Nuclear Inst. and Methods in Physics Research, A 883 (2018) 83–89

Contents lists available at ScienceDirect

Nuclear Inst. and Methods in Physics Research, A

journal homepage: www.elsevier.com/locate/nima

A software framework for pipelined arithmetic algorithms in field programmable gate arrays

J.B. Kim, E. Won *

Physics Department, Korea University, Anam-ro 145, Seongbuk-gu, 02841 Seoul, Republic of Korea

ARTICLE INFO

Keywords:
Software framework
FPGA
Pipelined arithmetic algorithms
VHDL
C++
Code generation

ABSTRACT

Pipelined algorithms implemented in field programmable gate arrays are extensively used for hardware triggers in the modern experimental high energy physics field and the complexity of such algorithms increases rapidly. For development of such hardware triggers, algorithms are developed in C++, ported to hardware description language for synthesizing firmware, and then ported back to C++ for simulating the firmware response down to the single bit level. We present a C++ software framework which automatically simulates and generates hardware description language code for pipelined arithmetic algorithms.

1. Introduction

In the modern experimental high energy physics field, detectors with massive number of channels are used to identify physical processes that occur when colliding particles. Because the rate of colliding particles including uninteresting background are in the scale of MHz [1–3] and data readout from detectors are in the scale of megabytes [4–6], it is currently impossible to record all the collision data which would be produced in the terabyte per second scale. Therefore a hardware trigger which determines whether the data should be recorded or not is required. The trigger should filter the detector data in such a way that only the physics processes of interest are written to a permanent storage at an acceptable rate. The trigger response also needs to be prompt in making the decision, because each sub-detector can hold its data for only a limited amount of time due to hardware limits which is in the scale of micro seconds [1,7,8]. The trigger should perform all of its logic before this limited amount of time is reached.

Field programmable gate arrays (FPGAs) are integrated circuits that are programmed using hardware description language. Due to their programmable and parallel nature, they have been used for event triggers in the modern experimental high energy physics field [9–11] extensively.

FPGA based trigger algorithms generally use integer based calculations [10,12–14]. Although floating-point calculation can be implemented in FPGAs, the calculation latency, FPGA resource usage are significantly higher as discussed in Refs. [15,16].

On the other hand, physics related data are generally handled using floating-point calculations with general purpose computers and physics analysis software are built with floating-point calculations for precise results. One needs to use these software to study the performance of trigger algorithms. The implemented trigger should also be simulated in these software in such way that the effects of the trigger on the recorded physics of interest can be studied, as well as to be compared with the output from the real hardware down to the single bit level.

Due to these facts, trigger algorithms are usually developed in two software versions. One that uses floating-point calculations and one that uses integer calculations [10,12–14]. The floating-point version shows the pure algorithm performance while the integer version shows the degradation of performance due to the constraints of integer calculation and the performance of the FPGA algorithms. Due to the coexistence of the two versions, one constantly needs to synchronize them when the algorithms are modified in one of the versions. To make matters worse, FPGAs are programmed using a hardware description language so that there can even be three versions of the same algorithm that are not necessarily developed by one individual. These various versions make the maintenance of the level one trigger software extremely difficult.

A solution to these problems could be to use high level synthesis (HLS) packages such as Vivado HLS [17]. One can write code in C++ and let HLS convert it into a hardware description language. If latency or resources of a FPGA is not an issue, this would be the best solution. However, as discussed earlier, floating-point calculation implemented in FPGAs take up much more latency and resources, which is also a concern for Vivado HLS [18]. Integer or fixed-point algorithms can be written in C++ using the classes provided by Vivado HLS, but then the precision

* Corresponding author.
E-mail address: eunil@hep.korea.ac.kr (E. Won).

https://doi.org/10.1016/j.nima.2017.11.064
Received 19 October 2017; Received in revised form 16 November 2017; Accepted 17 November 2017
Available online 29 November 2017
0168-9002/

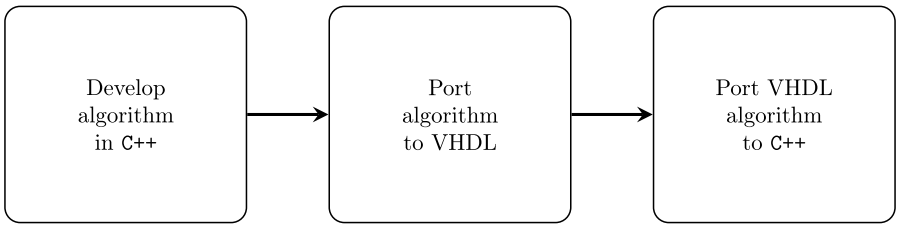

Fig. 1. A development procedure for firmware algorithms. An algorithm is developed with floating-point based calculations using simulated data as input. It is then ported to VHDL with integer based calculations to synthesize firmware for a given FPGA. To study the performance of the synthesized firmware, it is ported back to software which simulates the firmware response exactly bit-by-bit.

and bit widths for each calculation should to be additionally considered. Also there would still be the issue of maintaining two versions, floating-point and fixed-point, of the same algorithm.

We have developed a framework that solves the multiple version problem which uses integer based calculations for the hardware description language. Once an algorithm is implemented in the framework, one can obtain the floating-point calculation result, integer calculation result, and very high speed integrated circuit hardware description language (VHDL) code simultaneously.

In this work, a framework for pipelined arithmetic algorithms in FPGAs is reported. The goals and design of the framework are explained. The three C++ classes that were developed for the framework are described. Algorithms that were developed using this framework are discussed as examples. We also compare between our framework and Vivado HLS for a linear regression algorithm.

2. Goals

A typical procedure of firmware algorithm development is shown in Fig. 1. Algorithms are developed and tested in C++ first. After the algorithms are validated they are ported to hardware description language such as VHDL. There are several issues that should be considered when porting to VHDL. Floating-point numbers should be converted to integers. The bit width of all variables should be determined. Division and non-linear operators such as trigonometric operators should be implemented using look-up tables (LUTs). The inputs to an operator should be properly buffered so that the clock cycle between them are in synchronization. Overflow and underflow should be prevented when doing addition, subtraction, and multiplication. In order to limit the FPGA resource usage, the bit width of the inputs to the multiplication operator should be small enough to be implemented in a digital signal processing (DSP) slice [19]. After porting to VHDL, the resources used by the algorithm should be small enough to fit in to the chosen target FPGA. One way to reduce the resource is by controlling the bit widths for the LUTs. Due to the loss of calculation precision when porting, the VHDL codes need to be simulated, a priori to confirm if they can achieve their goals. A floating-point calculation of the algorithm should be performed to confirm the loss of precision due to this integer conversion. The VHDL codes should be simulated in C++ in such a way that the results can be used in studying other algorithms. Simulation in C++ will also help in debugging the firmware algorithm most efficiently.

A framework is developed to simplify the entire process of pipelined arithmetic firmware algorithm development. The framework can execute the algorithm using floating-point calculations, simulate the integer-valued version of the algorithm, automatically generate VHDL code, and deal with all the issues described previously on arithmetic algorithms. After an algorithm is developed, the framework will handle the rest of the development process most efficiently.

3. Design

Three classes have been developed in total. The first one is for simulating VHDL signals and the second one is for LUTs that use block random access memories (BRAMs) [20]. The third class is to store the information related with our VHDL codes. Clock cycles are taken into consideration so that the signals are properly buffered for the pipelined algorithms in the VHDL code.

3.1. Signal class

The signal class has been implemented to simulate the signed and unsigned VHDL types. Since algorithms generally use floating-point variables but VHDL signals are integer variables, a conversion from floating-point values to integer values are executed when the range of the floating-point variables and bit widths are given. For signed variables, the conversion is done by the following equations

$$\text{symmetric max} = \max(\text{maximum float value}, |\text{minimum float value}|) \quad (1)$$

$$\text{convertion constant} = \frac{2^{(n-1)} - 0.5}{\text{symmetric max}} \quad (2)$$

$$\text{integer variable} = \lfloor \text{float variable} \times \text{convertion constant} \rceil, \quad (3)$$

where n is a given the bit width, max is the maximum function, $\lfloor\rceil$ is a round-off function, and float refers to a floating-point. For unsigned variables, the conversion is done by the following equation

$$\text{convertion constant} = \frac{2^{n} - 0.5}{\text{maximum float value}} \quad (4)$$

$$\text{integer variable} = \lfloor \text{float variable} \times \text{convertion constant} \rceil, \quad (5)$$

where n is a given bit width. The real value which the integer value represents can be calculated using following equation

$$\text{real value} = \frac{\text{integer value}}{\text{convertion constant}}. \quad (6)$$

Addition, subtraction and multiplication operators have been implemented as class methods. The maximum and minimum values are calculated and stored in the class so that bit widths can be reduced to a minimum for each operator. Before adding and subtracting, the input's conversion constants should be matched. They are similarly matched by multiplying a factor of two which is done by bit shifting. The multiplication method is implemented so that only one DSP slice is used to reduce FPGA resources. One DSP slice can perform 25 bit × 18 bit calculations so that the bit width of the input is constrained to 25 bits or 18 bits by applying bit shifts. An if-else method is also implemented to be able to control the flow of the algorithm. It consists of a comparing component and an assigning component. Two signals can be compared with a compare method which receives ==, !=, >=, >, <=, <, &&, and || as an argument and returns a Boolean type signal. Depending on the comparison, different arithmetic operations can be preformed by setting the assigning component.

Each method for this class has logic which can generate VHDL code. To reduce calculation overhead, a flag is used to turn it on and off. All the methods also perform floating-point calculations where the results are stored in the class so that it can be compared with the integer-valued calculations.