Ph.D. DISSERTATION

# FLOATING-POINT SUPPORT FOR COARSE-GRAINED RECONFIGURABLE ARCHITECTURES

재구성형 연산 구조를 위한 부동소수점 지원

BY

MANHWEE JO

FEBRUARY 2014

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

# FLOATING-POINT SUPPORT FOR COARSE-GRAINED RECONFIGURABLE ARCHITECTURES

재구성형 연산 구조를 위한 부동소수점 지원

BY

MANHWEE JO

FEBRUARY 2014

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

# FLOATING-POINT SUPPORT FOR COARSE-GRAINED RECONFIGURABLE ARCHITECTURES

재구성형 연산 구조를 위한 부동소수점 지원

지도교수 최 기 영

이 논문을 공학박사 학위논문으로 제출함

2013 년 11 월

서울대학교 대학원

전기 컴퓨터 공학부

조 만 휘

조만휘의 공학박사 학위논문을 인준함

2013 년 12 월

| | | |
|---|---|---|
| 위 원 장 | 채 수 익 | (인) |
| 부위원장 | 최 기 영 | (인) |
| 위 원 | 백 윤 흥 | (인) |
| 위 원 | 이 종 은 | (인) |
| 위 원 | 김 윤 진 | (인) |

# Abstract

With a huge increase in demand for various kinds of compute-intensive applications in electronic systems, researchers have focused on coarse-grained reconfigurable architectures because of their advantages: high performance and flexibility. Besides, supporting floating-point operations on coarse-grained reconfigurable architecture becomes essential as the increase of demands on various floating-point inclusive applications such as multimedia processing, 3D graphics, augmented reality, or object recognition. This thesis presents FloRA, a coarse-grained reconfigurable architecture with floating-point support. Two-dimensional array of integer processing elements in FloRA is configured at run-time to perform floating-point operations as well as integer operations. More specifically, each floating-point operation is performed by two integer processing elements, one for mantissa and the other for exponent. Fabricated using 130nm process, the total area overhead due to additional hardware for floating-point operations is about 7.4% compared to the previous architecture which does not support floating-point operations. The fabricated chip runs at 125MHz clock frequency and 1.2V power supply. Experiments show 11.6x speedup on average compared to ARM9 with a vector-floating-point unit for integer-only benchmark programs as well as programs containing floating-

point operations. Compared with other similar approaches including XPP and Butter, the proposed architecture shows much higher performance for integer applications, while maintaining about half the performance of Butter for floating-point applications.

This thesis also proposes novel techniques to enhance utilization of integer units for high-throughput floating-point operations on CGRA. The approach to implementing floating-point operations on CGRA presented in this thesis enables floating-point functionality with less area overhead compared to the traditional approach of employing separate floating-point units (FPUs). However the total latency of a floating-point operation is larger than that of the traditional approach and the data dependency between split integer operations restricts further enhancement in terms of utilization of integer functional units in an operation. In order to overcome such inefficiency, two techniques are proposed in this thesis. One is overlapping two distinct floating-point operations, which increases the efficiency in terms of utilizations of integer functional units in the architecture. Free integer functional units in a floating-point operation can be used for another floating-point operation with this technique. The other is forwarding between two data-dependent floating-point operations, which decreases effective latency of the floating-point operations. The basic idea is to remove unnecessary calculations such as formatting which is normally done in between the two data-dependent floating-point operations. To implement the

overlapping or forwarding, FSMs and control paths in each PE are modified and temporal/communication registers are added. Light-weight sub-module such as increment units and registers for intermediate values are added for releasing resource conflict. Experiment is done with several arithmetic functions that are widely used in floating-point applications. The base architecture and the new architecture implementing the proposed technique are compared in terms of throughput and area overhead. The experimental result shows that the proposed technique increases the throughput by 33.9% on average with 20.9% of area overhead.

**Keywords**: Coarse-Grained Reconfigurable Architectures, floating-point numbers

**Student Number**: 2007-21094

# Contents

# List of Figures

# List of Tables

# Chapter 1

# INTRODUCTION

Not many years ago, phones are used just for calling, but today they are used for playing audios/videos, surfing web, image processing, and enjoying games. Not only phones but also tablets have been gaining popularity rapidly and are now a part of our daily lives. However, as the functionality of such mobile devices becomes more diverse and complex, supporting them with limited resources is a big challenge. Multicores are not enough to meet the requirement of compute-intensive programs even though they are suitable for running several control-intensive problems simultaneously. ASICs can hardly support various programs since we cannot put tens or hundreds of them into a single chip. In addition to that, we have encountered another challenge. While conventional compute-intensive programs such as multimedia applications are based

on integer calculation, new ones such as 3D graphics, augmented reality, object recognition, or face recognition require real number operations. Thus, efficient support for both fixed- and floating- point operations with limited resources is also important in future embedded systems.

With a huge increase of various high-performance multi-media applications running on a portable device, great attention to reconfigurable array architectures has been built up since such architectures can be a key to performance and flexibility [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. According to the granularity, we can classify such architectures into fine-grained reconfigurable array (FGRA) and coarse-grained reconfigurable array (CGRA). A representative example of FGRA is FPGA, which has an array of gates. On the other hand, CGRAs typically have an array of arithmetic and logic units (ALUs) or processing elements (PEs) so they can accelerate the execution of programs by parallel processing. In addition, they can run various applications by changing the functionality of hardware dynamically through the reconfiguration of the PEs and the interconnections between them. CGRAs have an advantage over FGRAs in that they can quickly adapt to a new application through dynamic reconfiguration. It is mainly due to the coarser granularity that renders less configuration overheads. CGRA has been proved to be one of viable solutions since it can provide performance and flexibility at the same time.

In spite of the advantages of CGRAs, most of the existing architectures are

limited to integer-based applications such as audio-visual data codec [2, 11], wireless communication [7], cryptography [8], and so on. Thus they are not able to meet the demands for floating-point-based applications effectively. Physics engines in 3-dimensional (3D) graphics are representative examples that cannot be handled efficiently by a conventional CGRA. There have been researches on implementing 3D rendering [12, 13] and ray tracing [14] with reconfigurable architectures. However, their approaches have limitations in generating high quality results due to the lack of floating-point computation.

Adding floating-point units (FPUs) to the integer-only reconfigurable architecture can be a solution to the above-mentioned problem. There have been several related researches on FPGAs. [15] introduces an FPGA including FPUs. [16] shows a design-space exploration for efficient implementation of floating-point operations on an FPGA by adding extra modules such as multipliers and FPUs or by modifying look-up tables (LUTs) for efficient binding. However, the area cost due to those approaches is significantly high, especially when multiple floating-point units are added. Moreover, the added units are not utilized at all when integer-based applications are running on the system. By the same token, the integer PEs will be useless when floating-point-based applications are running.

Another solution to the problem is reconfiguring existing units such that they can perform the floating-point operations [17, 18, 19, 20, 21, 22, 23].

[17] suggests a novel FPGA architecture and efficient implementation methods of floating-point operations on that architecture. However, implementing floating-point units using LUTs in FPGAs requires much more time to reconfigure the circuit than coarse-grained reconfigurable architecture. Thus it is hard to accelerate applications mixed with integer operations and floating-point operations. Coarse-grained reconfigurable architectures support reconfiguration with much less reconfiguration time (one cycle in our architecture). Thus the processing elements (PEs) in the architecture can be reconfigured to execute floating-point operations right after executing integer operations. The approaches in [18, 19, 20, 21] combine a pair of integer PEs to perform a floating-point operation. Since there are many PEs in an array, it is possible to perform multiple floating-point operations in parallel. For an efficient implementation of the floating-point operations, they use separate FSMs in addition to the configuration of the architecture. In this thesis, we present details of the chip implementation and experimental results of a coarse-grained reconfigurable architecture called FloRA (Floating-point-capable Reconfigurable Array), which supports floating-point operations as well as integer operations. Since the floating-point operations are performed with multiple integer PEs, the architecture does not have any separate floating-point units. This allows the architecture to have extended applicability with less hardware overhead.

There are other approaches to implementing floating-point operations ex-

ploiting the existing integer functional units in a CGRA. One of them is PACT XPP [24], a commercial coarse-grained reconfigurable architecture [23]. Their approach relies only on configurations of the existing architecture without any additional hardware support for floating-point operations and thus results in an inefficient implementation in terms of performance-to-area ratio. Another approach uses Butter architecture [25, 26, 27] where floating-point operations are implemented using its integer addition/subtraction units and multiplication units. Those architectures will be compared with our architecture in the later chapter.

Proposed approach of sharing integer functional units for floating-point operations has more latency than stand-alone floating-point units while the area overhead is much less. Besides, there could be utilization losses when floating-point operations are executed on PEs since each single functional unit in those PEs are not busy during the whole execution cycles. In order to overcome the implications above, two techniques are proposed in this thesis. One is overlapping two floating-point operations and the other is forwarding between the two floating-point operations which are data-dependent. The former enhances the utilization of the architecture while the latter increases the effective latency of the floating-point operations.

The organization of the thesis is as follows. Chapter 2 describes the template of the target CGRA in detail. Chapter 3 explains the design for floating-point

operations on the target architecture. Chapter 4 presents the characteristics of the fabricated chip of the target architecture and the experimental results obtained from chip test. Chapter 5 compares the target architecture with other architectures where similar floating-point implementation techniques are applied. Chapter 6 introduces the enhancement techniques for the implemented floating-point operations. Finally, Chapter 7 concludes.

# Chapter 2

# TARGET ARCHITECTURE

## 2.1 Overall Architecture

Figure 2.1 shows the overall architecture of FloRA. It consists of a RISC processor, a main external memory block, a DMA controller, and a reconfigurable computing module (RCM). All the components are connected through a data bus. Before executing an application on the architecture, the RISC processor initializes all other components in the architecture. It also controls them during the execution of the application. In addition, it executes control-intensive and irregular code blocks of the application while the RCM accelerates data-intensive and repetitive code blocks such as DSP kernels or matrix-vector calculations, which can be easily parallelized. The DMA controller is used for efficient communications between the RCM and the main memory.

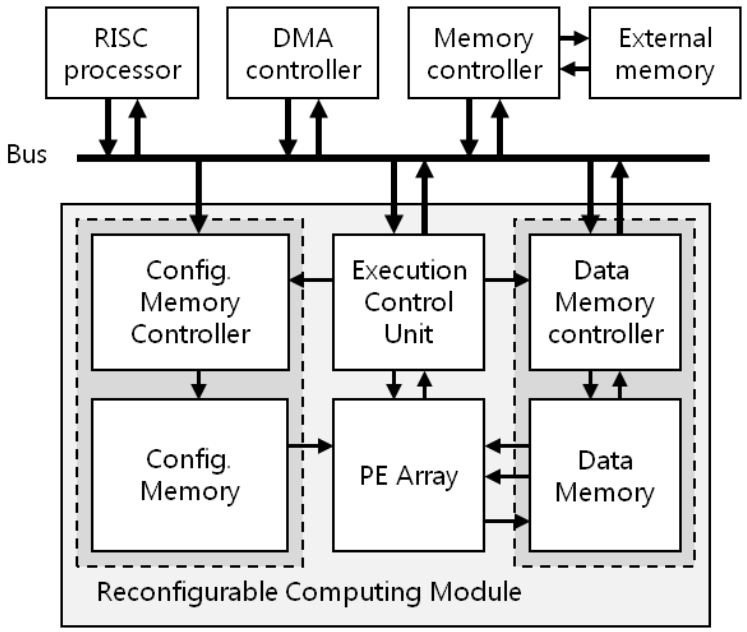


Figure 2.1 The overall structure of the target architecture.

## 2.2 Reconfigurable Computing Module

The RCM is in charge of accelerating data-intensive code blocks using an array of PEs. A PE is an ALU-like functional unit that can handle 16-bit integer values. The PE array is designed for accelerating data-intensive kernel code blocks by parallelizing independent operations in a code block on the array of PEs. As shown in Figure 2.2, each PE in the array has interconnections to its neighbor PEs (top, bottom, left, and right). Each PE also has interconnections to the PEs in two-hop distance in vertical and horizontal direction, and so on, so that it

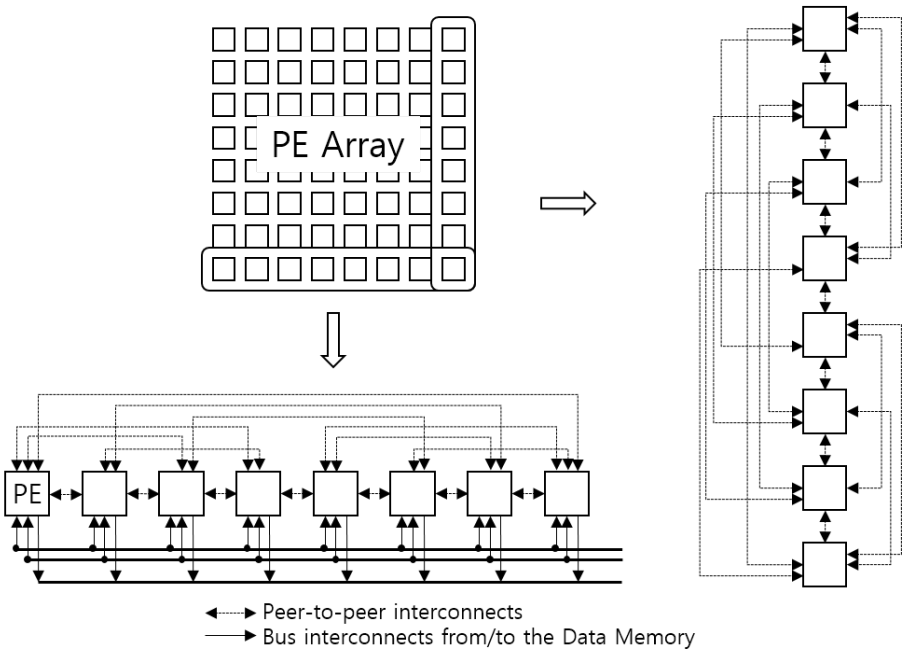


Figure 2.2 The interconnection topology in PE array. Solid line means one-way bus interconnects from/to data memory while dotted line means peer-to-peer interconnects. Each dotted line is physically implemented as two one-way interconnects.

can communicate directly with other PEs in a cycle via those interconnections without having to pass through neighbor PEs one-by-one along the paths. Such abundant interconnection resources make it easy to map data-flow graphs onto the array [28, 29].

Each PE can perform arithmetic operations and logical operations including shift operations and compare and select operations. Thus a PE can be considered

as a small processor without instruction fetch unit and branch unit as shown in Figure 2.3. Some operations (critical operations) such as multiplication and division require functional units that require much larger area and delay than other operations. Each of the critical functional units such as multipliers and dividers is shared by a set of PEs. In the Figure 2.3, pipelined multipliers (blue boxes) are shared by a row of PEs. Because of the reason, the executions of those functional units by the PEs are scheduled ahead of time [6]. Since the critical functional units typically have longer delays, they are pipelined so that they do not degrade the overall system throughput. The number of critical functional units integrated into the array is much less than the number of PEs, thereby saving much area and power consumption at the cost of ignorable performance degradation. In case of division, it is a common practice to change divisions into shift operations for applications that are not very sensitive to accuracy. Thus the number of dividers can be further reduced compared to the number of multipliers.

The PEs are configured by configuration control unit (CCU) and a set of configuration memory elements (CEs). Configuration Memory in Figure 2.1 is basically an array of CEs. Each CE provides the configuration data to the corresponding PE or row of PEs depending on mapping strategies. The architecture supports two different mapping strategies: spatial mapping and temporal mapping. In the spatial mapping, every PE has its own configuration data fetched

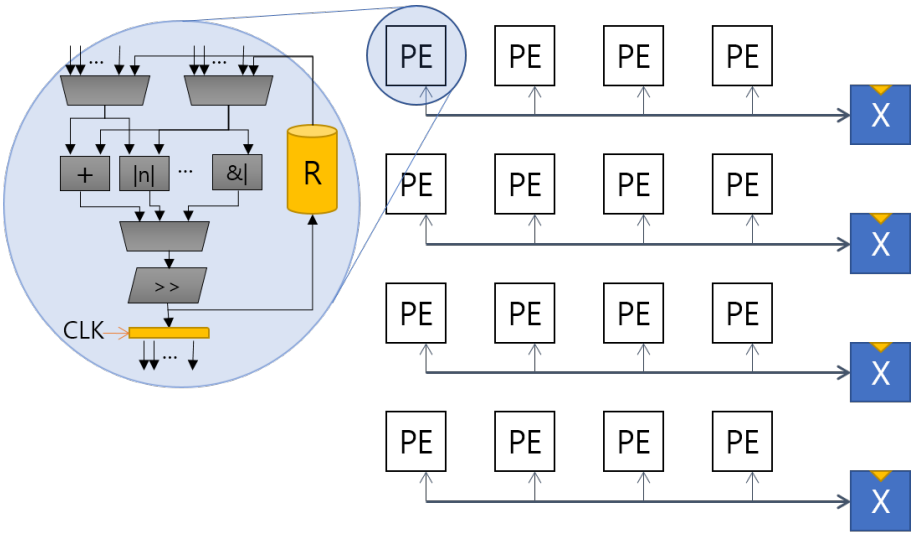


Figure 2.3 The inner-structure of PE and the shared modules in PE array.

from its corresponding CE in order to implement the kernel's dataflow on the two-dimensional array of PEs. The data stream loaded from the data memory in the RCM flows along the array of PEs and the results are stored back to the data memory. In this strategy, if the dataflow is larger than the array size, it cannot be mapped onto the array. On the other hand, in the temporal mapping, the kernel's dataflow is implemented on a column of PEs. Every cycle, new configuration data fetched from the CEs change the configuration of the column according to the dataflow. In this strategy, only a column of PEs can run the entire dataflow, since the PEs can be reconfigured every cycle to perform any necessary operations in the dataflow. Therefore, any dataflow can be mapped onto the array regardless of the size (provided that the CEs can store

all the configuration data). Since there are multiple columns of PEs, multiple iterations of a kernel loop can be executed in parallel through loop pipelining. The configuration memory has a hybrid structure [30] in order to support both strategies.

CCU has a simple address generation mechanism for the configuration memory, called macro configuration. In this mechanism, macro configuration operations (MCOs) are used for address generation. Each MCO is 2-byte long and consists of start address, address count, etc. MCOs are stored in a memory block, called MCO table, in CCU. To execute RCM, CCU fetches MCOs from the MCO table and generates a sequence of addresses corresponding to each fetched MCO. The generated addresses are sent to the configuration memory one by one to load the configuration words onto the PE array. With this mechanism, frequently used chunks of configuration words can be reused just by duplicating MCOs, which saves a lot of configuration memory space.

The data memory in the RCM contains data to be processed in the PE array. There are two sets of data memory used to support double buffering for hiding communication overhead. Each set of data memory consists of three banks. By adjusting control registers, each bank of the data memory can be attached to one of three read/write data buses (shown in Figure 2.2) in the PE array. Two banks are used for input and the remaining one is used for output. All PEs in a column can access the data memory at the same time. But PEs in each row

share the data buses so that only one PE can read/write data from/to memory through a bus at a time (since there are three buses, three PEs in a row can access the memory at a time).

# Chapter 3

# DEGISN OF FLOATING-POINT OPERATIONS

## 3.1 Floating-point Numbers

### 3.1.1 Representation of floating-point numbers

Floating-point representation internally used in the PE array is different from the IEEE standard 754 [31]. In the single precision of the IEEE standard, the floating-point representation consists of 1-bit sign, 8-bit exponent, and 23-bit mantissa as shown in Figure 3.1(a). In the PE array, the floating-point format is rearranged (Figure 3.1(b)) and separated (Figure 3.1(c)) since a floating-point value is managed by a pair of PEs: one treats the signed mantissa part of the floating-point value, while the other treats the exponent part of the floating-point value.

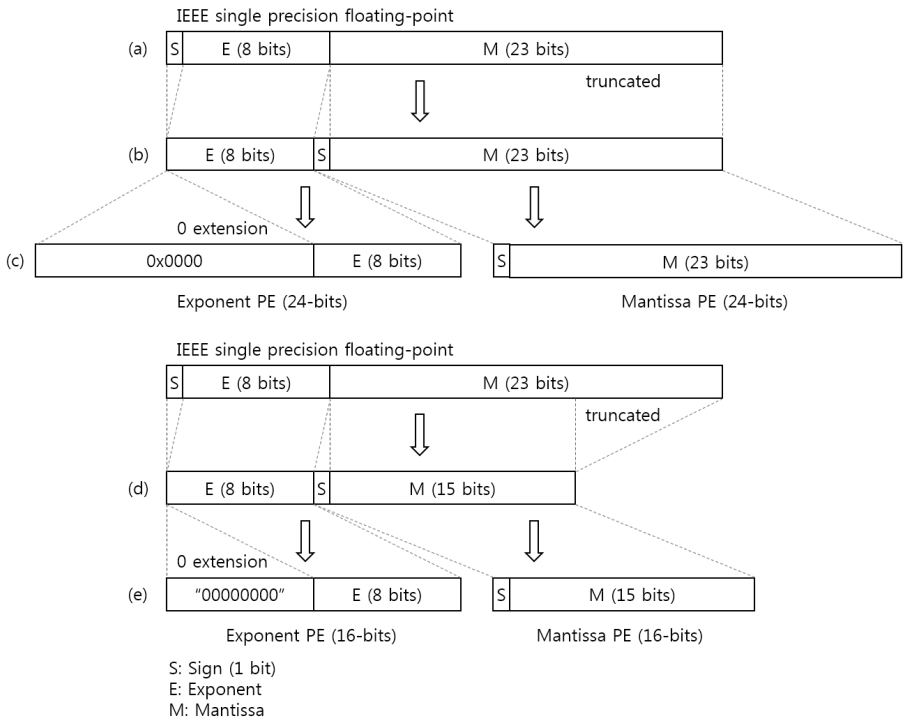


Figure 3.1 Floating-point formats: (a) 32-bit IEEE-754 simple precision, (b) 32-bit format internally used in PE array which has 24-bit data-path, (c) separation of a 32-bit floating-point value, (d) reduced 24-bit format internally used in PE array which has 16-bit data-path, and (e) separation of a 24-bit floating-point value.

In the case that the bit-width of the data-path of each PE is less than 23 bits of mantissa part, the least significant bits are truncated. Figure 3.1(d) and (e) shows the representation of floating-point values used in the PE array which has 16-bit data-path. 8 bits of the mantissa part is truncated so that the reduced floating-point format has 24-bit in total: 1-bit sign, 8-bit exponent, and 15-bit mantissa, as shown in Figure 3.1(d). If the precision of 15-bit mantissa is good enough for the targeting situation, hardware cost of the architecture can be decreased easily by reducing the bit-width of the data-path. Reduced floating-point formats are often used for low-end embedded systems because they have larger dynamic ranges than integer formats in case that the precision is not important [32, 33].

We assume that floating-point values are stored in the data memory of the RCM confirming to the IEEE single precision standard. In other words, floating-point numbers are transformed where they are transferred between the data memory and the PE array. If we truncate 8 least significant bits of mantissa part while loading data from the data memory to the PE array. When we store the floating-point results back to the data memory, we attach zeroes for the 8 least significant bits of the mantissa part. Figure 3.2(a) shows the hardware structure of the format converter from the data memory to the PE array. IEEE-754 floating-point numbers (32-bits) are converted as the separated format used in the CGRA and transferred to the corresponding PEs. On the other hand, Figure

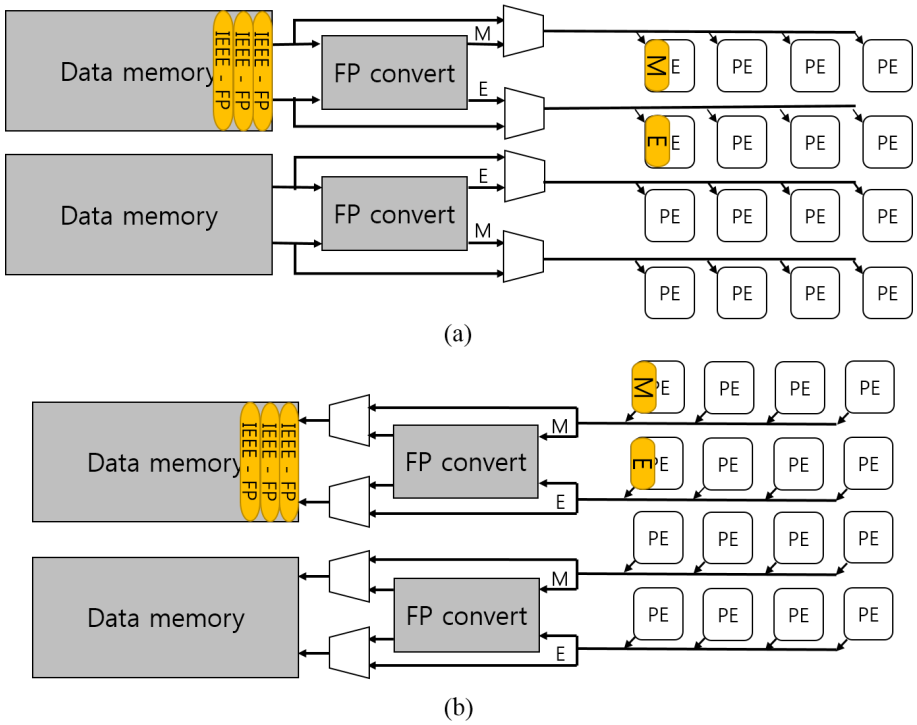


Figure 3.2 Floating-point format between PE array and the data memory. (a) shows the data-path from the data memory to the PE array, and (b) shows the data-path from the PE array to the data memory. Yellow tokens represent floating-point data.

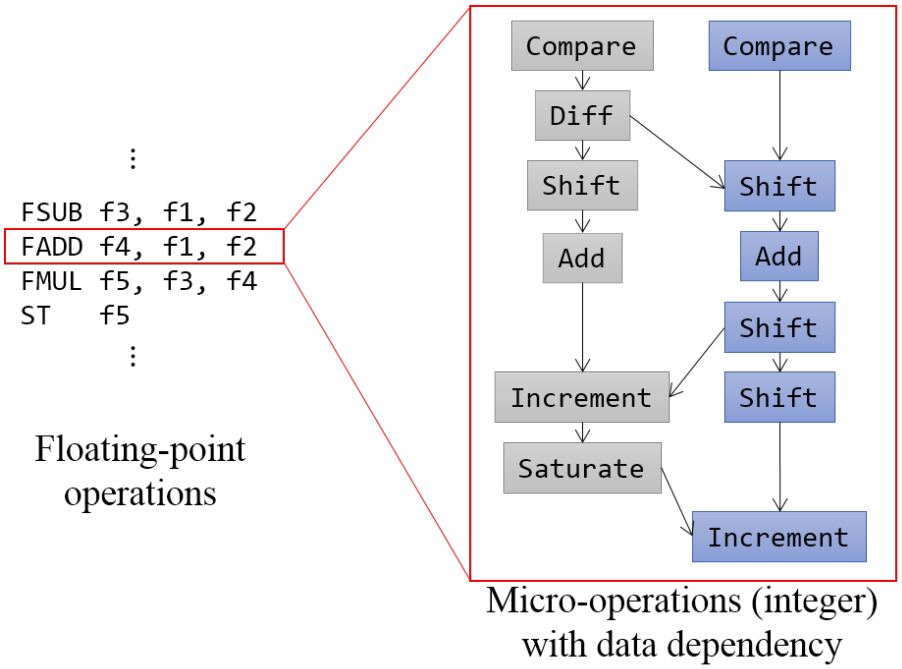


Figure 3.3 A floating-point addition operation boxed in the left side is split into several integer micro-operations with data dependency.

3.2(b) shows the format conversion from the PE array to the data memory.

### 3.1.2 Floating-point operations

A floating-point operation can be divided into several micro-operations each of which deals with a part of the whole floating-point operation. For example, a floating-point addition can be split into several micro-operations, as shown in Figure 3.3. Micro-operations have one or two fixed point values as its input, and outputs a fixed point value, in other words, they are integer operations

each of which can be dealt by a PE in the PE array. Execution of floating-point operations on a PE array is similar to the emulation of floating-point operations in integer-only processors, except that the micro-operations from floating-point operations are scheduled by the configurations and they are executed in parallel.

## 3.2 FPU-PE Cluster

### 3.2.1 Construction of FPU-PE Cluster

Each PE in the PE array has enough functionality to manipulate integer values. However it by itself cannot handle floating-point values efficiently. So, for a floating-point operation, we combine two PEs such that one PE takes charge of sign and mantissa parts of the floating-point operation, while the other PE takes charge of exponent part with remaining most significant bits set to zero as shown in Figure 3.1(c) or (e). The former is called mantissa PE and the latter is called exponent PE. Such a pair of PEs that co-operates to execute floating-point operations is called FPU-PE cluster (Figure 3.4).

To execute floating-point operations more efficiently, we add several sub-modules to each PE. Since the operations applied to the mantissa part and the exponent part are quite different, the sub-modules added to the mantissa PEs are different from those added to the exponent PEs, which makes the PE array heterogeneous. For instance, each mantissa PE has a leading-one detection

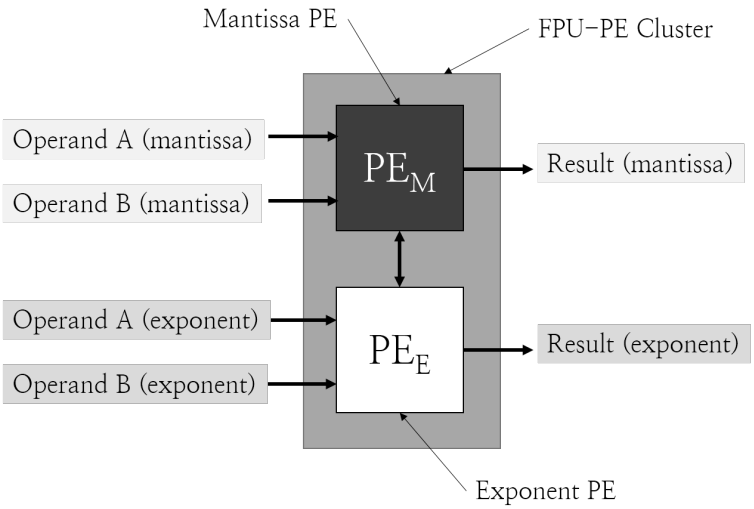


Figure 3.4 Overview of FPU-PE cluster.

module used for normalizing mantissa values, while each exponent PE has a saturation module used for limiting the exponent value not to exceed the value of infinity (0xFF in the single precision floating-point standard). If the architecture were designed with a homogeneous array of PEs, the hardware cost would be much higher without any performance gain.

### 3.2.2 Construction of Array of FPU-PE Clusters

Figure 2.2 shows the interconnection topology of the PE array. The basic topology is mesh but some of 2-hop, 3-hop, and pair-wise interconnects are added. With these abundant interconnects, the PEs can efficiently transfer their output data to others, allowing easy mapping of applications onto the array. The

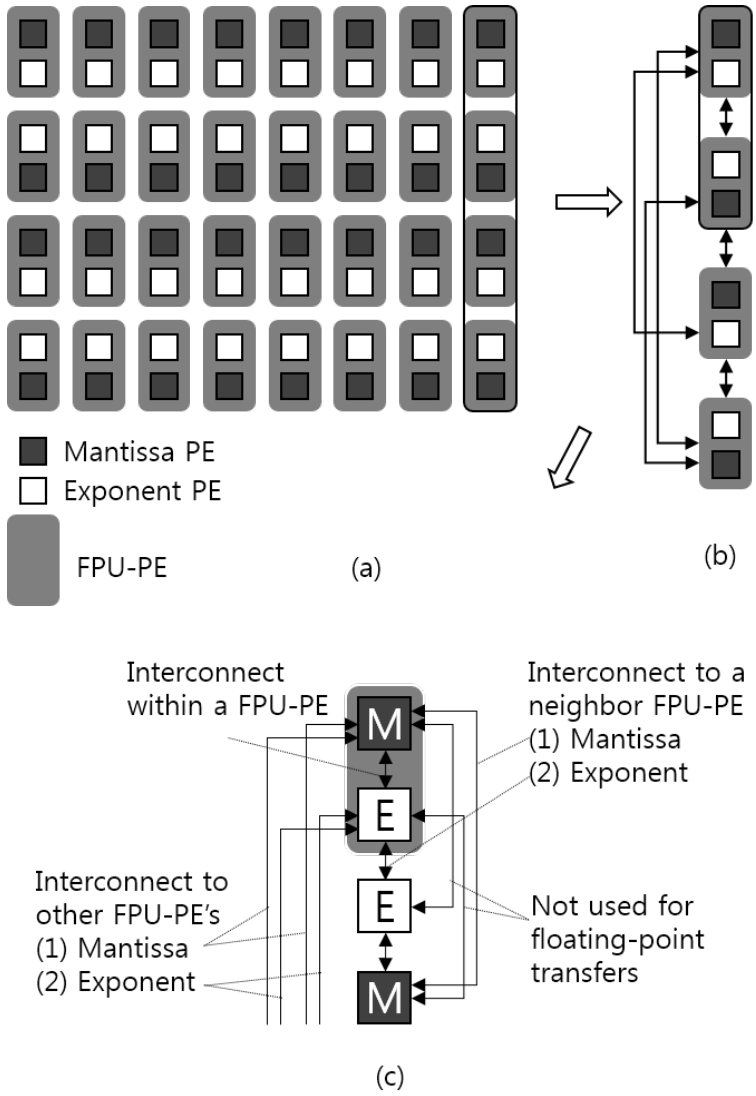


Figure 3.5 Construction of FPU-PE cluster array: (a) locations of Mantissa PEs and Exponent PEs in PE array and constructed FPU-PE array, (b) abstracted interconnection among RPU-PE clusters, and (c) interconnection of an FPU-PE cluster in detail.

Table 3.1 Comparison between different FPU-PE clusters

| Clustering case | Fixed | Paring1 | Paring2 |
|---|---|---|---|
| Functionality of PE | Either M or E | Either M or E | Both |
| Clustering | Fixed cluster | Flexible<br>(1 $PE_M$ and 1 $PE_E$) | Flexible<br>(2 PEs) |
| Hardware overhead | FSM / FU / control<br>for one of M or E | FSM / FU / control<br>for one of M or E,<br>pointer to the pair | FSM / FU / control<br>for both of M and E,<br>pointer to the pair |
| Configuration | FP-operation only | FP-operation<br>Pair selection | FP-operation<br>Pair selection |

mantissa PE and the exponent PE of an FPU-PE are located side by side for frequent communications between them. Also the mantissa PEs and exponent PEs are properly placed in order to construct an array structure of FPU-PEs as shown in Figure 3.5. Thus we make best use of the array structure in case of floating-point operations as well as integer operations.

### 3.2.3 Comparing Different FPU-PE Clusters

There are several ways to construct FPU-PE clusters. We can just fix the PEs to construct a FPU-PE cluster, or dynamically change the PEs in a FPU-PE cluster. A PE can deal with either mantissa part or exponent part, or a PE can deal with

both of the parts. Table 3.1 shows the three candidates for construction of FPU-PE clusters. "Fixed" is the case that behavior of every PE is fixed so that a PE is either a mantissa PE or an exponent PE, and a pair of PEs constructing a FPU-PE cluster is also fixed. The additional hardware cost of each PE is for the FSM, additional functional units, and modified control/data paths. "Paring1" is the case that the behaviors of the PEs are fixed, but the cluster is flexible so that any pair of a mantissa PE and an exponent PE which are interconnected directly can construct FPU-PE cluster and calculate floating-point operations. In this case, additional hardware cost for the pointer to the pair PE in the same cluster is required compared to the "Fixed" case. "Pairing2" is the case that all the PE have functionality of both mantissa and exponent and any two PE can construct a FPU-PE cluster. According to the operations, PE's behavior is determined whether it manages mantissa part or exponent part. Multi-cycled floating-point operations implemented in the every clustering cases are using FSMs to control PEs during the execution cycles, instead of configuring with the configuration bits from the configuration memory, which will be explained and analyzed in detail in 3.3.

Figure 3.6 shows the normalized hardware area costs of the PE arrays of the three different clustering cases. Since the FPU-PE clusters are not fixed in the "Pairing1" and "Pairing2" cases, they requires additional configurations such as selecting the pair PEs for constructing FPU-PE clusters. It is implemented as

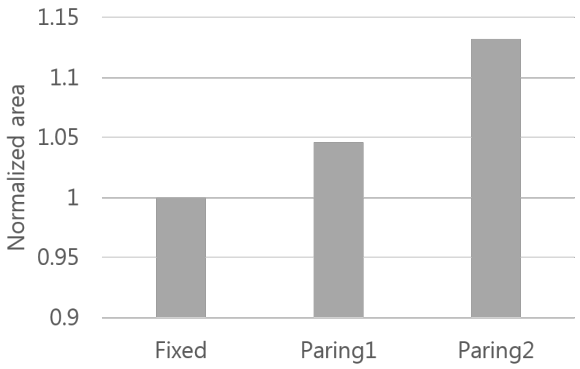


Figure 3.6 Normalized hardware area of the three different clustering cases.

an operation so that it may increase the number of operations of the kernels compared to the "Fixed" case. Compared to the "Fixed" case, "Paring1" requires 4.6% more and "Paring2" requires 13.2% more than the "Fixed" case, shown in the Figure 3.6.

Since "Paring1" and "Pairing2" allows more flexibility when constructing FPU-PE clusters, it is not effective floating-point conversion between the data memory and the PE array shown in Figure 3.2 limits their effects. Mantissa parts and exponent parts of floating-point numbers are sent to/received from certain PEs. Floating-point transfer in the FPU-PE cluster array shown in Figure 3.5 is efficient enough in that there are the same number of FPU-PE clusters in the PE array even though FPU-PE clusters are changed dynamically which are supported in "Pairing1" or "Pairing2" cases.

Table 3.2 Properties of implemented floating-point operations

| Operations | Input | Output | Latency (cycles) | Method |
|---|---|---|---|---|
| Add/sub | float | float | 6 | Internal ALU only |
| Multiply | float | float | 4 | Shared multiplier |
| Div | float | float | 7 | Shared divider |
| Square root | float | float | 7 | Shared square-root |

## 3.3 Implementation of Multi-Cycle Operations

Floating-point operations are much more complex than integer operations so that they are implemented as multi-cycle operations. To perform a floating-point operation, the pair of PEs in an FPU-PE co-operates for several cycles with its intermediate data stored in the local register files. Table 3.2 shows cycle counts of the floating-point operations implemented on each FPU-PE.

There are two different approaches to reconfiguring a PE for a multi-cycle operation. One is updating the PE's configuration every cycle with the code fetched from configuration memory until the operation is completed as shown in Figure 3.7(a), and the other is implementing a separate FSM in the PE to control the multi-cycle behavior of the PE as shown in Figure 3.7(b). The former does not require additional hardware like FSM, but it requires a lot of memory area

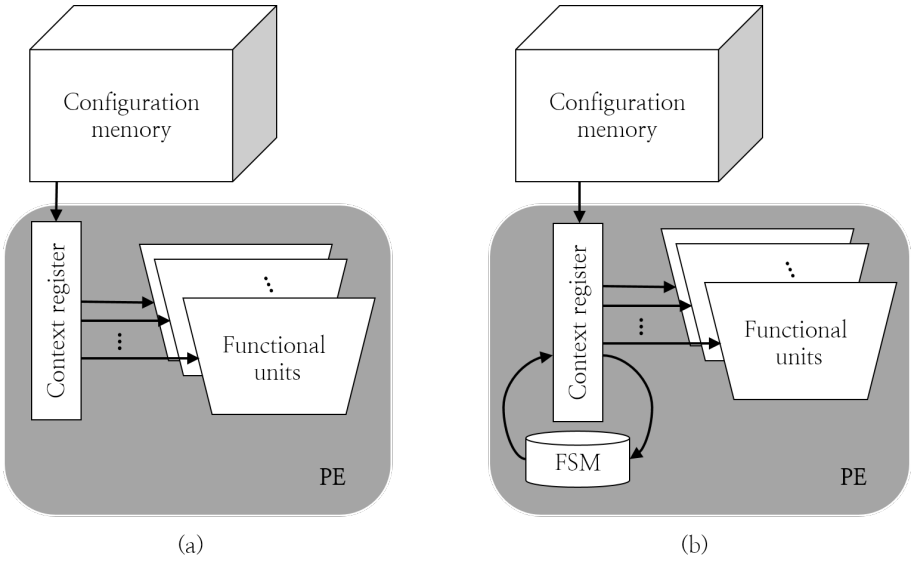


Figure 3.7 Reconfiguration of a PE in the (a) base architecture, and (b) FSM-included architecture.

for every-cycle configuration. It is also inevitable to consume more power for loading configuration words from the configuration memory. The latter requires implementation of FSM to control functional units in each PE, but it requires less memory space for the same functionality compared to the former. The both approaches accepts the fixed FPU-PE clustering.

Table 3.3 shows the hardware cost of the two different floating-point implementations. "FSM approach" indicates the case that the functional units in each PE are controlled by the corresponding FSM during floating-point executions, while "Config. memory approach" indicates that the functional units in

Table 3.3 Comparison of hardware cost per PE

| | FSM approach | Config. memory approach |
|---|---|---|
| Hardware cost per PE ($um^2$) | 8820.2 | 6160.8 (69.8%) |

each PE are controlled by the configuration bits from the configuration memory during floating-point executions. The numbers are effective hardware area of a PE. This shows that "FSM approach" requires more hardware cost than "Config. memory approach" for the FSMs to control the functional units in the corresponding PEs.

We choose the former approach in spite of its hardware cost and complexity as shown in Table 3.3, due to the following reason. Each floating-point operation requires as many steps as the latency shown in Table 3.2. If the implementation of floating-point operations relies only on reconfigurations (without an FSM), then, at each step, it needs to fetch a new configuration for a proper operation corresponding to that step (note that, in the FSM approach, each floating-point operation requires only one configuration word which yields the control to the FSM). For an application kernel that uses floating-point operations frequently, the required configuration memory space will increase easily beyond our control. Moreover, accessing the configuration memory every cycle while floating-point operations are executed causes much more power con-

Table 3.4 Comparison of memory usages for 9/7tap wavelet transforms

| Type of config. memory | FSM approach | Config. memory approach | |
|---|---|---|---|
| | Size(bytes) | Size(bytes) | Ratio(%) |
| Temporal config mem | 800 | 3264 | 408 |
| Spatial config mem | 1024 | 1024 | 100 |
| MCO mem | 44 | 44 | 100 |
| Total | 1868 | 4332 | 232 |

sumption compared to the case of using FSM. Table 3.4 shows memory usage of the two different approaches for a simple benchmark program of 9/7tap wavelet transform implemented with floating-point operations. Compared to the FSM approach, the approach of updating configuration memory every cycle requires about 2.3 times of memory space. For a larger application, it will require larger configuration memory space to exceed easily the existing local configuration memory space. Then the configuration words should be fetched from the external memory while the application is running. This implies that the requirement of large configuration memory space may bring about not only area overhead but also performance overhead.

## 3.4 Implementation of Floating-Point Operations

Floating-point addition and subtraction share the same data-path. Subtraction can be implemented easily with the data-path for floating-point addition by toggling the sign of the subtrahend. Floating-point addition/subtraction takes six cycles in total by an FPU-PE. In the first cycle, the exponent PE in the FPU-PE compares the exponents of the two operands (at the same time, the mantissa PE also compares the absolute values of the mantissas in case the two exponents are same). It keeps the larger exponent value and sends the difference to the other PE so that the mantissa PE can align the radix points of the two mantissa values in the second cycle. In the third cycle, the mantissa PE performs addition/subtraction on the two operands depending on the sign difference of the two operands. In the fourth cycle, the mantissa PE obtains the position of leading-one in the addition/subtraction result in order to normalize it. In the next two cycles, the two PEs perform normalization by shifting the mantissa part and updating the exponent part accordingly and then check the range of the result. The behaviors of floating-point addition and subtraction are shown in Figure 3.8.

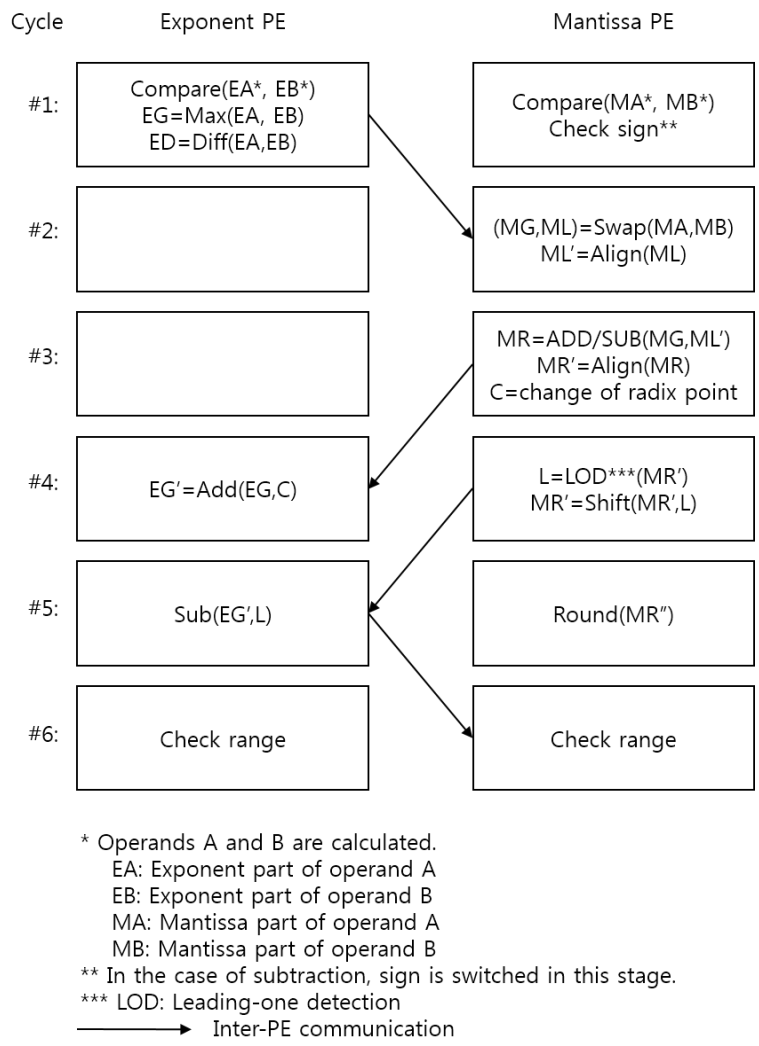


Figure 3.8 Behavior of floating-point addition for each cycle.

## 3.5 Implementation of Floating-Point Operations Using Shared Modules

Floating-point multiplication, division, and square-root operations are also implemented on the RCM. For those operations, separate shared integer functional modules (multipliers, dividers, and square-root units) are utilized for mantissa calculation. Each of the integer functional modules is shared by a set of PEs as mentioned in Section 2.2. For a floating-point operation on an FPU-PE, while the exponent PE calculates the exponent part, the mantissa PE sends two operands to the shared module and stalls until it receives the result back several cycles later when the shared module completes the calculation. Once the mantissa PE gets the intermediate result from the shared module, both the mantissa PE and exponent PE construct the final result and check its range and compensate it not to be out of range. The behavior of floating-point multiplication is shown in Figure 3.9.

Floating-point division and square-root operation are similar to multiplication. Floating-point division uses a mantissa PE and a shared divider in order to obtain the quotient of the mantissas. The exponent PE subtracts two exponents and then adds the bias number while mantissa PE waits for the result from the shared divider. Then the PEs follow the procedure of checking ranges of the result similarly to the floating-point multiplication shown in Figure 3.9. Since the division of the mantissa part takes more cycles than multiplication does, the

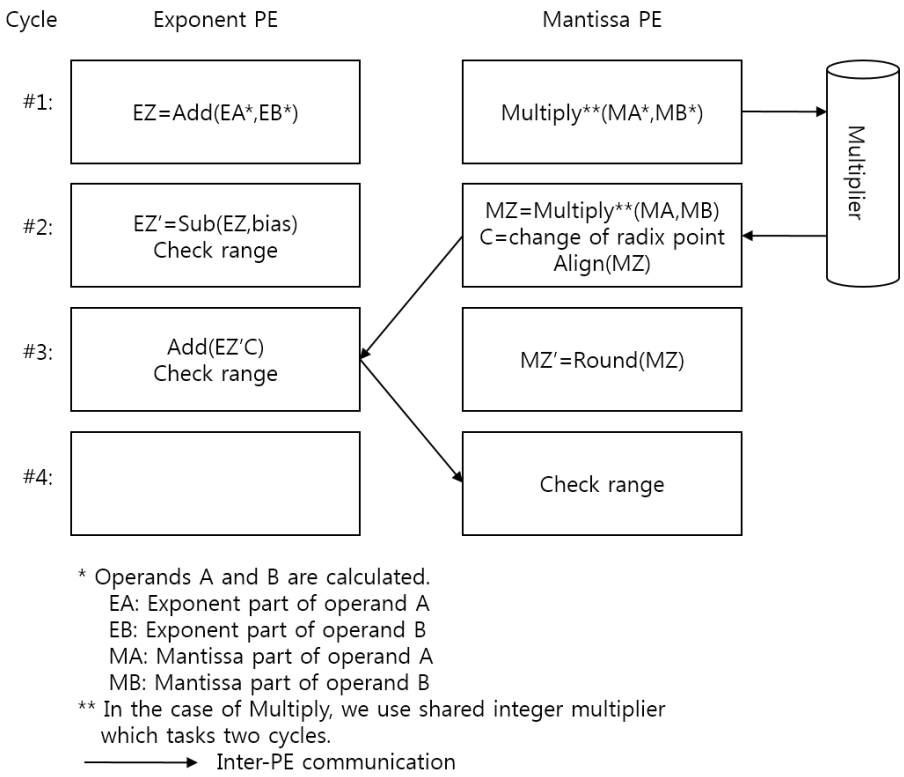


Figure 3.9 Behavior of floating-point multiplication for each cycle.

latency of the whole floating-point division operation is greater than that of the floating-point multiplication.

In case of floating-point square-root operation, one operand is given in an FPU-PE cluster. The exponent PE first checks to see whether the exponent part is odd or even. The flag is sent to the mantissa PE and the mantissa part is shifted by the flag in the mantissa PE. Then the mantissa PE sends the mantissa part to the integer square-root module, while the exponent PE divides the exponent

part by 2 (subtract bias rom the exponent part, shift right, and then add the bias). After the mantissa PE obtains the result from the shared square-root module, both PEs in the FPU-PE cluster check exception. Since the shared square-root module also takes long time, the total execution latency of the floating-point square-root operation is also long.

The properties of all the implemented operations are shown in Table 3.2. All the operations deal with 24-bit reduced precision floating-point values.

# Chapter 4

# Chip Implementation

## 4.1 Specification of Chip Implementation

Our architecture, which has been implemented to a chip, consists of an ARM7-compatible processor, a DMA controller, and an RCM, all of which are connected to an AHB bus. An AHB slave port and the processor's JTAG port are used as the chip's external interfaces. The RCM has an 8x8 array of PEs so that it can perform 64 integer operations or 32 floating-point operations at the same time. All PEs in each row of the array share a multiplier, all PEs in each of the 1st, 4th, 5th, and 8th rows share a divider, and all PEs in each of the 1st and 8th rows share a square-root unit. We have designed the architecture this way mainly because dividers and square-root modules occupy a quite larger area. Besides division and square-root operations are rarely used compared to

multiplication. Integer functions such as DCT, FFT, FIR filters, which are frequently accelerated on the RCM, do not require division and square-root at all. And even floating-point functions rarely use square-root operations. Normalization, which is one of the most frequently used functions in 3D graphics, requires one square-root operation and three division operations among nine arithmetic operations in total. Since only the mantissa PE in an FPU-PE cluster uses a shared divider during floating-point division, having only one divider shared by two rows of PEs is enough to execute normalization.

The configuration memory size is 5632 bytes and data memory size is 6144 bytes. CCU has an MCO table of 128 bytes for the generation of configuration memory addresses. The chip has been fabricated using Dongbu HiTek 130nm CMOS technology. Figure 4.1 shows a chip photograph of the proposed architecture laid on 4.1 mm × 4.1 mm area. The equivalent gate count of the whole architecture implemented on the chip is 1338k. Compared to an integer-only design, the size of the RCM is increased by 7.4% due to the implementation of floating-point operation, while maintaining the clock frequency, compared to our previous RCM architecture where the proposed technique is not applied. 17% of the increment is from the addition of functional units such as leading-one detectors, saturators, while 34% is from the added FSM and decoder modification, The rest of the increment is from other control- and data-paths like flag management for the pair PEs in an FPU-PE, flag registers, and so on. The

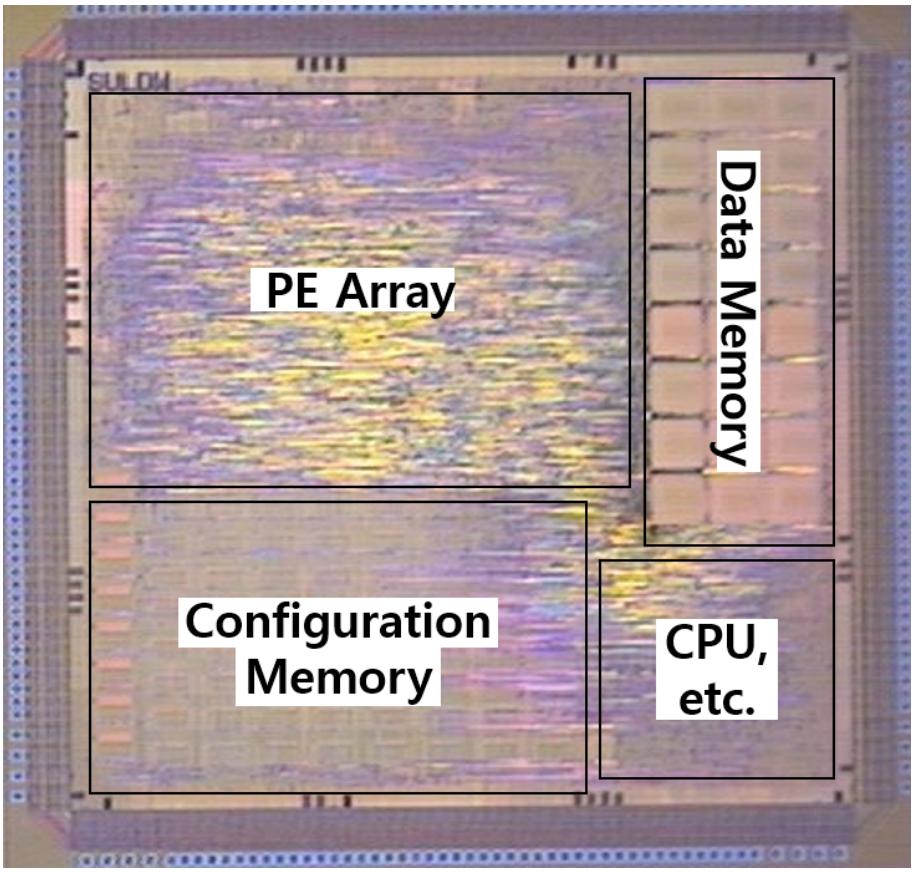


Figure 4.1 Micrograph of the fabricated chip.

proposed design, which we call FloRA, achieves maximum 6625M integer operations and 667.8M floating-point operations per second at 125 MHz operating frequency with 1.2V supply voltage.

Breakdown of 7.4% of the increased hardware area cost is shown in Figure 4.2. The largest portion of the increment is from the FSM for control of multi-cycle operations. Additional modules such as saturation unit, leading-

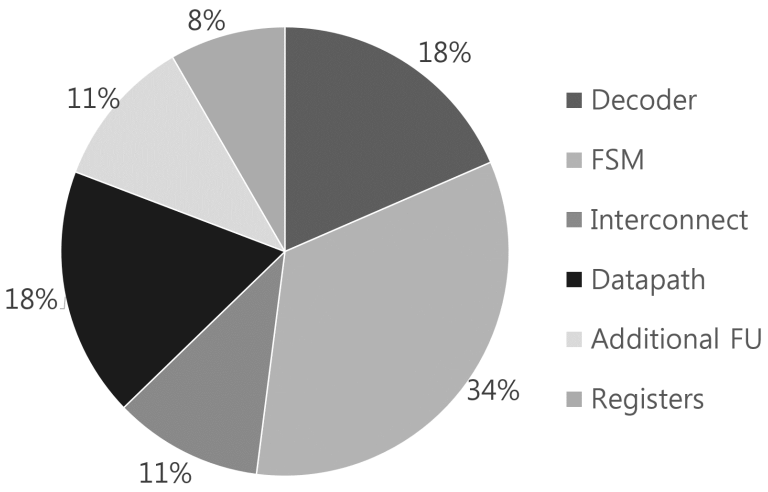


Figure 4.2 Area breakdown of the increased hardware.

one detector and registers are not very large compared to the modification of decoder and data-paths in each PE. Impacts of additional interconnects is also not very much.

## 4.2 Experimental Setup

For the test of the fabricated chip, we have constructed the prototype board by connecting external memory blocks to the chip through the chip's external interfaces. The board has been designed as a daughter board mounted on top of an existing FPGA board. The AHB slave port – one of the external interfaces of the chip – has been connected to an external memory block via an AHB bridge

module implemented on the FPGA. Besides, JTAG port for the RISC processor has been connected to a PC to control the processor. With this setup, we can download test programs and data into the external memory on the test board via a JTAG cable. We can also control the status of the processor and run/stop the execution of the processor. The FPGA board runs at 24MHz and the fabricated chip operates at 1.2V and the same clock speed as the board. Note that the maximum clock frequency of the chip is 125MHz.

Power consumption of the chip has been measured on the prototyping board. Power for the core in the chip has been supplied from a different power source from the FPGA board. By measuring the voltage and current of the separately supplied power source, we can obtain the power consumption of the chip.

## 4.3 Experimantal Results

### 4.3.1 Performance Comparison

In this subsection, we present the evaluation result of the fabricated chip implementing our coarse-grained reconfigurable architecture having floating-point functionality. We have tested it with several integer and floating-point benchmark kernels as shown in Table 4.1. FloRA has been tested at 24MHz on the prototype board, but the performance has been calculated as if it were running at 125MHz, which is the maximum operating frequency of the fabricated chip. Also we have run the same kernels on the ARM7 platform and ARM9 platform

Table 4.1 Performances of benchmark kernels accelerated by FloRA

| Kernel name | Type | Software-only (ARM7) | | ARM9+VFP | | FloRA | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Latency (cycles) | Thruput @133MHz (iter/sec) | Latency (cycles) | Thruput @210MHz (iter/sec) | Latency[2] (cycles) | Thruput @125MHz (iter/sec) | Thruput improv. over ARM7 | Thruput improv. over ARM9+VFP | Utiliz. of PE array (%) |
| Complex Mult. | int | 34 | 3.91M | 32 | 6.56M | 4 | 166.7M | 42.62 | 25.40 | 29.2 |
| Dot product (1×4) | int | 44 | 3.02M | 37 | 5.68M | 5 | 76.9M | 25.44 | 13.55 | 23.1 |
| Normalize (1×3) | float | 1186 | 0.11M | 116 | 1.81M | 31 | 25.6M | 228.28 | 14.14 | 35.3 |
| Mat. Mult. (4×4) | float | 5691 | 0.02M | 378 | 0.56M | 53 | 4.3M | 183.99 | 7.74 | 79.3 |
| Complex Mult. | float | 248 | 0.54M | 30 | 7.00M | 11 | 52.6M | 98.08 | 7.51 | 42.1 |
| Cross prod. (1×3) | float | 421 | 0.32M | 46 | 4.57M | 21 | 34.5M | 109.21 | 7.56 | 54.3 |
| Dot product (1×4) | float | 271 | 0.49M | 48 | 4.38M | 16 | 41.7M | 84.97 | 9.53 | 34.0 |
| Convolution (len:32) | float | 2663 | 0.05M | 188 | 1.12M | 51 | 2.1M | 42.05 | 1.88 | 33.4 |
| FIR filter(4-tap)[1] | float | 275 | 0.48M | 23 | 9.13M | 43 | 156.9M | 324.42 | 17.18 | 82.4 |
| Average | - | 1204 | 0.99M | 99.8 | 4.53M | 26.1 | 62.37M | 126.56 | 11.61 | 45.9 |

[1] Integer complex multiplication, integer dot product, and floating-point matrix multiplication execute two iterations concurrently, and FIR filter processes eight samples concurrently within the given latencies.

[2] In case of FIR filter, the throughput is in samples/s.

with a vector-floating-point (VFP) unit for comparison. The ARM7 processor emulates a floating-point operation as a series of integer operations since there is no floating-point unit in the processor. All the codes running on the processors have been compiled with -O2. Third to fifth columns of Table 4.1 show the latencies obtained for the benchmark kernels running on each platform. Latencies of the benchmarks running on the ARM7 and ARM9 with a VFP unit are obtained by the ARM instruction-set simulator (AXD). Compared to software-only implementations on a processor (ARM7 at 133MHz), the FloRA implementations operating at 125MHz are 126.6 times faster on average. And compared to those on ARM9 processor with a VFP unit operating at 210MHz, the FloRA implementations are 11.6 times faster. By executing two iterations concurrently in the cases of integer complex multiplication, integer dot product, and floating-point matrix multiplication, 2X throughput improvement can be obtained. In the case of FIR filter, eight samples are processed at the same time on the PE array of FloRA, which has also been considered in the throughput calculation. The utilization in the table means the average ratio of number of active PEs over number of total PEs in the PE array within execution cycles. It is one of the main factors that affect the power consumption in FloRA.

### 4.3.2 Power Consumption Comparison

For comparison of power consumption among different architectures, we measured the power consumption of the fabricated chip. The consumed power per clock frequency of the benchmarks running on the proposed architecture is 3.70mW/MHz on average. On the other hand, power consumption of ARM7 (ARM7TDMI-S in 130nm process whose maximum frequency is 100-133MHz) is 0.11mW/MHz [34], and that of ARM9 (ARM968E-S with VFP9 in 130nm process whose maximum frequency is 180-210MHz) is about 0.5mW/MHz [35, 36]. It has not been confirmed whether the power consumptions of both ARM7 and ARM9 architectures are obtained from macro-cells or chips. Even though the power consumption per unit frequency of the proposed architecture is higher than those of the two ARM architectures, the execution time of benchmark programs on the proposed architecture is much shorter than those on the other architectures. The energy consumption for the benchmark programs in the proposed architecture is 2.45% (7.57% for integer programs and 2.42% for floating-point programs) of that of ARM7, and 17.7% (24.0% for integer programs and 17.6% for floating-point programs) of that of ARM9 with VFP9. The results show that the proposed architecture is efficient than the software approach in terms of both performance and power consumption, and comparable with floating-point unit approach in terms of power consumption.

Figure 4.3 shows the dynamic power consumption of the benchmark ker-

nels on the fabricated chip. The numbers on the vertical axis represent dynamic power consumed by performing benchmark programs on the RCM. As can be seen in this figure, floating-point-based programs consume power comparable to the integer-based programs. When floating-point operations are executed, some PEs use multiple functional units simultaneously, but some other PEs do nothing but just wait while other PEs perform the calculations. Thus the power consumption of a floating-point operation is similar to that of integer operation. The overall power consumptions of applications depend on the utilization of the PEs rather than whether the application programs are based on floating-point operations or integer operations as shown in Figure 4.3. Two kernels, matrix multiplication and fir filter, consume much more power than other kernels, since they have much higher utilization of the PEs.

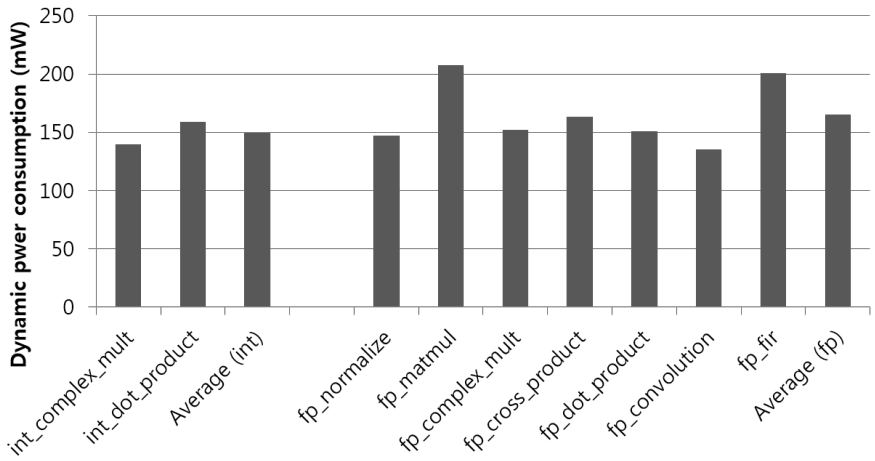


Figure 4.3 Dynamic power consumption of the benchmark kernels on the fabricated chip.

# Chapter 5

# Comparison with Other Architectures

## 5.1 Preparation for the comparison

The proposed architecture has 16bit-wide data-paths and does not support IEEE single precision floating-point standard, and thus it is not fair to compare the architecture with other architectures that support the IEEE standard. For a fair comparison, we have modified some features of FloRA. First, we have enlarged the bit-width of the data-paths of the PE array from 16 bits to 24 bits to support the IEEE single precision floating-point standard. We have removed modules for floating-point division and square-root, which are not supported in the other architectures to be compared. We have further optimized the design to reduce the critical path delay by placing shifter and ALU in parallel, which were serialized in the original design. It increases the cycle count

Table 5.1 Comparison of different FloRA implementations

| Features | FloRA Chip implementation | new FloRA implementation |
|---|---|---|
| FADD/FSUB | 6 cycles | 8 cycles |
| FMUL | 4 cycles | 5 cycles |
| FDIV | 7 cycles | – |
| FSQRT | 7 cycles | – |
| Clock freq. | 200 MHz (post-synthesis) | 328 MHz (post-synthesis) |

of floating-point addition and subtraction from 6 cycles to 8 cycles and that of floating-point multiplication from 4 cycles to 5 cycles. However, it increases the clock frequency significantly. Modified design is synthesized by Synopsys Design Compiler with TSMC 130nm library. The estimated clock frequency after logic synthesis of the 24-bit FloRA is 328 MHz while that of 16-bit architecture is 200MHz (125MHz after fabrication). Besides, area of 24-bit FloRA is a bit reduced compared to the 16-bit FloRA. It is because the integer division units, integer square-root units, and their corresponding control logic have been removed. The features of chip implementation of FloRA and the modified FloRA are summarized in the Table 5.1.

## 5.2 Comparison with PACT XPP

The approach in [23] also implements floating-point operations on PACT XPP, a commercial coarse-grained reconfigurable architecture. XPP has 2-dimensional array of integer ALUs - so called ALU-PAEs - and it accelerates various integer kernels with those ALUs. ALUs are connected with crossbar switches and registers. There are only integer ALUs in the architecture, it divides floating-point operations into several integer micro-operations each of which can be handled in the architecture. The approach to implement floating-point operations in XPP architecture is similar to our approach, but XPP architecture does not fix the ALUs to execute either mantissa part or exponent part.

In order to compare FloRA to the architecture, we have implemented the same benchmark kernel described in [23] for FloRA. There are two kinds of discrete wavelet transforms (DWTs) in [23]: reversible 5/3 tap DWT using fixed-point numbers and irreversible 9/7 tap DWT using floating-point numbers, and both are included in JPEG2000 standard. For our comparison the 9/7 tap DWT has been selected.

XPP64-A1, a commercial implementation of XPP architecture, is used for the comparison. It is also fabricated as a chip with 130nm technology [37]. It contains 64 ALU-PAEs (PAE: Processing Array Element), which is the same as the number of PEs in FloRA. However, the bit-width of ALU-PAE is 24 bits. Thus we can fairly compare it with FloRA with the bit-width of the data-paths

Table 5.2 Comparison of hardware features of PACT XPP and FloRA

| Features | XPP [37] | FloRA |
|---|---|---|
| CMOS process (nm) | 130 | 130 |
| Data bit-width (# bits) | 24 | 24 |
| Clock frequency (MHz) | 64 | ~ 205[1] |
| Area ($mm^2$) | 35.1 | ~ 16.8[2] |

[1] Clock frequency is from estimation on the basis of the chip implementation of 16-bit FloRA.

[2] Area cost is from estimation on the basis of the chip implementation of 16-bit FloRA.

enlarged to 24 bits.

Table 5.2 shows the hardware features of both XPP architecture and FloRA with 24-bit data-path. The maximum clock frequency of the modified FloRA with 24-bit data-path was estimated with the information of FloRA with 16-bit data-path. The estimated maximum clock frequency after fabrication was 205MHz, which was much greater than 125MHz, the clock frequency of the 16-bit FloRA chip. Although the area has been reduced from that of 16-bit FloRA, the difference is not big compared to the whole chip size.

Table 5.3 compares the execution times of 9/7 tap DWT running on XPP and FloRA for various image sizes. Although FloRA takes more cycles for the

Table 5.3 Comparison of performances between PACT XPP and FloRA running 9/7 tap discrete wavelet transform

| Image size | XPP @64 MHz [23] | | FloRA @205 MHz[1] | | |
|---|---|---|---|---|---|
| | Cycles | Time (ms) | Cycles | Time (ms) | Speed-up (X) |
| 256×256 | 398,000 | 6.200 | 709,536 | 3.461 | 1.80 |
| 512×512 | 1,577,758 | 24.652 | 2,813,652 | 13.725 | 1.80 |
| 1024×1024 | 6,296,340 | 98.380 | 11,164,680 | 54.462 | 1.81 |

[1] Time and speed-up are calculated with clock frequency of 205MHz, which is of maximum frequency from the estimated 24-bit modified FloRA chip implementation. Since the modified 24-bit FloRA has only synthesis result, the clock frequency of the modified 24-bit FloRA has been estimated with the synthesis result and the fabrication information of 16-bitFloRA.

same sizes of image than XPP, the actual execution time is about 80% faster than XPP, since the clock frequency of FloRA is 3-4 times higher than that of XPP. Besides, XPP needs much larger area than FloRA as shown in Table 5.2. There are two reasons why the area of XPP is larger than that of FloRA. First, an ALU-PAE in XPP has three ALUs - one supporting all possible operations and the other two supporting simple operations, whereas a PE in FloRA has only one ALU. This allows XPP to execute the application with less number of clock cycles, but causes large area cost.

Secondly, each ALU-PAE has its own multiplier, which also allows XPP to execute the application with less number of clock cycles, but causes large area cost as well as lower clock frequency. In the chip implementation of XPP, there are only reconfigurable computing modules such as 64 ALU-PAEs and some Function-PAE, memory modules such as 16 RAM-PAEs (for storing data) and configuration cache memory blocks, JTAG debug interface and I/O interfaces, but not the whole architecture including large memory blocks [38]. It is the same as the case of FloRA chip implementation, which contains a RISC processor, RCM with local memory blocks, peripherals and interface for external memory. The fact that chip size of XPP is more than two times larger than that of FloRA shows the area efficiency of FloRA. If the number of PEs in the proposed architecture increases to fit the area of XPP chip implementation (from 16.8 mm2 to 35.1 mm2), the estimated speed-up of the proposed architecture can be up to 3.76 times of XPP architecture considering that the target application kernels are parallelizable according to the number of PEs.

## 5.3 Comparison with Butter Architecture

Butter [25] is a CGRA implemented in a similar way to ours for manipulating floating-point numbers. A floating-point unit in the Butter architecture is implemented using existing integer units in a PE. The mantissa part of a floating-point number is calculated by the integer units, while the remaining parts - ex-

ponent and sign parts – are treated by special modules added for floating-point operations.

Since the Butter architecture was implemented on an FPGA, we have also implemented FloRA targeting the same FPGA for a fair comparison. FloRA was synthesized by Altera Quartus II for Altera Stratix II EP2S180 FPGA chip, which was also used for implementing the Butter architecture in [25]. Since the Butter architecture can execute three floating-point operations: addition, subtraction, and multiplication, we have implemented FloRA that can execute floating-point addition, subtraction, and multiplication only, without dividers, square-root units, and corresponding control logic. Shared multipliers in FloRA were implemented with DSP blocks in FPGA. Bit-widths of the multipliers of both architectures are different, and the numbers of DSPs for constructing multipliers are also different. For fair comparison, the number of DSPs is converted as the numbers of ALUTs, which will be explained later in this section.

Table 5.4 shows the area and the clock frequency of a PE in the three different architectures implemented on the FPGA: Butter architecture and two different instances of FloRA. FloRA_1Mul in Table 6 has one column of integer multipliers, each of which is shared by one row of PEs. FloRA_2Mul has two columns of integer multipliers; in other words, PEs in a row share two multipliers in FloRA_2Mul. FloRA_2Mul has better performance with more area cost by reducing resource conflicts, in general.

Table 5.4 Comparison of hardware features of PEs in different architectures

| Features of a PE | | Butter | FloRA_1Mul[1] | FloRA_2Mul |
|---|---|---|---|---|
| Comb. Area(ALUTs) | (a)Comb. | 1456 | 1430.33 | 1487.66 |
| | (b)DSPs | 1728 (12 DSPs) | 83.75 (1 DSPs) | 167.5 (2 DSPs) |
| | (a)+(b) | 3184 | 1514.08 | 1655.16 |
| Registers (Bits) | | 251 | 253 | 253 |
| Frequency (MHz) | | 34 | 68.0 | 62.1 |

[1] FloRA_1Mul has one shared integer multiplier per row, while FloRA_2Mul has two shared integer multipliers per row. The area of each PE is obtained by dividing the area of the entire PE array including shared modules by the number of PEs.

PEs in the Butter architecture use similar number of ALUTs but utilize much greater number of DSP blocks compared to the PEs in FloRA. For the purpose of comparison, we need to obtain the equivalent ALUT count for a DSP block, which depends on the bit-width of the module. The Butter architecture utilizes 12 DSP blocks per PE and among them eight DSP blocks are used to implement four 16x16 multipliers [26], each of which is made of 2 DSP blocks and is equivalent to 288 ALUTs.

The usage of the other four DSP blocks is not described clearly, so we assume that the equivalent ALUT count for a DSP block is 144. FloRA utilizes 25x25 multipliers, which can be implemented with either eight DSP blocks or

670 ALUTs. The effective ALUT counts for the DSP blocks in Butter architecture and FloRA are shown in Table 5.4. According to the sizes of PEs in the different architectures, the total area costs in terms of number of ALUTs are similar to each other (note that the number of PEs in Butter is a half of that in FloRA).

Table 5.5 shows the performance of several benchmark programs running on Butter architecture and two different versions of FloRA. There are two types of benchmark programs: one for processing integer data and the other for processing floating-point data. Latencies for the benchmark programs running on Butter architecture except for Itrans+dequant are estimated on the basis of the internal structure and interconnection topology of the Butter architecture explained in [25] and [27]. Data transfer overhead between local memory in the Butter architecture and outer memory is not considered for those benchmarks. However, the results of Itrans+dequant, which are from [25], include data transfer overhead. The performance of the kernel (Itrans+dequant) running on FloRA also considers the overhead, of course. Configurations of FIR filter (12-tap) kernel is divided into several sub-kernels, each of which requires reconfiguring the entire PE array. Within each sub-kernel, a data chunk is processed in a pipelined manner. However, once a chunk of data is processed by a sub-kernel, the output data chunk should wait until the PE array is reconfigured for the next sub-kernel. While the floating-point kernels do not show good performance on FloRA compared to the Butter architecture, the integer kernels show

Table 5.5 Comparison of performance between Butter architecture and FloRA

| Kernel name | Type | Butter (34MHz) | | FloRA_1Mul (68MHz) | | | FloRA_2Mul (62MHz) | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Latency (cycles) | Throughput (Miter/s) | Latency (cycles) | Throughput (Miter/s) | Ratio | Latency (cycles) | Throughput (Miter/s) | Ratio |
| Complex multiplication | int | 2 | 68.00 | 8 | 135.68 | 2.00 | 4 | 247.15 | 3.63 |
| Dot product(1 × 4) | int | 3 | 68.00 | 8 | 135.68 | 2.00 | 5 | 197.93 | 2.91 |
| Itrans+Dequant[1] | int | 354.2 | 0.10 | 145 | 0.45 | 4.67 | 145 | 0.42 | 4.38 |
| Complex multiplication | float | 2 | 68.00 | 13 | 38.40 | 0.56 | 13 | 35.07 | 0.52 |
| Cross product(1 × 3) | float | 2 | 34.00 | 24 | 22.51 | 0.66 | 22 | 22.41 | 0.66 |
| Dot product(1 × 4) | float | 3 | 68.0 | 021 | 40.92 | 0.60 | 21 | 37.37 | 0.55 |
| FIR filter(12-tap)[2] | float | 14 | 33.75 | 155 | 14.03 | 0.42 | 153 | 12.98 | 0.38 |

[1] Results of Itrans+Dequant kernel include data-transfer overhead.

[2] FIR filter (12-tap) kernel is too large to be mapped on a 4×8 array in Butter architecture, and thus the kernel is partitioned into several sub-kernels, each of which has a different configuration. The Latency value is just a summation of latencies of all the contexts and the Throughput value is obtained by running each sub-kernel in a pipelined manner.

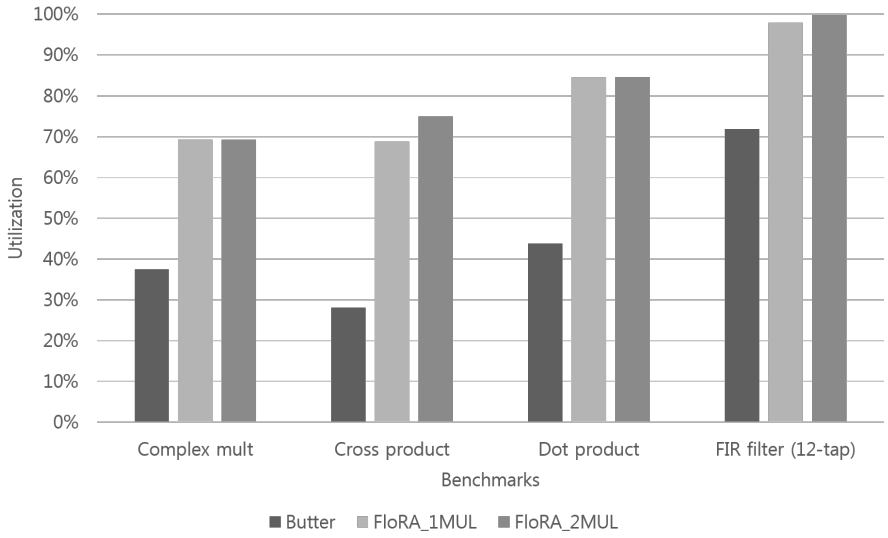


Figure 5.1 Comparison of the utilizations of the architectures for the floating-point kernels.

much higher performance on FloRA. It is mainly because the clock frequency of FloRA can be made much higher than that of Butter due to the difference in implementing floating-point operations. Floating-point operations in the Butter architecture take one clock cycle, while those in FloRA take several cycles. With one-cycle floating-point operations, latencies (in number of cycles) of floating-point kernels can be reduced, but the clock frequency decreases. On the other hand, FloRA allows multi-cycle floating-point operations to increase the clock frequency. Higher clock frequency has a merit when executing integer kernels.

Figure 5.1 shows the utilizations of the Butter architecture and two different versions of FloRA for the floating-point benchmark kernels. Even though the latencies of the floating-point operations are much longer than those of the Butter architecture (2.5 to 4.4 times longer), the utilization of the proposed architectures are relatively higher than that of the Butter architecture so that the performance degradation of the proposed architectures are less (1.8 to 1.9 times longer). The difference of the utilization results from the different mapping styles between the Butter architecture and FloRA. In FloRA, the benchmark kernels are mapped in a temporal mapping style, where the configuration for PEs changes every cycle and the data are kept by PEs. It is similar to a processors executing a program. The behavior of a processor changes according to its operation every cycle while the data are stored in the register file. On the other hand, in the Butter architecture, the benchmark kernels are mapped in a spatial mapping style, where the configuration of PEs is fixed and the data flows along the data-paths constructed by the configuration. When using spatial mapping style, the size of the kernel has a decisive effect on the utilization of the architecture. For example, if the kernel size is too small to the size of the PE array the utilization may be low. If the kernel size is too large to the size of the PE array, the kernel could mapped inefficiently or not be mapped onto the PE array. Besides, the interconnection topology is restricted in the Butter architecture. An intermediate value can only flow the interconnection topology in the spatial

mapping, while it can be stored in a local register file in order to be used later in the temporal mapping. A data register in a local register file in the temporal mapping is regarded as a vertical (in time-axis) global bus interconnect in the spatial mapping since a value in the register file can be used whenever it is needed after it is stored.

Additionally, there is an implication of the one-cycle floating-point operations implemented in the Butter architecture. Floating-point operations such as division, square-root operation and so on cannot be easily implemented in the Butter architecture since the area overhead and the critical path delay will be too large to make those operations execute in one cycle. FloRA allows shared and pipelined modules [6] so that such floating-point operations can be implemented without degrading the clock frequency or incurring too much area overhead.

## 5.4 Implication of the proposed architecture

As described in the above sections, one of the main differences of the proposed architecture from the other architectures is that the floating-point operations are implemented as multi-cycle operations. Execution times of multi-cycle floating-point operations can be longer than those of single cycle floating-point operations due to overheads including register load/store delays. Implementing single cycled floating-point operations, however, results in significant in-

crease in the critical path delay and decrease in the maximum clock frequency. Moreover, different floating-point operations can have very different delays and packing them within a cycle may impose unnecessary delay on the floating-point operations with shorter execution delay. Besides, simple integer operations are not executed efficiently since they are performed at lower clock frequency. On the other hand, multi-cycle floating-point operations proposed in our approach take different execution cycles according to their own execution delay. Short and simple data-path of the proposed approach achieves much higher clock frequency and thus single-cycle integer operations can be executed much faster.

The performances of FloRA is higher than those of Butter-like architecture when integer benchmark kernels are executed while the performances of FloRA is lower than those of Butter architecture when floating-point benchmark kernels are executed as described in the above section. Thus the proposed architecture can achieve the good results according to the ratio between integer computation and floating-point computation. From the Table 5.5, the average speed-up for the integer kernels and the floating-point kernels of FloRA_1MUL are 2.89 and 0.56, respectively. Those of FloRA_2MUL are 3.64 and 0.53, respectively. With this information, we can estimate the performance gain of FloRA compared to the Butter architecture for any application where integer kernels and floating-point kernels are contained together.

Let's assume there is an application which contains both integer kernels and floating-point kernels together. The computation of all the integer kernels is $C_i$ and the computation of all the floating-point kernels is $C_f$, then the ratio of the integer computation to the floating-point computation $R_C$ is:

$$R_C = \frac{C_i}{C_f} \tag{5.1}$$

The computation is defined as the number of instructions executed in kernels. Thus integer computation of an application means the number of integer instructions executed in the kernel loops in the application, which are executed on the accelerator such as FloRA or the Butter architecture. The total execution time of the integer kernels and floating-point kernels ($T_{Butter}$) in the Butter-like architecture is:

$$T_{Butter} = T_{Butter,i} + T_{Butter,f} = \alpha_i C_i + \alpha_f C_f \tag{5.2}$$

where $\alpha_i$ and $\alpha_f$ are the coefficient for the integer kernels and the floating-point kernels of the Butter-like architecture, respectively. By the way, the latencies of the integer operations and the floating-point operations are the same, the total computation time coefficient $\alpha_i$ and $\alpha_f$ are the same and the Equation 5.2 is simplified as below:

$$T_{Butter} = \alpha(C_i + C_f) \tag{5.3}$$

The total execution time of the integer kernels and the floating-point kernels ($T_{FloRA}$) in the FloRA is:

$$T_{FloRA} = T_{FloRA,i} + T_{FloRA,f} = \beta_i C_i + \beta_f C_f \tag{5.4}$$

where $\beta_i$ and $\beta_f$ are the coefficient for the integer kernels and the floating-point kernels of FloRA, respectively. FloRA is more beneficial than the Butter-like architecture if the inequality satisfies:

$$T_{FloRA} < T_{Butter} \tag{5.5}$$

Equation 5.5 is derived as follows:

$$\begin{aligned}
& T_{FloRA} && < && T_{Butter} \\
\Rightarrow \quad & \beta_i C_i + \beta_f C_f && < && \alpha(C_i + C_f) \\
\Rightarrow \quad & \gamma_i C_i + \gamma_f C_f && < && C_i + C_f \\
\Rightarrow \quad & \gamma_i R_C + \gamma_f && < && R_C + 1 \\
\Rightarrow \quad & \gamma_f - 1 && < && R_C(1 - \gamma_i)
\end{aligned} \tag{5.6}$$

where $\gamma_i$ and $\gamma_f$ are $\frac{\beta_i}{\alpha}$ and $\frac{\beta_f}{\alpha}$, respectively. $\gamma_i$ and $\gamma_f$ are actually the ratio of the execution time of FloRA to that of Butter-like architecture for integer kernels and for floating-point kernels, respectively, since:

Table 5.6 Minimum ratio of the integer computation to the floating-point computation of applications which is better to be executed on FloRA

| | FloRA_1MUL | FloRA_2MUL |
|---|---|---|
| $R_C$ | 1.201 | 1.223 |
| Portion of integer computation | 54.5% | 55.0% |

$$\frac{T_{FloRA,i}}{T_{Butter,i}} = \frac{\beta_i C_i}{\alpha C_i} = \frac{\beta_i}{\alpha} = \gamma_i = \frac{1}{Speedup_i} \tag{5.7}$$

$$\frac{T_{FloRA,f}}{T_{Butter,f}} = \frac{\beta_f C_f}{\alpha C_f} = \frac{\beta_f}{\alpha} = \gamma_f = \frac{1}{Speedup_f} \tag{5.8}$$

From Equation 5.6, 5.7 and 5.8 with the speed-up values of FloRA_1MUL and FloRA_2MUL to the Butter architecture, we can obtain minimum $R_C$ as shown in Table 5.6. It means that any application satisfying that the ratio of the integer computation to the floating-point computation of the application greater than the value is better to be executed on FloRA. With this information, we can obtain the lower-bound of the percentage of the integer computation of the whole kernel computation for Flora_1MUL and FloRA_2MUL as 54.5% and 55.0%, respectively.

# Chapter 6

# Enhancement Techniques

## 6.1 Introduction

FloRA, the architecture proposed in the above chapters, can execute floating-point operations as well as integer operation in the same architecture as the floating-point operations are divided into several integer micro-operations and they are calculated by the integer functional units in PEs. One of the most important features of the floating-point operations implemented on the proposed architecture is they are multi-cycled. It allows the clock frequency of the architecture high so that the integer operations can run faster than other architectures supporting both integer and floating-point operations.

The execution delays of the floating-point operations in FloRA, However, are somewhat longer than those of the floating-point units even though the

clock frequency of FloRA is high. It is because that the integer micro-operations divided from a floating-point operation are data-dependent to each other. It occurs unbalance of the utilizations of between mantissa PE and exponent PE so that the latency of the floating-point operations cannot be decreased enough. In order to release the problem, two enhancement techniques – overlapping and forwarding between two floating-point operations – are proposed.

Above all, the conventional floating-point operations in the base architecture are introduced. Then the two enhancement techniques are proposed. Lastly, the experimental results show the efficiencies of the techniques.

## 6.2 Conventional Approach

### 6.2.1 Base Architecture

Since each PE has only integer functional units inside in the base architecture, a couple of PEs can cooperate in order to execute floating-point operations [10]. A floating-point operations are divided into a bunch of integer operations and they are executed on a pair of PEs, named an FPU-PE cluster. Specifically, floating-point numbers to be calculated are divided into two parts: signed mantissa part and exponent part, then one PE (mantissa PE: $PE_M$) in a FPU-PE cluster manipulates mantissa parts while the other PE (exponent PE: $PE_E$) operates exponent parts. PEs in the cluster store the intermediate data in their local registers and communicate with each other via network interconnection be-

tween them while executing floating-point operations. Each PE has a finite state machine (FSM) for configuring the behavior of the data-path during multi-cycle floating-point operations rather than fetching configuration bits from the configuration memory every cycle. The floating-point operations implemented in this work include floating-point addition (FADD), subtraction (FSUB) and multiplication (FMUL). The latencies of FADD and FSUB are 8 cycles, while that of FMUL is 4 cycles. In FMUL operation, the mantissa PE multiplies two floating-point mantissa parts using integer multiplier module, which is shared by all the PEs in a row for reducing area cost [18].

### 6.2.2 Utilization of Floating-Point Operations

Figure 6.1 shows the utilization of the integer functional units in a FPU-PPE cluster while a floating-point operation is executed. Colored boxes indicate which functional unit is used in which cycle. In the case that there are no colored boxes at a certain cycle in a PE, then just dedicated logic in the PE rather than functional units are used or the PE is idle at the cycle. In Figure 6.1(a), there are no colored boxes on the PE_man in the second cycle (cycle 1). At this cycle, the mantissa PE selects the least of the two mantissa and store a certain storage so that it can be aligned next cycle. It is done by the dedicated logic in the mantissa PE but it is not shown in the figure.

We can easily define that a PE is utilized at a certain cycle by one or more

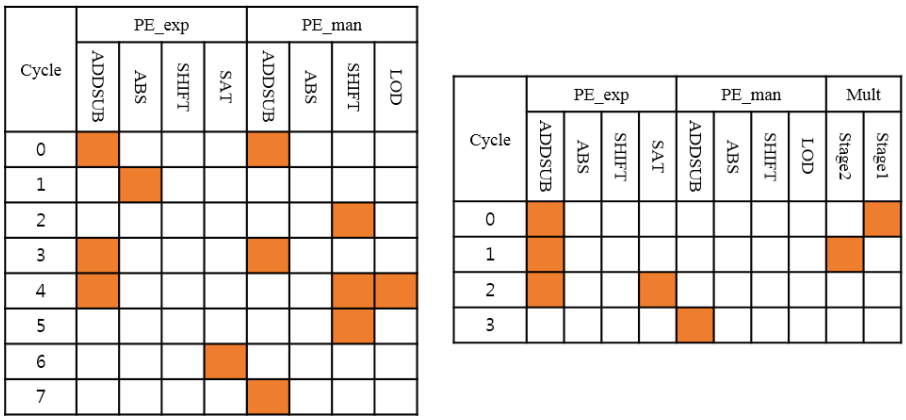


Figure 6.1 Utilization of integer functional units in an FPU-PE cluster. (a) is for an FADD/FSUB operation, while (b) is for FMUL operation. Colored box indicates that the functional unit is utilized at the cycle.

functional units in the PE are utilized at the cycle. In other words, the dedicated logic rather than functional units are ignored. With this definition, we can obtain the utilization of the PEs in a FPU-PE cluster during floating-point operations as shown in Table 6.1.

## 6.3 Proposed Enhancement Techniques

### 6.3.1 Overlapping Technique

Although using integer functional units in order to perform floating-point operations increases utility of the architecture, the long latencies of the floating-

Table 6.1 Utilization of PEs in a FPU-PE cluster

| Operation | Exponent PE | Mantissa PE | Shared multiplier |
|---|---|---|---|
| FADD/FSUB | 62.5% | 75% | - |
| FMUL | 75% | 25% | 50% |

point operations limit the performance of floating-point applications (since the base architecture is optimized for executing various single cycle integer operations, scheduling integer operations split from a floating-point operation requires unnecessary idle cycles due to extra mux and functional unit delays). In order to mitigate it, a technique to share functional units that are temporarily freed during a floating-point operation with another floating-point operation is proposed in this section. Simply, it is overlapping two floating-point operations with different operands by sharing functional units between the two floating-point operations. That can be done with minimal overhead of additional control logic and registers. Such overlapping of two different floating-point operations improves performance by enhancing the utilization of functional units when multiple independent floating-point operations are to be executed.

During the execution of floating-point operations, each PE employs its functional units to manipulate the input values, a temporal register to keep the intermediate value, and output register to communicate with each other. Since all

the functional units in each PE are not used every cycle during execution of a floating-point operation, they can be used for another floating-point operation. Unlike the functional units, which are frequently freed due to data dependency, two specific registers are utilized almost every cycle so that they cannot be shared with other floating-point operations. This problem is solved by inserting an additional register with dedicated communication channel for communication in an FPU-PE cluster for another floating-point operation to be overlapped. For intermediate values, there is no need of inserting additional register, but utilizing one of the registers in the local register file in the PE works. Thus just dedicated interconnects to store intermediate value to a register in the register file are added. For the simplicity of the FSM in each PE, time difference (overlap) between two floating-point operations is fixed as a half of the latency of the operation, i.e., 4 cycles for FADD, and 2 cycles for FMUL.

Since the overlapping technique cannot be applied to two operations that have data-dependency between them, we expect it to be mainly applied in unrolled iterations of kernels. In general, however, FADD and FMUL in a kernel program are likely to have data dependency, and thus overlapping FADD and FMUL is not considered in this thesis. Figure 6.2 shows the usage of functional units of PEs in an FPU-PE cluster for an FADD operation. When another FADD operation starts four cycles later, there are resource conflicts at two different positions shown as blended-colored box in Figure 6.2(b). There are two

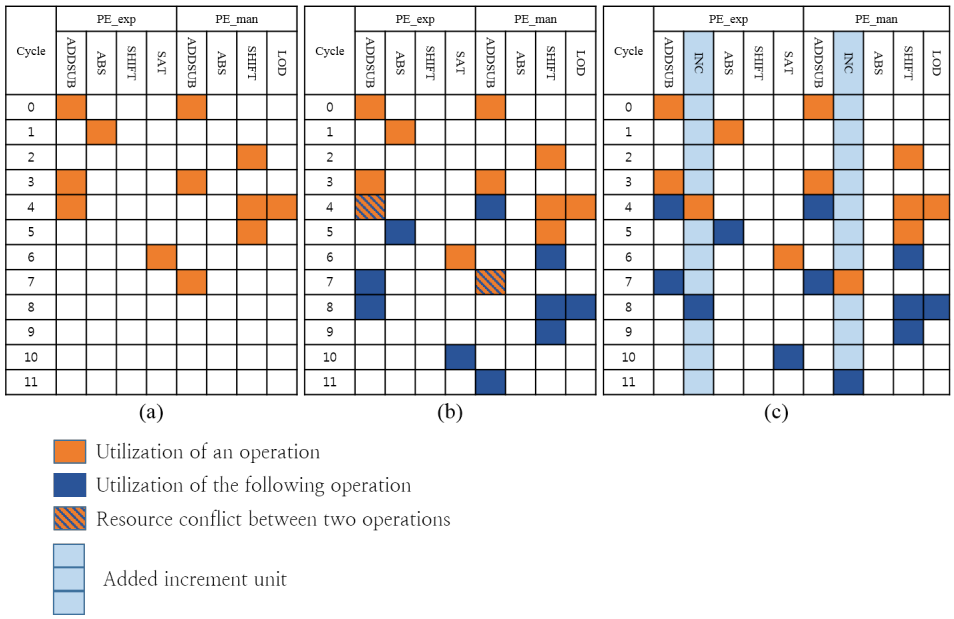


Figure 6.2 Utilization of integer functional units in an FPU-PE cluster for a FADD operation: (a) for an operation, (b) for two overlapped operations without additional functional unit, (c) for two overlapped operations with functional unit.

alternative ways of avoiding the conflict: duplication of the conflicted functional unit, or executing the following floating-point operation one cycle later. In fact, both of the two conflicts are for incrementing the input value by 1 for rounding. Thus inserting an increment unit instead of another add/subtract unit is good enough for overlapping two floating-point operations with 4 cycles of time difference (Figure 6.2(c)).

On the other hand, there is one conflict on add/subtract unit in an expo-

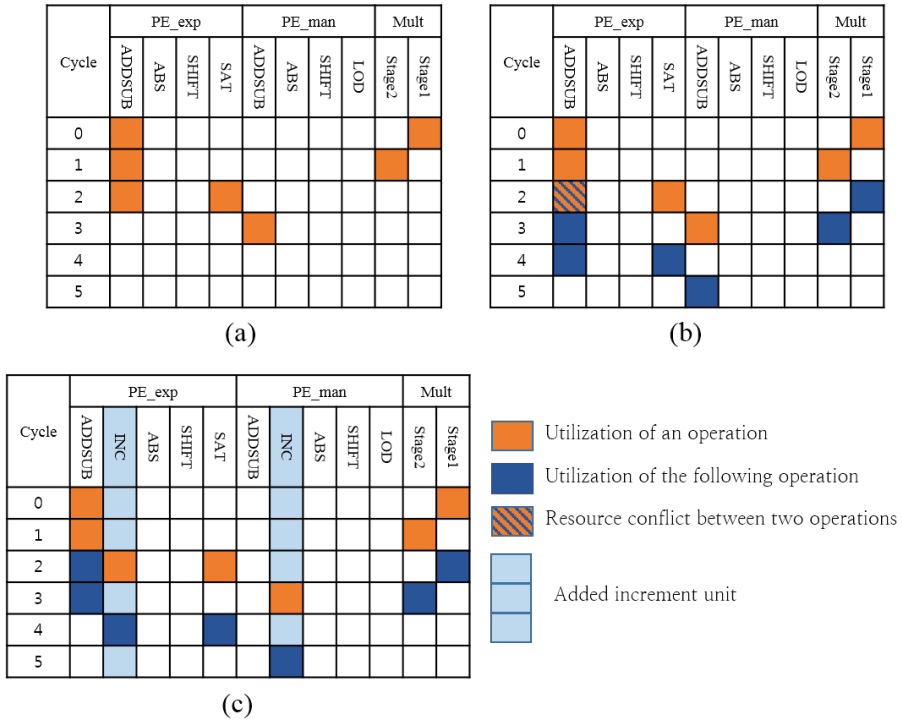


Figure 6.3 Utilization of integer functional units in an FPU-PE cluster for a FMUL operation: (a) for an operation, (b) for two overlapped operations without additional functional unit, (c) for two overlapped operations with functional unit.

nent PE when overlapping two 4-cycle FMUL operations with 2 cycles of time difference as shown in Figure 6.3(b). The conflict is also due to a rounding operation, and thus the increment unit inserted for the conflict on FADD is utilized. Shared multiplier is pipelined so that it is not overlapped unless the two FMUL operations are mapped at the same cycle. Same to the case of FADD, two FMUL operations are overlapped with 2 cycles of time difference with additional increment unit as shown in Figure 6.3(c).

The FSM is also changed to support overlapping two floating-point operations. Since the cycle difference is fixed, there is no need of parallelizing FSMs but only merging states of the two overlapping floating-point operations is needed.

### 6.3.2 Forwarding Technique

Floating-point implementation on the target architecture enables floating-point operations as well as integer operations on the same architecture. However the latency of the floating-point operations are longer than that of stand-alone floating-point units since the architecture is basically optimized for rapid integer operations instead of short latencies of floating-point operations.

Main idea of the forwarding between two floating-point operations is skipping unnecessary functions of the first operation and send the intermediate data directly to one of the inputs of the second operation. Those functions are formatting and exception checking. Formatting is not necessary to calculate, but

exceptions – especially, overflow and underflow – can be effect the result. Thus behavior of the second operation should be changed in order to consider those effect.

There are four different cases: (1) the following operation is FADD/FSUB, and (2) the following operation is FMUL.

### Case 1. FADD/FSUB follows floating-point operations

In the last two cycles of the leading operation, the exponent PE checks saturation of the exponent result and the mantissa PE check if the mantissa value should be rounded. Thus the exponent PE should consider whether the forwarded exponent value is overflowed or not when comparing two operands at the first cycle in the following operation and the add 1 to the forwarded value if needed (refer to the dual-colored box at cycle B1 in Figure 6.5(a)). The mantissa part should consider the round bit when comparing two operands and storing them to the temporal registers. Consequently, additional work in the following FADD operation is summarized into two types of works: (1) storing the rounded forwarded value, and (2) comparing of two input values considering round bit. The former can be solved by increment units in the PEs. The new forwarded value $Fwd_{new}$ can be obtained by:

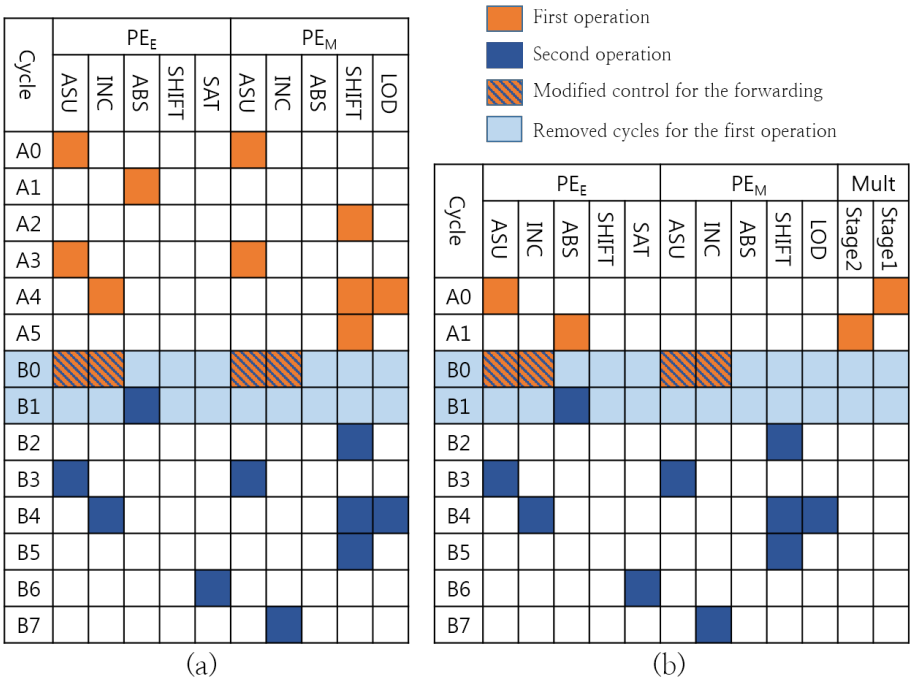


Figure 6.4 Utilization of integer functional units in an FPU–PE cluster in the case that the output of the leading operation is forwarded: (a) from FADD to FADD, and (b) from FMUL to FADD.

$$Fwd_{new} = \begin{cases} Fwd, & if round = 0 \\ Fwd + 1 & if round = 1 \end{cases} \Rightarrow Fwd + round \qquad (6.1)$$

Where $Fwd$ is the forwarded value from the leading operation, $Fwd_{new}$ is the value to be stored in the temporary register in the following operation, and *round* is a flag if the forwarded value should be rounded or not. It is calculated by increment units in the PEs.

The latter can be solved using carry-in port of add/subtract unit. When comparing two values, we subtract a value from the other value with adjusting carry-in properly:

$$\begin{aligned} In - Fwd_{new} &= In - Fwd - round \\ &= In + (\neg Fwd + 1) - round \\ &= In + \neg Fwd + (1 - round) \\ &= In + \neg Fwd + \neg round \end{aligned} \tag{6.2}$$

Where $In$ is the other input value of the following operation. $(1 - round)$ is 1 if the round flag is zero, otherwise 0 so that the problem of considering round flag in the case of comparing is solved by setting carry-in as $\neg round$.

Exception is checked by the exponent PE first then the information is propagated to the mantissa PE. The flag is generated and stored in the exponent PE and then the exponent PE send signal to cancel the calculation of the mantissa part at the end of the following operation.

### Case 2. FMUL follows floating-point operations

In the case that the following operation is FMUL, round flag in the exponent PE can be handled easily by assigning to the carry-in at the first stage when the two exponent operands are added as shown in Equation 6.3.

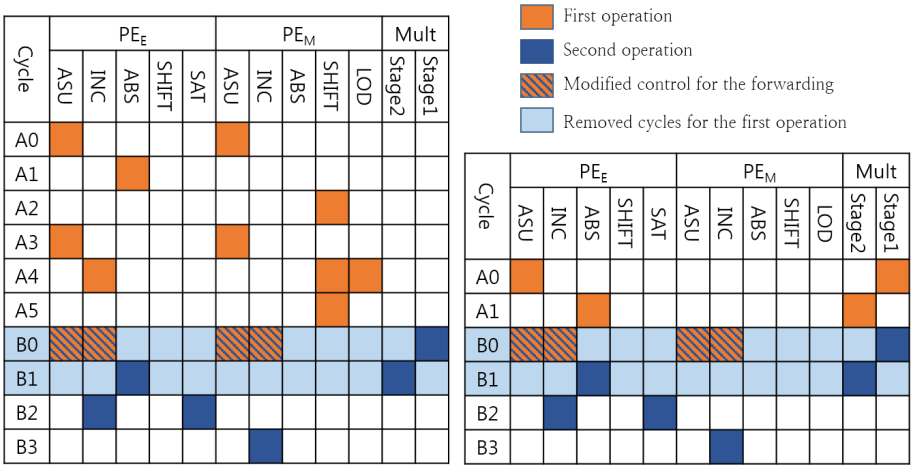


Figure 6.5 Utilization of integer functional units in an FPU-PE cluster in the case that the output of the leading operation is forwarded: (a) from FADD to FMUL, and (b) from FMUL to FMUL.

$$\begin{aligned} Exp_{new} &= In + Fwd_{new} \\ &= In + Fwd + round \end{aligned} \tag{6.3}$$

Rounding in the mantissa part is delayed to the end of the following FMUL operation. It implies that the error caused by forwarding FMUL could be more than the IEEE standard.

$$(Fwd_M + round) \times In = Fwd_M \times In_M + round \times In_M$$

$$\sim \quad Fwd_M \times In_M + round$$

$$(\because 1.0 <= In_M < 2.0) \tag{6.4}$$

Where $Fwd_M$ and $In_M$ are the mantissa value of the forwarded floating-point number and the other input number, respectively. Equation 6.4 shows the approximation of the multiplication of forwarded value with round flag and normal input value.

The exception checking is the same to the FADD.

## 6.4 Experiments

### 6.4.1 Performance Comparison

Figure 6.6 shows the performance of the simple functions running on the existing architecture and the architecture where the proposed approaches are applied. Employed benchmark functions are basic arithmetic functions, which are frequently used in signal-processing, multimedia applications, 3D graphics, and other mathematical algorithms. Normalized throughput shown in Figure 6.6 is throughput ratio of the proposed architecture to the base architecture for each function. By applying the proposed approach of sharing functional units between two floating-point operations, we obtain up to 42.9% better performance and 27.5% better performance on average compared to the base architecture in the case that the overlapping technique is applied, and 33.9% better performance on average in the case that both overlapping and forwarding techniques

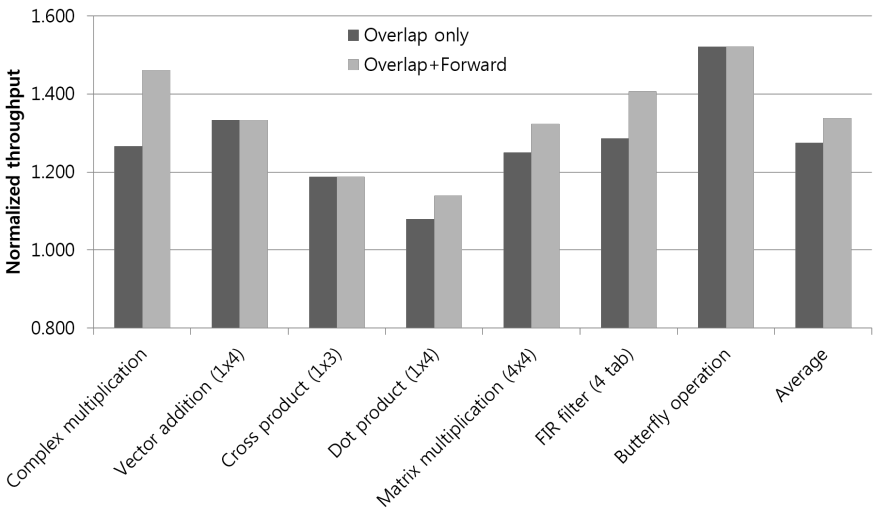


Figure 6.6 Normalized throughputs of the architectures where the enhancement technieuqs are applied compared to the base architecture.

are applied.

### 6.4.2 Hardware Cost of the Proposed Techniques

Both of the proposed architecture and the existing architecture are synthesized by Synopsys Design Compiler with TSMC 130nm technology library. Each architecture has 8x4 array of PEs, one column of shared integer multipliers (i.e., eight multipliers), and configuration and data memory blocks. Synthesis result shows that the area overhead caused by the proposed approach is about 13.8% (for only overlapping technique applied), and 20.9% (for both overlapping and forwarding techniques applied) at the same clock speed of 330MHz. They are

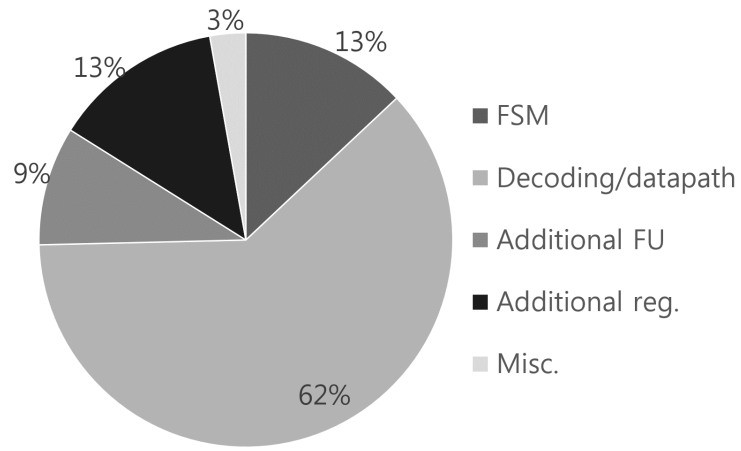


Figure 6.7 Area breakdown of the additional hardware for applying overlapping technique to the base architecture where any enhancement techniques are applied.

only about 2.9% and 4.2% of the whole RCM with configuration and data memory blocks for only overlapping technique applied and both techniques applied, respectively. The area overhead is from FSM and decoder on the control path as well as additional registers and increment units.

Figure 6.7 shows the breakdown of the additional hardware area of the architecture where the overlapping technique is applied compared to the base architecture where any enhancement techniques are not applied. Most of the increment is from the modification of the decoding logic and data-paths. Since

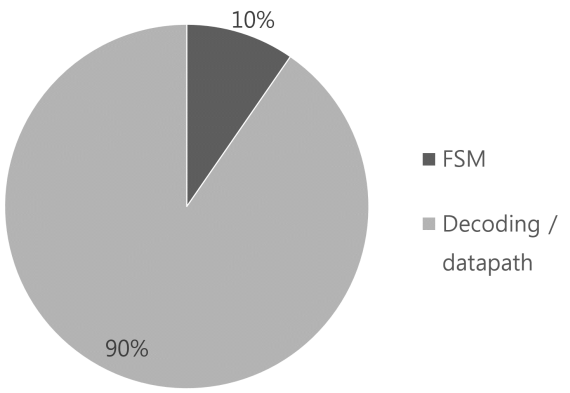


Figure 6.8 Area breakdown of the additional hardware for applying forwarding technique to the architecture where the overlapping technique is applied.

the decoder in each PE should control the behaviors of the two different operations at the same time, it becomes more complicated. Furthermore, it should control the additional hardware such as increment units and registers. It causes the data-paths in each PE modified. Additional functional units and registers are not that much compared to the modification of the control-paths. FSMs are changed for describing the overlapping states.

Figure 6.8 shows the breakdown of the additional hardware area of the architecture where both the forwarding and overlapping techniques are applied compared to the architecture where only the overlapping technique is applied. There is neither additional functional units nor registers to support forwarding technique so that the additional hardware area is from modification of the FSM,

the decoding logics, and the corresponding data-paths.

To consider both performance and area cost at the same time, we simply introduce a term, gain, as throughput ratio divided by area ratio. The gain of the application of only overlapping technique and both techniques to the base architecture is 1.12 and 1.11, which indicates about 12% and 11% improvement over the base architecture, respectively.

### 6.4.3 Utilization Enhancement by the Proposed Techniques

The proposed techniques can enhance the utilization of the PEs in the PE array by sharing a PE for two different floating-point operations at a time. This subsection shows how the utilizations are increased by applying those enhancement techniques. In order to figure out the improvements in terms of utilization, I redefine utilization of PEs as below:

- A PE is utilized in the case that one or more functional units in the PE are utilized.
- A PE is not utilized in the case that the PE is not executing any operations.
- A PE is not utilized in the case that simple dedicated logic in the PE is only utilized.

Floating-point operations are multi-PE and multi-cycled operations and PEs are not busy all the time during whole execution cycles of the floating-point

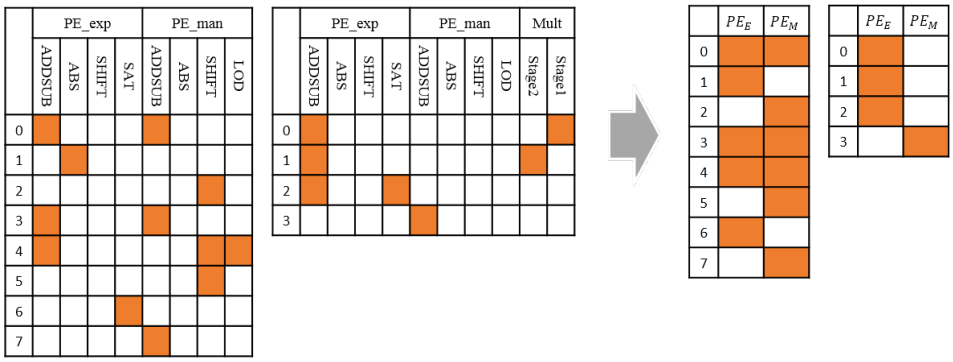


Figure 6.9 Utilizations of functional units of PEs during floating-point operations and their redefined utilizations.

operations. The third item in the above list provides a criteria for utilization of PEs during floating-point operations. With this definition, the utilizations of the PEs while PEs are executing floating-point addition, subtraction and multiplication are shown in Figure 6.9.

Figure 6.10 shows the utilization of PEs where the enhancement techniques are applied. The left-side diagram is for the overlapping techniques, and the right-side diagram is for the forwarding techniques. For the case of overlapping, overlapped region are fully utilized so that the overlapping method increases the utilization of PEs effectively. On the other hand in the cases of forwarding, the utilization in the overlapped region depends on the type of the following operation. In the case that the following operation is addition/subtraction, the PEs in the overlapped region are fully utilized as shown in the figure. But there are

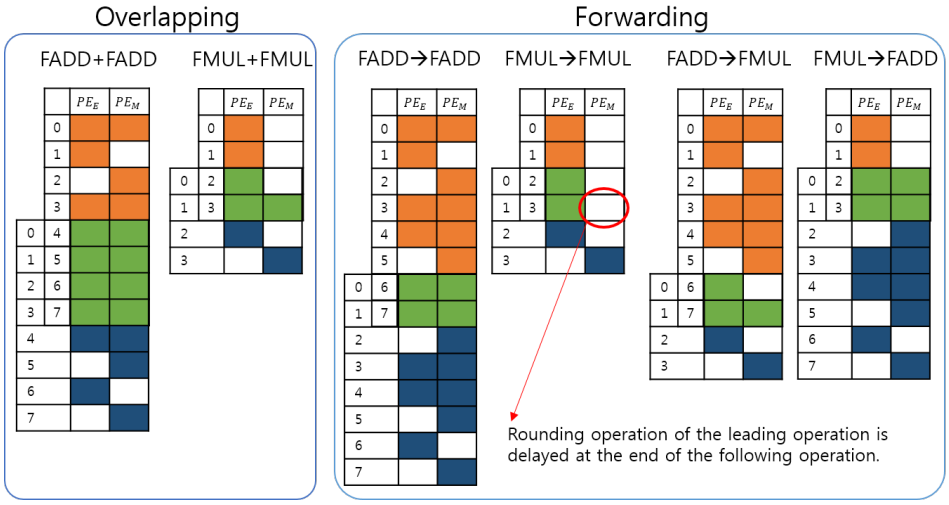


Figure 6.10 Utilizations of functional units of PEs where the enhancement techniques are applied.

idle PEs in other region which leads the utilization less compared to the case that the overlapping technique is applied. In case that the following operation is multiplication, the utilization of PEs in the overlapped region is not fully utilized. Besides the utilization decreases in case that the leading operation and the following operation are same to the multiplication, because the rounding step in the leading operation is delayed to the following operation.

### Utilization Enhancement by Overlapping Technique

Figure 6.11 shows the utilization trends where floating-point add/subtract operations are overlapped. Without overlapping, the utilization of the exponent

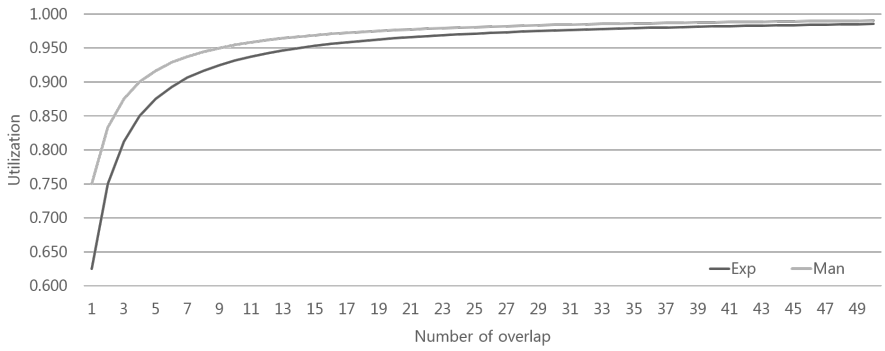


Figure 6.11 Utilization trends where FADD/FSUB are overlapped. "Exp" and "Man" stand for the exponent PE and the mantissa PE, respectively.

PE and the mantissa PE are 62.5% and 75%, respectively (refer to Table 6.1). As overlapping the operations more and more, the utilizations of the exponent PE and the mantissa PE are increasing up to 100% as shown in Figure 6.11. Overlapping technique makes the PEs fully utilized in the overlapped periods of the floating-point additions/subtractions.

Figure 6.12 shows the utilization trends where floating-point multiply operations are overlapped. In this case, the utilization of the mantissa PE is much lower than others because it does not manage the multiplication of the two mantissa input. Instead, it sends the input mantissa values to the corresponding shared integer multiplier and receives the result value from the multiplier. Since the utilization of the mantissa PE without overlapping is 25%, the utilization with overlapping is limited to 50% while the utilization of the exponent PE is

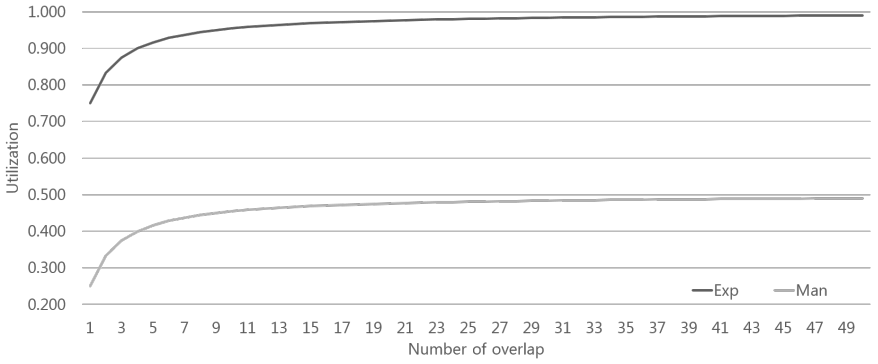


Figure 6.12 Utilization trends where FMUL are overlapped. "Exp" and "Man" stand for the exponent PE and the mantissa PE, respectively.

increasing up to 100%. Figure 6.12 shows the utilization of the mantissa PE is increasing up to 50%, which is the theoretical limit of the utilization in case that the overlapping technique – overlapping two data-independent floating-point operations – is applied.

### Utilization Enhancement by Forwarding Technique

The intention of the forwarding technique is reducing latency of the data-dependent floating-point operations, while that of the overlapping technique is sharing unused functional units of a PE for another floating-point operation. Thus the effectiveness of the forwarding technique is less than that of the overlapping technique in terms of utilization enhancement. The forwarding technique is partly helpful to increase the utilization of the PEs in the PE array.

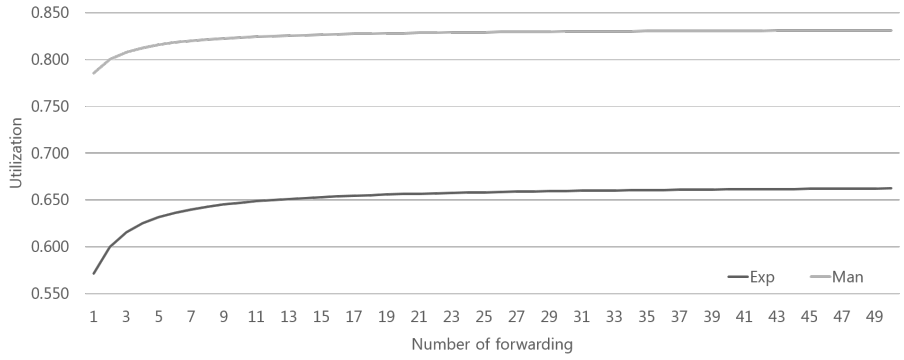


Figure 6.13 Utilization trends where FADD/FSUB are forwarded. "Exp" and "Man" stand for the exponent PE and the mantissa PE, respectively.

Figure 6.13 shows the utilization trends where the forwarding technique is applied to data-dependent floating-point add/subtract operations. The more the number of forwarding increases, the more the utilizations of both the exponent PE and the mantissa PE increases as shown in Figure 6.13. However the following operation does not cover the leading operation enough, the utilizations are limited to a certain level. The limits of the utilizations of the exponent PE and mantissa PE is 66.7% and 83.3%, respectively.

Figure 6.14 shows the utilization trends where the forwarding technique is applied to data-dependent floating-point multiply operations. The utilization of the exponent PE is increasing up to 100% while the utilization of the mantissa PE is decreasing down to 0% as the number of forwarding increases. In case of exponent PE, the following operation shares the idle functional units for the

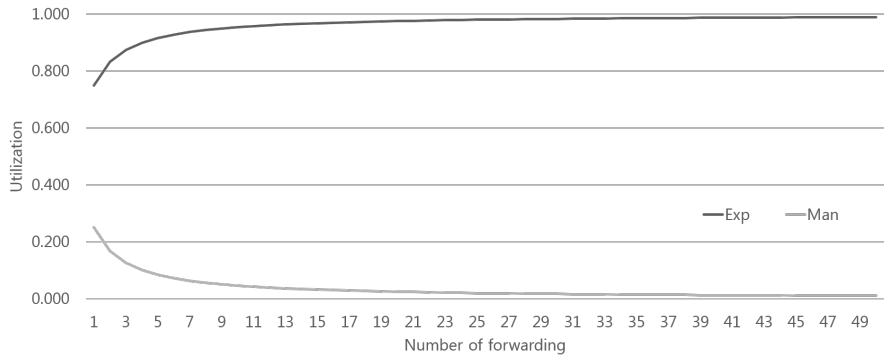


Figure 6.14 Utilization trends where FMUL are forwarded. "Exp" and "Man" stand for the exponent PE and the mantissa PE, respectively.

leading operation well so that the idle cycles are hidden. On the other hand, in the case of mantissa PE, following multiply operation postpones the rounding step of the leading operation to the end of the following operation. As a result the rounding steps of the leading operation and the following operation are combined so that the utilization is decreased.

Figure 6.15 shows the utilization trends where the forwarding technique is applied to data-dependent floating-point add/subtract and multiply operations. The add/subtract operations and multiply operations follow alternately. In the case of exponent PE, the direction of the utilization trends are the same but the limits are different. Thus the utilization of the exponent PE increases with fluctuation and converges to 75%, which is in the middle of the limits for the add/subtract operation and multiply operation as the number of forwarding in-

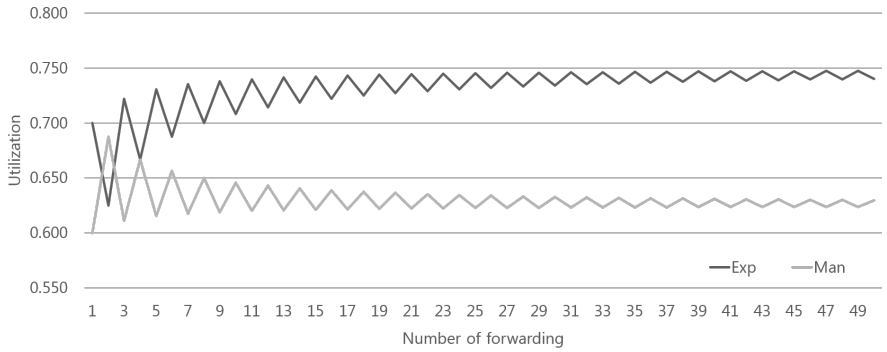


Figure 6.15 Utilization trends where FADD/FSUB and FMUL are forwarded alternately. "Exp" and "Man" stand for the exponent PE and the mantissa PE, respectively.

creases. In the case of mantissa PE, the utilization also fluctuate and converges to 62.5%, which is in the middle of the limits for the add/subtract operation and multiply operation, similar to the case of exponent PE. When the utilization fluctuates, it decreases where the following operation is multiply operation and increases where the following operation is add/subtract operation as the reason is explained in the above paragraph.

## 6.5 Comparison with Other Architecture

In order to explain the effects of the proposed enhancement techniques, I compared FloRA with those enhancement techniques to the Butter architecture. For the comparison, three different versions of FloRA is introduced. The first ver-

Table 6.2 Clock frequencies of different versions of FloRA

| FPGA type | FloRA(Base) | FloRA(Overlap only) | FloRA(Both) |
|---|---|---|---|
| Altera Stratix IV [39] | 94.5 MHz | 93.5 MHz | 91.4 MHz |
| Altera Stratix II [40] | 60.5 MHz | 59.8 MHz[1] | 58.5 MHz[1] |

[1] The clock frequencies of FloRA(Overlap only) and FloRA(Both) versions on Stratix II FPGA are estimated from the proportions of the clock frequencies of the different versions of FloRA on Stratix IV FPGA.

sion is the base architecture where no enhancement techniques are applied. It has 8 × 8 two-dimensional array of PE and two columns of shared integer multipliers, the latency of the floating-point addition/subtraction and multiplication are 8 cycles and 4 cycles, respectively. The second version is the architecture where the overlapping technique is only applied. The same types of data-independent floating-point operations can be overlapped in this version of FloRA. The third version is the architecture where both the overlapping and forwarding techniques are applied. Data-dependent floating-point operations are scheduled reducing 2-cycles of their total latency.

Though all the implementations of FloRA introduced in the above sections in this chapter had been synthesized for targeting ASICs, the architectures are modified and synthesized for targeting Altera FPGAs for the comparison with the Butter architecture. The Butter architecture was synthesized on a Stratix II

FPGA but the second and the third versions of FloRA cannot be synthesized on the FPGA chip because of lack of area even though the FPGA is the biggest one of the same 130nm technology. For fairness, the clock frequencies of the second and the third versions are obtained in order to compare with other architectures synthesized on the Stratix II FPGA by synthesizing three versions of FloRA – one the three can be synthesized on the Stratix II – for the larger FPGA and estimating the clock frequencies of the second and the third version on the Stratix II FPGA. Table 6.2 shows the real clock frequencies and the estimated clock frequencies of different versions of FloRA.

Figure 6.16 shows the normalized throughputs for several floating-point benchmark kernels of different architectures. The average utilization improvement of FloRA(Overlap only) from the FloRA(Base) is 3.04% and that of FloRA(Both) from the FloRA(Base) is 9.83%. The reason why the improvement by the overlapping technique is less than that of Figure 6.6 is related to the size of the PE array. PEs in a row in the PE array shares a bus connected to the input data memory. In the case that the number of columns is changed to 8 from 4, PEs should wait for 4 more cycles to access to the data memory again since the delay that all the columns of PEs in the PE array access to the data memory is also changed to 8 from 4. It hinders loop unrolling in that loop unrolling requires more bandwidth to the data memory, but the bottleneck of the input data bus occurs unnecessary delays for waiting input data from the data memory. Be-

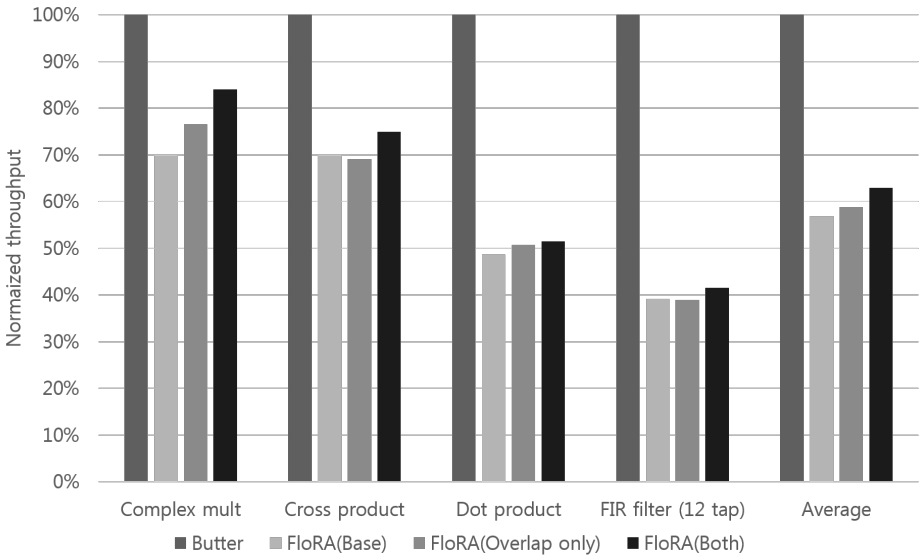


Figure 6.16 Comparison of the normalized throughputs for floating-point benchmark kernels. FloRA(base) is the base architecture, where the enhancement techniques are not applied, FloRA(overlap_only) is the architecture implementing the overlapping technique, and FloRA(Both) is the architecture implementing both overlapping and forwarding techniques.

sides, it barriers overlapping floating-point add/subtract operations at the end of the kernels, which is critical for short benchmark kernels whose execution cycles are about 20-30 cycles.

The lower-bound described in Section 5.4 could be found for the architectures implementing enhancement techniques. We can obtain the speedup of the floating-point kernels to the Butter architecture from Figure 6.16. We can

Table 6.3 Minimum ratio of the integer computation to the floating-point computation of applications which is better to be executed on FloRA with enhancement techniques applied

| | FloRA(Base) | FloRA(Overlap only) | FloRA(Both) |
|---|---|---|---|
| $R_C$ | 1.055 | 0.993 | 0.829 |
| LB int. comp.[1] | 51.3% | 49.8% | 45.3% |

[1] Lower bound of the portion of the integer kernel computation of the whole kernel computation.

calculate the speedup of the integer kernels to the Butter architecture easily since the integer operations in FloRA is not changed except for the clock frequency. The speedup of the integer kernels for FloRA(Overlap only) and FloRA(Both) are 3.51 and 3.342, respectively. Calculated ratio of the integer computation to the floating-point computation is as shown in Table 6.3. It implies that the enhancement techniques enlarges the favorable application domains.

# Chapter 7

# Conclusion

We have introduced a coarse-grained reconfigurable architecture supporting both integer operations and floating-point operations. A pair of integer processing elements called FPU-PE co-operate a floating-point operation for multiple cycles. FSM is included in each PE for reconfiguration for multiple cycles, and data-paths in PEs are modified to support various kinds of floating-point operations. By reusing most of the existing integer functional units for floating-point operations, we have achieved more flexibility with less hardware overhead.

We have compared the chip implementation of the proposed architecture with processor-based approaches. We have also compared the modified version of the proposed architecture with other architectures adopting similar approaches and described the benefits of the proposed approach.

This thesis also proposes an enhancement techniques to utilize integer functional units during execution of floating-point operations. By sharing functional units, we can support floating-point operation with minimal overhead. Especially the proposed overlapping technique enhances the utilization of functional units so that we can accelerate floating-point kernels effectively. Experimental results show that our approach has 11-12% better throughput-to-area ratio compared to the previous sharing schemes.

As a future work, sharing functional units among different types of floating-point operations, or operations with data dependency could be researched in order to reach further improvements. Various floating-point operations other than the already implemented ones including add, subtract, multiply, divide, and square-root can be implemented on the architecture. Or revised IEEE floating-point formats can be implemented on the proposed architecture.

Though the proposed enhancement techniques increases efficiency by hiding the idle slacks and removing unnecessary steps of floating-point operations, there are remaining idle slacks and inefficiency caused from the dependency of the micro-operations in a floating-point operation. It can be dealt as a future work. A possible direction with the problem is exploring design space of FPU-PE clusters. Inefficiency caused from the communication delays could be eliminated by combining the mantissa PE and the exponent PE more tightly. Another way is sharing data-paths of two FPU-PE clusters.

# Bibliography

[1] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine," in ACM SIGPLAN Notices, vol. 33, pp. 46--57, ACM, 1998.

[2] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications," Computers, IEEE Transactions on, vol. 49, no. 5, pp. 465--481, 2000.

[3] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Kressarray xplorer: A new cad environment to optimize reconfigurable datapath array architectures," in Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific, pp. 163--168, IEEE, 2000.

[4] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in Proceedings of the conference on Design, automation and test in Europe, pp. 642--649, IEEE Press, 2001.

[5] J. Becker, T. Pionteck, C. Habermann, and M. Glesner, "Design and implementation of a coarse-grained dynamically reconfigurable hardware architecture," in VLSI, 2001. Proceedings. IEEE Computer Society Workshop on, pp. 41--46, IEEE, 2001.

[6] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization," in Design, Automation and Test in Europe, 2005. Proceedings, pp. 12--17, IEEE, 2005.

[7] D. Novo, W. Moffat, V. Derudder, and B. Bougard, "Mapping a multiple antenna sdm-ofdm receiver on the adres coarse-grained reconfigurable processor," in Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on, pp. 473--478, IEEE, 2005.

[8] Y. Hasegawa, S. Abe, H. Matsutani, H. Amano, K. Anjo, and T. Awashima, "An adaptive cryptographic accelerator for ipsec on dynamically reconfigurable processor," in Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on, pp. 163--170, IEEE, 2005.

[9] P. Raghavan, A. Lambrechts, M. Jayapala, F. Catthoor, D. Verkest, and H. Corporaal, "Very wide register: an asymmetric register file organization for low power embedded processors," in Proceedings of the conference on

Design, automation and test in Europe, pp. 1066--1071, EDA Consortium, 2007.

[10] K. Choi, "Coarse-grained reconfigurable array: Architecture and application mapping," IPSJ Transactions on System LSI Design Methodology, vol. 4, no. 0, pp. 31--46, 2011.

[11] C. Arbelo, A. Kanstein, S. Lopez, J. F. Lopez, M. Berekovic, R. Sarmiento, and J.-Y. Mignolet, "Mapping control-intensive video kernels onto a coarse-grain reconfigurable architecture: the h. 264/avc deblocking filter," in Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07, pp. 1--6, IEEE, 2007.

[12] C. Park, Y. Kim, and K. Choi, "Domain-specific optimization of reconfigurable array architecture," in Proc. US-Korea Conference, Citeseer, 2005.

[13] J. Davila, A. de Torres, J. M. Sanchez, M. Sanchez-Elez, N. Bagherzadeh, and F. Rivera, "Design and implementation of a rendering algorithm in a simd reconfigurable architecture (morphosys)," in Proceedings of the conference on Design, automation and test in Europe: Designers' forum, pp. 52--57, European Design and Automation Association, 2006.

[14] M. L. Anido, N. Tabrizi, H. Du, M. Sanchez-Elez, N. Bagherzadeh, et al., "Interactive ray tracing using a simd reconfigurable architecture," in Com-

puter Architecture and High Performance Computing, 2002. Proceedings. 14th Symposium on, pp. 20--28, IEEE, 2002.

[15] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert, "Architectural modifications to enhance the floating-point performance of fpgas," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 16, no. 2, pp. 177--187, 2008.

[16] C. H. Ho, C. W. Yu, P. Leong, W. Luk, and S. Wilton, "Floating-point fpga: architecture and modeling," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 17, no. 12, pp. 1709--1718, 2009.

[17] M. J. Myjak and J. G. Delgado-Frias, "A medium-grain reconfigurable architecture for dsp: Vlsi design, benchmark mapping, and performance," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 16, no. 1, pp. 14--23, 2008.

[18] M. Jo, V. Prasad Arava, H. Yang, and K. Choi, "Implementation of floating-point operations for 3d graphics on a coarse-grained reconfigurable architecture," in SOC Conference, 2007 IEEE International, pp. 127--130, IEEE, 2007.

[19] M. Jo, D. Lee, and K. Choi, "Chip implementation of a coarse-grained reconfigurable architecture supporting floating-point operations," in SoC

Design Conference, 2008. ISOCC'08. International, vol. 3, pp. III--29, IEEE, 2008.

[20] D. Lee, M. Jo, K. Han, and K. Choi, "Flora: Coarse-grained reconfigurable architecture with floating-point operation capability," in SoC Design Conference Chip Design Contest, 2009. ISOCC-CDC'09. International, IEEE, 2009.

[21] D. Lee, M. Jo, K. Han, and K. Choi, "Flora: Coarse-grained reconfigurable architecture with floating-point operation capability," in Field-Programmable Technology, 2009. FPT 2009. International Conference on, pp. 376--379, IEEE, 2009.

[22] V. Baumgarte, G. Ehlers, F. May, A. Nuckel, M. Vorbach, and M. Weinhardt, "Pact xpp: a self-reconfigurable data processing architecture," the Journal of Supercomputing, vol. 26, no. 2, pp. 167--184, 2003.

[23] M. A. Syed and E. Schueler, "Reconfigurable parallel computing architecture for on-board data processing," in Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on, pp. 229--236, IEEE, 2006.

[24] "Pact xpp." http://www.pactxpp.com.

[25] C. Brunelli, F. Garzia, D. Rossi, and J. Nurmi, "A coarse-grain reconfigurable architecture for multimedia applications supporting subword

and floating-point calculations," Journal of Systems Architecture, vol. 56, no. 1, pp. 38--47, 2010.

[26] C. Brunelli, F. Garzia, and J. Nurmi, "A coarse-grain reconfigurable architecture for multimedia applications featuring subword computation capabilities," Journal of real-time image processing, vol. 3, no. 1-2, pp. 21--32, 2008.

[27] C. Brunelli, F. Cinelli, D. Rossi, and J. Nurmi, "A vhdl model and implementation of a coarse-grain reconfigurable coprocessor for a risc core," in Research in Microelectronics and Electronics 2006, Ph. D., pp. 229--232, IEEE, 2006.

[28] Y. Kim, C. Park, H. Song, J. Jung, and K. Choi, "Design and exploration of a coarse-grained reconfigurable architecture," in International SoC Design Conference, 2004.

[29] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," Design & Test of Computers, IEEE, vol. 22, no. 2, pp. 90--101, 2005.

[30] Y. Kim, I. Park, K. Choi, and Y. Paek, "Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture," in Proceedings of the 2006 international symposium on Low power electronics and design, pp. 310--315, ACM, 2006.

[31] I. std. 754-1985, "Ieee standard for binary floating-point arithmetic," ANSI/IEEE Std 754-1985, 1985.

[32] D. Etiemble and L. Lacassagne, "16-bit fp sub-word parallelism to facilitate compiler vectorization and improve performance of image and media processing," in Parallel Processing, 2004. ICPP 2004. International Conference on, pp. 540--547, IEEE, 2004.

[33] L. Lacassagne, D. Etiemble, and S. O. Kablia, "16-bit floating point instructions for embedded multimedia applications," in Computer Architecture for Machine Perception, 2005. CAMP 2005. Proceedings. Seventh International Workshop on, pp. 198--203, IEEE, 2005.

[34] ARM, "Arm7 thumb family," Tech. Rep. ARM DOI 0035-3/02.02(7), ARM Limited.

[35] "Arm968 processor." http://www.arm.com/products/processors/classic/arm9/arm968.php?tab=Performance.

[36] "Vector floating point." http://www.arm.com/products/processors/technologies/vector-floating-point.php.

[37] A. Rivaton, J. Quevremont, Q. Zhang, P. Wolkotte, and G. Smit, "Implementing non power-of-two ffts on coarse-grain reconfigurable architectures," in System-on-Chip, 2005. Proceedings. 2005 International Symposium on, pp. 74--77, IEEE, 2005.

[38] J. Becker and M. Vorbach, "Stream-based xpp architectures in adaptive system-on-chip integration," in New Algorithms, Architectures and Applications for Reconfigurable Computing, pp. 29--42, Springer, 2005.

[39] Altera, "Stratix iv device handbook," Tech. Rep. SIV5V1-4.6, Altera Corporation, 2012.

[40] Altera, "Stratix ii device handbook," Tech. Rep. SII5V1-4.5, Altera Corporation, 2011.

# 국문초록

최근 휴대 전화나 태블릿 PC 등의 전자기기에서 구동되는 응용프로그램의 성능과 연산량이 증가가 크게 증가하고 있으며 동영상 녹화/재생, 3D 그래픽스, 증강 현실, 물체 인식 등의 다양한 응용프로그램들이 그러한 전자기기들에서 수행되도록 요구되고 있다. 이러한 흐름과 함께 범용 프로세서보다 빠른 성능을 보이면서 ASIC에서는 볼 수 없는 유연성을 갖고 있는 재구성형 구조에 대한 관심이 높아지고 있다. 대부분의 재구성형 구조는 정수 연산이나 부동소수점 연산만을 위한 구조가 많았는데 최근의 다양성은 하나의 그 연산량이 굉장히 많을뿐더러 이 중 일부는 부동소수점 연산을 사용해야 할 정도의 정밀도를 요구하기도 한다.

본 논문에서는 FloRA라는 이름의 재구성형 연산 구조를 제안한다. 이 구조는 동일한 하드웨어 상에서 그 구성만을 달리함으로써 정수 연산과 부동소수점 연산을 모두 수행할 수 있는 배열 형태의 연산 유닛을 가지고 있기 때문에 앞서 언급한 다양한 응용 프로그램을 가속화하는데 효과적이다. 이 구조는 부동소수점 연산 시 정수 연산 유닛을 이용하기 때문에 7.4%의 적은 하드웨어 비용으로도 부동소수점 연산을 가속할 수 있다. 가능한 재구성형 구조에 비해 이 구조를 130nm CMOS 공정에서 실제 칩으로 제작하여 1.2V에서 최대 동작 주파수가 125MHz임을 확인하였으며 부동소수점 연산 장치를 포함하는 ARM9의 동작보다 평균 11.6배 성능이 향상됨을 보였다. 한편 본 논문에서 제안하는 방식과 유사한 방식의 다른 구조인 PACT XPP와 Butter architecture와의 비교를 통해 제안된 구조가 정수

연산에서는 월등히 빠른 성능을 보이면서도 부동소수점 연산 시에 적어도 다른 구조의 성능 대비 반 이상을 유지함을 보였다.

본 논문에서 제안된 부동소수점 구현은 하나의 부동소수점 연산을 여러 개의 작은 정수 연산으로 바꾸어 정수 연산 유닛을 통해 연산하는 것으로, 정수 연산에 최적화된 구조 및 변환된 정수 연산 간 데이터 의존성으로 인해 부동소수점 연산의 수행 시간이 길어지게 되고 정수 연산 유닛의 사용률이 감소한다는 단점이 있다. 이러한 단점을 극복하기 위해 두 가지 최적화 기법을 제안한다. 첫 번째 방법은 두 개의 서로 독립적인 부동소수점 연산을 겹쳐 수행하는 것이다. 이 방법을 통해 부동소수점 연산 수행 시 동작하지 않는 연산 유닛을 활용할 수 있어서 결과적으로 throughput을 높일 수 있다. 둘째로, 부동소수점 연산의 긴 수행시간을 단축시킬 수 있도록 데이터 의존적인 두 연산을 이어 붙이는 기법을 제안한다. 두 부동소수점 연산 사이에 포함된 불필요한 동작을 건너뛰고 바로 두 연산을 연이어 수행함으로써 부동소수점 연산의 유효 수행 시간을 줄이는 것이다.

위의 기법들을 적용하여 여러 가지 벤치마크에 대하여 성능 평가를 하는 한편 위의 기법이 적용된 하드웨어를 설계하여 기능 및 성능을 검증하였다. 제안된 기법들을 적용하여 20.9%의 하드웨어 비용으로 평균 33.9%의 성능 향상을 얻을 수 있었다.

**주요어**: 재구성형 연산 구조, 부동소수점

**학번**: 2007-21094

# 감사의 글

처음 설계자동화연구실에 들어와 동료들과 인사하던 때가 아직도 생생합니다. 아무것도 모르고 연구실로 무작정 찾아왔던 제가 어느 새 이렇게 학위 논문을 마무리하고 있다니 세월이 참 빠름을 다시 한 번 체감하게 됩니다. 돌아보면 제 연구는 저 혼자 이루어 낸 것이 아니었습니다. 많은 분들의 도움 위에 제가 이렇게 학위 논문을 완성할 수 있었습니다. 그래서 학위 논문의 한 켠에 글로나마 그 분들에 대한 제 감사의 마음을 담고자 합니다.

먼저 30년 넘게 저를 낳아 길러주신 부모님께 감사 드리고 싶습니다. 제가 대학원 생활 동안 열심히 연구에 매진할 수 있는 육체적, 지적, 영적 토대를 만들어 주셨으며 긴 학업의 기간 동안 묵묵하게 뒷바라지 해주시고 기다려주셨기에 오늘의 제가 있을 수 있었습니다. 마음 깊이 감사 드립니다.

대학원 기간 동안 저를 지도해 주신 최기영 교수님께 감사를 드립니다. 교수님께로부터 전공 분야와 연구 방법을 배울 뿐 아니라 연구자로서의 삶이 어떠해야 하는지에 대해서도 직접 보고 배울 수 있었습니다. 다양한 분야를 섭렵하시며 연구에 매진하시는 교수님의 열정은 앞으로도 계속 닮아가고 싶습니다.

바쁘신 중에도 제 논문을 지도해 주신 채수익 교수님, 백윤흥 교수님, 이종은 교수님, 김윤진 교수님께도 감사의 말씀을 드리고 싶습니다. 교수님들의 조언을 통해 논문이 학문적으로 완성도 높은 모습으로 다듬어질 수 있었습니다.

연구실의 선후배, 동기 분들께도 감사를 드립니다. 특별히 학위 논문의 주제인 재구성형 연산 구조의 연구를 같이 했던 김윤진 교수님, 일현 형, 정기 형, 철수 형, 혁중 형과 Prasad, 경욱에게 감사 드립니다. 특별히 칩 제작을 도와주신 동욱 형과 검증 작업을 도와준 규승에게 감사를 전합니다. 2009년의 반 년 동안 함께 미국에서 방문 연구자 생활을 할 때 함께 생활하며 도움을 주셨던 강희 형과 용진 형께도 감사 드립니다. 그리고 진용 형, 성현 형, 동관 형, 영철 형, 기성 형, 현직

형, 임용 형, 석현 형, 동엽 형, 준희 형, 형석 형, 재훈 형, 종경, Wu Di, 양수, 성식, Mingyang, 경훈 형, 준환, 학림, 동우, 재민, 선욱, 성주, 남형, Pierre, 그리고 유일한 동기인 한민. 여러 연구실 동료 선후배 여러분들과 함께 할 수 있어서 고마웠다는 말씀 드리고 싶습니다. 비단 연구뿐만 아니라 삶이라는 문제를 풀어가는 동료의 입장에서 많은 것들을 교감하고 배울 수 있었습니다.

공부 한다고 신경을 많이 써주지 못했는데 동생 은진이에게는 고마운 마음과 미안한 마음이 동시에 듭니다. 자주 만나지는 못하지만 가끔 안부 주고 받는 중학교, 고등학교, 대학교 친구들에게도 오랜 시간 동안 힘이 돼주어 고맙다는 말을 전하고 싶습니다. 연구 기간 중에 만나 많은 격려와 응원해준 여자친구 은미에게 고마움을 전하고 싶습니다.

영적인 면에서 많이 힘들었을 때 상담과 기도로 격려해 주신 정선아 전도사님, 이종태 목사님, 최성혜 전도사님께도 깊이 감사 드립니다. 또 여러 교회 친구들에게도 일일이 이름을 언급하지는 못하지만 감사를 드립니다.

마지막으로 이 모든 일을 주재하신 하나님께 모든 영광을 돌려드립니다. 이 논문을 완성하기까지 매 순간 하나님의 인도하심이 있었음을 인정하고 고백합니다.

논문을 마무리하는 제 마음은 박사과정의 끝보다는 새로운 시작을 향해 있습니다. 새로운 마음으로 더 열심히 정진하여 여러분들의 은혜에 보답하는 삶을 살도록 하겠습니다. 감사합니다.

2014년 2월

조 만 휘