T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	C/C++によるシステムレベルRTLデザイン手法およびLLVMコンパイラインフラストラクチャを用いた設計効率の評価
Title(English)	C/C++ Based System Level RTL Design Methodology and Its Design Efficiency Using LLVM Compiler Infrastructure
著者(和文)	貞末多聞
Author(English)	Tamon Sadasue
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第12388号, 授与年月日:2023年3月26日, 学位の種別:課程博士, 審査員:一色 剛,髙橋 篤司,本村 真人,劉 載勲,佐々木 広
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12388号, Conferred date:2023/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Science Tokyo Research Repository)

Doctoral Dissertation

C/C++ Based System Level RTL Design Methodology and Its Design Efficiency Using LLVM Compiler Infrastructure

Tamon Sadasue

Information and Communications Engineering,
Tokyo Institute of Technology

Supervisor: Professor Tsuyoshi Isshiki

February, 2023

Abstract

As the scale of digital circuit design increases, design and verification using conventional RTL such as verilog-HDL and VHDL, has become a problem in terms of efficiency. We consider that conventional HDLs have design efficiency problems from three points of view: description efficiency problems, verification efficiency problems, and system-level design and verification problems. The inefficiency of conventional HDL descriptions is mainly due to the lack of aggregate types for interfaces, poor preprocessing, and poor parameterization capabilities without support for generic programming. Major problems with verification efficiency are that conventional HDL simulation takes much longer than software debugging and lacks interactive debugger environments such as GDB. The problem in system-level design and verification is that bus and communication protocol abstractions are not built into the design language, resulting in increased simulation time and reduced signal visibility.

To solve these problems of conventional hardware description language (HDL), a lot of methods have been proposed. One major approach is a high-level synthesis (HLS) method that generates hardware from a software program, and the other is a domain-specific language (DSL) method that specializes a high-level language to describe the hardware. However, previous approaches have advantages and disadvantages, and are not effective in all cases of hardware description.

C2RTL is a framework developed in Isshiki Laboratory against this background. Although it takes C language as input, unlike HLS, it enables description at the Register Transfer Level of abstraction, and provides the efficient hardware implementation and verification as well as the full control to hardware resources. However, the previous C2RTL did not support C++ code, so object-oriented programming and generic programming using templates were not available. Only IP level description was supported, system description was out of scope. Lastly, since the previous C2RTL was an in-house compiler, it is difficult to maintain compiler quality.

At this work, we introduced LLVM as a compiler platform and developed a methodology to convert LLVM-IR, an intermediate language of LLVM, into RTL-IR, an intermediate language of hardware structure. The verilog code and its equivalent C simulation code are generated from this RTL-IR. This mechanism ensures the compliance with the latest C/C++ standards and allows the application of LLVM's generic and customized optimization passes.

Next, by developing a two-step RTL description method for IP level and system level, we constructed a framework that can design and verify systems such as SoCs using only C/C++ language. Thus, by extending C2RTL to system-level descriptions, it is possible to complete the design and verification of the entire chip in C/C++ language and utilize an extensive software development environment such as debuggers and IDEs for hardware development.

In this paper, we clarify how LLVM-C2RTL converts input code to hardware representation via LLVM-IR, and then evaluated the design efficiency of LLVM-C2RTL through several studies.

In order to evaluate that the preprocessing function of LLVM-C2RTL improves the parameterization ability of hardware design, we implemented and evaluated a machine learning algorithm based on decision trees. We present a fully pipelined hardware implementation of Gradient Boosted Tree training with high performance and flexibility to realize highly parameterized machine learning models. This experimental result shows that our FPGA implementation achieves a 11- to 33-times faster performance and more than 300-times higher power efficiency than a GPU accelerated software implementation.

We evaluated the design efficiency of LLVM-C2RTL. As a result of comparing the amount of code required to create the AES encryptor and RISC-V core with other DSLs that enable register level description, we confirmed that the amount of code can be reduced by approximately 40%. In addition, the compile time of LLVM-C2RTL is about an order of magnitude faster than its counterparts, and the simulation time is comparable to verilator, one of the fastest verilog simulators.

As for the quality of the generated hardware, we verified that the synthesizable clock frequency is comparable to other approaches and can be improved by the automatic pipelining feature of LLVM-C2RTL, and the area of generated hardware is reduced through compiling with LLVM optimization.

These experimental results shows that LLVM-C2RTL framework offers an extremely efficient design and test environment because a complete system-level hardware design and debugging can be performed on powerful software development platforms and tool sets. The generated verilog code gives almost the same performance as hand-designed logic using the conventional HDL and automatic pipelining and system-level design functionality facilitate the trade-off search between performance and area size.

Acknowledgments

I would like to acknowledge and give my warmest thanks to my thesis supervisor Professor Tsuyoshi Isshiki for his invaluable support and guidance through the hard moments of Research.

I would also like to thank the members of the thesis committee, Professor Atsushi Takahashi, Professor Masato Motomura, Associate Professor Hiroshi Sasaki, and Associate Professor Jaehoon Yu, for their brilliant feedback and suggestions.

I would also give my special thanks to all the members of my laboratory at Tokyo Institute of Technology for great discussions and research help.

Finally, I would like to thank my family for supporting me in difficult times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Currently, verilog-HDL and VHDL are the main hardware description languages that represent the digital circuit structure at the register transfer level (RTL) for the design of ASIC and FPGA. However, as the scale of digital circuit design increases, design and verification using conventional hardware description languages (HDLs), such as verilog-HDL and VHDL, limit efficiency.

We consider that conventional HDLs have design efficiency problems from three points of view: description efficiency problems, verification efficiency problems, and system-level design and verification problems.

As for the description efficiency, in some respects, the functionality supported by these conventional HDLs are inferior to modern software description languages. For example, the lack of object types, like class or struct, can increase code volumes and make the code prone to errors, even for simple module instantiation and connection. Lack of type inference requires strict declarations of each signals associated with connecting other modules. In addition, their limited parameter abstraction, which replaces a part of behaviours or attributes, complicates the reuse of their created libraries. Regarding the method of parameterization, it is necessary to describe parameter variables for all target modules, and to propagate them from upper modules to lower modules, which is very burdensome task.

Major problems with verification efficiency are that conventional HDL simulation takes much longer than software, and luck of interactive debuggers like GDB that can stop at breakpoints set in the source code and step execution, as is often used in software development environments.

In terms of system-level design and verification efficiency, conventional HDLs do not support the abstraction of interfaces such as bus and communication protocols directly, so the amount of verification code tends to be large, and special tools were needed for abstraction level conversion, such as signal-level and transaction-level conversion. In addition, there were also problems of slow simulation speed and poor visibility of internal signals.

Recently, the new specification of SystemVerilog has been supported by several EDA tools and vendors, and abstraction functionality has been enhanced by user-defined struct and interface. However, struct can not be directly parameter-

ized, and the support of interface function which can be used for logic synthesis varies widely between tools. Also, inference the type and direction of input and output signals are not supported as language specification. It is also impossible to parameterize the presence or absence of input/output ports in conventional HDL and SystemVerilog. The pre-processing ability of SystemVerilog to allow such advanced parameterization is poor compared to template or constant expressions in modern C++ language.

Another approach is to create descriptions in C or C-like software language and convert them into RTL descriptions using high-level synthesis (HLS) tools[1] [2] [3] [4] [5] [6] [7].

Recent C-based HLS tools, such as Vivado HLS [1], Catapult [6], and Intel HLS Compiler[2], support C/C++ standards such as template arguments and functions, which enable a high degree of parameterization.

In general, each HLS tool converts a software program written at the behavior level into a hardware structure. In particular, the characteristics of the tool are determined by the scheduling process, which determines the cycle in which the operation is executed, and the binding process, which maps operations to arithmetic units and variables and variable arrays to registers and memory. These steps are performed based on resource constraints and user directives, and various hardware structures can influence the delay and throughput.

Unfortunately, problems can emerge here. When users write software code, they can concentrate mainly on algorithms without paying significant attention to their implementation; however, the performance of hardware logic created by HLS tools significantly depends on implementation. Hence, the users must spend substantial amounts of time tuning performance directives. Therefore, when designing logic there must be strict communication between modules or strict delay cycles and resource constraints; RTL hand-coding is often preferred over the HLS approach. HLS is also not suitable for designs that require feedback control, such as processors [8].

Another disadvantage of HLS is that HLS is aimed at IP-level design and does not support system-level design. Therefore, the system design that usually contains bus interconnect to connects the CPU and peripheral circuits, and high-performance DMA controller is implemented by using conventional HDL.

In such a situation, as an alternative to conventional HDLs, new software description languages referred to as domain-specific language (DSL) or middle-level synthesis (MLS) that generate conventional HDL codes (verilog or VHDL) are currently garnering considerable attention. In these approaches, users sometimes must be aware of the hardware implementation at or near the register transfer level; however, designing with these approaches is more efficient because they are less descriptive and their libraries are more reusable [9] [10] [11] [12] [13] [14] [15] than conventional HDLs.

However, there are problems with these HDL alternatives. First, since DSL is not originally a language for describing hardware, it is necessary to use classes provided for hardware modules and signals on the base language. The amount of description increases due to class definition and instantiation, and we need to learn the base language and usage of classes and libraries for hardware modules.

Second, many of them elaborate and compile on interpreters such as Scala or Python [9] [13] [14], increasing their compilation and simulation time significantly. Using a fast simulator on generated RTL code partially mitigate this problem, but it also faces the problem of double-verification, where input code verification and generated HDL verification must be done in different languages and environments. Finally, they do not have automatic pipelining functionality; therefore, pipelining the complex combinational logic by hand is difficult when it does not meet at the target clock period.

We adopt the HDL alternative approach rather than HLS. While our design framework inputs the C code, it does not incorporate scheduling or binding, and require no special classes or structs. All variables that become registers and memory must be explicitly specified by the user, and all substitutions to these memory elements are converted to the combinational logic. In other words, we provide a register transfer level description language that can be written in C language.

Our previous research, C2RTL [16], implemented a conversion mechanism from C data flow description to pipelined logic with our original compiler. However, since it was a in-house compiler, it was difficult to extend the language to support generic programming such as `template` and constant expression (`constexpr`) function, to guarantee the quality of the generated hardware in terms of the compiler. In addition, our previous work only supported IP-level HDL generation, but had difficulty integrating system-level hardware descriptions.

In this study, we introduced Clang/LLVM [17] as a compiler infrastructure to solve this difficulty. By integrating it into the compiler rather than the interpreter or pre-processor, we could handle all hardware types as the compiler's built-in types. Our new "LLVM-C2RTL" supports all C/C++ standard descriptions, and enables to exploit an advanced description abstraction and a rich development environment such as debuggers of software programming languages for hardware design. Also it offers significant improvement of design efficiency through reduced code writing, faster compilation, faster simulators, and delay and area prediction.

Furthermore, we newly proposed a hierarchical system-level architecture description method and implemented the mechanism for hierarchical RTL model generation of hardware (RTL) and software (simulator) from a single C/C++ source code.

1.2 Related Work

Bluespec SystemVerilog (BSV) is a rule-based MSL where hardware is described as object-oriented modules [11]. The BSV compiler allows users who are accustomed to hardware design to describe hardware intuitively by defining rules for a set of terms and a set of updating variables, avoiding redundant descriptions of signal connections such as verilog-HDL and VHDL. BSV also enables user-defined types such as the `struct` of C/C++; however, the operators with the user-defined types are not as trivial as the C/C++ language.

Chisel [9] [10] is an HDL built on the Scala programming language with the Chisel library, and it has been used in the design of the RISC-V processor [18].

When using Chisel as a DSL on the Scala, the degree of parameterization becomes higher, because the variables are evaluated by the Scala before the logic written in Chisel's class is evaluated. In addition, a type template allow the generic expressions and high productivity with reusable libraries when the user becomes proficient the language.

However, Chisel defines hardware signals in the same way as verilog-HDL and VHDL; hence, it cannot be described as software engineers procedurally assign variables and call functions. For example, when updating a repetitive variable and evaluating it, or when writing the behavior of a recursive assignment, the amount of description for each signal definition and module classes tends to be increased. Furthermore, engineers who do not know Scala find it difficult to master Chisel modules and utility classes while learning the Scala language.

BSV, Chisel, and MyHDL process on Haskell, Scala, and Python, respectively, which increases cycle-level simulation time. To deal with this, Chisel and BSV simultaneously generate verilog-HDL code and C++ or System-C simulation code for compile-based high-speed simulations. However, in these environments, test code must be created separately from the original description code by adopting another language and converting the interface.

PyMTL [13] is a framework that integrates behavior level, cycle level, and register transfer level models. While it builds classes and libraries for hardware on Python for high productivity and parameterization, it applies the just-in-time (JIT) compiler for cycle-level simulation and Verilator [19] for RTL simulations like Chisel. With further study [15], to avoid the double verification load when using Verilator as a simulator, researchers improved the host language simulation speed by optimally generating JIT compiler hardware; however, the simulation speed is still orders of magnitude larger than that of Verilator.

ArchHDL [12] is a hardware modeling DSL that employs C++ class libraries, such as module, wire, and register, corresponding to verilog-HDL. This approach facilitates both the design and debugging of hardware using the C++ language, incorporating the efficiency of generic software debugging, while the class name and method must be explicitly described, including the directive equivalent to RTL, while the coding efficiency is insufficient.

In addition, these tools do not support automatic pipelining.

LLVM [17] is an open source compiler infrastructure that is widely used in both industry and academia, and HLS tools such as Xilinx Vivado HLS [1], Intel FPGA SDK for OpenCL [20], and LegUp [3], adopted the LLVM as a compiler platform [21]. By introducing LLVM, HLS tools can support the latest C/C++ standard and exploit the generic optimization passes those are built-in for software performance enhancement. For example, [22] incorporate LLVM for front-end and optimization purposes in a tool-chain to convert the software code with specific libraries to the VHDL code. [23] provides a detailed analysis of the effects of LLVM's generic optimization passes on the performance of generated hardware in LegUp. And many studies have investigated the hardware-specific optimization passes, in addition to generic optimization passes [24].

These approaches utilize LLVM in an attempt to generate accelerating hardware that matches the input software programs, and the products heavily rely on

the tools. By contrast, we aim to give the user full control of hardware resource and products' portability. Therefore, while the optimization passes that destroys the hardware structure can not be applied, some hardware attributes must be propagated through compilation. We clarified a method for generating hardware using LLVM from register-level C/C++ code, and show that the RTL design efficiency is enhanced.

1.3 Thesis Contribution

The main contributions of this thesis are follows:

- We reconstructed our C2RTL design framework with LLVM compiler infrastructure, to support the latest C/C++ standards and enable modern software descriptions with object-oriented programming and generic programming.
- LLVM-C2RTL significantly improves the design and verification efficiency by exploiting the software development environment, reduced description size, and fast compilation and simulation.
- We develop a methodology to convert LLVM-IR, the intermediate language of LLVM, into RTL representation for HDL generation.
- We established the description and synthesis method of not only IP-level RTL, but also system-level RTL by the C/C++ language.
- A system design integration method that automatically generates a system behavior model and a system hardware model from the same software source was realized for the first time.
- The large-scale hardware description and the high parameterization ability of LLVM-C2RTL, which enables to design fast and flexible decision tree training systems, was demonstrated.

1.4 Thesis Organization

The first Chapter is an introduction, and describes the background of this paper. We will describe the alternative developing methods and environments which have been proposed as alternatives to conventional HDLs.

In Chapter 2, we explain about the design framework "LLVM-C2RTL", developed in Isshiki Laboratory. We first describe the overview and concept of the LLVM-C2RTL framework, and clarify the definitions of constraints and rules introduced to correspond dataflow-style descriptions in C/C++ to circuit structures.

In Chapter 3, we describes how the LLVM compiler infrastructure was implemented in the LLVM-C2RTL framework. LLVM has LLVM-IR as an intermediate representation for generating machine code from C/C++ programs. We show how

this LLVM-IR is transformed into the final circuit description in LLVM-C2RTL compling flow. Furthermore, as features of LLVM-C2RTL, we describe the mechanisms of automatic bit width calculation, circuit delay prediction, and automatic pipelining.

Chapter 4 describes the support for system-level hardware descriptions, and the method for optimized simulation code generation.

In Chapter 5, in order to demonstrate the usefulness of the LLVM-C2RTL design framework, we present our experiments on hardware implementation for fast and low-power training of the Gradient Boosted Decision Tree, one of the most precise machine learning algorithm. There, a fully pipelined hardware-friendly algorithm is shown, and LLVM-C2RTL enables hyperparameter search, which is difficult with conventional HDL.

In Chapter 6, we discuss the design efficiency of LLVM-C2RTL. Chisel and PyMTL3 are design tools similar to LLVM-C2RTL developed as domain-specific languages that use scala and python as input languages and output verilog code. Therefore, we show a comparison of the amount of input code of each tools when implementing the AES encryption logic with the same algorithm. And we show a comparison of the quality of the generated hardware by synthesizing the output verilog.

The final chapter concludes this thesis by summarizing this thesis and presenting future work.

Chapter 2

C/C++ Based RTL Design Framework

2.1 Overview of LLVM-C2RTL Design Framework

Our design framework was originally proposed in our previous study [16], where the C2RTL compiler tool generates RTL codes in Verilog-HDL, RTL equivalent C codes (RTL-C) having cycle-accurate and bit-precise simulations, and testbench codes in Verilog-HDL.

In this previous study, we built the compilation process to convert C codes to RTL and parse C syntax. However, supporting C++ syntax and preprocessing is time consuming and lacks the flexibility to implement various optimization algorithms. In addition, it only supported module-level (or IP-level) hardware generation and does not have a mechanism for building SoC with CPUs, peripherals, and accelerators.

Against this background, we have introduced Clang/LLVM as a compiler frontend. Instead of parsing the C syntax and converting it directly, we use Clang to convert it to an intermediate representation (LLVM-IR), once applies appropriate common optimization algorithms and to convert to the register transfer level intermediate representation (RTL-IR), then finally generates the hardware (Fig.2.1). In addition, we newly made it possible to create hierarchical module structure and simulation code by C/C++ description.

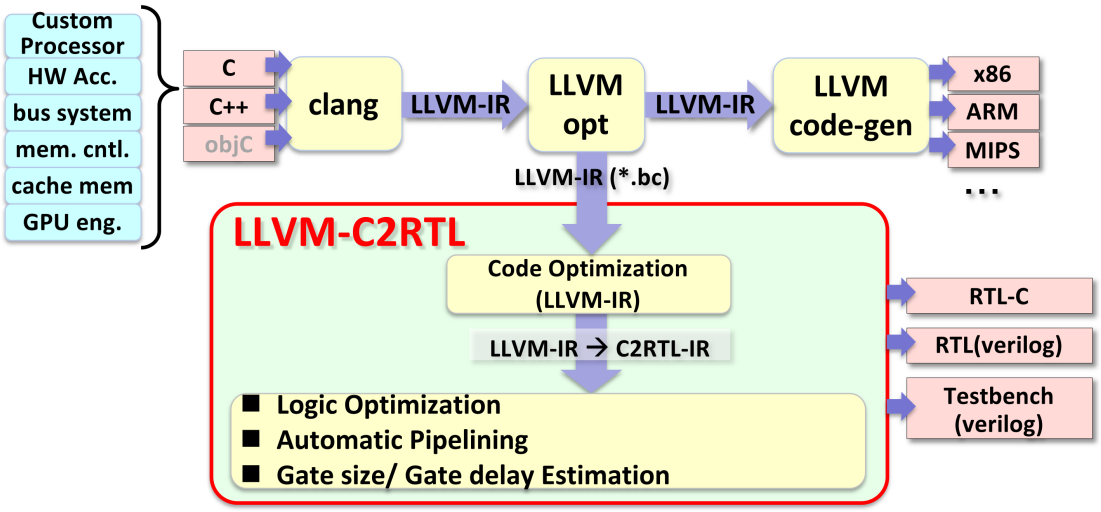

Figure 2.1: C/C++ based RTL design framework (LLVM-C2RTL)

The complete design and verification flow is shown in Fig.2.2. In the module-level design, the verilog code and equivalent C-simulation code are generated from the input C/C++ code. This C-simulation code is verified, and the equivalency check of these codes can be done with the dumped input and output patterns from the C-simulation. Next, system-level C/C++ code is compiled. This compilation yields the top-level verilog code, which instantiates the submodules, as well as the system-level C-simulation code, which simulate cycle-accurate and bit-precise top modules, including all submodules. A description example of the module-level and system-level integration is shown in Fig.4.2.

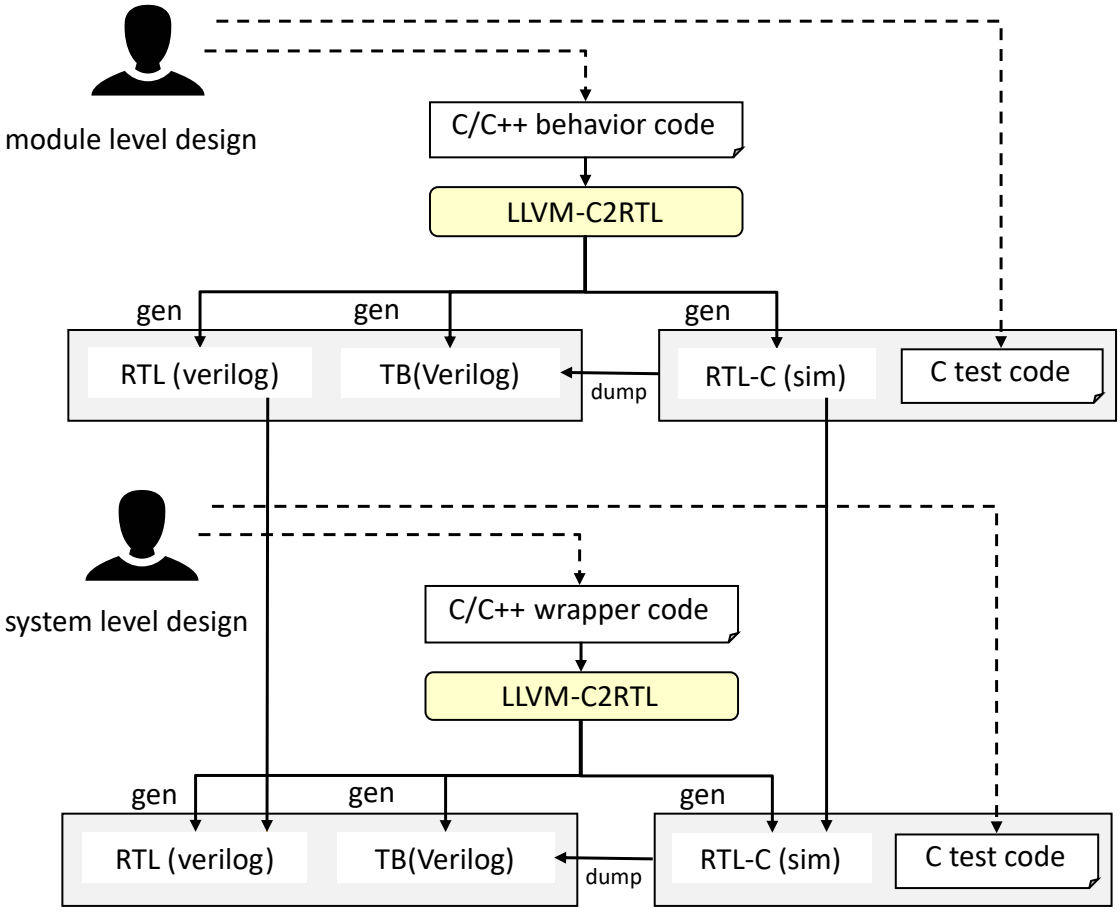

Figure 2.2: Design and verification flow with LLVM-C2RTL

The rest of this chapter clarifies the specifications of our description language and coding rules.

2.2 Concept of LLVM-C2RTL Language

The main concept of our approach is that if RTL can be expressed using a procedural language such as C/C++, the design flow will be significantly simplified by the data-flow style description and rich software development assets. This feature is also provided by HLS compilers; however, LLVM-C2RTL gives users full control over the clock cycle delay, memory elements, and pipeline elements, such as the number of stages, pipeline registers, and signal wire.

While HLS aims to achieve a hardware acceleration with as few modifications to the existing codes as possible, LLVM-C2RTL aims to provide full control to the hardware resource with the abstraction level of conventional HDLs, while minimizing the writing effort.

As an example of the benefits of full control over HLS, when designing a processor, HLS cannot be used to design multi-cycle instructions or forwarding mechanism, which uses the results of the execution or memory stage for the next instruction in that cycle [8]. To implement this, the processor pipeline stages must

be explicitly divided, and each stage should be carefully designed to end within one clock cycle. By contrast, with LLVM-C2RTL, if explicitly adding the state attribute to the variable to be forwarded, its value can be obtained from any pipeline stage.

Conventional HDLs are event-driven because the elements of the circuit are originally structural and event-driven. In contrast, the C code includes control flow, such as function calls and branches, so there is a gap between C code and conventional HDLs.

However, a normal synchronous digital circuit becomes a directed acyclic graph (DAG) if the signal transfer from register to register is cut out. Figure 2.5 illustrates the DAG created from the code shown in Fig.2.3. Registers correspond to the static variables in the C function and appear at the start and end of the DAG. 'stt (prev)' indicates the register state at the previous clock cycle, and 'stt' indicates the updated register state one clock cycle after. In this way, we can express arbitrary circuits using a data flow description limited to the behavior from the registers' output to input.

```
 1: typedef struct {
 2:   UINT16 din[2];
 3:   UINT16 dout;
 4: } ST_IO;
 5:
 6: _C2R_FUNC(1)
 7: void func1(ST_IO& io) {
 8:   static UINT16 stt _TYPE(state);
 9:   UINT16 a = 0;
10:   for (int i = 0; i < 2; i++) {
11:     a += io.din[i];
12:   }
13:   UINT16 c;
14:   if (stt == 0) {
15:     stt = a;
16:     c = a;
17:   } else {
18:     c = a + 1;
19:   }
20:   io.dout = c;
21: }
```

Figure 2.3: Simple example code

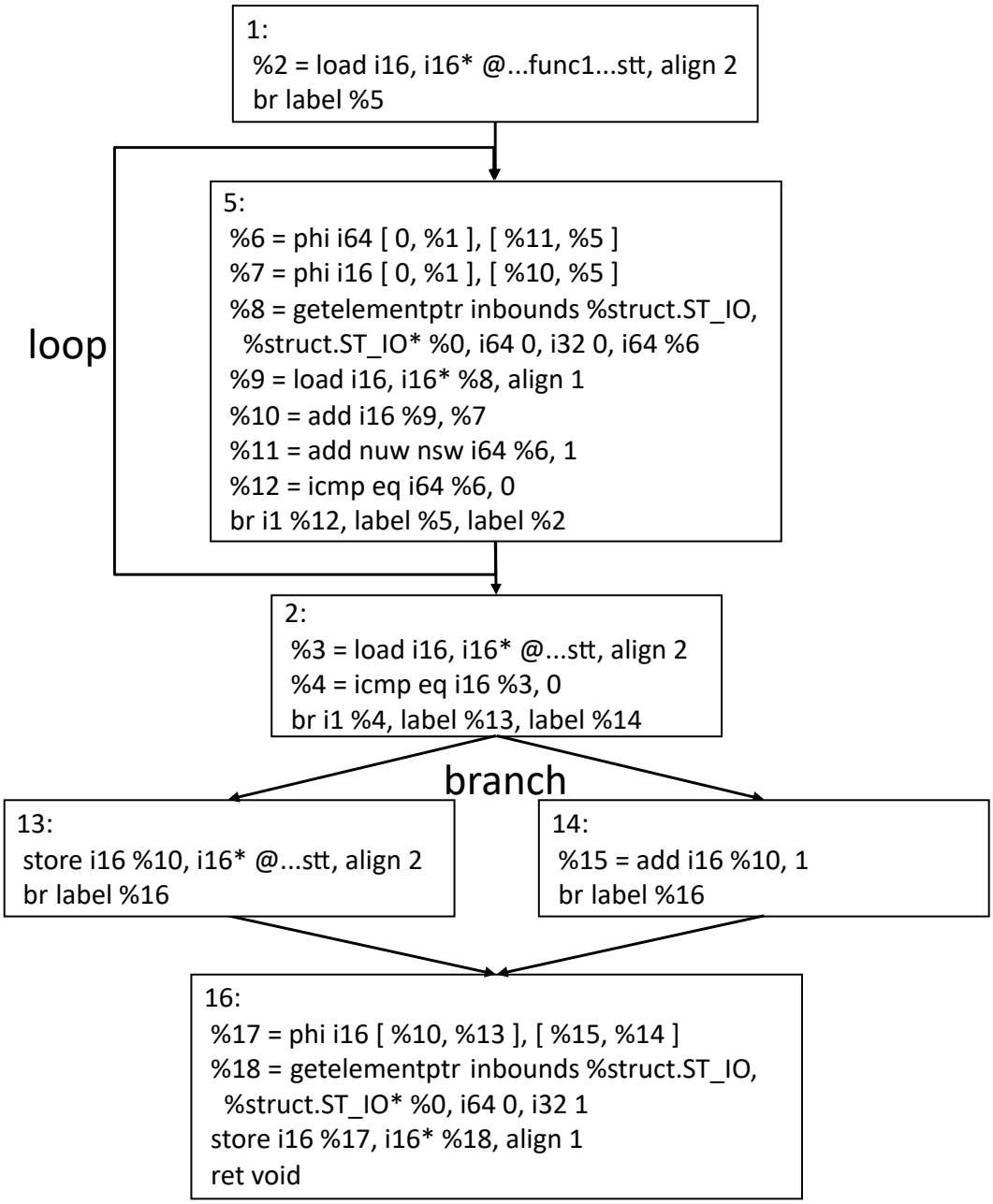

Figure 2.4: LLVM-IR of example code

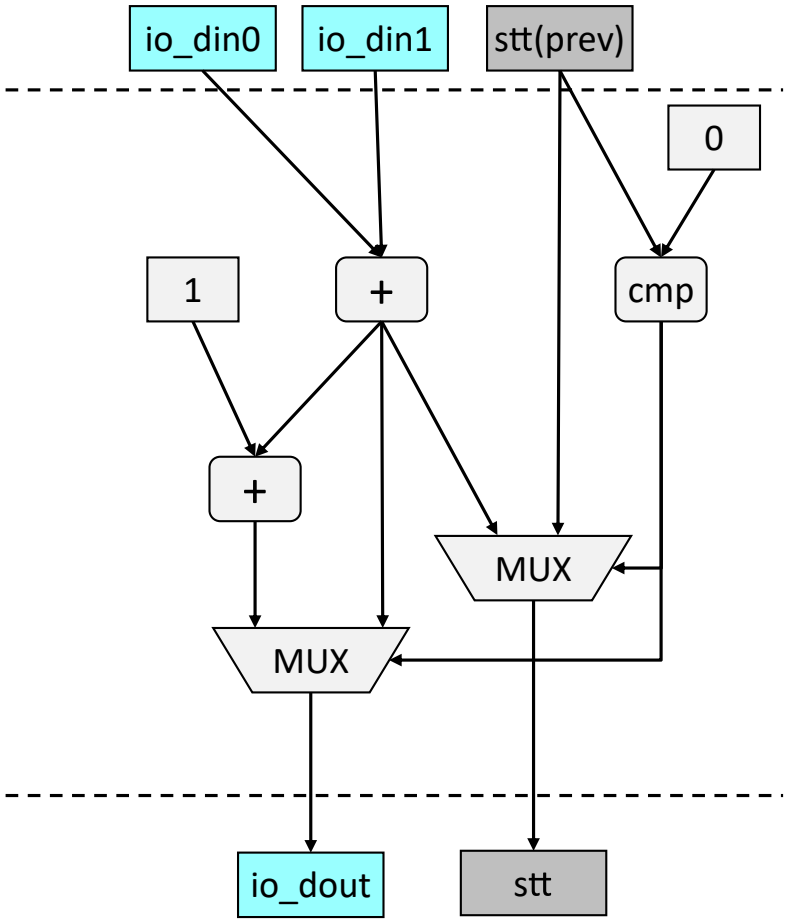

Figure 2.5: Directed acyclic graph

2.3 Hardware Specific Attributes Definition

```
#define _BW(N)           __attribute__((C2RTL_bit_width(N)))
#define _TYPE(N)         __attribute__((C2RTL_type(N)))
#define _C2R_FUNC(N)     __attribute__((C2RTL_function(N)))
#define _C2R_MODULE_     __attribute__((C2RTL_module))

typedef uint16_t  UINT16  _BW(16);
typedef uint32_t  UINT20  _BW(20);
```

Figure 2.6: Macro and inheritance of hardware-specific attributes

Some attributes are defined as follows to specify hardware-specific attributes. No special instructions nor pragmas are required in the user code for tuning performance. We use macros to shorten these attribute declarations. In addition, the given hardware attributes can be inherited using typedef or using declaration (Fig.2.6).

function the target function of RTL generation along with the number of pipeline stages

module the target function for hierarchical module instantiation

bit_width specifies the bit width of the signal

type (state) the variable as the register

type (memory) the variables as a single-port RAM

type (memory_dual_port) the variables as dual-port RAM

type (direct_signal) indicates that these Input/Output ports do not pass through registers

2.4 Generation Target

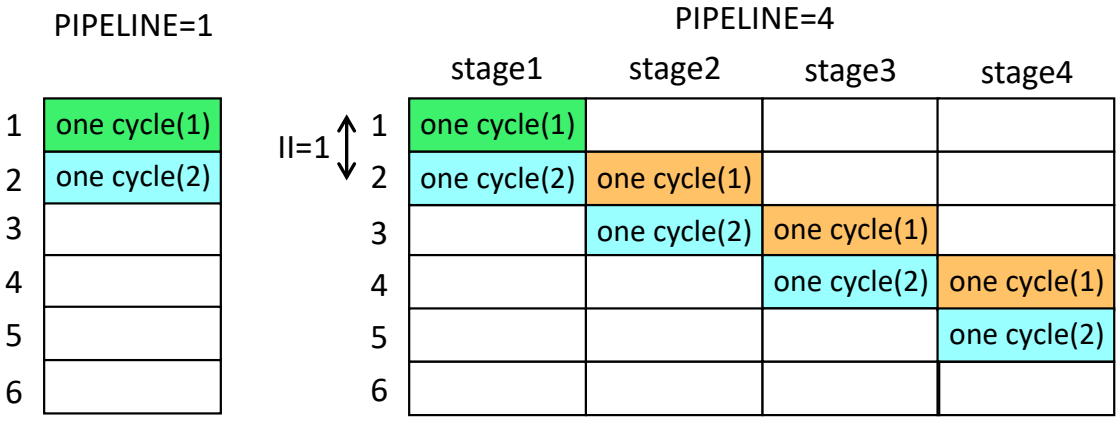

Figure 2.7: One cycle behavior

The function with the attribute of C2RTL_function along with the number of pipeline stages is the target of the single clock synchronous circuit's hardware generation, and the defined macro of _C2R_FUNC() is used in the code, as shown in Fig.2.3. The input and output ports of the generated module are defined from the arguments of the target function, and the argument referred in the function is inferred as the input, while the assigned argument is inferred as the output of the hardware module (line 7). The number of pipeline stages is specified by the user. One stage (not pipelined) is directed in this code.

Here, the description of the signal flow from input to output, which does not overlap any registers, is called "one cycle behavior". When the single pipeline stage is directed, it describes the register transition in one clock cycle like the traditional HDLs. When more pipeline stages are directed, "One cycle behavior" is realized through the number of pipeline stages. The initiation interval (II) of the generated hardware module is always one, unlike the module compiled by HLS, which does not always have a single II (Fig.2.7).

2.5 Description Rules

The input code must be written according to the following rules.

- All variables handled by LLVM-C2RTL are derivation from integer or float variables, including scalars, arrays, and aggregated types (struct).
- The bit width of the variable is inferred and extended automatically through arithmetic operations.
- Operations are described by a combination of C/C++ language built-in operators.
- Assignments are evaluated in the same order as the C/C++ process flow.
- Assignments to static variables with the state attribute become non-blocking assignment to the register of the sequential logic, and the new value is reflected in the next clock.
- On-chip RAM modules are generated by adding a memory attribute to an array of any type of variables, and the access to the RAM module is created using the array index as the address.
- A chain of assigning to variables without state or memory makes a chain of combinational logic.
- Loops are inline expanded. It is the same as the for statement of verilog rather than the C program and is used to duplicate the circuit.

2.6 Coding Restrictions

Writing a circuit static structure using C/C++ code has some restrictions and limitations. Unlike HLS, access to external memory is not created automatically. The DMA with bus interface must be explicitly coded. A simplified example of an image filter that directly inputs and outputs through a video stream interface suitable for hardware is shown in Fig.2.8.

- The behavior code must not contain non-constant loops.
- Heap or dynamic memory allocation is not allowed (Pointers can be used as long as they point to static variables).
- After assignment to a variable with the state attribute, it is not reassigned a value created by referring to the state variable (define-refer-define chain), because this contradicts the definition of "one cycle behavior".
- Access is read and written once to the array with the memory attribute in the "one cycle behavior".

By writing code based on the rules and restrictions above, the input code can be executed as a algorithm program by compiling with a normal C/C++ compiler, while verilog-HDL and simulation C code that behave as the input code in the specified clock cycles are generated by compiling with LLVM-C2RTL.

```
template <typename T>
struct Pixel {
  BIT vld;
  T val;
};

template <typename T, int WIDTH>
class Filter {
 protected:
  int x _TYPE(state);
  T win[9] _TYPE(state);
  T lbuf[2][WIDTH] _TYPE(memory_dual_port);
  virtual T filt(T* w);

 public:
  void step(Pixel<T>& pix_src, Pixel<T>& pix_dst) {
    T pix1 = lbuf[1][x];
    if (pix_src.vld) {
      win[0] = win[1];
      win[1] = win[2];
      win[2] = lbuf[0][x];
      win[3] = win[4];
      win[4] = win[5];
      win[5] = pix1;
      win[6] = win[7];
      win[7] = win[8];
      win[8] = pix_src.val;
      lbuf[0][x] = pix1;
      lbuf[1][x] = pix_src.val;
      x = (x == WIDTH - 1) ? 0 : x++;
    }
    pix_dst.val = filt(win);
    pix_dst.vld = pix_src.vld;
  };
};
```

Figure 2.8: C++ example code of abstract image filter class

2.7 Generated Code

In our framework, this code is converted to LLVM-IR (Fig.2.4), then converted to C2RTL-IR, and the verilog code (Fig.2.9) and C simulation code are generated from C2RTL-IR at the same time. This C simulation code performs the cycle-accurate and bit-accurate simulation with the same interface (input and output arguments) as the original code, so that one clock cycle behaviour is executed by each function call. All elements of the simulation code correspond one-to-one with the verilog code, including registers, memories, signals, and operations, so

normally we only need to verify the C simulation code, and also internal signals such as memory are highly observable. This feature is very useful, because the C language development environment has a wealth of flexible, extensible, low-cost tools such as compilers, debuggers, and IDEs, compared to the hardware development environment.

```
 1: module func1(clk, rst_n, G_io_din_0, G_io_din_1, G_io_dout);
 2:   parameter M_ID = 0; // module ID
 3:   input        clk;
 4:   input        rst_n;
 5:   input [15:0] G_io_din_0; /// <0,65535> [U16];
 6:   input [15:0] G_io_din_1; /// <0,65535> [U16];
 7:   output[15:0] G_io_dout; /// <0,65535> [U16];
 8:   reg   [15:0] G_io_dout;
 9:   reg   [15:0] G_func1_stt;
10:
11:  wire       CP_func1_B1_F1 = G_func1_stt == 16'h0;
12:  wire[15:0] I_func1_B1_2   = G_io_din_1 + G_io_din_0;
13:  wire[15:0] DM_1           = (CP_func1_B1_F1) ? I_func1_B1_2 : 16'h0;
14:  wire       CE_0           = ! CP_func1_B1_F1;
15:  wire[15:0] I_func1_B3_0   = I_func1_B1_2 + 16'h1;
16:  wire[15:0] DM_2           = (CE_0) ? I_func1_B3_0 : 16'h0;
17:  wire[15:0] DM_3           = DM_1 | DM_2;
18:   always @ (posedge clk or negedge rst_n) begin
19:   if (rst_n == 1'b0) begin
20:     G_func1_stt <= 16'h0;
21:     G_io_dout   <= 16'h0;
22:   end /// (rst_n == 1'b0)
23:   else begin
24:     if (CP_func1_B1_F1)  G_func1_stt <= I_func1_B1_2;
25:     G_io_dout   <= DM_3;
26:   end
27:   end /// always @ ...
28: endmodule
```

Figure 2.9: Generated verilog code from simple example code

2.8 RAM Description

In LLVM-C2RTL, synchronous RAM can be described by assigning memory attributes to array variables. The number of times that theses variables can be read and written is limited in "one cycle behavior". An array variable with the memory attribute can be read or written once in "one cycle behavior". This corresponds to single-port RAM. An array variable with memory_dual_port can simultaneously read with one index and write with another index within in "one cycle behavior". This corresponds to dual-port RAM. The behavior of reads and writes to the same index in the same cycle, such as write-first and read-first, can be directed by options to the compiler and reflected in both of verilog output code and simulation C code.

Figure 2.10 is an example of behavioral description for histogram calculation using dual-port RAM. This input code causes LLVM-C2RTL to generate the verilog code that includes the description to infers a RAM module as shown in the figure

2.11. This RAM module description can be easily replaced with a technology-specific RAM instance using ifdef.

```
struct HistIF {
  UINT8 idx;
  UINT32 din;
  UINT32 dout;
} _TYPE(direct_signal);

class HistL {
 private:
  UINT32 mem[256] _TYPE(memory_dual_port);
  UINT32 din_d _TYPE(state);
  UINT8 idx_d _TYPE(state);
  UINT32 rdata;

 public:
  void step(HistIF& hif) {
    mem[idx_d] = rdata + din_d;
    idx_d = hif.idx;
    din_d = hif.din;
    rdata = mem[hif.idx];
    hif.dout = rdata;
  }
};
```

Figure 2.10: Histogram calculation model using dual-port RAM

```
`ifdef RAM_DUAL_PORT_W1ST_NODEF
`elsif Hist_RAM_NODEF
`else
module Hist_RAM_DUAL_PORT (clk, we, din, ce, w_addr, r_addr, dout);
  parameter M_ID = 0; // module ID
  parameter MEM_SIZE = 1024; // default memory size
  parameter ADDR_WIDTH = 12; // default memory address width
  parameter DATA_WIDTH = 16; // default memory data width
  parameter WE_WIDTH   =  1; // default write-enable width

  input                  clk ;
  input                  we ;
  input [DATA_WIDTH-1:0] din ;
  input                  ce  ;
  input [ADDR_WIDTH-1:0] w_addr, r_addr;
  output[DATA_WIDTH-1:0] dout;
  reg   [DATA_WIDTH-1:0] mem[0:MEM_SIZE-1];

  reg   [DATA_WIDTH-1:0] dout;
  always @ (posedge clk)
  if (ce == 1'b1) begin
    if (we == 1'b1) begin
      mem[w_addr] = din; /// Write1st (DualPort)
    end
    dout = mem[r_addr]; /// (DualPort)
  end

endmodule
`endif
```

Figure 2.11: Generated dual-port RAM inference HDL code

2.9 C++ Extensions

The current incorporated LLVM version is 11.0.0; hence, our framework supports up to C11 standard [25] compliant C codes and up to ISO C++ 2017 standard [26] compliant C++ codes. These C/C++ standard supports polymorphism with virtual functions, generic programming, and preprocessing using template, constexpr functions, lambda functions, and auto type inferences. Owing to these features of the modern C++ language, we can create reusable codes for the change in the bit width of variables, depth of RAM, type of operator, data structures and number of instances and algorithms.

2.10 C++ Example Code

Figure 2.12 represents a code example of FIFO logic, which handles a user defined data structure. The example code comprises an interface definition, a class definition, an alias declaration, and a HDL generation part. To use classes including encapsulation and virtual functions can improve the reusability of the hardware

library. Also, static_assert prevents users from misusing, and alias are useful for grouping parameters or adding new constraints to parameters.

```
 1: template <typename T>
 2: struct FifoIF {                                    Interface definition part
 3:   BIT ren;
 4:   BIT wen;
 5:   BIT empty;
 6:   BIT full;
 7:   T din;
 8:   T dout;
 9: } _TYPE(direct_signal);
10:
11: template <typename T, int depth, int idx_w>
12: class FifoImpl {                                   Class definition part
13:  static_assert(depth == (1 << idx_w));
14: private:
15:   T dbuf[depth] _TYPE(state);
16:   unsigned int ridx _BW("idx_w") _TYPE(state);
17:   unsigned int widx _BW("idx_w") _TYPE(state);
18: public:
19:   void step(FifoIF<T>& fifo_if) {
20:     bool empty = (ridx == widx);
21:     bool full = (ridx == ((widx + 1) & C2R_MAXVAL(idx_w)));
22:     fifo_if.empty = (empty) ? 1 : 0;
23:     fifo_if.full = (full) ? 1 : 0;
24:     fifo_if.dout = dbuf[ridx];
25:     if (fifo_if.ren == 1 && !empty) {
26:       ridx++;
27:     }
28:     if (fifo_if.wen == 1 && !full) {
29:       dbuf[widx] = fifo_if.din;
30:       widx++;
31:     }
32:   }
33: };                                                  Alias
34: template <typename T, int depth_w>
35: using Fifo = FifoImpl<T, (1 << depth_w), depth_w>;
36:
37: struct ST {  BIT flag;  UINT32 data;};             RTL generation part
38: _C2R_FUNC(1)
39: void SyncFifo(FifoIF<ST>& fifo_if) {
40:   static Fifo<ST, 3> inst;
41:   inst.step(fifo_if);
42: }
```

Figure 2.12: C++ example code of FIFO logic

2.11 Using Generic Programming

The biggest advantage of taking C/C++ code as input and outputs Register Transfer Level code is that we can use the generic programming. Bacause the circuit description must has a statically determined structure, it can not be abstracted like dynamic polyformism in software programs, but it can be abstracted with the help of pre-processing and meta-programming. By using these abstraction functions, it becomes possible to implement algorithms that can not be directly described in conventional HDL.

For example, consider the case of a circuit that outputs an index that gives the maximum value for an arbitrary number of input arrays. A description that simply compares input arrays using a for statement would result in a circuit with a long critical path as shown on the left in the figure 2.13, but a binary search circuit as shown on the right is optimal.

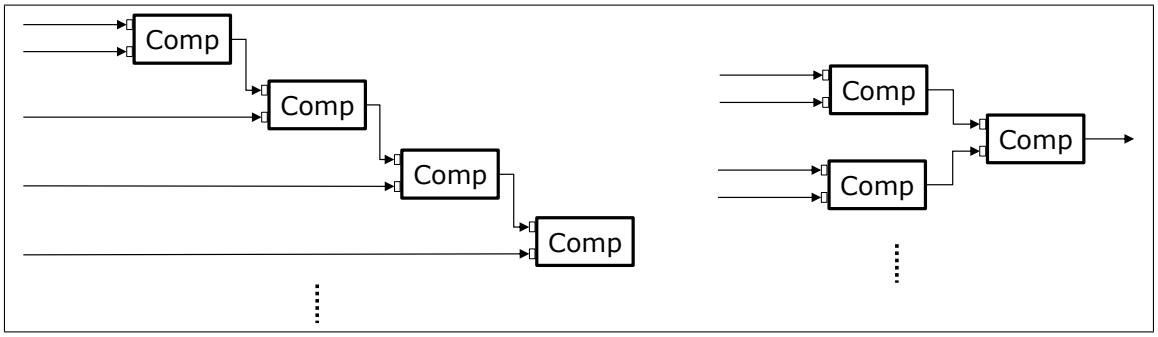

Figure 2.13: Example of layout of comparators for finding the maximum value

In order to describe such an optimal circuit in verilog, recursive instantiation of modules is not allowed and it is necessary to use the technique of using the generate syntax and passing part of the input signal array to the lower module. In contrast, in LLVM-C2RTL, maximal search with binary tree can be implemented simply by using template classes. (Fig.2.14). Also, by using a template of comparison conditions, binary search can be templated with the type of operators (Fig.2.14).

```
template <typename T, int N>
class BinMaxL {
  static int getIdx(T* vals, T& tmp_max) {
    T max_a, max_b;
    int idx_a = BinMaxL<T, (N + 1) / 2>::getIdx(vals, max_a);
    int idx_b = BinMaxL<T, N - (N + 1) / 2>::getIdx(vals + (N + 1) / 2, max_b) + (N + 1) / 2;
    if (max_a < max_b) {
      tmp_max = max_b;
      return idx_b;
    } else {
      tmp_max = max_a;
      return idx_a;
    }
  }
};

template <typename T>
class BinMaxL<T, 1> {
  static int getIdx(T* vals, T& tmp_max) {
    tmp_max = vals[0];
    return 0;
  }
};
```

Figure 2.14: Code of Binary Search for Maximum Value

```
template <typename T>
struct CMax {
  static bool cond(T& a, T& b) {
    return (a < b);
  }
};

template <typename T, typename C, int N>
struct BinSearch {
  static int getIdx(T* vals, T& tmp) {
    T tmp_a, tmp_b;
    int idx_a = BinSearch<T, C, (N + 1) / 2>::getIdx(vals, tmp_a);
    int idx_b = BinSearch<T, C, N - (N + 1) / 2>::getIdx(vals + (N + 1) / 2, tmp_b) + (N + 1) / 2;
    if (C::cond(tmp_a, tmp_b)) {
      tmp = tmp_b;
      return idx_b;
    } else {
      tmp = tmp_a;
      return idx_a;
    }
  }
};

template <typename T, typename C>
struct BinSearch<T, C, 1> {
  static int getIdx(T* vals, T& tmp) {
    tmp = vals[0];
    return 0;
  }
};
```

Figure 2.15: Code of templated binary search

Chapter 3

Transformation Methodology of LLVM-IR into Hardware Instance

3.1 Principle

The LLVM compiling process consists of three parts: the front-end that converts the input code to an intermediate representation (LLVM-IR), the middle-end that optimizes the LLVM-IR by applying multiple passes, and the back-end that converts the LLVM-IR into a machine language specific to the CPU architecture (3.1).

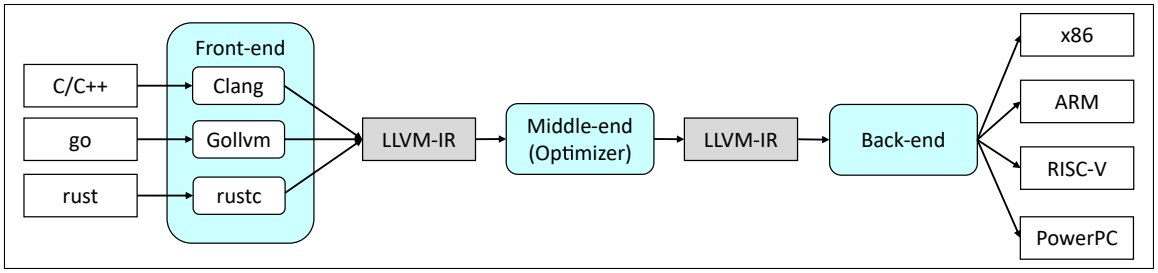

Figure 3.1: Three stage strategy of LLVM

Some LLVM-based HLS tools generate HDL at the back-end stage, but the DAG at the back-end stage may have been optimized by ignoring register and memory attributes. Therefore, we adopted Clang as the front-end and implemented passes to convert LLVM-IR to C2RTL-IR as a structural intermediate representation, and finally generate verilog-HDL in the middle-end Opt process, as illustrated in Fig.2.1.

A C/C++ program comprises a data-flow part, such as assignment to a variable and operation, and a control-flow part comprising function calls and branch instructions. Because the data-flow description has almost the corresponding description in RTL, the main problem is how to convert the control flow. This process is achieved by the following functional steps:

1. Detect RTL-generated target functions, extract input / output ports, and extract the specified hardware attribute information.

2. Convert the input code to a single static assignment (SSA) form and create a control flow graph (CFG).

3. Convert the original CFG to a flat CFG via loop-unrolling and full-inlining.

4. Reduce the branch and join and memory access of the flat CFG.

5. Lower the CFG to the DAG.

6. Insert the pipeline boundaries based on the specified number of stages and data dependency.

7. Generate the verilog-HDL code and simulation C code.

3.2 Compiling Flow from C/C++ Code to RTL

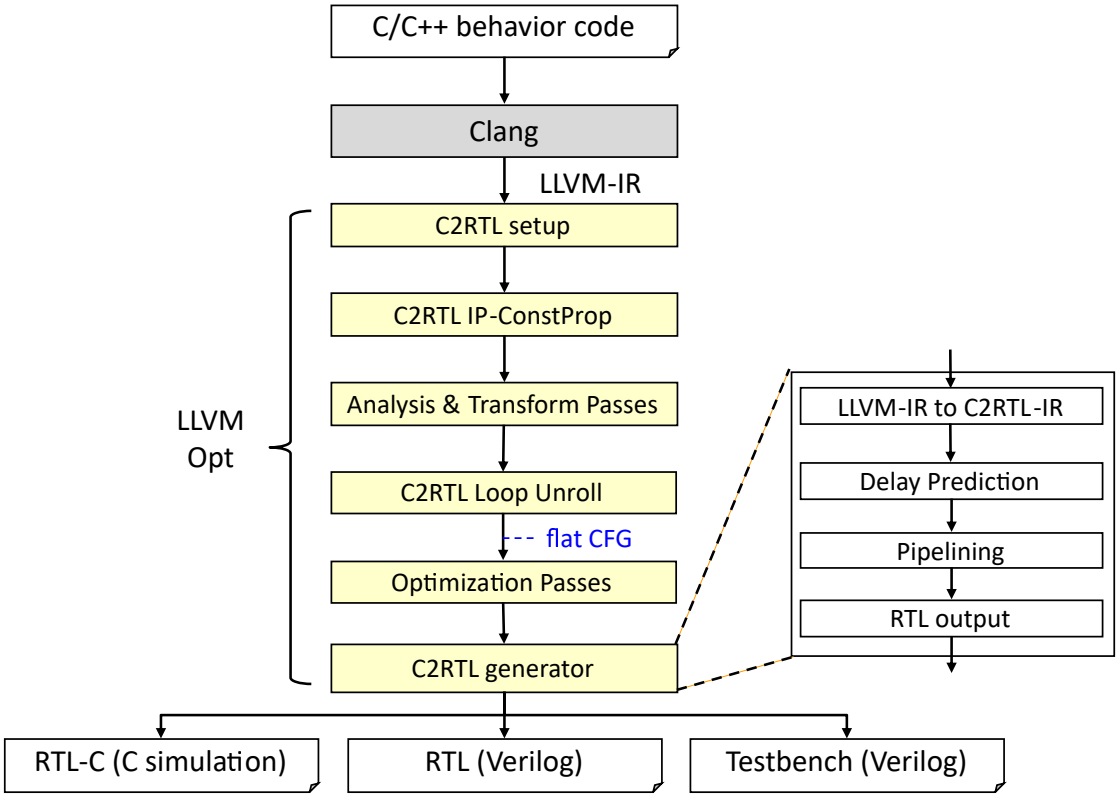

Figure 3.2: LLVM-C2RTL compiling flow

The module-level LLVM-C2RTL compilation flow will be explained, as illustrated in Fig.3.2.

First, a compiler front-end tool, Clang, performs C and C++ syntax parsing and preprocessing on the user written code, converts it to the SSA-formed intermediate language (LLVM-IR), and applies first optimizations with -O1 flag.

Next, because the Opt process commences, the first block, C2RTL setup, detects all the generation target functions to which the function attribute is attached and extracts all other LLVM-C2RTL attributes in the functions.

The second block, C2RTL IP-ConstProp, applies constant propagation in almost the same way as ConstProp in the LLVM. Unlike the ConstProp, it processes only the values that can be reached from the target functions, and it also propagates the constants to the variable and hardware attributes, such as the bit width and number of specified pipeline stages.

The third block, Analysis & Transform passes block, contains some common LLVM analyses and transform passes, including dominator tree construction, canonicalization of natural loops, loop-closed SSA form, scalar evolution, and strength reduction.

The fourth block, C2RTL Loop Unroll, inline passes all function calls to convert the original CFG of the target function into a flat CFG, detects all loops inside reachable functions, and then unloops them. All function calls must be non-recursive or expandable into a finite number of function calls if recursive. In addition, all loops must have constant times. Because we do not use HLS, this limitation is similar to that of conventional HDLs. LLVM-C2RTL exhibits errors when any violation is detected. After applying this pass, the CFG may still contain branches and joins but no function calls nor loops (Fig.3.3).

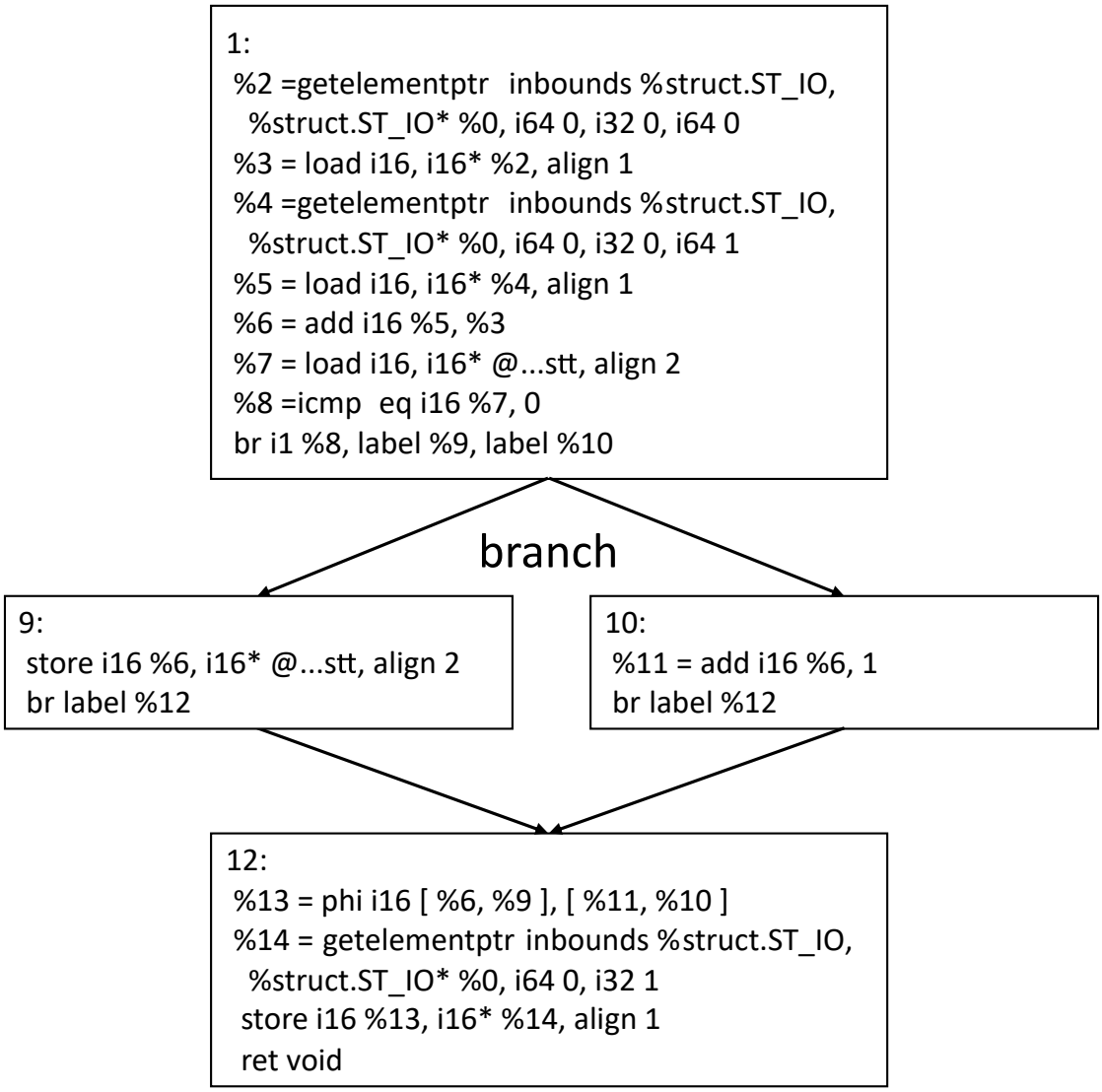

Figure 3.3: Loop unrolled LLVM-IR

The fifth block contains common LLVM-IR optimization passes, Sparse Conditional Constant Propagation, Simplify the CFG, and Dead Code Elimination. These optimizations reduce the hardware area and improve the performance without changing the behavior. Note that optimization passes that may change the order of assignments and references, are not applied because they may change the structure of the finite state machine.

The final block, C2RTL generator, re-analyses LLVM-IR, converts it to C2RTL-IR, and lowers the C2RTL-IR. The bit width of the calculation result is determined from the bit width of the source variables and arithmetic type. In addition, LLVM-C2RTL arranges the pipeline boundaries to minimize the maximum delay of one clock cycle based on the circuit delay prediction. Finally, verilog and C-simulation codes are generated from C2RTL-IR. Generation is a direct one-to-one correspondence; hence, their consistency is always maintained. More details on these LLVM-IR to C2RTL-IR conversion steps are given in the next section.

3.3 Conversion from LLVM-IR to C2RTL-IR and Lowering

C2RTL-IR is an intermediate representation for generating both the verilog and C-simulation codes. C2RTL-IR has two levels: The upper level corresponds to LLVM-IR, which contains control statements such as branch instructions. The lower level is a set of structural elements (nodes) that make up a circuit, such as constant values, memory elements, and arithmetic operations, as shown in Fig.2.5.

The C2RTL Generator pass takes the LLVM-IR, whose CFG is already flattened, as an input, and first converts the values and instructions to each corresponding C2RTL-IR node, creating edges from the nodes to other referred nodes for an arithmetic operation or assignment. Then, some instruction nodes and arithmetic operation nodes will be lowered into nodes that directly correspond to the RTL, as well as creating DAG through certain steps.

3.4 Conversion from Branch

The code example presented in Fig.2.3 includes the conditional branch, therefore, the unlooped LLVM-IR (Fig.3.3) also includes a branch. Local variables expressed in SSA form as %17, use the phi instruction at the joint point. In C2RTL-IR, the behavior of the original code is realized by executing all the operations related to local variables in all branches, and replacing the phi instruction with a multiplexer whose condition will be the branch condition (Fig.2.5).

Assignments to state registers and memory variables are expressed by memory access instructions like store. And these instructions in branches are replaced with conditional assignments as shown in Line 24 of generated verilog code (Fig.2.9). If they are assigned in multiple branches, the index of the memory variable and the assigned values are merged using a multiplexer like the phi functions for local variables.

3.5 Conversion from Memory Access Instructions

```
%class.FifoImpl = type { [16 x i32], i32, i32 }
%struct.FifoIF = type { i8, i8, i8, i8, i32, i32 }
...
%31 = getelementptr inbounds %struct.FifoIF, %struct.FifoIF* %1, i64 0, i32 4
%32 = load i32, i32* %31, align 1
%33 = zext i32 %6 toi64
%34 = getelementptr inbounds %class.FifoImpl , %class.FifoImpl* %0, i64 0, i32 0, i64 %33
store i32 %32, i32* %34, align 1
```

Figure 3.4: Example of an LLVM-IR code with memory access instructions

Figure 3.4 presents the LLVM-IR instructions corresponding to Line 29 of the FIFO module code (Fig.2.12). In LLVM-IR, all local variables are expressed in the SSA form as %33 and can be simply converted to wires or registers; however, access to the members of the array or struct is expressed by the load and store instructions of obtaining addresses (getElementPtr called GEP) that cannot be converted in a straight forward manner.

As the first argument of GEP is the structure or array type, the subsequent arguments are indices of the nested struct (or array). To convert GEP instructions to the node corresponding to the pointed data, LLVM-C2RTL holds all the existing information regarding the struct and type of attached attributes (indicating wires, registers, or memory) in the first analysis. After the struct specified as the first argument is analyzed, the nested struct or struct member is extracted as the assignment source or destination. In this example, %31 is the fifth element of the struct for the I/O, and thus it points to din, and %34 is the first member of the class, and thus it points to dbuf. By tracing the structure's information in the compiler, the destination and source of the assignment can be replaced with the corresponding data node.

Subsequently, if there is no variable in the index arguments of the GEP instruction, then the target data element is obtained directly. If the index argument is a variable, the assignment to an element of a variable array is converted into RTL that uses the index of the array (upper part of Fig.3.5), and the assignment to an element of the struct array (Fig.2.12) is converted into a set of conditional assignments of the variables (lower part of Fig.3.5).

```
// case of assignment to an element of array
wire[ 2:0] N_2 = G_SyncFifo_inst_widx[2:0];
always @ (posedge clk or negedge rst_n) begin
  G_SyncFifo_inst_dbuf[N_2] <= G_sfifo_if_din;
end
```

```
// case of assignment to an element of array of structure
wire[ 2:0] N_06 = G_SyncFifo_inst_widx[2:0];
always @ (posedge clk or negedge rst_n) begin
  if (N_06 == 3'h0) begin
    G_SyncFifo_inst_dbuf_0_flag <= G_sfifo_if_din_flag;
    G_SyncFifo_inst_dbuf_0_data <= G_sfifo_if_din_data;
  end
  if (N_06 == 3'h1) begin
    G_SyncFifo_inst_dbuf_1_flag <= G_sfifo_if_din_flag;
    G_SyncFifo_inst_dbuf_1_data <= G_sfifo_if_din_data;
  end
  if (N_06 == 3'h2) begin
    G_SyncFifo_inst_dbuf_2_flag <= G_sfifo_if_din_flag;
    G_SyncFifo_inst_dbuf_2_data <= G_sfifo_if_din_data;
  end
  if (N_06 == 3'h3) begin
    G_SyncFifo_inst_dbuf_3_flag <= G_sfifo_if_din_flag;
    G_SyncFifo_inst_dbuf_3_data <= G_sfifo_if_din_data;
  end
  ...
end
```

Figure 3.5: Example of a converted verilog code from GEP instructions

3.6 Bit width Calculation

LLVM-C2RTL automatically calculates the required bit width for variables and uses it for the output HDL. This is achieved by holding and propagating the maximum and minimum possible values of the variables. For constants, the constant value itself is held as the constant's maximum and minimum value.

It starts by giving input and register variables the maximum and minimum value according to specified bit width by the attribute, or the bit width of the built-in type, if not specified. Then the maximum and minimum possible values are calculated to all intermediate and output data with the operation (op) and maximum and minimum values of source data using Eq.(3.1), where $x.H$ and $x.L$ are the maximum and minimum values of the variable x, $y.H$ and $y.L$ are those values of the variable y. For example, when the addition of an unsigned 8-bit integer and a signed 8-bit integer is assigned to a signed variable, the calculated range is $[0 - 128, 255 + 127] = [-128, 382]$, and the bit width becomes 9-bit.

$$MRange(\mathbf{op}, [x.L,\ x.H]\ , [y.L,\ y.H]\ ) = \min\left(z0, z1, z2, z3\right) \tag{3.1}$$

where

$$\begin{aligned} z0 &= x.L\ \mathbf{op}\ y.L \\ z1 &= x.L\ \mathbf{op}\ y.H \\ z2 &= x.H\ \mathbf{op}\ y.L \\ z3 &= x.H\ \mathbf{op}\ y.H \end{aligned}$$

3.7 Delay Prediction

LLVM-C2RTL predicts the circuit delay within one clock cycle by using propagation time of the signals' MSB and LSB. The tool aims to give the user a rough estimate of the magnitude of the logic delay and provide an automatic pipelining, which is described in the following.

LLVM-C2RTL first evaluates the delays of each built-in operators, such as arithmetic, comparison, shift, and conditional operators, based on a three-parameter model. The basic propagation delay L in an arithmetic unit is common for MSB and LSB. The carry delay C is defined for an arithmetic unit having a carry to the highest bit, such as an adder or a multiplier. Division is decomposed into a combination of subtractions, comparisons and multiplexers, and the maximum delay after decomposition is MSB of the comparison. Therefore, in our model, the delay of MSB of an arithmetic unit is always greater than that of LSB. The synchronization flag F is used when the arithmetic unit needs to wait for the arrival of all bits, then LSB propagation time is aligned to the MSB.

Then, the propagation time of the destination signals corresponding to the source signals is defined using Eqs.(3.2)-(3.4) (where $S(i).TM$ and $S(i).TL$ are propagation time of MSB and LSB of i-th source signals respectively). Finally, the destination signals propagation time $D.TM$ and $D.TL$ are respectively obtained as the maximum value of $D(i).TM$ and $D(i).TL$ (Fig.3.6).

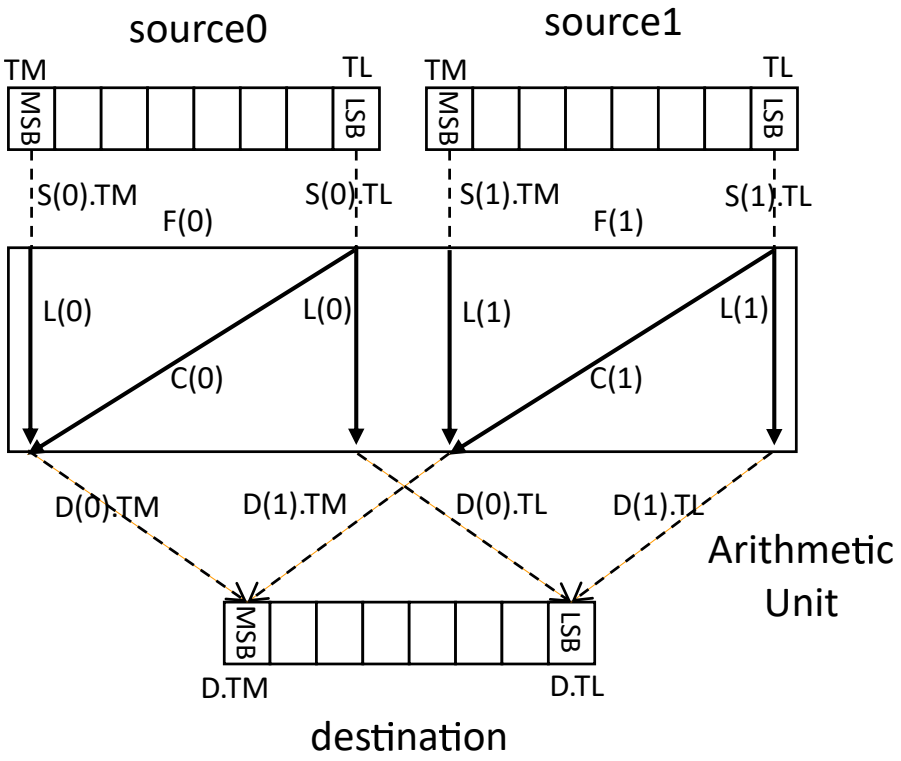

Figure 3.6: Model of Arithmetic Unit delay

$$TL(i) = \begin{cases} S(i).TM & (F(i) = 1) \\ S(i).TL & (F(i) = 0) \end{cases} \tag{3.2}$$

$$D(i).TL = TL(i) + L(i) \tag{3.3}$$

$$D(i).TM = \max\left(S(i).TM + L(i), TL(i) + C(i)\right) \tag{3.4}$$

$$D.TM = \max\left(D(i).TM \,|\, i \in I\right) \tag{3.5}$$

$$D.TL = \begin{cases} D.TM & (Bits = 1) \\ \max\left(D(i).TL, i \in I\right) & (Bits \neq 1) \end{cases} \tag{3.6}$$

Next, the pipeline delay is evaluated as the signal delay between clock boundaries of the pipeline stage (Fig.3.7). Starting with a propagation time of the source (register or input port) signals of 0, calculate the propagation time of the destination signals of each arithmetic unit with the source signals, then the final propagation time is estimated based on the maximum propagation time of the signals' destination (register or output port).

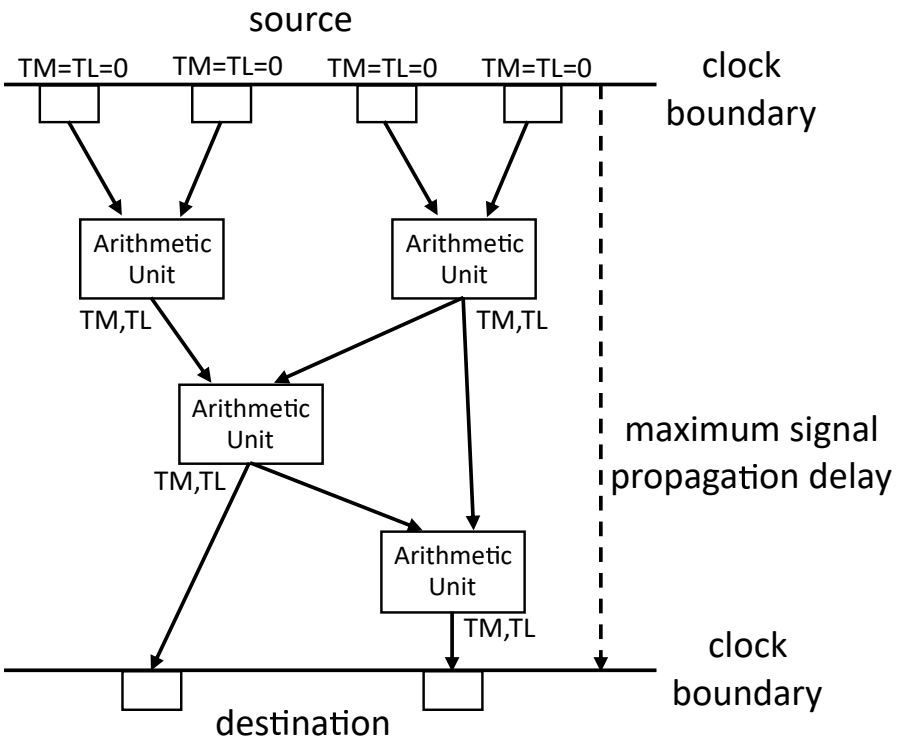

Figure 3.7: Model of pipeline delay

We explain the delay evaluation process of the complex arithmetic units based on multiplication and division examples (Fig.3.8). The mul instruction in LLVM-IR is first converted into the MULT operation node in C2RTL-IR. There are many ways to implement multiplication. Therefore, the implementation method is selected by the compile-time setting of LLVM-C2RTL. Depending on this option, MULT operation is lowered to the HDL level representation as a direct representation, an array-type multiplier, or a multiplier using the Booth-Recorder. The default setting is direct expression, and it is simply converted into a verilog-HDL multiplication operator *, and the final circuit depends on the logic synthesis process.

Similarly, operators that exist in C language but not supported by HDL are decomposed into HDL level expressions when lowering the C2RTL-IR. For example, dividing is decomposed into long division, which causes an extremely long delay but becomes practical when applying the pipelining described next. Floating operations are also implemented.

Finally, the delay of each HDL level arithmetic units such as +, -, & are given by each three model parameters and calculated depending on the bit width of signals, and the delay of complex arithmetic unit is obtained by concatenating those units.

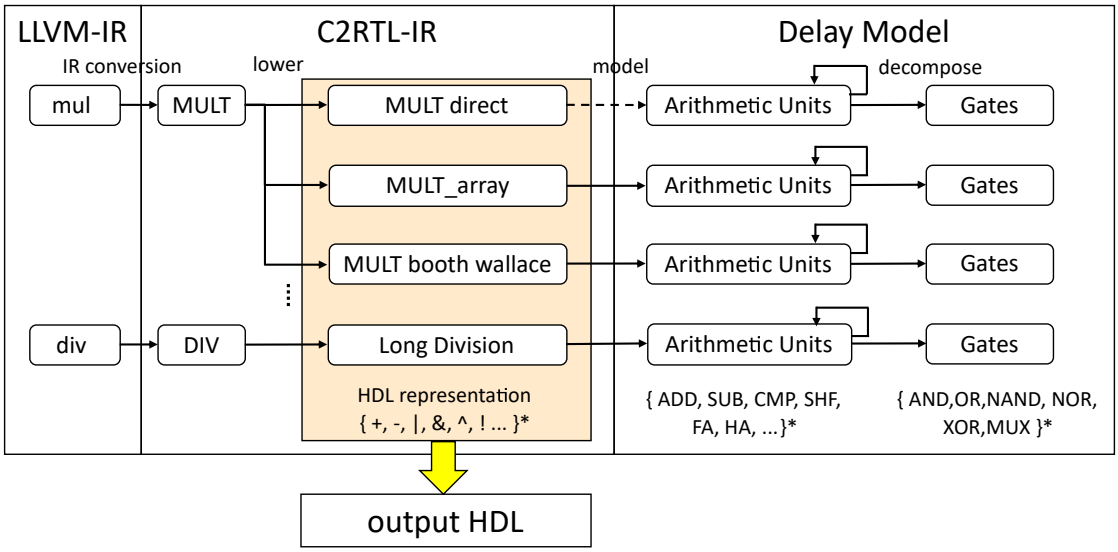

Figure 3.8: Delay prediction process

3.8 Pipelining

```
 1: _C2R_FUNC(3)
 2: void func3(UINT16& din, UINT16& dout) {
 3:   static UINT16 stt _TYPE(state);
 4:   bool b = (stt == 0);
 5:   UINT16 a = b ? din : din + 1;
 6:   if (b) {
 7:     stt = a;
 8:   }
 9:   UINT16 c = (stt + a) * 3;
10:   dout = c;
11: }
```

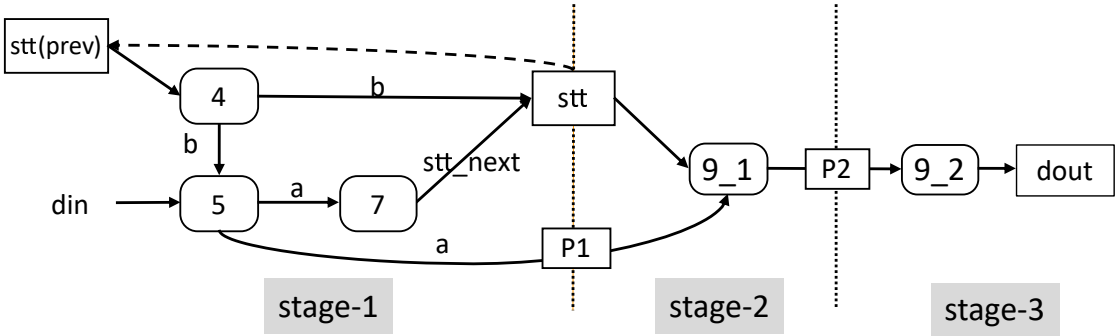

Figure 3.9: Example of pipeline assignment

LLVM-C2RTL is a pipeline-oriented tool that always generates the HDL, which operates the described behavior through the pipeline. This pipeline logic feeds

input signals for every cycle, and after a certain clock cycle delay, the corresponding result is output. In other words, the II is always one clock cycle.

Retiming [27] optimizes the critical path delay by moving registers without changing the output logic for the input. In [28], the authors exploit a pipeline algorithm for the software and enable scheduling for the HLS, which significantly improves throughput. In contrast, LLVM-C2RTL realizes high throughput and high portability by pipelining at the HDL level when the output signal's clock cycles delay can be changed, such as multi-cycle instructions for processors. Then, retiming can be further applied at the logic synthesis phase to improve the operating frequency.

LLVM-C2RTL's pipelining process assigns a pipeline stage to each operation node in the data flow graph while satisfying the constraints imposed by the user code, which is presented in Algorithm 2. Figure 3.9 is an example of the pipeline stage assignment.

For the ExtractConstraints, because the reference after assignment to the state variable must be in the next stage, the pipeline boundary attribute is added to the edge between them. Similarly, in synchronous SRAM, a pipeline boundary attribute is added to the edge from the read address to the memory variable owing to the constraint in which the value can be read at one clock after the address is given. The maximum number of pipeline boundaries in the input-to-output path provides the minimum number of pipeline stages (three stages in this case). The data flow from the reference to the state variable to the assignment must end within one clock, and they are therefore grouped by removing the edges between them for their location in the same pipeline stage. Nodes (4), (5), and (7) are grouped in this case. A node that refers to a state node after being assigned must be placed after the assigned node. A node that refers to the state node before assignment must be placed before the assignment node. The initial arrangement is conducted by sequentially allocating the nodes to the pipeline stages from the input to the output side, according to the pipeline boundary edges.

The optimization process consists of two phases of simulated annealing methods. The first phase is to balance the pipeline delay at each stage while reducing the maximum pipeline delay given by Eq. (3.7). In addition, the cost function is defined as Eq. (3.8). The second phase is to minimize the pipeline registers, and the cost function is defined as Eq. (3.9), where P indicates the edges across the pipeline boundary and b_e represents the edge's signal bits. At each step of the simulated annealing, randomly selected non-attributed nodes are moved to the adjacent pipeline stage to create new assignments, unless they do not violate the restrictions. If the maximum pipeline delay increases because of recalculating each pipeline delay, the old assignment is adopted. If a generated random number between O and 1 is greater than *prob*, the new assignment is then adopted.

$$max\_pipe\_delay = \max\{pipe\_delay_i \mid i = 1...num\_stage\} \tag{3.7}$$

$$balance\_pipe\_cost = \sum_{i=1}^{num\_stage} (pipe\_delay_i)^2 \tag{3.8}$$

$$register\_cost = \sum_{e \in P} b\_e \tag{3.9}$$

Because the delay is modeled for each arithmetic unit, one step calculation is extremely fast, and the SA method becomes suitable if the number of iterations is increased to approach the global solution. The total processing time for pipelining is mostly negligible. For example, the pipeline of ten parallel 64-bit divisions to 16-stages is completed within 1 s.

In the code shown in Fig.3.9, three pipeline stages are specified and the arithmetic in Line 9 is further divided into nodes (9_1) and (9_2). After the pipeline boundaries are determined, pipeline registers are inserted at the edge across the pipeline boundaries (P1 and P2), and the data flow graph is updated.

Algorithm 1 Pipelining process

Input: data flow graph $G = (V, E)$
Input: arithmetic unit delay d_v (for all $v \in V$), signal Bits b_e (for all $e \in E$)
Output: assigned pipeline stage $A = \{A_v : v \in V\}$

```
 1: G' ← ExtractConstraints(G)
 2: A ← DoInitialAssignments(G')
 3: OptimizePipe(balance_pipe_cost)
 4: OptimizePipe(register_cost)
 5:
 6: procedure OptimizePipe(CostFunc)
 7:     while t > t_th do
 8:         A' ← ModifyAssignments(A)
 9:         max_pipe_delay ← CalculatePipeDelay(A)
10:         max_pipe_delay' ← CalculatePipeDelay(A')
11:         if max_pipe_delay' ≤ max_pipe_delay then
12:             cost ← CalculateCost(A, CostFunc)
13:             cost' ← CalculateCost(A', CostFunc)
14:             prob = exp((cost − cost')/t)
15:             if prob > Random(0, 1) then
16:                 A ← A'
17:             end if
18:         end if
19:         t ← UpdateThermal(t)
20:     end while
21: end procedure
```

3.9 Evaluation of Delay Prediction and Pipelining

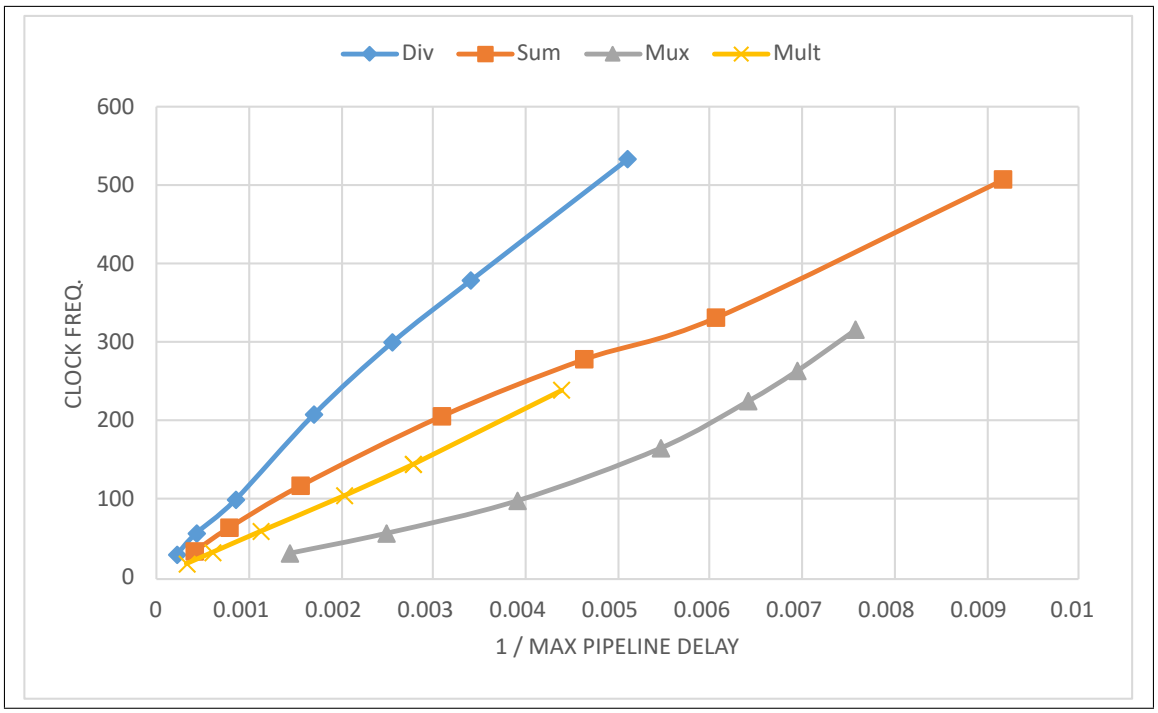

Figure 3.10: Clock frequency result of pipelining

Figure 3.10 presents the inverse of the maximum pipeline delay and post-synthesis clock frequency due to pipelining. We pipelined the chained logic of divisions, additions, multiplexers, and multiplications with 1, 2, 4, 8, 12, 16, and 24 stages. A clock frequency that is almost proportional to the number of stages is obtained, and the accuracy of the pipelining is ensured. However, a model that reflects the difference between devices and technologies that can absorb the variation of the arithmetic units will be introduced in the future.

3.10 HDL and Simulation Code Output

Because C2RTL-IR corresponds to the complete circuit structure, all nodes are output as variable declarations or instructions of verilog-HDL. Simultaneously, C code with the same structure as HDL is output. This C code executes the behavior of one clock cycle for a single call.

Figure 3.11 represents the RTL equivalent simulation code correspond to output verilog code (Fig.2.9) generated with simple code input (Fig.2.3). The first part (lines 3-5) is the part that substitutes the argument corresponding to the input port into the internal structure. Lines 13-24 correspond equivalently to the verilog code. In the actual output code, the corresponding line numbers in the original input code are indicated in the comments. The last part (lines 30-33) is the part

that assigns the argument corresponding to the output port from the internal structure member corresponding to the register. The macro of _U32_BP is defined as shown in Fig.3.12. Because C/C++ language do not have the arbitrary bit precision type, it is used to round values to arbitrary bit precision, and it can be turned off for faster simulation.

```
 1: TOP_FUNC_TYPE TOP_FUNC_NAME (ST_IO& io, ST_func1 *G)
 2: {
 3:   //// input wrapper :
 4:    G->_gw.io_din_0 = _U32_BP(16, (io).din[0]);
 5:    G->_gw.io_din_1 = _U32_BP(16, (io).din[1]);
 6:   const int func1_pSetup = 1; /// pipeStageCount = 1
 7: #if !defined(RTLC_In2FF)
 8:   //// latched outputs (current state) :
 9:    G->_gw.io_dout = _U32_BP(16, G->_out_d.io_dout);
10:   //// latched probes (current state) :
11:    G->_gw.func1_pSetup = _NO_BP (1, func1_pSetup);
12: #endif
13: {
14:   //// pipe-stage(1) : combinational-logic...
15: #if !defined(RTLC_FF2Out)
16: U32B CP_func1_B1_F1 = _NO_BP (1, G->func1_stt/*prv*/ == 0x0)   ;
17: U32B I_func1_B1_2 = _U32_BP(16, (U16B) (G->_gw.io_din_1 + G->_gw.io_din_0));
18: U32B DM_1    = _U32_BP(16, (CP_func1_B1_F1) ? I_func1_B1_2 : 0);
19: U32B CE_0            = _NO_BP (1, ! CP_func1_B1_F1)            ;
20: U32B I_func1_B3_0    = _U32_BP(16, (U16B) (I_func1_B1_2 + 1))   ;
21: U32B DM_2            = _U32_BP(16, (CE_0) ? I_func1_B3_0 : 0)   ;
22: U32B DM_3            = _U32_BP(16, (U16B) (DM_1 | DM_2))        ;
23: #endif /// !(RTLC_FF2Out)
24: #if !defined(RTLC_FF2Out)
25:   //// pipe-stage(1) : state-registers...
26:   if (CP_func1_B1_F1)  G->func1_stt      = _U32_BP(16, I_func1_B1_2);
27:   if (ROOT_CP       )  G->_out_d.io_dout = _U32_BP(16, DM_3)        ;
28: #endif /// !(RTLC_FF2Out)
29: }
30: #if !defined(RTLC_In2FF)
31:   //// output wrapper :
32:   (io).dout = _U32_BP(16, G->_gw.io_dout);
33: #endif
34: }
```

Figure 3.11: Logic equivalent simulation code generated simple example code

```
#if defined(EXT_BIT_PRECISE_C)
#define EXT_BIT_PRECISE_RTL_C   (EXT_BIT_PRECISE_C == 1)
#define EXT_BIT_PRECISE_RTL_C_N   (EXT_BIT_PRECISE_C == 0)
#else
#define EXT_BIT_PRECISE_RTL_C   0
#define EXT_BIT_PRECISE_RTL_C_N   0
#endif
#if BIT_PRECISE_RTL_C
#define _U32_BP_CORE(n, exp)   (((U32B)((exp) << (32 - (n)))) >> (32 - (n)))
#define _U64_BP_CORE(n, exp)   (((U64B)((exp) << (64 - (n)))) >> (64 - (n)))
#define _S32_BP_CORE(n, exp)   (((S32B)((exp) << (32 - (n)))) >> (32 - (n)))
#define _S64_BP_CORE(n, exp)   (((S64B)((exp) << (64 - (n)))) >> (64 - (n)))
#else
#define _U32_BP_CORE(n, exp)   (exp)
#define _U64_BP_CORE(n, exp)   (exp)
#define _S32_BP_CORE(n, exp)   (exp)
#define _S64_BP_CORE(n, exp)   (exp)
#endif
```

Figure 3.12: Macro difinition for arbitrary bit precision representation in simulation code

Chapter 4

System-Level Integration of RTL Modules on C/C++ Descriptions

4.1 Hierarchical RTL Generation

With the pipelining functionality of LLVM-C2RTL, even a complicated processing flow such as a processor can be efficiently designed by describing the behavior in software language. However, our previous work only supported hardware generation for individual modules, and it was difficult to describe all peripheral circuits, such as bus masters, and interconnect them in one function.

This work also supports system-level design to generate a hierarchical RTL code along with its equivalent simulation code. Hierarchical RTL generation is simple and achieved by system-level descriptions illustrated in the figure 4.2 in addition to module instantiations in the figure 4.2. The function with the C2RTL_module attribute (the macro of _C2R_MODULE_ is used instead in the code) becomes the target of hierarchical RTL generation, and submodules are instantiated by calling the functions that describe the submodule. Within the function for instantiating the submodule, the rule is to allow only to describe the wire, and to prohibit to describe the Finite State Machines or arithmetic circuit, but conditional statements and loops can be used.

```
_C2R_FUNC(PIPE_STAGES)
void AesEncRound(AesEncRoundDIF& rif) {
    static AesEncRoundL inst;
    inst.round(rif);
}

_C2R_FUNC(PIPE_STAGES)
void AesEncRoundLast(AesEncRoundDIF& rif) {
    static AesEncRoundL inst;
    inst.roundLast(rif);
}

_C2R_FUNC(1)
void AesEncFSM(AesEnc128IF& enc_if, AesEncRoundIFArr<11>& rif_arr) {
  static AesEncFSML inst;
  inst.run(enc_if, rif_arr);
}
```

Figure 4.1: Example code of module instantiations part of hierarchical RTL generation

```
_C2R_MODULE
void AesEnc(AesEnc128IF& enc_if, AesRoundKey128IF& round_key,
AesEncRoundIFArr<11>& rif_arr) {
    for (int i = 0; i < 11; i++) {
        rif_arr.rif[i].rkey = round_key.rkey[i];
    }
    for (int i = 0; i < 10; i++) {
        AesEncRound(rif_arr.rif[i]);
    }
    AesEncRoundLast(rif_arr.rif[10]);
    AesEncFSM(enc_if, rif_arr);
}
```

Figure 4.2: Example code of system-level instantiation part of hierarchical RTL generation

LLVM-C2RTL compiles user's system-level input code in two passes, as shown in Fig.4.3. In the first compilation, LLVM-IR is parsed and converted to C2RTL-IR to generate module unit verilog-HDL codes, as presented in Section 4. In the second compilation, like the first compilation, loop unrolling and constant propagation are performed for the target function first to support parameterized module instantiation. If a submodule function call is found inside the target function, the

compiler checks for data type and I/O port direction violations based on the assignment or reference within the submodules. Finally, a wrapper verilog-HDL code that instantiates the submodule is generated.

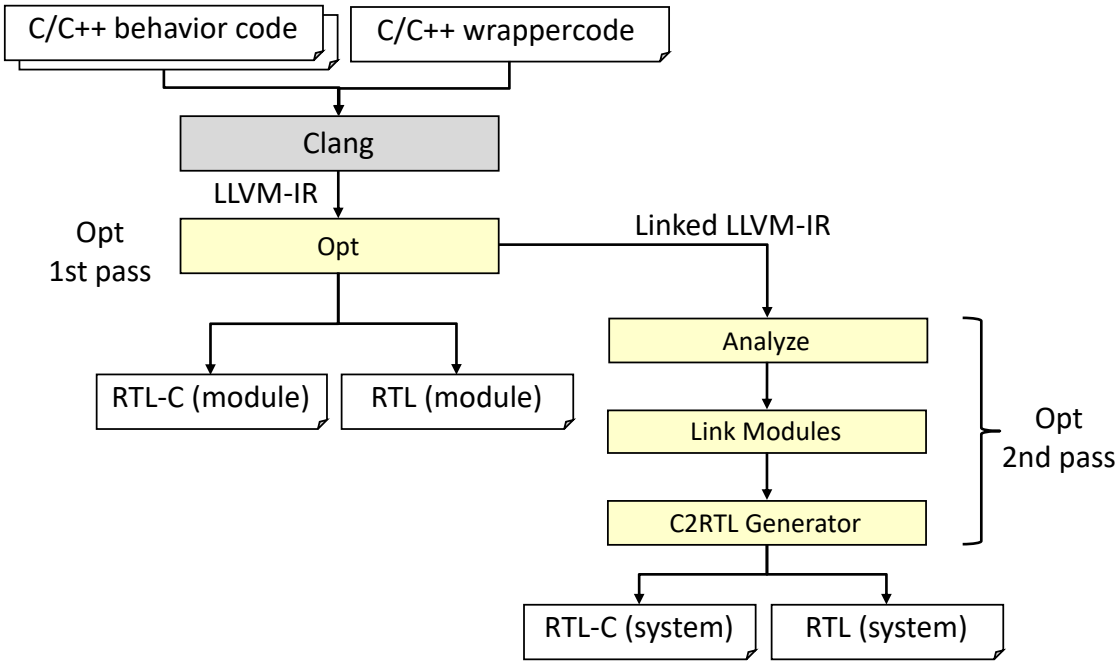

Figure 4.3: System level compile flow

4.2 Simulation Code Generation

Executing the original behavior code with each submodule call is not equivalent to the generated RTL. This inequivalence is because the convergence according to the combinational circuit is not completed. Each submodule must be iteratively executed until the output signals do not change (converged) in the simulation of a hierarchical module (Fig.4.4). Several RTL simulators take a similar approach[29][30][31]. However, LLVM-C2RTL creates optimized simulation codes that are faster in this situation.

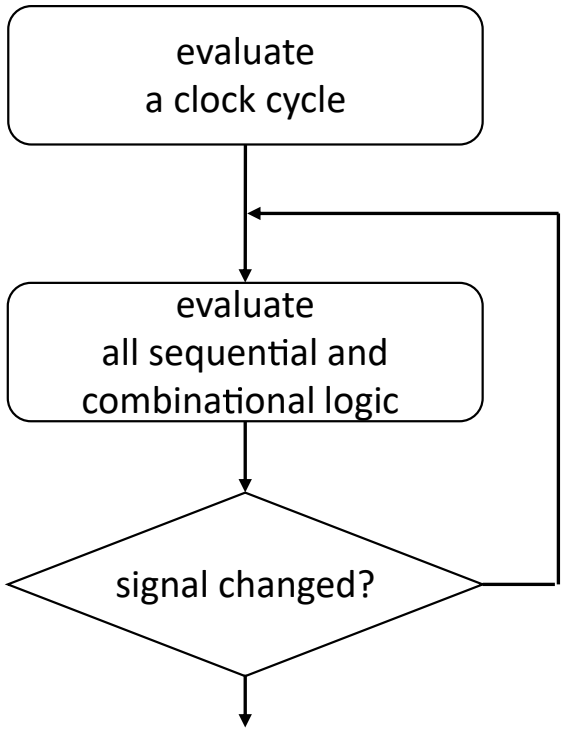

Figure 4.4: RTL simulation loop

First, LLVM-C2RTL creates the simulation code of every single module that is split into two parts based on the data flow graph. The InToFF routine executes only the data paths that update the register, while the FFToOut routine executes the other paths that make the output signals dependent on the current state (Fig.4.5). The single module simulation executes the InToFF routine after the FFToOut routine. A module whose FFToOut routine depends only on its current state and does not have an input-to-output combination path is called Moore type, otherwise, it is called Mealy type (Fig.4.6).

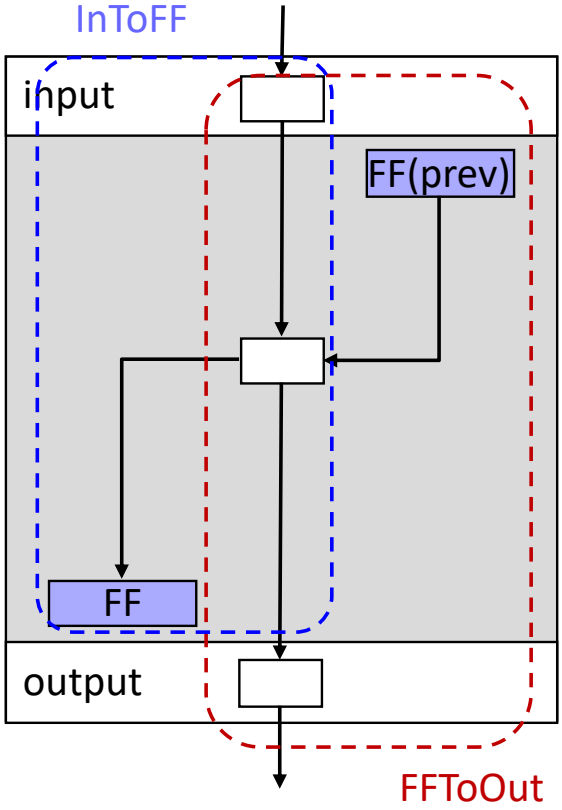

Figure 4.5: Split into two parts

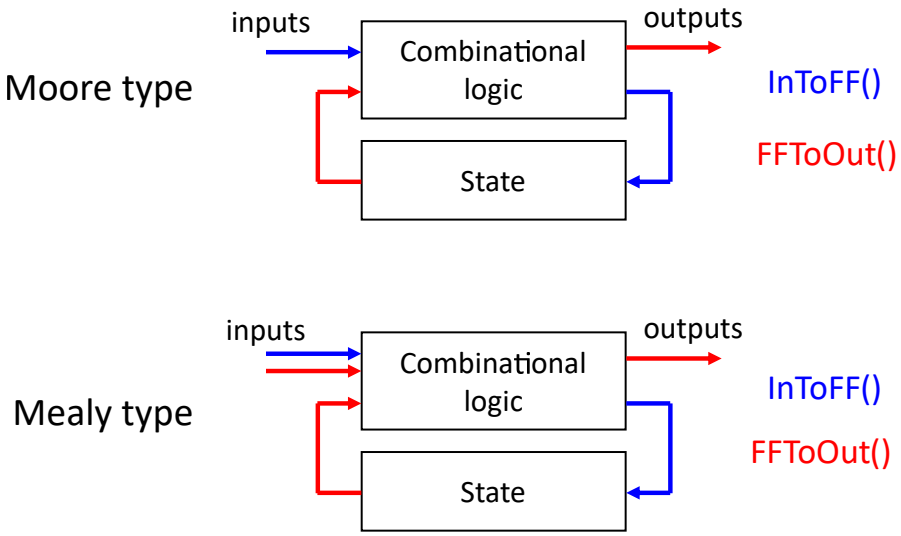

Figure 4.6: Moore type or Mealy type

Because output signals from the Moore type module are not affected by the current cycle's input signals, executing FFToOut routines is unnecessary. Hence, the simulation code first executes the FFToOut routines of all the Moore type modules once, then executes the FFToOut routines of all the Mealy type modules

until the input signals do not change, and finally executes the InToFF routines of all the modules once. This strategy can reduce simulation time (Fig.4.7).

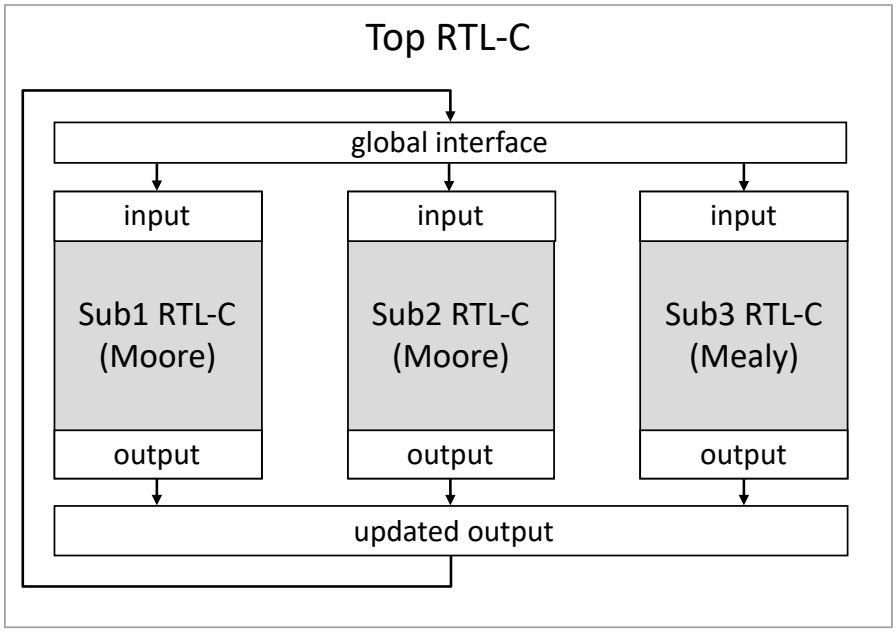

Figure 4.7: RTL equivalent hierarchical simulation

Chapter 5

Hardware Implementation of Fast Gradient Boosted Tree Training Algorithm

5.1 Introduction

In this chapter, as an example of utilizing the high flexibility and performance ability of LLVM-C2RTL for hardware implementation of machine learning algorithms, we present a study of implementing Gradient Boosted Tree (GBT) training algorithms on FPGA. It's a fully pipelined, efficient, high-speed logic architecture to speed up the GBT training. The design of a high throughput pipelined logic that can adapt to changes in data size, a variety of data parallelisms, and a variety of loss functions in the search for the optimal hyperparameters for a GBT is quite challenging. We propose a flexible hardware architecture on our LLVM-C2RTL design framework that enables to build the entire training system from C++ description and to facilitate hyperparameter search.

Regarding previous work on GBT training acceleration techniques, multiple CPU threads, distributed system and GPU accelerated method [32] [33] [34] [35] [36] have been reported, and some of their implementations are being widely used. However, the degree of speedup remains small compared to that of Random Forest, which is another popular decision tree ensemble technique.

The GBT algorithm produces a new tree to compensate for past losses, resulting in high accuracy and fast convergence. However, unlike Random Forest, where all trees can be independently constructed resulting in a high degree of parallelism, it is difficult to exploit parallelism at the tree level owing to dependency between consecutive tree constructions. In addition, GBT training process needs a high computation cost for an exhaustive split condition search, and the memory bandwidth can become a large bottleneck for such a process. Therefore, GPU acceleration for GBT training has limited success compared to Random Forest training.

The main contribution of this chapter is that we present a highly parameterized hardware implementation allows an efficient operation even on low-cost FPGA

devices with a small memory capacity. This was achieved by describing the entire GBT training system in C++ along with all configuration parameters such as the number of samples, the feature dimensions, the maximum tree depth, the loss functions gain evaluation methods and the memory interface between external DRAM and on-chip RAMs. These C++ descriptions of the entire GBT training system are then directly compiled into a system-level verilog-HDL, resulting in a highly efficient FPGA implementation in terms of both computational throughput, power consumption, and hardware flexibility.

5.2 Related Work

Regarding previous work on GBT training acceleration techniques, multiple CPU threads, distributed system and GPU accelerated method [32] [33] [34] [35] [36] have been reported, and some of their implementations are being widely used. However, the degree of speedup remains small compared to that of Random Forest, which is another popular decision tree ensemble technique.

The GBT algorithm produces a new tree to compensate for past losses, resulting in high accuracy and fast convergence. However, unlike Random Forest, where all trees can be independently constructed resulting in a high degree of parallelism, it is difficult to exploit parallelism at the tree level owing to dependency between consecutive tree constructions. In addition, GBT training process needs a high computation cost for an exhaustive split condition search, and the memory bandwidth can become a large bottleneck for such a process. Therefore, GPU acceleration for GBT training has limited success compared to Random Forest training.

An energy efficient GBT training method on an FPGA was first reported [37] as a prior aspect of this study. This prior approach had several drawbacks such as requiring all sample feature data in the expensive on-chip RAMs, and allowing only a fixed set of algorithm parameters in the FPGA implementation. These drawbacks were mostly due to the hardware design methodology using HDLs, which requires significant effort in the overall system design as well as in exploring different system-level architectural options.

5.3 Gradient Boosted Tree Algorithm

5.3.1 Overview of GBT Training Algorithm

The GBT training algorithm for consecutively constructing new trees is shown in Algorithm 2. The first process is to search for the best split condition evaluated by a gain formula that represents the separation degree of the classification result (lines 6-13), the second is to split the samples contained in the node into two child nodes according to the condition found (line 18), and after the above two processes are repeated until reaching the maximum depth of the tree, the predicted values of the samples are updated with the weight of the leaf node calculated using the teacher labels included in that leaf node (lines 20-22). Then, for the next tree,

the gain is calculated based on the difference between the updated predicted value and the training value, the prediction error will be compensated at each additional tree construction.

Algorithm 2 Making a new tree of GBT training

Input: training samples
Input: F: list of features
Input: maxdepth: maximum depth
Output: t: decision/regression tree

```
1: InitTree(t)
2: N ← RootNode(t)
3: for all n ∈ N do
4:     S_n ← SamplesInNode(n)
5:     maxgain ← 0, threshp ← (0, 0)
6:     for all f ∈ F do
7:         for all v ∈ FeatureValues(f) do
8:             gain ← ComputeGain(S_n, f, v)
9:             if gain > maxgain then
10:                maxgain ← gain, threshp ← (f, v)
11:            end if
12:        end for
13:    end for
14:    t ← t ∪ (n, f, v, maxgain)
15:    if maxgain < 0 then
16:        UpdateLeafNode(n,N)
17:    else
18:        (n1, n2) = SplitNode(n,N,f*,v*)
19:    end if
20:    if depth(n) + 1 = maxdepth then
21:        UpdateLeafNode(n1), UpdateLeafNode(n2)
22:    end if
23: end for
```

The gain for the split condition candidates at the current node is evaluated using Eq.(5.1) where γ is the regularization term. In addition, G_L and G_R the sum of gradient g_i of all current samples in the left node and right node, respectively. Likewise, H_L and H_R are the sum of Hessian h_i. Gradient g_i is the first-order derivative of the loss function denoted by $l(y, (y_i))$ and Hessian h_i is the second-order derivative of the loss function, as shown in Eq.(5.2).

$$Gain = \frac{1}{2}\left[\frac{{G_L}^2}{H_L + \lambda} + \frac{{G_R}^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right] - \gamma \tag{5.1}$$

$$g_i = \frac{\delta l(y, \hat{y})}{\delta \hat{y}}, h_i = \frac{\delta^2 l(y, \hat{y})}{\delta \hat{y}^2} \tag{5.2}$$

When samples reach a certain leaf node j, the prediction values of these samples will be updated by adding a leaf weight ${w_j}^*$ given by Eq.(5.3), where

$G_j = \sum_{i_j \in I_j} g_i$, $H_j = \sum_{i_j \in I_j} h_i$, and I_j is the set of samples reached at leaf node j. Then these new prediction values will be used for constructing the next tree.

$$w_j{}^* = \frac{-G_j}{H_j + \lambda} \tag{5.3}$$

5.3.2 GBT Training Hotspots

The most compute-intensive part of the algorithm is finding the split condition at each node. This part can be further divided into two parts: (A) sorting all samples at every feature, and (B) a maximum gain search over all threshold candidates. Since the processing time of the GBT algorithm is dominated by these two hotspots, accelerating these parts in hardware results in significant throughput improvement.

Sorting based versus Histogram based Split Finding Algorithm

As reported in previous work [38] [32], sorting all feature values in advance in order to find the optimal split condition from all possible candidates requires heavy computing resources when the number of distinct feature values is large, but a histogram based algorithm can reduce this computing cost drastically. Therefore, we quantize the feature values to fit them into a fixed number of bins for each attribute and make a histogram by accumulating the gradient and hessian for each bin, finally choosing the threshold that gives the maximum gain.

Acceleration of Gain Calculation

The gain calculation in Eq.(5.1) is needed for all features and threshold candidates during the split condition search, resulting in a large computational load. On multi-core CPUs and GPUs, acceleration of the gain calculation can be achieved by data parallelisms applied on different features and thresholds, although this approach may create a data transfer bottleneck of distributing the sample data over multiple processing elements. Instead, we directly connect a dedicated pipelined gain calculation hardware to the gradient/hessian histogram memories to avoid any data transfer bottleneck and achieve a sufficiently high gain calculation throughput.

5.4 Hardware Architecture

Our approach has five main characteristics. The first is the histogram based approach mentioned earlier. A histogram-based approach is well suited to hardware implementation because not only does it reduce computing cost of a comparison from O(#sample * #feature) to O(#bin * #feature), the memory consumption also becomes a constant (#bin * #feature) and thus it can be easily stored in on-chip memory. The second characteristic is a pipeline architecture, which avoids a random memory access and often results in a degraded performance to a memory

bandwidth bottleneck, and achieves a high throughput in most parts of the training process. The third characteristic is a pointer memory architecture that keeps both a high throughput and low memory consumption at the same time. The fourth characteristic is the efficient parallelism based on the throughput per hardware area. The last is a methodology that uses LLVM-C2RTL, enabling flexibility and scalability to support various types of hyperparameters and environmental constraints.

5.4.1 Pipeline Architecture

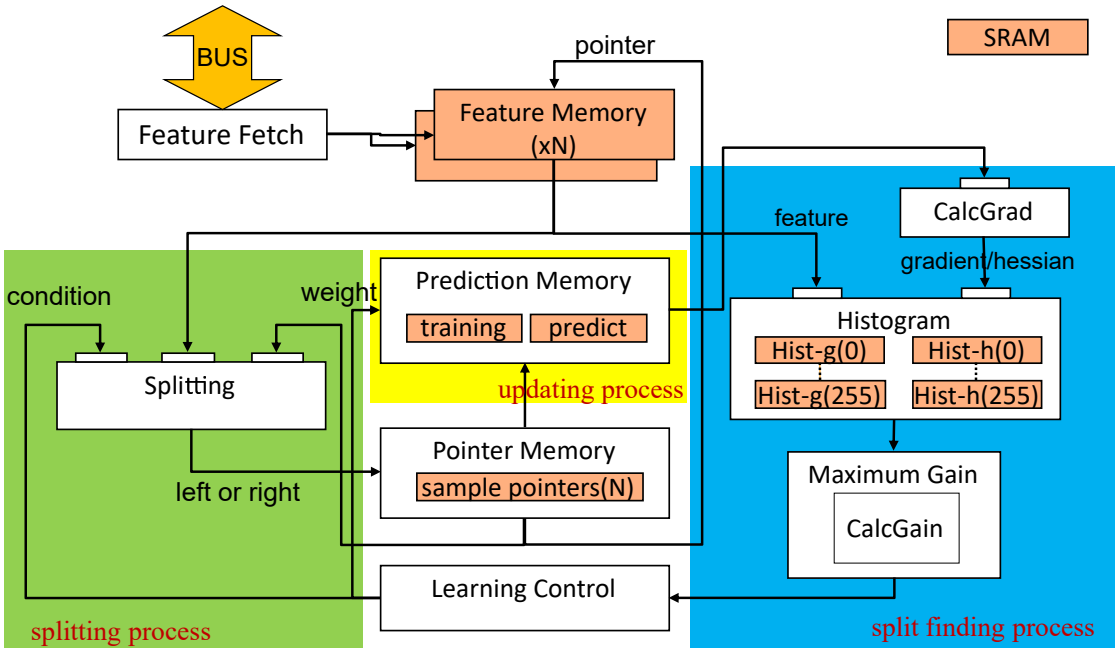

Figure 5.1: Overview of system module architecture

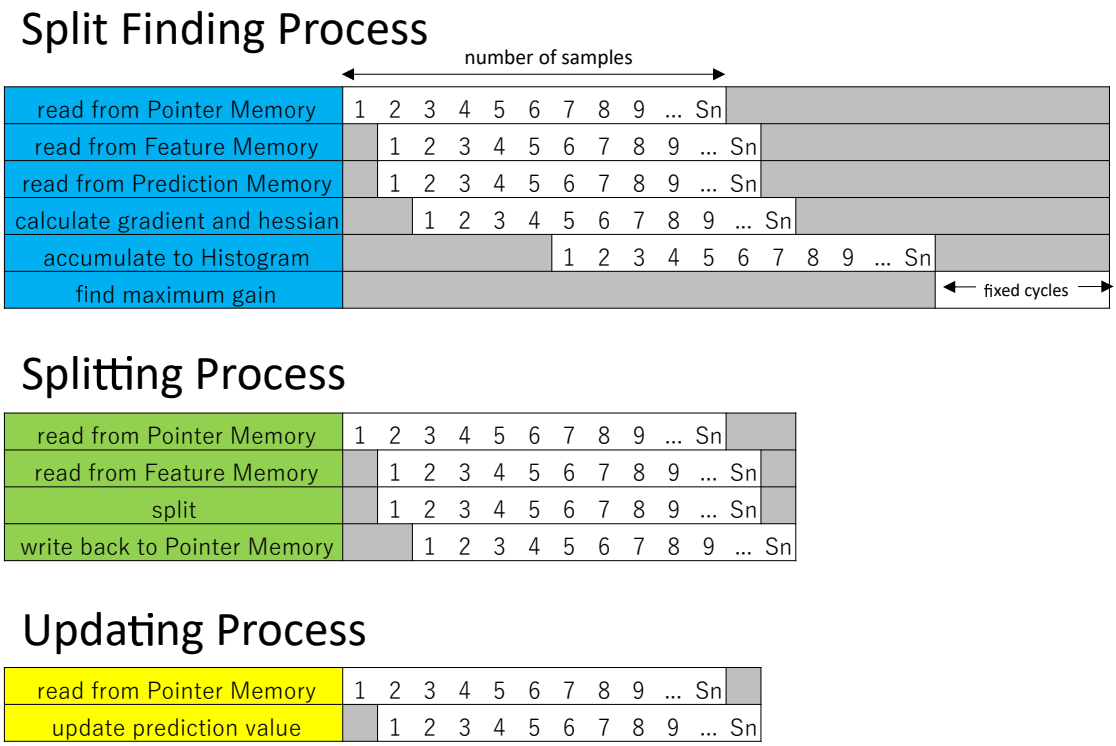

Figure 5.2: Pipeline of three processes

Our architecture consists of the modules shown in the 5.1. There are four modules holding on-chip memory. The pointer memory module holds the pointers for the samples, the prediction memory module holds the labels and the predicted values for the samples, and the histogram module stores the accumulated gradient and hessian values for each bin in on-chip memories. The feature memory holds the number of feature values determined by the number of fetchable feature dimensions and the samples. The three main processes that are repeatedly executed are the split finding process, splitting process, and update process, that correspond to the colored parts in the 5.1. The split finding process and the splitting process are repeated as many times as there are nodes at a certain depth. This set of processes are further repeated as many times as the maximum depth.

These three processes must be sequential, because the splitting process cannot be started until the split condition search after histogram generation is completed, and the predicted value cannot be updated until the samples entering each node are determined. However, as shown in 5.2, the sub-processes within each process are pipelined to handle one sample per clock cycle (the number in the sub-process block indicates the processing sample). This architecture provides the most effective acceleration while avoiding duplicate and random access to the memory. Here, the gradient calculation module has delay of 4 cycles because this module is further pipelined with for stages internally to increase the clock frequency in this case.

Split Finding Process

Feature values for each sample are read from feature memory, the predicted value is read from prediction memory, and the gradient and hessian values are simultaneously calculated from the label and the predicted value through the CalcGrad module. These values are fed to the histogram module, which accumulates these gradient and hessian values at each histogram bin.

After gradient and hessian values corresponding to all samples are accumulated, we search for the condition that maximizes the gain after splitting using the selected feature and threshold. This gain is evaluated using Eq.(5.1), where the first two terms represent the separation degree of classification at the two child nodes, and the third term represents the separation degree of classification at the original node. The last term is the regularization term. Here, using the maximum gain module, we employ a full exhaustive search where Eq.(5.1) is evaluated for all threshold candidates located at the histogram bin boundaries. The maximum gain module spends constant cycles according to the number of bins and the feature dimensions.

Splitting Process

After the split condition is found, every samples in the node will be split into two nodes at the next depth. Node splitting is performed by reading the corresponding feature values of all samples in the node from the feature memory (through the pointer memory mechanism described in section 5.4.2), comparing the values against the splitting threshold, and updating the pointer memory to reflect whether each sample resides in the left or right child node. These processes are executed by pipeline one sample per cycle. After all nodes at current depth are processed, the split finding and splitting process continue to the next depth.

Updating Process

After reaching the maximum depth, in the prediction memory module, the predicted value of every sample is updated by adding the weight of the leaf node that contains the sample while reading the sample index and node index from the pointer memory. The weight of the leaf node is calculated using Eq.(5.3).

5.4.2 Pointer Memory Structure

Although many software tree structure implementations are possible such as dynamic 2D arrays or a linked list, a hardware tree structure implementation requires special care for sustaining high throughput processing. Here, we employ the pointer memory which is an array of sample indices used as address values for accessing the on-chip feature memory. The tree structure is maintained by a set of registers that indicate the boundaries of the tree nodes on the array. Two sets of such arrays are used in a ping-pong fashion to facilitate the tree splitting process.

At each splitting stage, the current sample indices in one RAM array are read one by one, where each sample index is then written to the other RAM array,

either from the left side (left split) or from the right side (right split), as shown in 5.3 (1). After all sample indices have be relocated from one RAM to the other, the roles of the read/write RAMs are reversed at the next node splitting shown in 5.3 (2). This pointer memory scheme keeps the memory consumption at a constant size independent of the tree depth, and is effective in a hardware tree structure implementation.

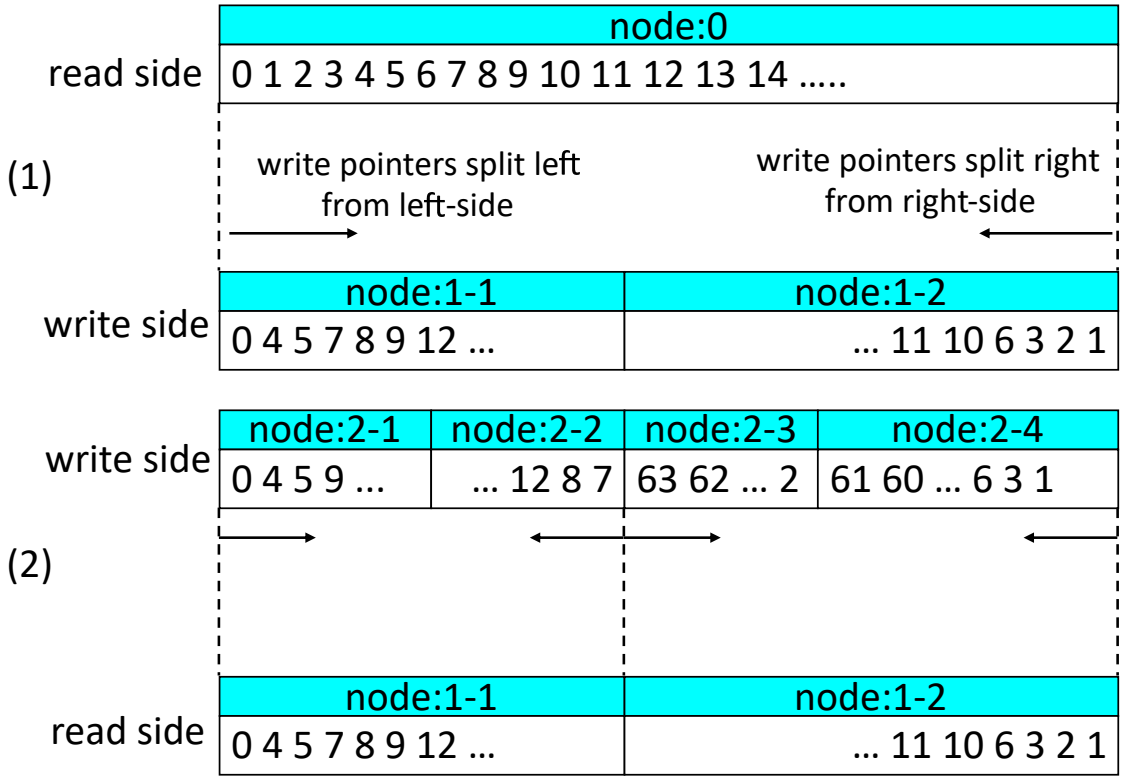

Figure 5.3: Example of pointer memory read/write

5.4.3 Feature Prefetching

Because it is difficult to retain the feature data of 100,000 or more samples in on-chip RAM of a small FPGA, the features must be stored in external memory. In addition, to sustain the process throughput of one sample per cycle, feature fetch module shown in 5.1 with a DMA engine transfers the required features from the external memory into the on-chip RAM. Input labels are also transferred beforehand into the on-chip RAM.

Depending on the on-chip RAM size, sample size, and feature size, different prefetching strategies are considered. The first case is the most simple in which all features of all samples can be stored in on-chip RAM. In this case, DMA completes a prefetching of all data at once before making the trees. The second case is when it is not possible to hold all features (#sample * #feature), but is possible to hold all the feature data for several attributes (#sample * K). In this case, the feature memory can be used as a temporary buffer, and the prefetching is overlapped with the histogram generation and the maximum gain search processing. To execute

the splitting stage immediately after the split finding stage is completed, another memory is prepared to hold the feature data giving the maximum gain. The third case is in which the on-chip memory cannot store (#sample) the feature data, we can accumulate the gradient and hessian of all samples with feature values by fetching several times repeatedly. We assume all samples input labels, and prediction values can be held in on-chip RAM.

5.4.4 Parallelism

Parallelism is necessary to further accelerate each stage, and several types of parallelism can be considered here. Tree level parallelism is not applicable because a new tree depends on the construction of previous trees. To consider the node level parallelism with our architecture, one choice is to divide the pointer memories such that multiple nodes residing in different pointer memories can be processed in parallel. However, tree splitting may result in a highly unbalanced tree, which may introduce a large overhead in the total pointer memory size. Another way is to prepare multiple histogram generation modules and gain calculation modules for parallelizing these processes. This approach may also result in a large circuit overhead. Overall, we have observed little advantage of employing node level parallelism.

Based on the above observation, we adopted feature level parallelism and sample interleaving parallelism instead. feature level parallelism involves processing N feature dimensions in parallel using the same number of histogram generation modules and gain calculation modules, which can directly contribute to the speedup of the split finding stage. Sample interleaving parallelism uses the technique of dividing all samples in an interleaving fashion and also duplicating the pointer memory, feature memory, and prediction memory for simultaneously accessing the interleaved samples in parallel. We can apply the same sets of processes to these interleaved sample groups independently, and then merge these multiple set of gradient and hessian histograms before a gain calculation. 5.4 shows an example of a two-sample interleaving case.

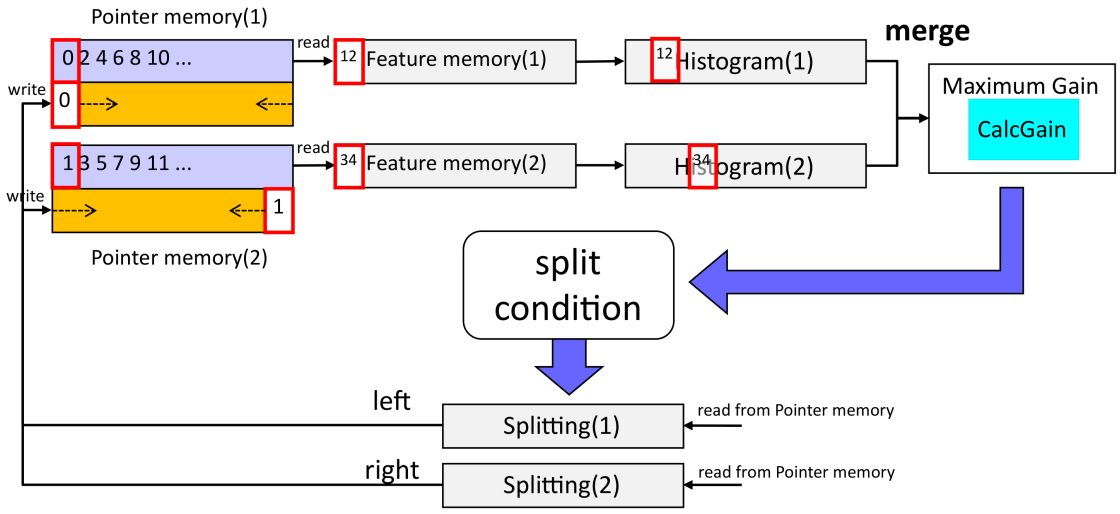

Figure 5.4: Example of duplicated modules for 2 sample interleaving

5.4.5 Support for Data Subset and Random Sampling

When we want to terminate the training early (early-stop) based on certain criteria, we use a data subset for calculating the validation error. Or when we want to increase the effect of the regularization and mitigate the risk of an over-fitting, we use random sampling each time a new tree is made.

For both of these cases, we can pick the sample feature values randomly from sequential memory in external memory into different feature memory modules. This is done in a similar way as the interleaving method described before. For example, we make four sets of pointer memory and feature memory modules: three sets are used to hold the training subset, one set is used to hold the validation subset. A feature memory module for validation does not participate in a split finding pipeline. In a random sampling case, one of four or one of two samples will be chosen randomly to participate in the split finding stage. For this function, the index of the feature memory module that determines the destination memory is given by the pseudorandom generator within the feature fetch module.

5.4.6 Numerical Computation

Because floating-point arithmetic takes a high cost for both hardware and the gate delay, we replaced it with fixed-point arithmetic involving all variables and constants including the training value, prediction value, gradient, hessian, histogram, and leaf weight. We use 8 bits for the fraction part in most cases.

Loss Function

The most commonly used loss function for classification is logistic regression loss, and predicted value at that time is calculated by the sigmoid function denoted in Eq.(5.4). Because the sigmoid function includes the exponential term, we replaced the sigmoid function with an approximate expression by using a piecewise linear

approximation [39] with the division of seven symmetrical sections and the clipping of infinite areas. In addition, we tested replacing the loss function, with the squared error function which is commonly used for regression problems.

$$S(x) = \frac{1}{1 + exp(-x)} \tag{5.4}$$

Gain Calculation

The gain calculation is the most compute-intensive part. To avoid using a divider, we transform the variable part of the original equation in Eq.(5.1) into Eq.(5.5). We convert the divider into subtraction in the base 2 logarithmic space, and then convert back to the linear space through exponential arithmetic. The same transformation technique is applied to the left leaf weight W_L given by Eq.(5.6) and the right leaf weight W_R given by Eq.(5.7), these are originally Eq.(5.3).

Logarithmic arithmetic is implemented by counting the most significant bit (MSB) and exponential arithmetic, which is implemented through a shift operation (5.5). The fraction part is approximated with a linear interpolation. Despite the low hardware area and little gate delay, a better resolution and approximation can be achieved compared to a naive linear interpolation of the equation through this transformation. The synthesis result of two variations of gain calculation logic with four pipeline stages is shown in Table 5.1. The optimization of the gain calculation module has a significant effect on improving the clock frequency and reducing the hardware area.

$$\frac{{G_L}^2}{H_L + \lambda} + \frac{{G_R}^2}{H_R + \lambda} = exp(2log\,|G_L| - log\,|H_L + \lambda|) + exp(2log\,|G_R| - log\,|H_R + \lambda|) \tag{5.5}$$

$$W_L = -exp(log\,|G_L| - log\,|H_L + \lambda|) \tag{5.6}$$

$$W_R = -exp(log\,|G_R| - log\,|H_R + \lambda|) \tag{5.7}$$

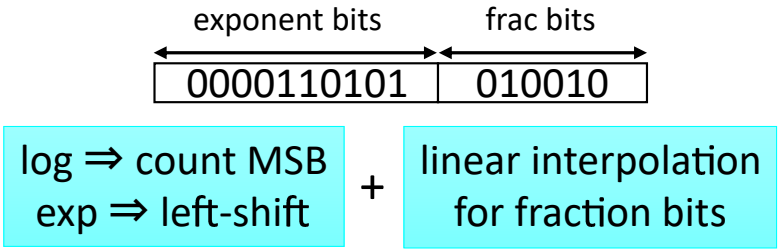

Figure 5.5: Transformation of gain calculation

Table 5.1: Comparison of gain calculation implementation(pipelines=4)

Implementation	LUT	FF	DSP	Clock Freq.
Naive (Divider)	4438	705	4	100MHz
Our approximation	1601	379	0	250MHz

5.5 Description

5.5.1 Control Path Description

When we describe a control path such as finite state machine, accessing the RAM or dealing with a bus transaction issue using a handshake procedure, we describe the explicit definition of the state registers and transition across the clock cycle boundary. Reading from and writing to the on-chip memory can be achieved using a normal C-array with a specified attribute annotation.

5.5.2 Data Path Description

When we describe the data path, such as a calculation of the gain without any state transition, we can determine an arbitrary number of pipeline stages for the target data flow description. Depending on the type of gain calculation, the computational complexity and logic delay can be large and degrade the system operable clock frequency. Therefore, we define multiple pipeline stages for the data flow path, then optimizer of LLVM-C2RTL distributes the gate delays to multiple pipeline stages and inserts registers between the clock boundaries automatically to optimize the clock frequency at the expense of the specified clock cycle latency limit.

Figure 5.6 shows an example code of the gain calculation data path description. At the class definition part, we use template parameters to make a configurable module with the bit-width of the variables and the number of classes to classify, in addition, the bit-width for each variable is defined through an attribute annotation for HDL generation with different parameters.

```
// hi_bw : bit-width of histogram
// gn_bw : bit-width of gain
// num_class : number of class to classify
template <int hi_bw, int gn_bw, int num_class>
class CalcGain : public CalcGainBase<hi_bw, gn_bw, num_class> {
public:
    virtual void calc(int32_t gl, int32_t hl, int32_t gr, int32_t hr, int32_t lamda,
        int32_t& gain_out, int32_t& weight_l_out, int32_t& weight_r_out) {
        // Gain' = GL^2 / (HL + lamda) + GR^2 / (HR + lamda)
        int32_t left = (gl * gl) / (hl + lamda);
        int32_t right = (gr * gr) / (hr + lamda);
        gain_out = left + right;
        int32_t weight_l = -gl * ONE / (hl + lamda);
        int32_t weight_r = -gr * ONE / (hr + lamda);

        weight_l_out = weight_l;
        weight_r_out = weight_r;
    }
    ...
};
```

Figure 5.6: Example of data path description

5.5.3 System Level Description

An example of a top level module is shown in 5.7. The first part of the code example focuses on the configurable parameters. Because all configuration parameters must be constant before instantiation of the hardware modules, the template technique and constant expression code of C++ are used as the parameter definitions and the conditional module instantiations are evaluated at the preprocessing of our LLVM-C2RTL compiler using the LLVM frontend.

```
constexpr LossType loss_type = LossType::SQUARED;
constexpr int FRAC_BITS = 8;      // Number of fractional bits of Fixed point format
constexpr int PREDICT_BW = 12;    // bit-width of predicted value
constexpr int GRADIENT_BW = 22;   // bit-width of gradient(hessian) value
constexpr int HISTOGRAM_BW = 30; // bit-width of histogram(accumulated gradient in each bin)
constexpr int GAIN_BW = 30;       // bit-width of calculated gain value
constexpr int TREE_DEPTH = 6;
constexpr int NUM_NODES = (1 << TREE_DEPTH); // number of nodes at maximum depth
...

_C2RTL_MODULE_
void GBTTop(LearnCtlCmd<SAMPLE_ID_BW>& lcmd,
    ...
    LearnCtlIF<NODE_ID_BW, BIN_ID_BW, SAMPLE_ID_BW, FETCH_FEATURE_DIMS, GAIN_BW>& lif,
    SplitUIF<NODE_ID_BW, BIN_ID_BW, FETCH_FEATURE_DIMS>& spu_if, SplitDIF& spd_if,
    PointerMemUIF<NODE_ID_BW, SAMPLE_ID_BW>& pmu_if,
    PointerMemDIF<NODE_ID_BW, SAMPLE_ID_BW>& pmd_if,
    CalcGradUIF<PREDICT_BW, 1>& cgru_if, CalcGradDIF<GRADIENT_BW, 1>& cgrd_if,
    CalcGainUIF<HISTOGRAM_BW, 1, FETCH_FEATURE_DIMS>& cg_u_if,
    CalcGainDIF<HISTOGRAM_BW, GAIN_BW, FETCH_FEATURE_DIMS>& cg_d_if) {
    PointerMem(pmu_if, pmd_if);
    PredictMem(predu_if, predd_if, weight_if);
    if (loss_type == LossType::SQUARED) {
        CalcGradSquare(cgru_if, cgrd_if);
    } else {
        CalcGradLRegr(cgru_if, cgrd_if);
    }

    FeatureMemArr(fmawt_if, fmaru_if, fmsru_if, fmard_if, fmsrd_if);
    Split(sp_cmd, spu_if, spd_if);
    Histogram(lif.hist_on, histar_wt_if, histar_rd_if);
    CalcGain(cg_u_if, cg_d_if);
    CalcMaxGain(histar_rd_if, cmgf_if, cg_u_if, cg_d_if);
    ....
}
```

Figure 5.7: Example of system level description

For example, when we use the squared loss as the loss function for regression problems, the type of loss function is set to SQUARED, CalcGradSquare() is called, and the squared loss module is instantiated to calculate the slope of the loss function. Otherwise, CalcGradRegr() is called and the logistic regression loss module is instantiated instead. Both the CalcGradSquare() and CalcGradRege() functions use a class derived from the same base class to calculate the slope of the loss function. Any loss function can be employed in this way.

The configurable parameters used in our GBT experiments are listed in Table 5.2.

Table 5.2: Configurable Parameters

Parameter	Range
LOSS_TYPE	Squared, LogReg
NUM_BINS	16, 32, 64, 128, 256
TREE_DEPTH	1–15
FRAC_BITS	1–16
PREDICT_BIT_WIDTH	12–32
GRADIENT_BIT_WIDTH	12—64
HISTOGRAM_BIT_WIDTH	24—64
GAIN_BIT_WIDTH	12—64
SAMPLES	100–1000000
SUB_SAMPLES	100-–10000
FEATURE_DIMS	1—100
FETCH_FEATURE_DIMS	1–50
NUM_INTERLEAVE	2^n
CALC_LOGREGR_PIPELINES	4
CALC_GAIN_PIPELINES	2

5.6 Implementation Results

5.6.1 Synthesis Result

We synthesized the generated RTL of a typical GBT training model, the parameters of which are denoted in Table 5.3 and implemented on a Xilinx Kintex-7 UltraScale+ KCU116 evaluation board with the characteristics shown in Table 5.4. As a result of synthesis at a 250 MHz clock frequency, resource utilizations are shown in Table 5.5. The 'Number' indicates the number of each instances according to the numbers of parallel features and sample interleaving. 'Others' includes the DMA controller to read from external memory and the host interface module through the bus. The values of this row are constant for any parameter configuration.

Table 5.5: Synthesis result at 250MHz

	Number	LUTs	Util	Registers	Util	BRAM	Width	DSPs
Learning Control	1	5122	2.36%	3535	0.81%	0	0	0
Pointer Memory	8	6357	2.93%	6264	1.44 %	16	11(13K)	0
Prediction Memory	8	5615	2.59%	904	0.21 %	8	12(14K)	0
Calc Grad	8	11871	5.47 %	168	0.04 %	0	0	8
Feature Memory	224	1807	0.83 %	2121	0.49 %	224	8(9K)	0
Split	8	1047	0.48 %	440	0.10 %	0	0	0
Histogram	224	73099	33.69 %	27764	6.40 %	448	30(7K)	0
Calc Gain	28	29762	13.72 %	10988	2.53 %	0	0	0
Calc Max Gain	28	28787	13.27 %	6476	1.49 %	0	0	0
Others	-	1681	0.77 %	1465	0.34 %	0	0	0
Total	-	165148	76.12 %	60125	13.86 %	696	(593K)	8

Table 5.3: Configuration parameters for the experiments

Parameter	Value
Num_Samples	10000,100000
Num_Bins	256
Feature_Parallel	28
Num_Interleaves	8
Max_Depth	6
Frac_Bits	8
prediction_Bits	12
Gradient_Bits	22
Histogram_Bits	30

Table 5.4: Target FPGA

Device	XCKU5P-2FFVB676E
Logic Elements	474600
Block RAM	34.9 Mbits
DSP slice	1824
Speed grade	-2

5.6.2 Evaluation Setup

In terms of accuracy and performance, we compared our FPGA implementation with a software implementation using the 'xgboost 0.90' library for both CPU only and GPU accelerated versions. We used an Intel Core i7-8700 CPU, and an NVIDIA GeForce-RTX2060 GPU with CUDA version 10.1. We used the xgboost library from the Python interface. For all of our experiments, we used the same parameters (Table 5.3) and variables (number of trees, learning rate, and regularization parameters) as used in our FPGA implementation and xgboost library. In addition, we used the same binned dataset for both the FPGA and software cases which are quantized by using the frequency-based algorithm such that the appearance frequencies are even. In the case of the FPGA, the processing time was measured after the embedded CPU transferred the training data to DRAM on the FPGA board.

5.6.3 Accuracy Result

For the accuracy test, we use the binary classification dataset of Higgs [40] and Airline [41], the loss function of which is binary logistic regression and the evaluation function is a binary error. We randomly extracted 100,000 samples for both the Higgs and Airline datasets, and then used 80,000 samples as the training dataset and 20,000 samples as the validation (test) dataset. The validation error is obtained by applying the validation dataset to the tree created with the training dataset. We use an early-stop when the validation error does not improve further.

5.8 shows the curve of the training and validation errors with the Airline dataset where the number of bins is set to 16, 32, 64, 128, and 256. The bold curves correspond to the xgboost software implementation. Although the training error of the software implementation is constantly lower than our FPGA implementations, validation errors of the software implementation and FPGA implementations with 64 or more bins are quite similar on 200 or more trees. Table 5.6 shows the validation errors with early-stop condition which also confirms our observations on the validation errors between the software implementation and FPGA implementations.

Table 5.6: Validation error with different number of bins

Number of Bins		256	128	64	32	16	xgboost
Error	Higgs	0.2755	0.2695	0.2715	0.275	0.275	0.2719
	Airline	0.2715	0.271	0.271	0.300	0.344	0.2739

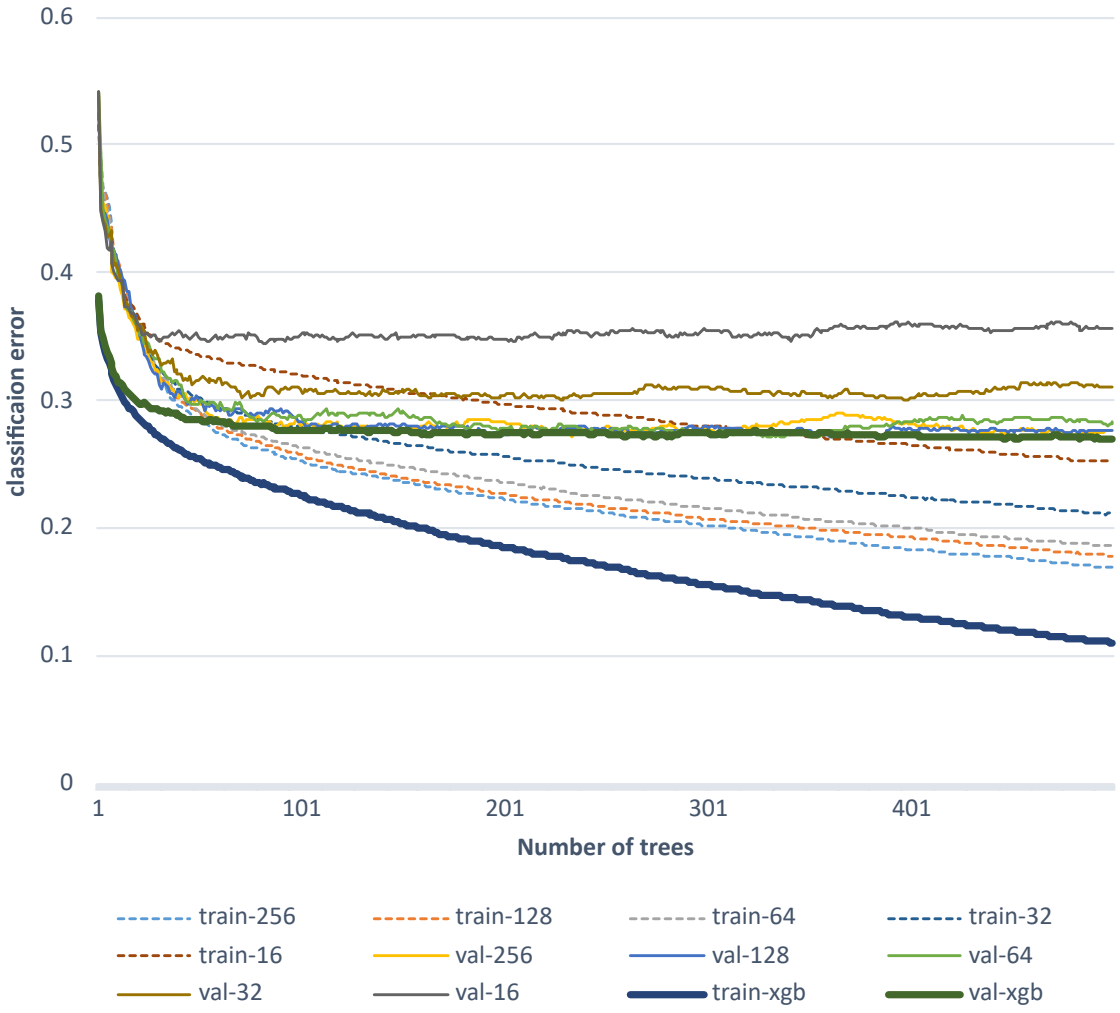

Figure 5.8: Training and validation error curves with 16, 32, 64, 128, 256 bins and xgboost

5.6.4 Performance Results

We compared the execution time for constructing 500 trees on a FPGA with a CPU-only version and a GPU-accelerated version of xgboost. The first case is 10,000 samples extracted from the Higgs (28 features) and Airline (13 features) datasets, respectively. In this case, all extracted feature data can be prefetched into an on-chip RAM concurrently, achieving a speed up of more than 17-times that of the GPU version (Table 5.7), if we increase the number of interleaves from 8 to 64, and synthesize on a larger capacity device such as a Vertex UltraScale+ VU5P at the same clock frequency (920K LUTs is needed), a speed-up of more than 70-times that of the GPU version is estimated through our cycle level simulation. Theoretically, our implementation has more advantages when a smaller bin number is used because the maximum gain search of each feature can be finished quickly, although in some cases, CPU processing with a small number of bins is advantageous possibly owing to the effect of the cache efficiency. In the second case of 100,000 samples (Table 5.8), our FPGA system needs to fetch samples 10

times during each histogram generation. This can disturb an additional speed-up particularly at a lower depth.

Table 5.7: Training time [seconds] for 10,000 samples

Dataset	Bins	CPU	GPU	FPGA	Speed-up (vs. CPU)	Speed-up (vs. GPU)
Higgs	256	8.67	3.85	0.212	×40.9	×18.2
	64	4.41	3.66	0.0726	×38.3	×31.8
Airline	256	3.67	3.63	0.134	×17.3	×17.1
	64	2.95	3.85	0.0726	×25.7	×33.5

Table 5.8: Training time [seconds] for 100,000 samples

Dataset	Bins	CPU	GPU	FPGA	Speed-up (vs. CPU)	Speed-up (vs. GPU)
Higgs	256	24.6	9.12	0.825	×29.8	×11.1
	64	20.24	9.16	0.727	×27.8	×12.6
Airline	256	15.54	8.8	0.825	×18.8	×10.7
	64	14.59	8.21	0.727	×20.1	×11.3

5.6.5 Energy Consumption

The estimated power of the FPGA device using the Xilinx Vivado tool was 3.29 W, including the power of the static and dynamic logic power, clock, and Block RAM, not including the off-chip memory. The estimated GPU power when using the Nvidia-smi utility was 95 W (max.160 W) at a utilization rate of 88%. A training power efficiency of 300 fold is achieved because it speeds up at least 10.7 times with 3.29 W of power when compared to a 95 W GPU.

5.6.6 Configuration Parameter Search

Some parameters have a trade-off; in particular, the buffer size for the feature data, and the amounts of sample interleaving and feature parallelism affect both the performance and hardware significantly. Therefore, a comparison between different parameters is needed to determine the optimal configuration parameters for a certain device.

We show the total number of LUT cells of each synthesis when the feature parallelism to 7, 14, and 28 and the sample interleaving to 1, 2, 4, 8, and 16, as shown in 5.9. All other parameters are fixed in this experiment. The number of LUTs is mostly linear to the degree of sample interleaving and feature parallelism. The number of training samples processed per cycle is shown in 5.10. The throughput is mostly linear to the degree of sample interleaving but not to the feature parallelism for both 10,000 and 100,000 samples, because feature parallelism accelerates only the split finding stage wheres sample interleaving accelerates all three stages.

The area efficiency curve (throughput/LUTs) is shown in 5.11. When the amount of feature parallelism is fixed, increasing the amount of sample interleaving always improves the area efficiency. By contrast, when the amount of sample interleaving is fixed, the area efficiency decreases when the amount of feature parallelism is increased. This indicates that it is more effective to increase the number of interleaves when the area is in a surplus. Thus, the strategy could to start with a small amount of feature parallelism and as much sample interleaving as possible, and then increase the amount of feature parallelism depending on its available hardware area.

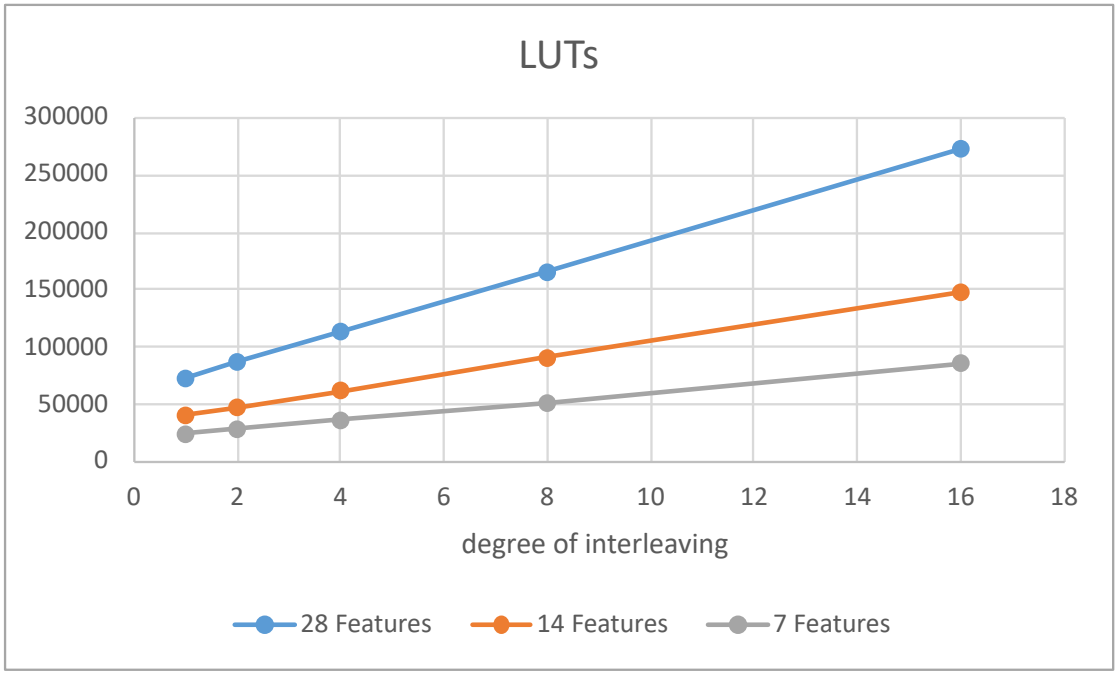

Figure 5.9: Comparison of number of LUTs with different amounts of sample interleaving for 7, 14, and 28 feature parallelism

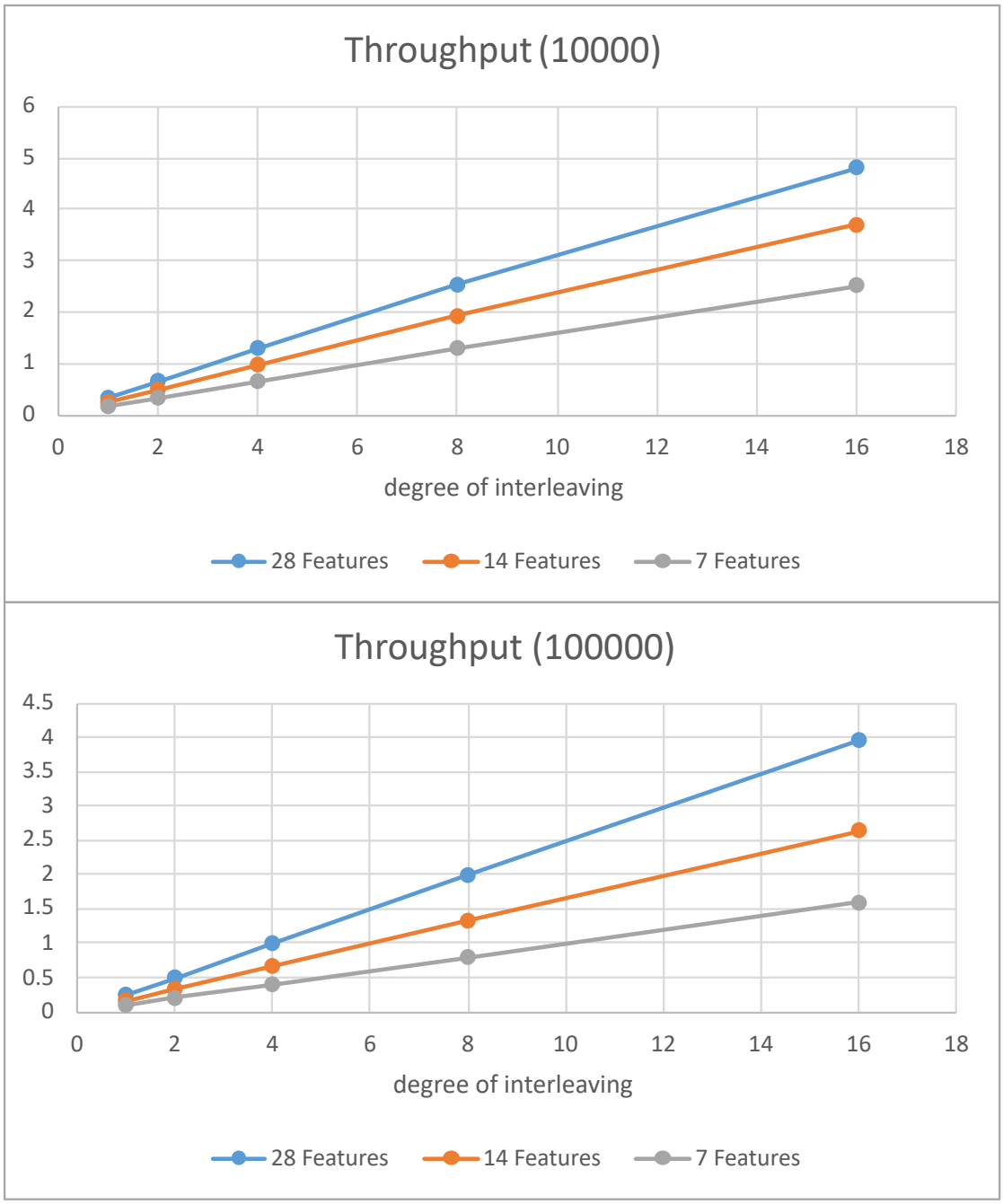

Figure 5.10: Throughput of 10,000 and 100,000 samples with sample interleaving (1, 2, 4, 8, 16) and feature parallelism (7, 14, 28)

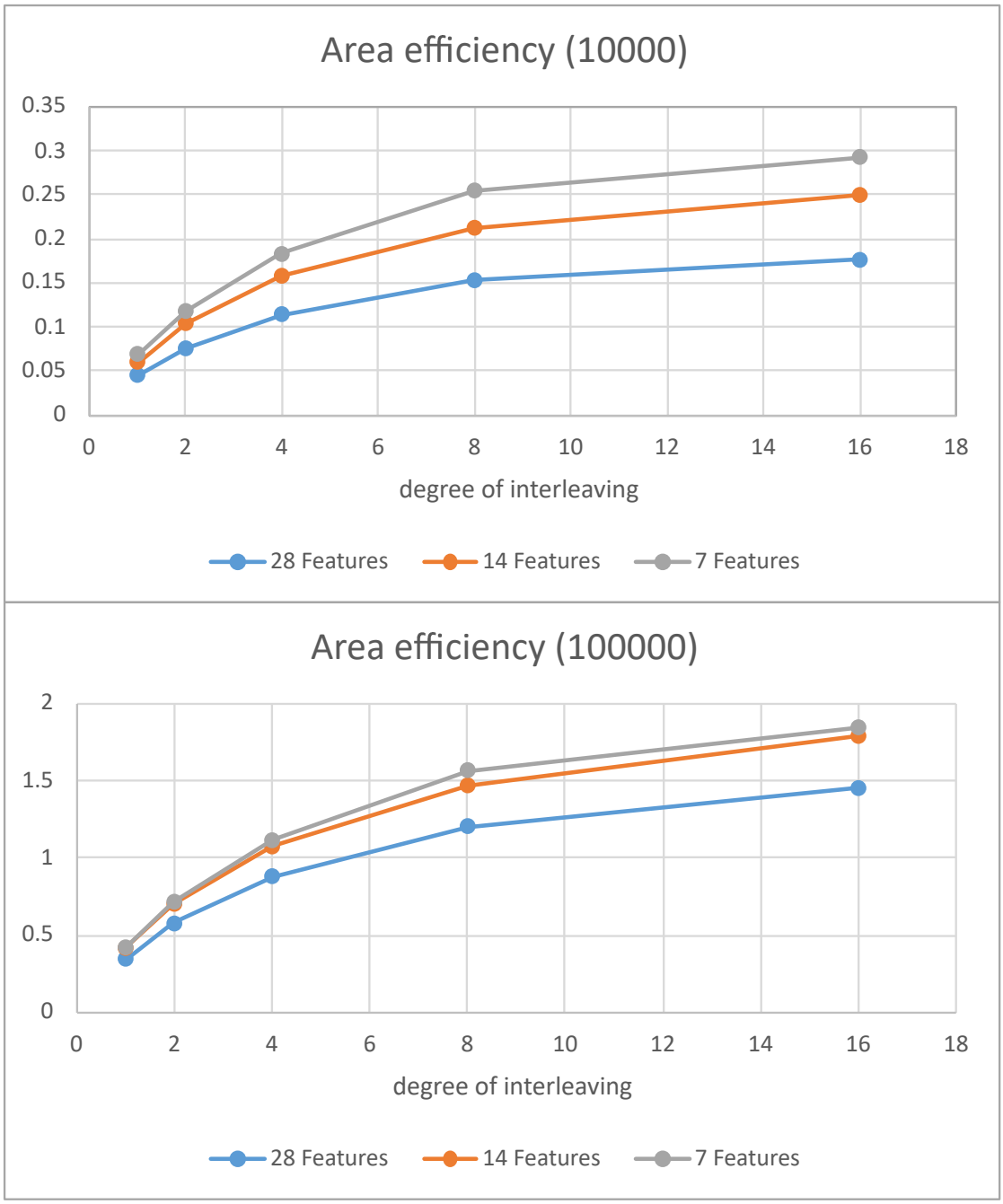

Figure 5.11: Area efficiency of 10,000 and 100,000 samples with sample interleaving (1, 2, 4, 8, 16) and feature parallelism (7, 14, 28)

5.6.7 Conclusion

We proposed a pipeline-oriented hardware architecture for GBT training, which uses feature prefetching with pointer memory structures, and exploits feature-level and sample interleaving parallelism. Optimization techniques for numerical computations by transformation to logarithmic arithmetic and linear interpolation are employed to speed-up and reduce the amount of hardware.

LLVM-C2RTL design framework enables high-quality hardware descriptions of both the data-path and control-path, and flexible configuration to meet the system requirements. Experimental result shows that an implementation to the small FPGA device achieves a 11-33 times speed-up compared to the GPU approach, and more than a 70-times speed-up is possible by simply changing the parameters when we use a large capacity FPGA device.

Chapter 6

Evaluation of Design Efficiency and Quality

In this chapter, we verify the design efficiency of LLVM-C2RTL through comparative experiments using LLVM-C2RTL and other middle synthesis tools that generate HDL. In addition, the quality of the generated HDL is confirmed with the hardware area and the operable clock frequency. Lastly, we consider the effect of LLVM optimization from the results of logic synthesis.

We implemented a 128-bit key-length AES encryptor and 64-bit RISC-V core using LLVM-C2RTL. For the AES encryptor, we implemented the exact algorithm using Chisel [9] (version 3.3) and PyMTL3[15] with register transfer level modeling. We did not use any library. Then, we compared the efficiency of the description, compilation speed, and performance of the generated verilog codes.

The AES algorithm is presented as illustrated in 6.1. SubBytes, ShiftRows, MixColumns, and AddRoundKey are performed in one round, and encryption is performed through ten rounds. Only the final round does not include MixColumns. We used the implementation technique of [42]. This method does not employ a lookup table for the SubBytes and replaces inversions in the Galois field with isomorphic composite arithmetic, solely with combination logic.

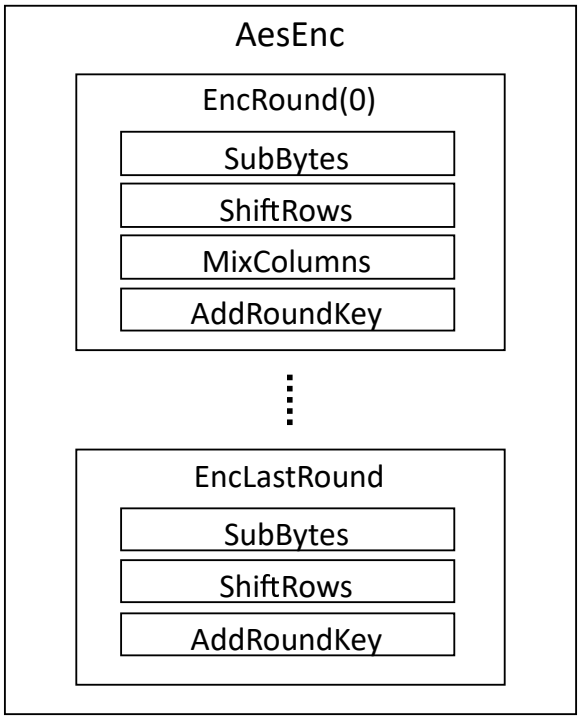

Figure 6.1: AES algorithm

For the RISC-V core, we implemented a highly parameterized RV64IMA (Integer, Multiply, and Atomic instructions) 5-stage pipeline processor with I-cache, D-cache, TLB, and ALU without the floating point arithmetic unit while using the simple bus model instead of interconnect. We set the configuration parameters of the Rocket-core [43] to be equivalent and compared the verilog code generated by Chisel (Table 6.1).

Table 6.1: Configuration parameters for RISC-V

Parameter	Value
XLEN	64
MMU(virtual memory)	true(Sv39)
FPU	None
D-cache-Ways	1
D-cache-Sets	64
I-cache-Ways	1
I-cache-Sets	64
TLB-Sets	1
TLB-Ways	4

6.1 Design Efficiency

Figure 6.2 presents the number of lines of input code for AES encryption described in Chisel, PyMTL3, and LLVM-C2RTL, and figure 6.3 presents the number of lines

of generated verilog code. Figure 6.4 presents the number of lines of input code for RISC-V core. The code for the random initialization of the register automatically generated by Chisel are excluded. Similarly, the wrapper module code for the signal probe, which is automatically inserted by LLVM-C2RTL, is excluded. Comment lines are excluded for all.

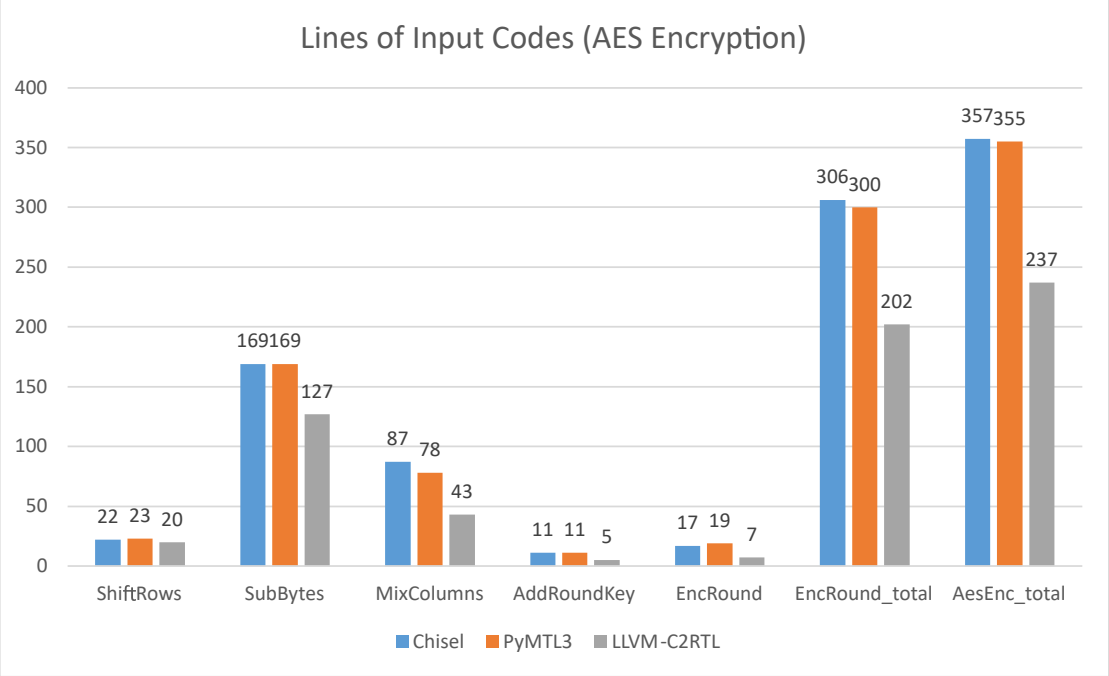

Figure 6.2: Lines of Input Code for AES encryption

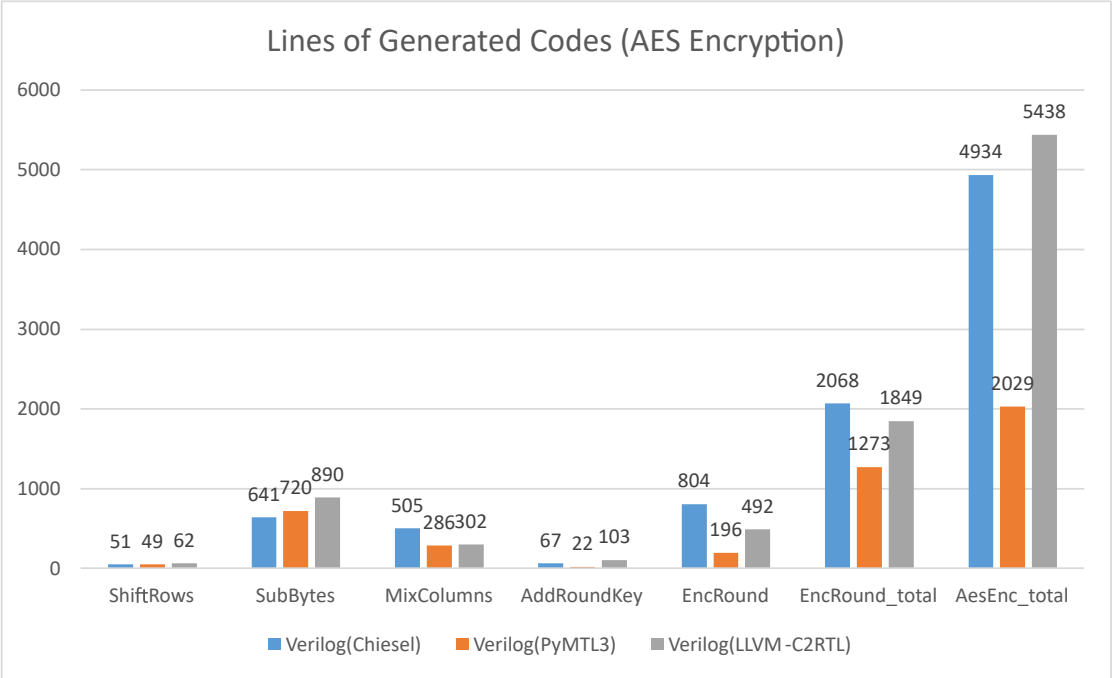

Figure 6.3: Lines of Generated Code for AES encryption

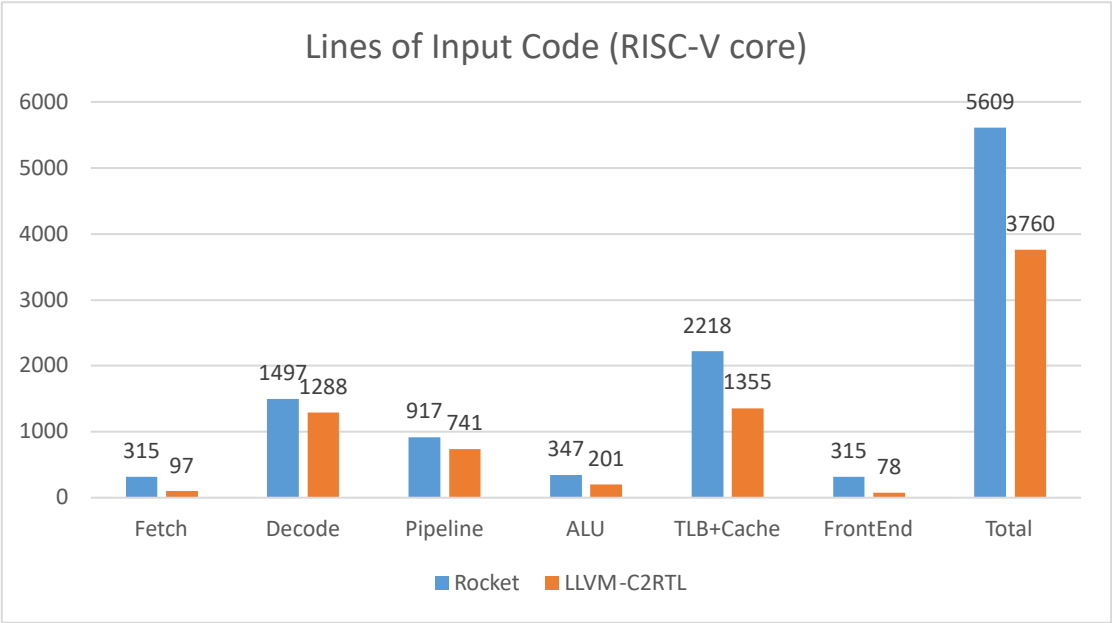

Figure 6.4: Lines of Input Code for RISC-V Core

As an example, we present the codes for the multiplication on an 8-bit Galois field, whose polynomial is defined by Eq.(6.1) used in MixColumn process. Figure 6.5, figure 6.6 and figure 6.7 are Chisel, PyMTL3 and LLVM-C2RTL code respectively. Both Chisel and PyMTL3 require dedicated declarations of I/O ports and cannot be handled directly as variables, the amount of description increases. Also,

since Chisel cannot be reassigned to a signal defined by val, it is necessary to define intermediate signals (the amount of description can be some reduced by using variable type of var).

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{6.1}$$

```
class GfMul8() extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(8.W))
    val b = Input(UInt(8.W))
    val ret = Output(UInt(8.W))
  })

  val a = Wire(Vec(8, UInt(8.W)))
  val a_tmp = Wire(Vec(8, UInt(8.W)))
  val ret = Wire(Vec(8, UInt(8.W)))
  when(io.b(0) === 1.U){
    ret(0) := io.a
  } .otherwise {
    ret(0) := 0.U
  }
  a_tmp(0) := (io.a << 1) & 0xff.U
  when ((io.a & 0x80.U) =/= 0.U) {
    a(0) := a_tmp(0) ^ 0x1b.U
  } .otherwise {
    a(0) := a_tmp(0)
  }

  for (i <- 1 until 8){
    when (io.b(i) === 1.U){
      ret(i) := ret(i - 1) ^ a(i - 1)
    } .otherwise {
      ret(i) := ret(i - 1)
    }
    a_tmp(i) := (a(i - 1) << 1) & 0xff.U
    when ((a(i - 1) & 0x80.U) =/= 0.U) {
      a(i) := (a_tmp(i) ^ 0x1b.U)
    } .otherwise {
      a(i) := a_tmp(i)
    }
  }
  io.ret := ret(7)
}
```

Figure 6.5: Chisel code of multiplication on the 8-bit Galois field

```
class GfMul8( Component ):
  def construct( s ):
    s.a = InPort ( Bits8 )
    s.b = InPort ( Bits8 )
    s.out = OutPort ( Bits8 )

    s.a_tmp = Wire( Bits8 )
    s.b_tmp = Wire( Bits8 )
    s.ret = Wire( Bits8 )

    @update
    def up_comb():
      s.a_tmp @= s.a
      s.ret @= 0
      for i in range( 8 ):
        if s.b[i] == 1:
          s.ret @= s.ret ^ s.a_tmp
        if s.a_tmp[7] == 1:
          s.a_tmp @= s.a_tmp << 1
          s.a_tmp @= s.a_tmp ^ 0x1b
        else:
          s.a_tmp @= s.a_tmp << 1
      s.out @= s.ret
```

Figure 6.6: PyMTL3 code of multiplication on the 8-bit Galois field

```
UINT8 gfmul8(UINT8 a, UINT8 b) {
  UINT8 ret = 0;
  for (int i = 0; i < 8; i++) {
    if (b & 1) {
      ret ^= a;
    }
    UINT8 x8 = a & 0x80;
    a <<= 1;
    if (x8) {
      a ^= 0x1b;
    }  // x^8 = (-x^4 - x^3 - x - 1)
    b >>= 1;
  }
  return ret;
}
```

Figure 6.7: LLVM-C2RTL code of multiplication on the 8-bit Galois field

The followings are codes of one round process of AES encryption as an example of hierarchical module instantiation. Figure 6.8, figure 6.9 and figure 6.10

are Chisel, PyMTL3 and LLVM-C2RTL code respectively. Chisel and PyMTL3 requires strict definitions of all the signals and wire connection to submodule instances, LLVM-C2RTL requires only the function call for submodule instantiation, which reduce the description amount. Totally, the amount of description for PyMTL3 was almost the same as that for Chisel. Regarding the description for the entire AES encryption, LLVM-C2RTL requires more than 40% fewer lines than Chisel and PyMTL3.

```
class AesEncRound() extends Module {
  val io = IO(new Bundle {
    val src = Input(Vec(16, UInt(8.W)))
    val key = Input(Vec(16, UInt(8.W)))
    val dst = Output(Vec(16, UInt(8.W)))
  })

  val sub_bytes = Module(new SubBytes)
  sub_bytes.io.src := io.src
  val shift_rows = Module(new ShiftRows)
  shift_rows.io.src := sub_bytes.io.dst
  val mix_columns = Module(new MixColumns)
  mix_columns.io.src := shift_rows.io.dst
  val add_roundkey = Module(new AddRoundKey)
  add_roundkey.io.src := mix_columns.io.dst
  add_roundkey.io.key := io.key
  io.dst := add_roundkey.io.dst
}
```

Figure 6.8: Chisel code of one round of AES encryption

```
class AesEncRound( Component ):
  def construct( s ):
    s.in_ = [ InPort ( Bits8 ) for _ in range(16) ]
    s.key = InPort ( Bits128 )
    s.out = [ OutPort ( Bits8 ) for _ in range(16) ]

    s.sub_bytes = SubBytes()
    s.shift_rows = ShiftRows()
    s.mix_columns = MixColumns()
    s.add_round_key = AddRoundKey()
    for i in range( 16 ):
      s.sub_bytes.in_[i] //= s.in_[i]
      s.shift_rows.in_[i] //= s.sub_bytes.out[i]
      s.mix_columns.in_[i] //= s.shift_rows.out[i]
      s.add_round_key.in_[i] //= s.mix_columns.out[i]
      s.add_round_key.key[i] //= s.key[i*8:(i+1)*8]

    @update
    def up_comb():
      for i in range( 16 ):
        s.out[i] @= s.add_round_key.out[i]
```

Figure 6.9: PyMTL3 code of one round of AES encryption

```
_C2R_FUNC(AES_PIPELINES)
void AesEncRound(RoundIF& rif) {
  UINT8 step[3][16];
  subBytes(rif.src, step[0]);
  shiftRows(step[0], step[1]);
  mixColumns(step[1], step[2]);
  addRoundKey(step[2], rif.dst, rif.rkey);
  rif.dvld = rif.den;
}
```

Figure 6.10: LLVM-C2RTL code of one round of AES encryption

Notably, the description of PyMTL3 is verilog-HDL compatible with register transitions, assignments, wire signals, etc., and the compilation is more like direct conversion. However, since PyMTL3 did not detect the errors in the hardware structure, such as looped signals or unbinded signals, there were cases where the generated verilog resulted in an error during logic synthesis. In addition, PyMTL3 does not flatten the multi-dimensional array used for I/O ports, which is required to be one dimensional at logic synthesis phase, so it must be rewritten before logic synthesis phase. In contrast, LLVM-C2RTL analyses the hardware structure and notifies the user in such error cases; it also automatically flattens the multi-dimensional array, so all the generated verilog codes did not fail logic synthesis. Similarly, Chisel generated codes did not fail logic synthesis.

Table 6.2 and Table 6.3 present the comparison of compilation and simulation time for AES Encryptor and RISC-V Core respectively. LLVM-C2RTL has a substantially shorter compilation time. This shorter time is because the LLVM-C2RTL process is embedded in a native program while Chisel evaluates Scala and compiles Chisel's classes on the Java VM, and PyMTL3 is also a language which is processed on python although it has introduced JIT compiler. In the simulation of AES, 2000 cycles of AES encryptions were simulated and compared using Chisel's PeekPoke tester, PyMTL3's python simulation, Verilator [19] version 4.028 simulation codes from Chisel and PyMTL3 outputs, and an LLVM-C2RTL simulation code. In this case, all modules were Moore type, and the simulation time was nearly the same as that of the Verilator.

In the simulation of the RISC-V core, we executed 153 official RISC-V assembler tests for rv64ui-p, rv64ui-v, rv64ua-p, rv64ua-v, and rv64um-p . Our RISC-V core simulates a bus with a transaction model. Therefore, we compared the simulation time per core clock cycle. The simulation time per cycle of Verilator is 0.000107 seconds, and that of LLVM-C2RTL is 0.000124 seconds. Because LLVM-C2RTL can adopt the information on the module type (Moore or Mealy) of the user code and be optimized for minimal repetitive execution on a module-by-module basis as described before, its simulation speed is comparable to that of the Verilator, which is one of the fastest verilog-HDL simulators.

Table 6.2: Compilation and simulation time for AES Encryptor

	compile [s]	sim cycles	sim time [s]
Chisel	40.4	2000	1254
Chisel + Verilator	47.3	2000	0.129
PyMTL3	22.9	2000	375.9
PyMTL3 + Verilator	33.4	2000	0.683
LLVM-C2RTL	3.4	2000	0.159
LLVM-C2RTL 3-stages	4.16	2000	0.234
LLVM-C2RTL 5-stages	4.85	2000	0.307

Table 6.3: Compilation and simulation time for RISC-V Core

	compile [s]	sim cycles	sim time [s]	sim speed [s/cycles]
Chisel + Verilator	341.3	18483416	1976	0.000107
LLVM-C2RTL	12.9	3916910	485	0.000124

6.2 Performance of Generated Hardware

We synthesized and implemented the compiled verilog of the AES encryptor using Vivado 2021.2 for the Xilinx's Zynq UltraScale+ ZCU104 Evaluation Board (Device: xczu7ev-ffvc1156-2-e). The first row of Table 6.4 presents the results obtained by Chisel and the second row by PyMTL3. Because one round is processed in one cycle, the throughput is 128-bit x 350 MHz. This performance should be the same as when the same algorithm is described in the conventional HDL. The third row presents the results obtained by LLVM-C2RTL. Compared to Chisel and PyMTL3, the hardware area size has been significantly reduced and the maximum frequency is slightly inferior.

Table 6.4: Performance of generated hardware (AES encryption)

	LUT	FF	CLB	BRAM	Freq. (MHz)	Throughput (Mbps)
Chisel	14852	1348	2647	0	350.0	44,800
PyMTL3	15305	1282	2455	0	350.0	44,800
LLVM-C2RTL	9156	1419	1379	0	333.33	42,666
LLVM-C2RTL 3-stages	12026	8944	1865	0	500	64,000
LLVM-C2RTL 5-stages	11635	11392	1914	0	550	70,400

Table 6.5: Performance of generated hardware (RISC-V core)

	LUT	FF	CLB	BRAM	Freq. (MHz)
Chisel(Rocket)	18857	9368	3760	92	75.0
LLVM-C2RTL	16512	10607	3393	31.5	110.0

A more detailed analysis revealed that this result is brought by the difference in RTL representation and the contribution of LLVM-C2RTL optimizations. Table 6.6 shows the test implementation result at 100MHz for the primitive blocks of AES. This is a comparison of RTL representation for logically equivalent combinational logic. AddRoundKey and ShiftRows are excluded because they contain almost no combinational logic.

Table 6.6: Performance of generated hardware of AES submodules

		LUT	CLB	FMax (MHz)
GfMul4	Chisel	7	3	748
	LLVM-C2RTL	8	2	689
GfMul8	Chisel	36	8	645
	LLVM-C2RTL	20	6	876
MixColumns	Chisel	141	35	609
	LLVM-C2RTL	129	37	598
SubBytes	Chisel	1377	257	350
	LLVM-C2RTL	1345	238	356

One example is the code for multiplication on the 8-bit Galois field shown in Fig.6.7. LLVM-C2RTL generates HDL that uses the bit extraction with the minimum truncated bit width from bit-wise AND and bit shift operations in for statements. This feature minimize the hardware area and the delay than Chisel and PyMTL3. Furthermore, since MixColumns calls multiple GfMul8 with constants, LLVM-C2RTL's inlining and constant propagation provides the bit width optimization across hierarchical boundaries. In such cases, manually specifying the minimum bit width for all intermediate values would be very complicated.

In contrast, in the case of calling submodules that have no room for optimization by constant propagation, inline expansion may hinder the optimization for duplicated modules by the logic synthesis tool. GfMul4 is an example of that. The hierarchical module description works better for LLVM-C2RTL in this case.

The second row from bottom in Table 6.4 presents the results obtained by LLVM-C2RTL with the pipelining of the one round encryption logic into three stages. This pipelining was achieved by setting the static variable PIPE_STAGES as three for AesEncRound and AesEncRoundOut. Further, it is possible to synthesize at a high frequency while suppressing the hardware size. In [42], a sub-pipeline technique is used to process one round encryption in five cycles; however, manually dividing pipelining to optimal sub-pipelines is very difficult. The five-stages pipelining brings a slight frequency improvement for the increase of registers from the three-stages. Thus, with LLVM-C2RTL, automatic pipelining simplifies the search for the optimum number of stages in terms of area size, throughput, and delay. In addition, the hierarchical RTL generation is also beneficial when data and control paths are combined.

RISC-V described in LLVM-C2RTL achieved a clock frequency of up to 110 MHz in synthesis and implementation for the same FPGA device.

Chapter 7

Conclusion

We introduced LLVM-C2RTL a novel design framework for the SoC design and verified its efficiency. In this chapter, we summarize this thesis, then, we describe our future work.

7.1 Summary

In chapter 2, we first clarified the concept and description rules of the LLVM-C2RTL design framework. LLVM-C2RTL uses the C/C++ language as the hardware description method at the register transfer level, making it possible to express circuits using only simple data-flow style descriptions. Next, we showed, by incorporating the LLVM compiler infrastructure, it can provide the high functional parameterization, such as type and behavior abstraction, with the benefit of generic programming capabilities of the latest C++ language specifications.

In chapter 3, we showed that the LLVM-C2RTL compilation flow and the methodology of translating the LLVM-IR generated by the LLVM front-end into an RTL representation after LLVM's optimization passes applied. The key is the moethod to translate from branch and memory accesses instructions that are the instructions for software behavior. Furthermore, we presented that pipelining function is the process of assigning a pipeline stage to each node of the dataflow graph, while satisfying the constraints imposed by the user code in order to increase the operating clock frequency. This is achived by iterative solution of the optimization problem of minimizing the circuit delay balance cost and minimizing the cost of registers using the simulated annealing method.

In Chapter 4, we proposed the system-level description method, and extension of LLVM-C2RTL, enabling SoC design including interconnects and peripheral circuits. It also analyzes whether the module type is Mealy type or Moore type to generate faster simulation code.

In Chapter 5, we showed the case study to implement the Gradient Boosted Tree training which process the entire training process in pipelined hardwired logic, achieving a speed-up of more than 10 times of GPUs implementation. We also demonstrated the ability to design large-scale system including memory sysytem and to parameterize behavior and module constrcution.

In Chapter 6, we conducted a comparative experiment targeting Chisel and PyMTL3, and verified the amount for code writing, compilation time, and simulation time as the productivity of LLVM-C2RTL. While LLVM-C2RTL framework requires approximately 40% less description compared to Chisel and PyMTL3, it provides a highly productive design verification environment with advanced parameterization capabilities and high-speed compilation and simulation. In addition, the automatic pipelining functionality at the register transfer level and the powerful optimization ability of LLVM provide high-performance and portable logic design.

As a result, the benefits of LLVM-C2RTL design framework are the following:

- We can easily design hardware with less effort owing to the data-flow description style and pipelining capabilities without learning a new language.
- We can use powerful generic programming capabilities, and rich software development environment (debuggers, tool chain, IDE) for hardware design.
- Design efficiency is better than DSLs, because the behavior and verification description are unified to C/C++ language and only built-in types are used.
- Reduced description size, fast compilation and fast simulation result in high productivity.
- Enables to design and verify the system architectures such as highly parallelized data-paths and high-bandwidth memory systems, compared to HLS's scope of IP-level.

7.2 Future work

Current LLVM-C2RTL does not require the clock description, and assumes that a single clock resource is provided into the module. In other words, there is a limitation that the behavior of an asynchronous circuit driven by multiple clock resources cannot be described. When describing such a circuit, after describe the modules driven by each clock resource independently, describe the wrapper HDL (verilog or VHDL) to connect the modules generated from those C2RTL descriptions. However, if a module that is driven by multiple clocks can be described, the range of application will be further expanded.

Because LLVM-C2RTL uses the LLVM compiler infrastructure, 64-bit integers have the maximum bit width as built-in integer types. Therefore, when using with registers, wire signals, and RAM with a width greater than 64 bits, it is necessary to express them as built-in integer arrays of 64 bits or less bits. It is desirable to have compiler support or an arbitrary precision library provided to handle arbitrary precision bit-width hardware elements in a concise manner.

The maximum clock frequency that can be achieved by pipelining LLVM-C2RTL depends on the accuracy of the delay estimation mentioned in section 3.7, since it is better to divide the delays as evenly as possible. The evaluation results in section 3.9 are generally good, but there is room for improvement. The

reason for the drop in accuracy here is that ideal ASIC standard cells are modeled as primitives and delay is predicted based on them. Therefore, it is required to improve by preparing the tables corresponding to individual architecture libraries such as other ASIC libraries and FPGA. As another method, it is conceivable to use feedback of actual synthesis results in the model, by using machine learning.

Publications

Journal Papers

- Tamon Sadasue, Takuya Tanaka, Ryosuke Kasahara, Arief Darmawan, and Tsuyoshi Isshiki. "Scalable hardware architecture for fast gradient boosted tree training." IPSJ Transactions on System LSI Design Methodology 14 (2021): 11-20.
- Tamon Sadasue, and Tsuyoshi Isshiki. "LLVM-C2RTL: C/C++ Based System Level RTL Design Framework Using LLVM Compiler Infrastructure." IPSJ Transactions on System and LSI Design Methodology Vol.16 1-15 (Feb. 2023).

International Conference Proceedings

- Tamon Sadasue, and Tsuyoshi Isshiki. "Scalable Full Hardware Logic Architecture for Gradient Boosted Tree Training." 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2020.

References

[1] Xilinx Vitis High-Level Synthesis. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[2] Intel High Level Synthesis Compiler. `https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html`.

[3] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, et al. From software to accelerators with legup high-level synthesis. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 1–9. IEEE, 2013.

[4] Sean O Settle et al. High-performance dynamic programming on fpgas with opencl. In *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*, pages 1–6, 2013.

[5] Malte Vesper, Dirk Kocha, and Khoa Phama. Pciehls: an opencl hls framework. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*, pages 1–6. VDE, 2017.

[6] Thomas Bollaert. Catapult synthesis: a practical introduction to interactive c synthesis. In *High-Level Synthesis*, pages 29–52. Springer, 2008.

[7] David Pursley and Tung-Hua Yeh. High-level low-power system design optimization. In *2017 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 1–4. IEEE, 2017.

[8] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. What you simulate is what you synthesize: designing a processor core from c++ specifications. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[9] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[10] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson,

et al. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216. IEEE, 2017.

[11] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, pages 69–70. IEEE, 2004.

[12] Simpei Sato and Kenji Kise. Archhdl: a new hardware description language for high-speed architectural evaluation. In *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, pages 107–112. IEEE, 2013.

[13] Derek Lockhart, Gary Zibrat, and Christopher Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–292. IEEE, 2014.

[14] Jan Decaluwe. Myhdl: a python-based hardware description language. *Linux journal*, (127):84–87, 2004.

[15] Shunning Jiang, Berkin Ilbeyi, and Christopher Batten. Mamba: closing the performance gap in productive hardware development frameworks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[16] Tsuyoshi Isshiki, Koshiro Date, Daisuke Kugimiya, Dongju Li, and Hiroaki Kunieda. C-based rtl design framework for processor and hardware-ip synthesis. In *Proc. SASIMI*, pages 40–45, 2015.

[17] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[18] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[19] verilator. `https://www.veripool.org/verilator`.

[20] Deshanand Singh. Implementing fpga design with the opencl standard. *Altera whitepaper*, 1, 2011.

[21] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2015.

[22] Teemu Rinta-Aho, Adnan Ghani, D Sameer, and Pekka Nikander. Towards software-defined silicon: Applying llvm to simplifying software. In *WISH-3rd Workshop on Infrastructures for Software/Hardware co-design, Chamonix, France*, 2011.

[23] Qijing Huang, Ruolong Lian, Andrew Canis, Jongsok Choi, Ryan Xi, Nazanin Calagar, Stephen Brown, and Jason Anderson. The effect of compiler optimizations on high-level synthesis-generated hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(3):1–26, 2015.

[24] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 127–136, 2018.

[25] International Organization for Standardization. Programming languages - C (ISO/IEC 9899:2011). `https://www.iso.org/standard/57853.html`.

[26] International Organization for Standardization. Programming languages - C++ (ISO/IEC 14882:2017). `https://www.iso.org/standard/68564.html`.

[27] Charles E Leiserson and James B Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[28] Greg Snider. Performance-constrained pipelining of software loops onto reconfigurable hardware. In *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 177–186, 2002.

[29] Wilson Snyder. Verilator: Open simulation-growing up. *DVClub Bristol*, 2013.

[30] Wilson Snyder. Verilator: Speedy reference models, direct from rtl. *Presentation to University of Massachusetts Amherst*, 2017.

[31] William K Lam. *Hardware design verification: simulation and formal method-based approaches (Prentice Hall Modern semiconductor design series)*. Prentice Hall PTR, 2005.

[32] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in eural information processing systems*, pages 3146–3154, 2017.

[33] Rory Mitchell, Andrey Adinets, Thejaswi Rao, and Eibe Frank. Xgboost: Scalable gpu accelerated learning. *arXiv preprint arXiv:1806.11248*, 2018.

[34] Zeyi Wen, Bingsheng He, Ramamohanarao Kotagiri, Shengliang Lu, and Jiashuai Shi. Efficient gradient boosted decision tree training on gpus. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 234–243. IEEE, 2018.

[35] Win-Tsung Lo, Yue-Shan Chang, Ruey-Kai Sheu, Chun-Chieh Chiu, and Shyan-Ming Yuan. Cudt: a cuda based decision tree algorithm. *The Scientific World Journal*, 2014, 2014.

[36] Zeyi Wen, Jiashuai Shi, Bingsheng He, Qinbin Li, and Jian Chen. Thundergbm: Fast gbdts and random forests on gpus, 2019.

[37] Takuya Tanaka, Ryosuke Kasahara, and Daishiro Kobayashi. Efficient logic architecture in training gradient boosting decision tree for high-performance and edge computing. *arXiv preprint arXiv:1812.08295*, 2018.

[38] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

[39] Hesham Amin, K Memy Curtis, and Barrie R Hayes-Gill. Piecewise linear approximation applied to nonlinear function of a neural network. *IEE Proceedings-Circuits, Devices and Systems*, 144(6):313–317, 1997.

[40] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5(1):1–9, 2014.

[41] Airline dataset. `http://kt.ijs.si/elena_ikonomovska/data.html`.

[42] Xinmiao Zhang and Keshab K Parhi. High-speed vlsi architectures for the aes algorithm. *IEEE transactions on very large scale integration (VLSI) systems*, 12(9):957–967, 2004.

[43] rocket-chip. `https://github.com/chipsalliance/rocket-chip`.