembertonet al., "Chipyard: Integrated design, simulation, and implementation framework for custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[42] D. A. Patterson and C. H. Sequin, "RISC I: A reduced instruction set VLSI computer," in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 216–230.

[43] M. G. Katevenis, R. W. Sherburne Jr, D. A. Patterson, and C. H. S´equin, "The RISC II micro-architecture," *Advances in VLSI and Computer Systems*, vol. 1, no. 2, pp. 138–152, 1984.

[44] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," in *Proceedings of the 11th AnnualInternational Symposium on Computer Architecture*, 1984, pp. 188–197.

[45] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. D. Adams, B. K.Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Konget al., *SPUR: a VLSI multiprocessor workstation*. University of California, 1985.

[46] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz et al., "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[47] K. Asanovic, D. A. Patterson, and C. Celio, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized RISC-V processor," University of California at Berkeley Berkeley UnitedStates, Tech. Rep., 2015.

[48] S. Karandikar, H. Mao, D. Kim, D. Biancolin, A. Amid, D. Lee, N. Pemberton, E. Amaro, C. Schmidt, A. Chopra, Q. Huang, K. Kovacs, B. Nikolic, R. Katz, J. Bachrach, and K. Asanovic, "Firesim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.

[49] E. Wang, C. Schmidt, A. Izraelevitz, J. Wright, B. Nikoli´c, E. Alon, andJ. Bachrach, "A methodology for reusable physical design," in *2020 21st International Symposium on Quality Electronic Design (ISQED)*, 2020, pp. 243–249.

[50] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A TileLink case study," in *1st Workshop on Computer Architecture Research with RISC-V*, 2017.

[51] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629-2640, Nov. 2019, doi: 10.1109/TVLSI.2019.2926114..

[52] Y. Lee, C. Schmidt, A. Ou, A. Waterman, and K. Asanovi´c, "The hwacha vector-fetch architecture manual, version 3.8.1," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262*,2015.

[53] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual DesignAutomation Conference (DAC)*, 2021.

[54] Z. Salcic, M. Nadeem, H. Park, and J. Teich, "Optimizing latencies and customizing noc of time-predictable heterogeneous multi-core processor," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, 2016, pp. 233–240.

[55] Z. Salcic, M. Nadeem, and B. Striebing, "A time predictable heterogeneous multicore processor for hard real-time GALS programs," in *ARCS2016; 29th International Conference on Architecture of Computing Systems*, 2016, pp. 1–8.

[56] Z. Salcic. (2019). Advanced Embedded Systems [PowerPoint slides]. Available: https://canvas.auckland.ac.nz/courses/38282

[57] E. Fix and J. L. Hodges, "Discriminatory analysis. Nonparametric discrimination: Consistency properties," International Statistical Review/ Revue Internationale de Statistique, vol. 57, no. 3, pp. 238–247,1989. [Online]. Available:

http://www.jstor.org/stable/1403797

[58] N. S. Altman, “An introduction to kernel and nearest-neighbour non-parametric regression,” The American Statistician, vol. 46, no. 3, pp.175–185, 1992.

[59] J. Zhang, “KNN numbers recognition,” cnblog.com. https://www.cnblogs.com/sjzh/p/6104105.html (accessed Feb. 1, 2020)

[60] *Chipyard*. (2020). UC Berkeley Architecture Research. Accessed: March 11, 2020. [Online]. Available: https://github.com/ucb-bar/chipyard/releases

[61] A.N.Habermann, “Parallel neighbor-sort (or the glory of the induction principle),” Carnegie Mellon University. Jun. 30, 2018. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/Parallel_neighbor-sort_or_the_glory_of_the_induction_principle_/6608258

[62] *SiFive TileLink Specification*, Version 1.8.0, Aug. 2019. [Online]. Available: https://sifive.cdn.prismic.io/sifive%2Fcab05224-2df1-4af8-adee-8d9cba3378cd_TileLink-spec-1.8.0.pdf

[63] P. Avouris, Z. Chen, and V. Perebeinos, “Carbon-based electronics,” *Nanoscience and Technology: A Collection of Reviews from Nature Journals*, pp. 174–184, 2010.

[64] G. Mounce, J. Lyke, S. Horan, W. Powell, R. Doyle, and R. Some, “Chipletbased approach for heterogeneous processing and packaging architectures,” in *2016 IEEE Aerospace Conference*, 2016, pp. 1–12.