
Electron Nest
Graph Processors
Specification Note

Electron Nest
Specification Note

Shigeyuki Takano

August 15, 2024

Contents

1 Introduction **7**
- 1.1 Background . . . 7
- 1.2 Proposal Architecture . . . 9
- 1.3 Outline . . . 10

2 Computation Model **11**
- 2.1 Electron Nest Microarchitecture . . . 11
  - 2.1.1 Concept of Microarchitecture . . . 11
  - 2.1.2 Composition and Basic Work of Microarchitecture . . . 11
  - 2.1.3 Contribution of Our Research . . . 12
- 2.2 Computing Model . . . 13
  - 2.2.1 Dependency-Chain Computing . . . 13
  - 2.2.2 Attribute Word . . . 13
  - 2.2.3 Push and Pull . . . 14
- 2.3 Components . . . 14
  - 2.3.1 Link Element . . . 14
  - 2.3.2 Processing Element . . . 14
  - 2.3.3 Retiming Element . . . 14
- 2.4 Attribution . . . 14
  - 2.4.1 Concept of Attribution . . . 14
  - 2.4.2 Attribution Decode . . . 15
  - 2.4.3 Attribution Control . . . 15
- 2.5 Message and Execution . . . 16
  - 2.5.1 Execution Example . . . 16
  - 2.5.2 Conditional Loading Message . . . 17
- 2.6 Extensions . . . 17
  - 2.6.1 Compressed Operation . . . 17
  - 2.6.2 Dynamic Routing . . . 18
- 2.7 Summary . . . 18

3 Synchronization Architectures **19**
- 3.1 Tokens . . . 19
- 3.2 Pipeline Register with Token . . . 20
  - 3.2.1 Concept of Undrop Pipeline Register . . . 20

3.2.2 Summary of Undrop Register . . . 20
3.2.3 Pipeline Register without Data-Drop . . . 21
3.3 Buffer Control . . . 23
3.4 Synchronization on Port . . . 23
3.5 Wait Unit: Waiting for Operands . . . 23
3.5.1 Wait Unit Architecture . . . 24
3.5.2 ALU Configuration . . . 24
3.6 Summary . . . 25

4 Dynamic Routing for Indirect Memory Access 27
4.1 Concept of Indirect Memory Access . . . 27
4.1.1 Common Indirect Access . . . 27
4.1.2 Indirect Access on Distributed Memories on a Chip . . . 27
4.2 Dynamic Routing Architecture . . . 29
4.2.1 Routing Data Generation . . . 29
4.2.2 Routing Data for Element Selection . . . 29
4.2.3 Multi-Level Indirect Access . . . 29
4.3 Control Indirect Access . . . 29
4.4 Summary . . . 30

5 Compressed Operations 33
5.1 Motivation . . . 33
5.1.1 Sparseness in Applications . . . 33
5.1.2 Summary of Benefits from Sparse Data . . . 33
5.1.3 Concept of Compressed Operation . . . 33
5.2 Sequence of Compressed Operation . . . 34
5.2.1 Initialization of Compressed Operation . . . 35
5.3 Compression Architecture . . . 36
5.3.1 Most Frequently Appeared Value Detection Architecture . . . 36
5.3.2 Index Compression . . . 36
5.3.3 Index Decompression . . . 37
5.4 Extension in RE . . . 37
5.4.1 Baseline CRAM . . . 37
5.4.2 Extended CRAM . . . 38
5.4.3 MFA Unit . . . 38
5.5 Extension in PE . . . 39
5.5.1 Sync Unit . . . 39
5.5.2 Skip Unit . . . 39
5.5.3 Re-order Buffer . . . 39
5.6 Summary . . . 39

6 External Access Subsystem 41
6.1 Block RAM and Front-End . . . 41
6.2 Front-End Unit . . . 41
6.2.1 Instruction Controller . . . 42
6.2.2 Instruction-Set and Decoder Logic . . . 42

6.2.3 Rename Unit 42
6.2.4 Port Map Unit 42
6.2.5 Commit Unit 43
6.3 Single Chip Electron Nest 43

Chapter 1

Introduction

This document is a specification note of a CGRA (coarse-grained reconfigurable array) architecture and its computation model. This section briefly explains our computing model and its microarchitecture. Firstly, we present background trends in computing systems. Second, we explain the concept of our microarchitecture.

1.1 Background

Modern processors use pipelined architecture to increase instruction throughput by dividing the execution of instructions into several stages. However, to achieve maximum performance, the pipeline stages need to be fully filled, and empty stages can be caused by various factors, such as the misprediction of dynamic control-flow caused by conditional branch instructions and the delay caused by waiting for data to arrive from external memory. To mitigate these issues, modern architectures can be limited by control-flow delays and data dependency issues. To address these concerns, processors have introduced techniques such as branch prediction, cache memory, and out-of-order execution.

The advantage of branch prediction is that it can significantly reduce the delay caused by conditional branch instructions, resulting in higher instruction throughput and improved performance. However, branch prediction is not always accurate, and incorrect predictions can lead to wasted processing cycles and decreased performance. Additionally, predicting the correct branch path requires analysis of the program's control-flow, which can be a complex and computationally expensive task. As a result, improving the accuracy of branch prediction is an ongoing challenge for modern processor designers. Cache memory reduces the impact of loading data from external memory, but cache misses can lead to delays and empty pipeline stages. In addition, finding independent instructions is a challenging problem, and it is done by selecting candidate instructions in an instruction window, which is a pool of fetched instructions. This task requires data dependency analysis on a chip, which can limit the scope of the instruction window. Out-of-order execution is another technique used by modern processors to maintain performance and overcome the impact of control-flow delays caused by branch misprediction. This technique allows the processor to execute instructions out-of-order that maximize the utilization of pipeline stages and computational resources, even if the instructions are not in the original program order. By analyzing the dependencies among instructions and rearranging their execution, the processor can fully fill pipeline stages and maintain a higher level of instruction throughput.

While these techniques have improved performance, they also present ongoing challenges in terms of accuracy, scalability, and power efficiency. As technology markets continue to evolve at a rapid pace, modern processors have become increasingly important for meeting the demands of high-performance and power efficient computing. However, even with advanced pipelined architectures, processors can still face issues such as branch misprediction and delays

caused by loading data from external memory. In today's rapidly evolving technology markets, there is a growing demand for high-performance and power efficient computing systems that can meet the needs of a wide range of applications. Given the trends and these challenges, there is a need for innovative solutions that can balance the performance, flexibility, and cost considerations in designing computing systems. In this context, the coarse-grained reconfigurable array (CGRA) architecture and compiler have emerged as promising alternatives to traditional CPU and GPU-based systems. By offering customizable and efficient hardware and software components, CGRAs can be tailored to meet specific application requirements and provide a flexible platform for future adaptation. CGRA architectures and compilers allow developers to customize and optimize both the hardware and software components of a computing system, providing a high degree of performance and efficiency for specific application domains. However, developing CGRA-based systems can be challenging and expensive, requiring specialized hardware and software expertise, as well as complex design and manufacturing processes.

Our proposal aims to address these challenges by exploring new approaches to developing a general-purpose CGRA architecture and compiler that can simplify the design and implementation process, while also providing the flexibility and adaptability required to meet the changing requirements of the market. The goal of this proposal is to create an architecture and compiler that are well-suited for a wide range of applications and can be easily customized and adapted to meet specific requirements. Using RISC philosophy for both the architecture and compiler can simplify the design and implementation of both components and can help ensure that they are well-suited for a wide range of applications. Using an existing common compiler frontend can simplify the development process; since it leverages existing infrastructure and tools to generate the necessary IR code for the CGRA architecture. The CGRA architecture can offer advantages in terms of power efficiency and design flexibility, which can be important for certain types of applications.

Compared to ASICs and DSL-based accelerators, developing a general-purpose CGRA architecture and compiler can potentially offer lower non-recurring engineering (NRE) and manufacturing costs, since they can be used for multiple applications and do not require specialized design and manufacturing processes. This can make the CGRA architecture and compiler more accessible to a wider range of applications and developers. The adaptability and flexibility of a general-purpose CGRA architecture and compiler can help ensure that the system remains relevant and valuable over the long term, even as the requirements and technology of the market change. This can help reduce the risk of being locked into a specific hardware or software technology that may become obsolete over time and can help to ensure that the investment in CGRA-based systems remains valuable and relevant over the long term.

Our approach to solving the steering control-flow is to prepare a path for a condition signal to select a source and or a destination. This approach could help to reduce the impact of control-flow on the pipeline by ensuring that instructions are executed in the correct order. By dynamically selecting the appropriate source and/or destination based on the condition signal, it could avoid pipeline stalls and maintain the original algorithm, even in the presence of complex control-flow.

By using distributed memories on a chip, which is located close to the processing elements, the latency of data transfers can be reduced, which can help to minimize the impact of data dependencies and control-flow on the performance. The on-chip memory can reduce the latency and area cost associated with the memory hierarchy, and it can simplify the design of the system, by eliminating the need for a separate memory hierarchy. This could reduce the area cost of the system, and also reduce the complexity of the memory access architecture, making it easier to manage. While on-chip memory is expensive, by applying many chips of CGRA, the cost can be reduced. Using many chips of CGRA can also provide other benefits, such as scalability and flexibility. By using multiple chips, it may be possible to scale the system to handle larger workloads or to add or remove chips as needed to optimize the system for specific workloads. Additionally, by using a

distributed architecture, the system can be designed to be more fault-tolerant, as failures in one chip can be isolated and handled without affecting the rest of the system.

The elements on a chip are connected with a two-dimensional mesh, and the chips are also arranged in a two-dimensional array which can be extended to connect the datapaths on different chips. This approach can simplify the design of the system, by using a regular array structure for the chips and the mesh connections. Additionally, by using data dependency-based configuration to connect the datapaths, the system can be designed to be highly configurable and optimized for specific workloads. The chaining-based execution approach with data dependency, could be used to connect datapaths on different chips. This approach can simplify the communication and synchronization between the different chips, as the chaining approach ensures that the data is transferred in the correct order and that there are no pipeline stalls.

One potential issue with this approach is that it may not offer the same level of performance and optimization for specific application domains as ASICs and DSL-based accelerators. This is because ASICs and DSL-based accelerators are specifically designed for the targeted application domains, which allows them to achieve high levels of performance and optimization. However, by carefully considering the design and implementation process and optimizing the hardware and software for the specific application domain, general-purpose CGRA and its compiler may be able to overcome this issue and provide competitive performance and optimization with ASICs and DSL-based accelerators, while still offering the benefits of a more accessible and adaptable platform.

1.2 Proposal Architecture

As mentioned in the previous section, steering the data dependency has a key role in microarchitecture. We aggressively use data dependency to configure datapath rather than finding independent operations. Our research focuses on the nature of data-level parallelism obtained from the spatial configuration like a graph configuration which can not be obtained in the case of traditional microprocessor-based computers.

Key components are a processing element, a memory element, and an interconnection network. Two elements are replicated on a chip with a two-dimensional arrangement of a checkered pattern. The regular pattern makes scaling the array possible, and simplifies and supports a process of compilation, especially for placement and routing. The processing element performs common operations such as arithmetic and logic operations. The memory element stores data that may be temporally used and copied for distribution to relax communication to many destinations.

Both types of elements consist of the same interconnection network components like a template that standardizes the microarchitecture. By embedding a user's module into the template, a user-specific element can be implemented. In addition, the template-based interconnection helps to design and verify it by the precise verification points. Our interconnection network consists of two types of simple routers. One router is called a FanOut link element that transfers data and supports unicast. Another router is called a FanIn link element that selects one source from candidates coming from the FanOut link elements. These two types of link elements are connected, FanOut link element is connected to FanIn link elements and vice versa. We do not take a hierarchical interconnection network that introduces complexity and thus higher workload for placement and routing, difficulty to scale the array as less compatibility for variants.

Any information in our microarchitecture has the same rules and format to run. Any information is treated as a message having its length. The message consists of one or more blocks that consist of the same type or attribution of routing data, configuration data for processing and memory elements, and data. The block has a limited length. The first word of the block is an information word about the block called an attribute word. The attribute word has some roles such as the type indication, length of the block, and so on. The message flows on the array with a worm-whole routing. Therefore, the proposed microarchitecture is a kind of message-passing. The message is stored in the element, and loaded from

the element. We call these actions, a push and a pull, respectively. At first, a message is pushed and pulled the message to another place. Our computation model belongs only to these two types of actions.

There is no central controller. In order to keep an order of processing, a message has an Identification (My-ID) and two other IDs called true-ID and false-ID. FanIn link element checks the My-ID of a message and selects one when My-ID is matched with one of the true-ID and false-ID that are stored by a previous message. One of the true-ID and false-ID is selected by a condition signal making in-order use of the link element and a conditional routing. The condition signal used for conditional routing comes from the processing or memory element.

In order to remove a controller for pipeline stall, we also propose a new pipeline register that avoids dropping data by stalling. We use a handshake protocol of request (req) and not-acknowledge (nack). A common protocol needs double cycles to transfer thus making half throughput. Our pipeline register architecture keeps a single data word transfer per cycle. This approach removes an effort to place a path for source operands that need the same timing to feed into ALU, thus common architecture needs unnecessary paths and routings. Thus our architecture reduces the workload for scheduling, placement and routing at compilation, and thus data-flow graph can be configured on the array without taking care of the timing.

1.3 Outline

The second chapter explains our proposal computation model, its baseline microarchitecture, and its components. The third chapter explains our proposal message-passing method. The fourth chapter explains a pipeline register architecture in order to be fully scalable. In addition, how synchronizing source operands on ALU is also explained through an explanation of the proposal method and its architecture. The fifth to seventh chapters explain interconnection architecture, processing element architecture, and retiming element architecture, respectively. The eighth chapter explains a performance enhancement method based on value-sharing with indexing. The ninth chapter discusses the die-stacking method for our microarchitecture.

Chapter 2

Computation Model

2.1 Electron Nest Microarchitecture

2.1.1 Concept of Microarchitecture

Our main idea is to configure a datapath of an algorithm with its many dependencies. In order to realize the concept, one approach is a coarse-grained reconfigurable array architecture that configures such the datapath on the array. Data is formed as a stream or a vector. We also follow this concept as a baseline.

Our architectural concept is to directly and spatially configure a graph formed by the algorithm with a *dependency-chain*. By appropriately programming the dependencies as signal-flow scheduling, we can predicate its behavior and cost, and thus we can control the latency and the traffic for data streaming.

When low power consumption is needed, data traffic should be the smallest to suppress the energy consumption involved in the signal propagation. Therefore, it should take programming data movement as stational as possible. When high throughput and or short latency is needed, the number of stalls on the signal movement should be as few as possible. To solve or relieve these issues, we propose a new concept of *dependency-chain computing* and its *dependency-chain programming*. Data and or programs are composed as a *message*. The message is transmitted between storage. The transmission is controlled by a user program. This is a slightly similar concept to the register transfer level (RTL) programming model for logic circuit design and to the message-passing on traditional distributed parallel computers.

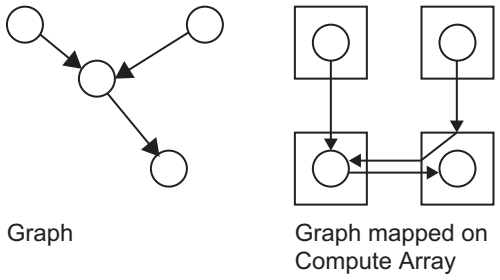

Figure 2.1: Graph and Its mapping

Almost all accelerators take synchronized control such as an SIMD method which consumes much data and produces many results at a time. In addition, almost all processing units take a fence function to wait for the completion of all threads. Such synchronization restricts the opportunity to enhance latency reduction, and thus unnecessary power consumption is introduced by which other all data must wait for the final data element coming from a critical path.

2.1.2 Composition and Basic Work of Microarchitecture

We use a wormhole reconfiguration[**?**] to configure paths on the array. The message runs on the routes. Data is followed after the routing to configure the datapath across elements of the array. A key for the concept is how to realize the routing to configure. We propose a new concept for the routing architecture.

Simple two kinds of router form interconnection networks between processing and memory elements in the array. We call the router a link element. Data movement needing a particular connection takes the link element when it is available. It continues to supply data until the link element gets termination information and or gets another termination information coming from the destination as a backpropagation. This idea reduces the idling time of a part of the elements and contributes to a reduction of the entire execution time.

There is one type of instruction to control data movement in our microarchitecture; The instruction is routing data that defines the actual topology of the graph configured on the array of elements. There are two types of instructions to perform processing and retiming, respectively; one instruction is used for a processing element (PE), which is called processing configuration data which defines the common arithmetic and logic operations. Another instruction is used for a retiming element (RE), which is called retiming configuration data which defines the loading from and the storing in the memory.

Messages of operand data are put on the input terminal(s) of the graph, and then they flow on configured routes, consumed by elements that produce some new vector element data (message), and the producer-consumer flows of signals are on the configured graph by end-terminal(s). When the configured path is no longer necessary, putting the release signal indicating the termination onto start-terminal(s), the release token (signal) flows on the path and makes the release of the elements and their link elements in a data-flow manner. Microarchitecture works a **execute-by-route** by repeating the routings, storing and loading messages between the elements.

In order to be a capable variable-length message on limited storage capacity, the message is divided into a set of blocks that all the blocks except for the last one have a fixed length in the baseline idea. The block has a special data word on its head, which indicates an attribution of the block such as what kind of block, the block's length, and a set of flags indicating how it must be treated by the elements. We call the special data word, an attribute word.

2.1.3 Contribution of Our Research

Proposing architecture introduces speeding up for compute-intensive workloads by the data-flow on the graph configuration and for the data-intensive workloads by the distributed on-chip memories (REs). The kind of coarse-grained reconfigurable array architectures has negative side effects to compilation needing high workloads for placing and routing to make the graph. We also propose a new pipeline register architecture and special logic circuit in order to reduce the workload.

1. Computing Model: Dependency-chained computing benefits from data-flow nature for high performance and throughput. Simply data-dependency representation is translated to routing data. The model realizes simple execute-by-route for computation.

2. Message-based Execution: Message flows on the elements with conditional routing by maintaining data-flow order.

3. Pipeline Register: We focus on distributed stall-control which every pipeline register autonomously stalls its register write. Data is not dropped by stalling.

4. Compilation: almost reconfigurable architectures need to adjust a path length to synchronize the source operands that introduce more workloads to the placement and routing during compilation. Our new pipeline register and proposing a special logic circuit attached to every ALU eliminates the workload.

The baseline architecture has significant potential, we also propose other ideas to get more performance as extensions.

One extension idea supports data compression and execution-skipping. A data block is compressed, and the compressed block is used for processing. For example, sparse vector data has many data having zero value, thus it can be compressed, in addition, the value is not needed to operate multiplication and addition, thus the operation can be skipped. We define general-purpose compression as data-sharing with a

block that removes the redundant data in the block, and operations using the value are skipped. We call such operation, *compressed operation*. These effects reduce the footprint of data and relax the pressure of the storage unit capacity on a chip.

Another extension supports dynamic routing. Applications often use indirect memory accesses involving random memory access. Our baseline architecture configures paths of the graph of the application and keeps the paths until release. Static routing can not work for random access. In order to support the access, we propose that random access is translated to dynamic routing by looking up the route at the retiming element. Our contributions by extensions are as follows;

1. Compressed Operation: Method to operate in message with compressed data. The method aims to reduce the footprint of data and eliminate unnecessary operations by skipping.
2. Dynamic Routing: Method to support indirect access in an application that is translated to dynamic routing across retiming elements.
3. Approximate Arithmetics: Special functions such as sigmoidal, softmax, and division can be configured on the ALUs without series expansion to approximate.

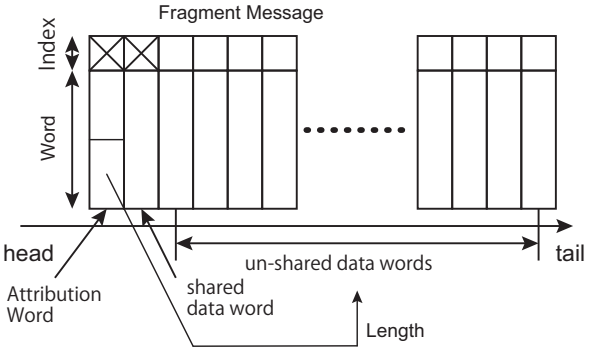

Figure 2.2: Uniform Formation of Data Stream

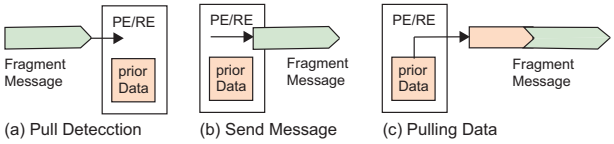

Figure 2.3: Pulling Prior Data

2.2 Computing Model

2.2.1 Dependency-Chain Computing

Our approach is on Coarse-Grained Reconfigurable Architecture (CGRA) and does not focus on a particular domain. Our architecture provides higher flexibility for local memory access, however, it limits the functionality of the element array consisting of a processing element (PE) and the retiming element (RE). PE can be configured to user-defined arithmetics with user-defined data types. We call our graph processor, Electron Nest (EN). EN consists of an interface load/store unit, and a sea of elements, as shown in Figure**??**.

Dependency-chain computing aims to control signal propagation mainly for data movements by the user's program in order to be appropriate for energy consumption and data movement latency, or, to coordinate balancing between them. Memory load and memory store are the data-flow's starting terminal and ending terminal, respectively. Namely, at a concept level, repeating loading and storing on storage makes the data move on a chip. Thus, by constructing a large-scale datapath but shorter critical path entirely with fewer uses of retiming elements, and by locally placing data having infrequent movement onto the retiming elements, we keep higher execution performance and lower energy consumption.

2.2.2 Attribute Word

Figure 2.2 shows the block formation, which flows from the left side to the right side datum. The block has a header data word that holds information of the block such as the type of the block, the block's length, how to treat the block by the element, and so on. We call the header word, attribute word.

2.2.3 Push and Pull

After storing data in RE or external memory, we need to fetch the data and use it. In order to do so, the push/pull technique is introduced in the model. The calculated result is on the output register in PE. Some data are stored in RE or external memory. These actions are called a push. After that, we need to get the data and use it for another operation. The action is called a pull. When pulling is necessary, the pull flag is asserted in the attribute word, and it is sent data block after transmission of preceding blocks as shown in Figure 2.3.

2.3 Components

PEs and REs form a two-dimensional mesh topology and compose the sea of elements called a grid array as shown in Figure2.4. This topology is very simple and makes high-density integration. Element's core part and routing link work independently. Thus, during execution in the core, the routing can be done, or vice versa.

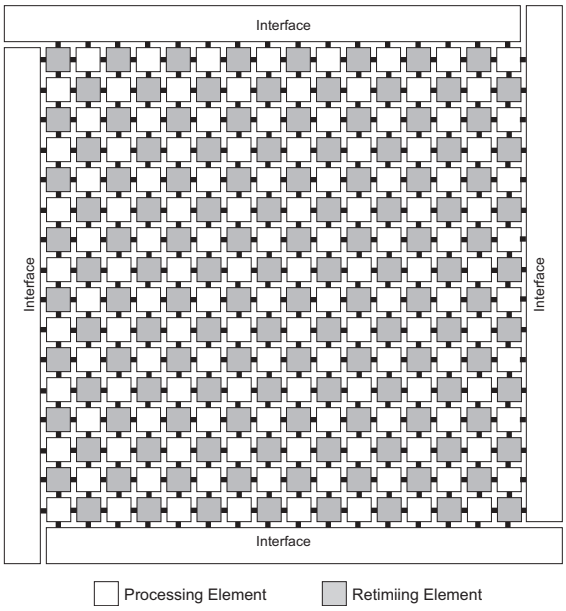

Figure 2.4: The Grid Array consisting of PE and RE

2.3.1 Link Element

We replaced the networks-on-chip (NoC) router with a **link element** connecting among neighbor PE and RE, and inside of PE and RE. The same part between PE and RE is the link element forming a template of the grid array. The arrangement of the grid array is symmetrical, because of user defines the routing path, thus, we need to keep high routability and support placement and routing by the compiler. The link element realizes conditional branching of streaming.

2.3.2 Processing Element

The processing element (PE) performs user-defined arithmetic, logic, bit operations, and so on. User-defined arithmetic operation is configured at run time. It can perform division, logarithm, exponential, and power functions as approximate functions. By translating data value into logarithm space, division can be operated by subtraction, thus the division can have a shallow pipeline depth and need no iterative work which decreases throughput.

2.3.3 Retiming Element

Retiming element (RE) maintains blocks. One RE can be shared with neighbor PEs, for example, a north RE in the grid is shared as a south RE between them. RE can consist of multiple memory banks. Moving the message is to load from and store to RE(s), just like a cut-through, or to store the message in the storage unit (in this case, it is equivalent to multicast and or unicast), and to load the message.

2.4 Attribution

2.4.1 Concept of Attribution

Proposing architecture takes a message-passing concept that communication is based on message transmission. It does not need a central management and a central memory unit. Thus, our architecture does not need to consider a memory coherency problem.

Any data including routing data and configuration data is treated as the message and flows on a common interconnection network consisted by a link element.

A message is a unit of communication. The message can make a path on the grid array with routing data. In order to ease control the communication, a message is subdivided into multiple sub-messages called a block. Follower blocks can flow on the configured paths. By putting a block on the start-terminal of the path, it flows on the path consumed on PE, PE produces an updated block, and the block reaches the end of the path.

The block has a header data word that has information about the block such as the type of the block, the block's length, how to treat the block by an element such as compressed operation, and so on. We call the header word, an attribute word. The block has one of the types, routing data, configuration data, auxial data, and so on. The type is used for serving on link elements, processing elements, and retiming elements, in order to detect their own order. The block's length is used for detecting the next block to serve based on the type of block.

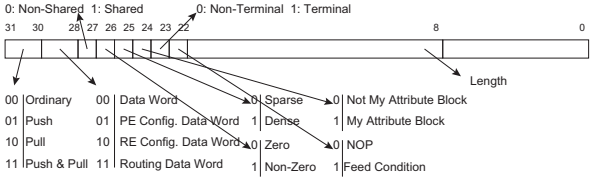

Figure 2.5: Attribute Word

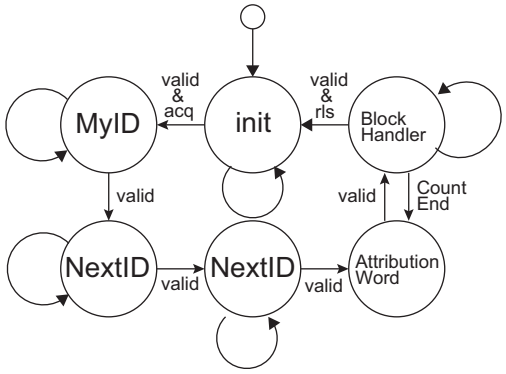

Figure 2.6: Finite State Machine for First Message

2.4.2 Attribution Decode

Figure2.5 shows a format of the attribute word. The 32nd and 31st bits indicate push and pull functions. The 30th and 29th bits indicate a type of block. The 28th bit indicates whether the block is shared or not. The 27th bit indicates whether a shared value is zero or non-zero. The 25th bit indicates the block is taken by RE. The 24th bit indicates the block is a terminal of its message. The 23rd bit indicates condition signal is fed by FanOut element and sent to the connected FanIn element. The 22nd bit to the 9th bit indicates the block's uncompressed length. The actual length is the value plus one.

When the attribute word indicates routing data on a link element, the 15th to first bit-field is a set of routing flags indicating destination link element(s) in the case of the 32-bit word.

2.4.3 Attribution Control

Figure2.6 shows a finite state machine (FSM) to handle a first message.

On the first block, it should have a header that consists of three identifications; The first one indicates the message's identification, called My-ID. Remained two IDs are used for a follower message to use a link element (FanIn Link). After the header, the next word should be the attribute word. An element (ex. link element, processing element, and retiming element) checks a type of block. When the block is for the element, the element consumes the block or a part of the block. The consumed part is removed from the (fragment) message, and the element can produce an updated or new part.

2.5 Message and Execution

2.5.1 Execution Example

Figure2.7 shows an example execution of a multiply-add kernel. The kernel consists of a multiplier and an adder and feeds three vectors, A, B, and C. Vector A and B are used for multiplication, and vector C is used for addition. There are multiple messages to configure the kernel on the grid array.

The first message attempts to configure a path for vector A and to pull the vector A from external memory and push it into a retiming element. The path configuration on a link element consumes one routing data word, and the consumed data word is removed from the message. When the message arrives on the retiming element, the message orders to store the follower auxial data block (step-1). Vector B is also stored in another retiming element with the same procedure (step-2).

The next message pulls vector A after the message passes the retiming element. In addition, the message configures a path to the processing element and orders to configure a multiplication on the element. The pulled vector A is pushed to the processing element as one source operand data block. Another source operand, vector B does not yet arrive on the processing element, so the transmission of the data word of vector A is stalled by a nack token. At the same time, the message configures a path to the next processing element for an addition, and the message reaches a retiming element used for access to external memory, orders store configuration to the store-unit serving external memory access.

The next message pulls vector B from the retiming element and configures a path to the processing element performing the multiplication. After both source operand data words arrive, the processing element does not send a nack token to the path of vector A. Thus, the processing element starts to perform the multiplication. The result data word of the multiplication is sent to the next processing element through the configured path. As the same processing element is configured as the multiplication, another source operand does not yet arrive on the follower processing element used for the addition. The pro-

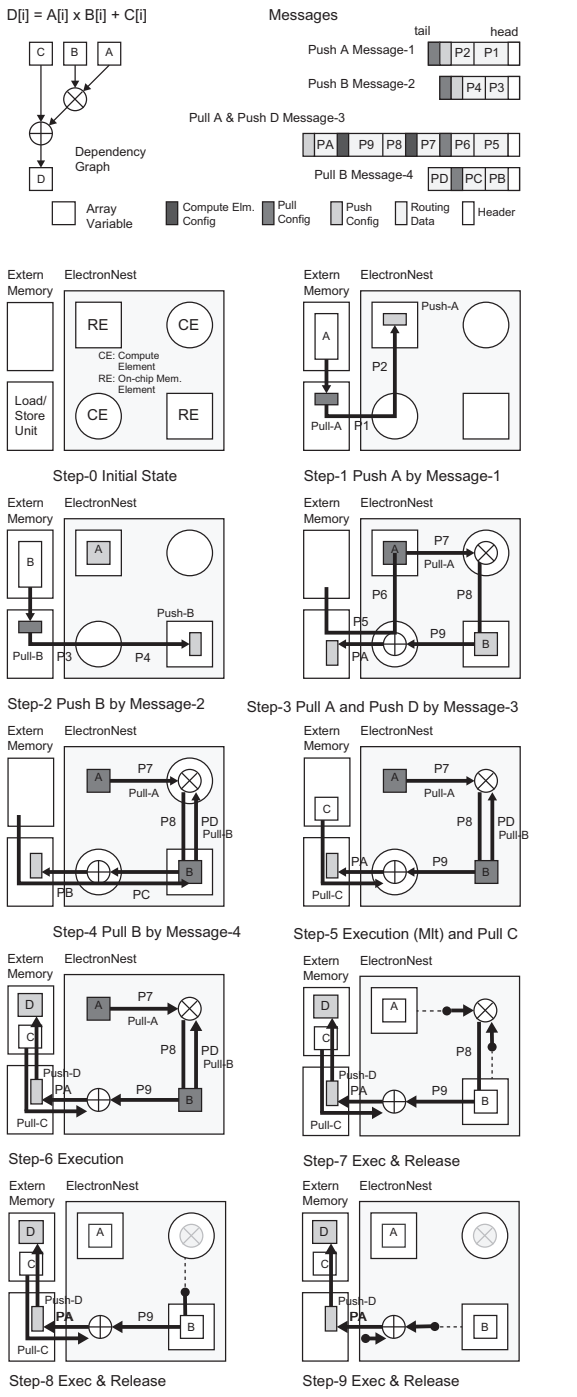

Figure 2.7: Execution Behavior Example

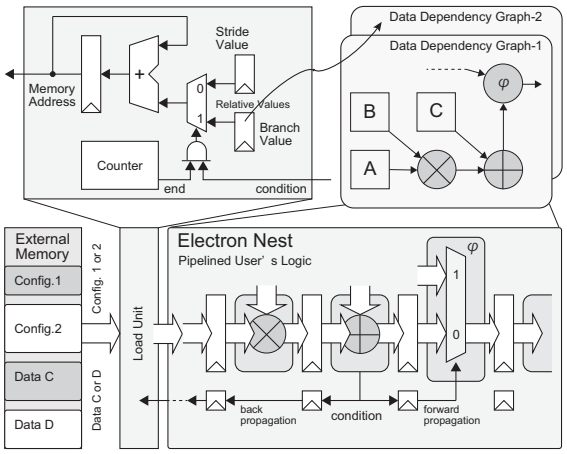

Figure 2.8: Conditional Branch to load Next Message

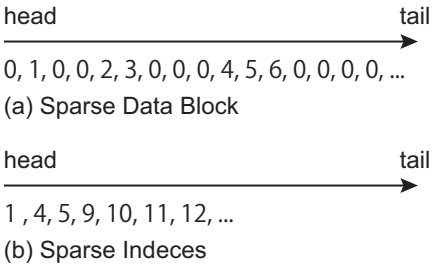

Figure 2.9: Indexed Compression

cessing element sends a nack token to the path of multiplication.

After pulling vector C and pushing it to the processing element, the element starts to perform the addition. The processing element stops sending nack signals to another processing element. The result of the addition is stored in external memory. At the same time, end of the pull for the vector A and B, the retiming element appends a release token to the last data word. The release token releases the configured path on the link elements. By firing of release token on the processing element, the configuration of multiplication and addition are also released. Finally, storing in external memory is also released.

2.5.2 Conditional Loading Message

Load/Store unit composes a message (we call this sequence a message composition or simply composition) and transmits it by a wormhole routing. Or, after the datapath configuration, we can put a message having only data block onto the start-terminal(s) of the configured paths, and it can flow on the path by the end-terminal(s).

Firstly, the interface or an external host requests a loading of raw data from external memory to the interface. Loaded data are composed of a root message that describes how to construct (sub-) graph(s), just like the main function in the traditional software program that the main function invokes functions. When the interface composes a message, it adds a header consisting of the message ID, and the next message IDs. There can be a message having only the header and routing data to extract (configure) a path. Data is defined as a message which is a unit of transfer just like mail and has a user-defined size. We can compose the message by loading particular part(s) of data from the original data and storing it in particular storage. The header includes My-ID and IDs for the next message. Let us call these, Next True ID, and Next False ID, respectively. The next message IDs can be used for structuring a linked list on a stream, and conditional branching on the streams.

2.6 Extensions

2.6.1 Compressed Operation

Data are compressed by proposing indexed compression which uses an index to address elements of not-shared data in a block. The shared data is inserted between a header word of a data block and the block. The most frequently appeared (MFA) value is shared within a block, which block has a maximum length just like a scope. Figure2.9 shows an example of that block consisting of 16 data words.

MFA value is zero in the example and shared in the block. When the shared value is "zero", multiplication can be skipped and simply results in "zero", an addition can also be skipped and other operands simply pass through as the result. The index addresses

the position of the data word before the compression. Thus, source operands for one operation should have the same index value. By checking a position of non-zero values in the data block of both source data blocks for operation, the operation can be skipped or passed through. Therefore, our architecture supports both zero-skipping a data-sharing.

The first data in a compressed data block is a shared data for the block. Thus, a new shared value can be calculated by ALU before working for source data blocks. Therefore, in the case of both operand messages having different redundant shared values, sending the new value to destination(s) and skipping its operation is sufficient. Therefore, the compressed operation method reduces the number of operations without decompression of the source data blocks.

2.6.2 Dynamic Routing

Baseline microarchitecture supports only static routing decided at the compilation. It limits flexibility for dynamic behavior on the grid array. In order to support it, indirect memory access is formed as dynamic routing by looking up a table in a retiming element. A message looks up RE and finds a destination as routing data, and the message uses the routing data, the message repeats this work by reaching to necessary data.

2.7 Summary

This chapter introduced our computation model of dependency-chain computing that chains between graph's nodes through wormhole reconfiguration. Data can be put on the start-terminal of the configured user's graph, and it flows on the path, calculated and or updated on the junction node, and reaches the end-point terminal, this phase can continue until the firing of a release token (explained in next chapter) to release the configured user's graph in data-flow manner.

Elements of processing and retiming were explained. The processing element (PE) performs arithmetics, logic, and bit operations. It can also perform a division, logarithm, exponential, and power of any number. The retiming element (RE) stores an intermediate data set in order to share it among follower graphs and or to explicitly retime in order to bridge the bandwidth gap between graphs.

We showed extension ideas. The compressed operation was also introduced in this chapter. The compressed operation aims to detect the most frequently appeared (MFA) value and a block of data with the value. When the MFA value is zero, zero-skipping is a common approach to enhance execution performance and reduce external memory accesses. The compression can be used for not only zero value but also any value. Dynamic routing was also introduced. Indirect memory access in an application is translated to dynamic routing across REs.

Chapter 3

Synchronization Architectures

This chapter explains the proposal pipeline register and synchronization architectures. The pipeline register removes length-adjustment efforts during compilation with enlarging length to be the same as the critical path length between processing elements. The effort make a huge workload for the compilation and thus takes a long time or can not get a routing result. In addition, the synchronization architecture provides timing to feed an appropriate set of source operand data on the datapath.

3.1 Tokens

We use a handshake method on the synchronous (clocking) logic circuit to retime on a pipeline register. We call the two signals to do so, a request (valid) and a not-acknowledge (nack), and simply call a token.

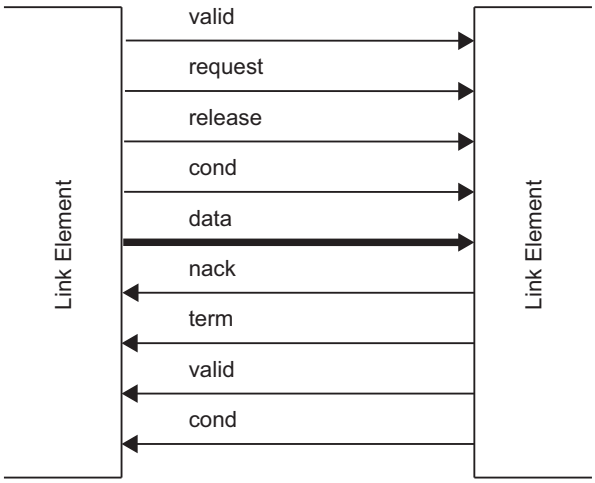

Figure 3.1: Link Configuration

Word has several tokens to indicate the input as shown in Figure3.1. There are two types of tokens. One is a forward token that flows on a configured path with a forwarding direction. Another one is a backward token which flows back direction on the configured path just like a backpropagation.

One forward token is a valid token. The forward valid token indicates the transmission is valid, then the word must be captured by the follower pipeline register. An acquirement token, which assertion means that the word having the token is the first word of the message, and follower data will arrive after the clock cycle. A release token, which assertion means that the word having the token is a tail word of the message. The configured path should be released by "firing" the release tokens.

Putting word onto the start node of data-flow path makes a natural data-flow until the release token is asserted on the path and fired on the ALU, which the release token makes a release of the configured path in data-flow order. A forward condition token is a conditional flag to decide a conditional action on the followed path and a conditional branch on the loading next message.

Backward tokens are a nack token, term token, valid and condition tokens. Nack token is not-acknowledge (nack). The assertion of the nack token indicates that the follower path needs to stop

the forwarding, it is to stall the pipelining on PE, RE, and LE. The term token is to force the termination of the message on PE, RE, and LE on the following configured path. The condition (cond) token with backward valid is to back-propagate the condition generated by PE in order to dynamically change the preceding path for data-flow on the Fan-in link by Next IDs, or the branch taken on loading message in RE or interface element.

3.2 Pipeline Register with Token

Almost all logic circuits have a pipeline register to keep their throughput. The pipeline has to be stalled when the predecessor pipeline stage makes a wait-state to write into its pipeline register. Case of pipelining on a synchronous logic circuit, data can be dropped by the stall signal. In order to avoid the dropping, the stall signal must be broadcasted to every pipeline stage in the pipeline segment in general cases.

Figure3.2 shows the drop of data by nack assertion. Three data, A, B and C arrived with valid signals. Let us see how data can be dropped in the pipelined logic circuit. A clocked pipeline with the tokens can also drop data when a nack arrives.

The word is captured by a pipeline register which outputs the word (Data) at the next clock cycle. The valid and nack tokens should also be captured by a pipeline register in order to avoid a combinatorial loop. The output of the nack token from the pipeline register makes a valid token low on the predecessor pipeline register. When the nack token is high, valid (V_Out) should not be high at the next clock cycle. Unfortunately, the nack and valid tokens can arrive at the same time. The data can be dropped.

3.2.1 Concept of Undrop Pipeline Register

In order to guarantee word having properly on a synchronous logic circuit to un-drop, we propose a distributed control for pipeline registers, therefore, we

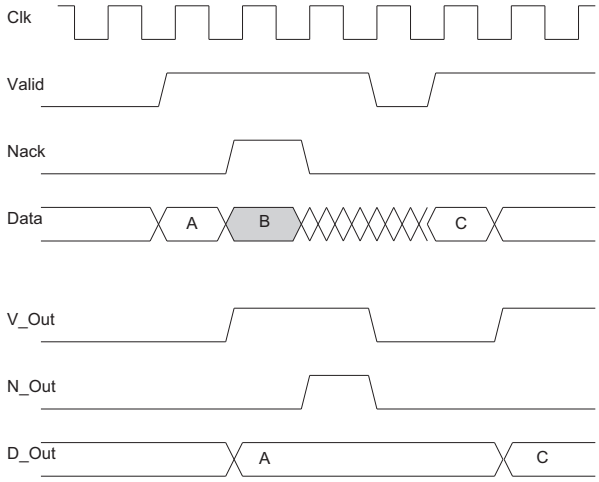

Figure 3.2: Timing Problem by Nack Token

take a hand-shaking protocol with the tokens on a synchronous logic circuit. Figure3.3 shows expected un-drop behavior on the pipeline register with the token of valid and nack.

3.2.2 Summary of Undrop Register

Our approach to making un-drop is summarized as follows.

1. Double-register-based pipeline register

   Two registers are used to avoid dropping. This seems to take a higher cost. We look into traditional microprocessors, these have cache memories to enhance memory access and reduce a wait state in the core. Microprocessor architects invest silicon resources to reduce or cancel the negative impact of microarchitectures. The double register is also one of them.

2. Remove Unnecessary Nack token propagation

   Nack token propagates inverse directions from predecessors to followers. However, the nack token should not be propagated when the valid token does not arrive at the pipeline register because the unnecessary backpropagation of the nack token can stop data transmission forever.

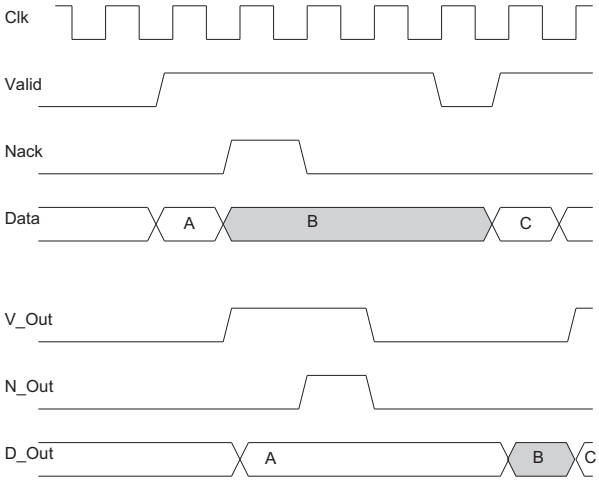

Figure 3.3: Expected Behavior with Nack Token

Therefore, if the valid token is captured, namely data is captured and not empty then nack should be propagated.

3. Buffering on Series of Pipeline Registers

   The nack token propagates and makes a stall of the stages in the inverse direction without a central controller to make the stalling entirely. This series of pipeline stages holds data and thus works as a buffer. In addition, bubbles that are non-data in the data stream, and the series of pipeline registers can be removed by the stalling.

3.2.3 Pipeline Register without Data-Drop

Figure3.4 shows the un-drop pipeline register. It has a double register, write-enable, write register select, and read register select. A controller, called TinyCTRL controls the capturing and write and read selects by arriving at the nack token.

In our approach, changing the write and read register is decided mainly by the nack's behavior. When the nack token arrives, the valid token should be negated. After the nack token is negated, the valid assertion is done to send a captured word.

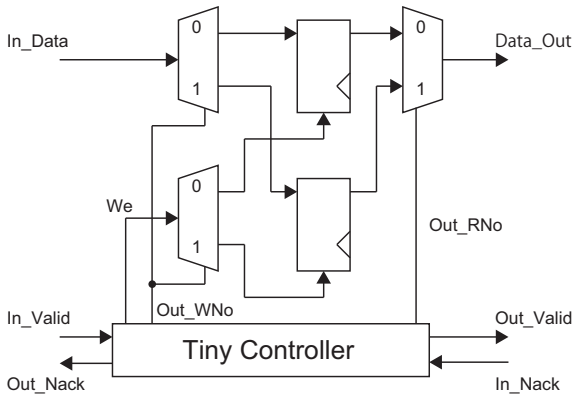

Figure 3.4: Pipeline Register

FSM	State	Input			Next State
	Empty	I_Valid	I_Nack	R_Revert	
0		T	T	T	Wait
0		T	X	F	Fill
		F	X	F	Empty
1	Fill	T	F	F	Fill
		T	T	T	Wait
		F	T	T	Wait
		F	F	F	Empty
2	Wait	T	F	F	Revert
		F	T	F	Wait
		F	F	T	Revert
		F	F	F	Wait
3	Revert	T	R_Revert	F	Fill
		T	R_Revert	F	Wait
		T	F	F	Revert
		T	T	F	Empty

Table 3.1: FSM Transition for Tiny Controller.

Data-flow stops when a nack token arrives. This function forms a queue by the pipeline registers on the path. There is no central control for the pipeline registers to stall. The stalling is made only by predecessor primitive (ex. PE and RE).

Token Unit

Figure3.5 shows one example of the implementation of the pipeline register's controller. The controller feeds the valid token and the nack token from the follower stage and the predecessor stage, respectively. The control takes a ring buffer concept which has only two entries. The controller appropriately updates the write pointer (write select) and the read pointer (read select) by the valid token and the nack token.

Most upside one of two registers captures a valid token signal. After sending successfully, the valid's

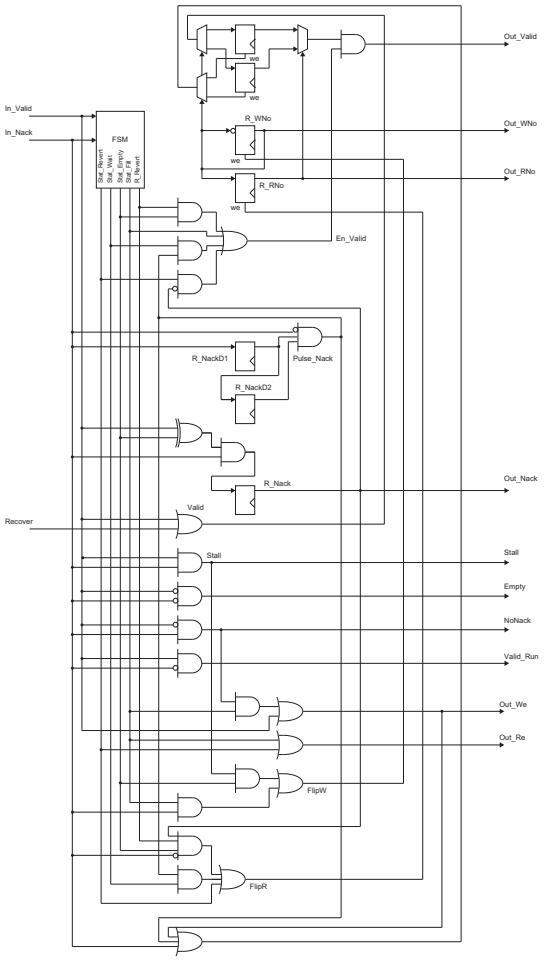

Figure 3.5: Tiny Controller

register should indicate an empty. The most downside side register captures incoming nack tokens. Remained two registers hold a select number for write and read, respectively. Register holding the read-register number captures the write-register number when the nack token leaves. Write-enable (We) is asserted when valid and nack tokens are arrived and not arrived, respectively. When the read-register number is changed, the write signal is also used for the We to capture a signal that could be dropped by a coming nack token.

Table3.1 shows FSM for the controlling. The

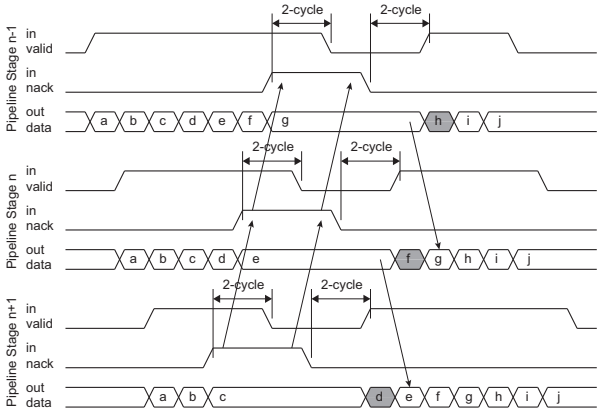

Figure 3.6: Timing Chart on Pipeline

R_Revert is a single-bit register that cooperates with the FSM in order to revert the valid token.

Example Works

Figure3.6 shows an example timing chart on three pipeline stages. There is an in-coming valid token, an in-coming nack token, and a read-out word on every stage.

The word and the valid token are propagated forward direction. The nack token is propagated to follower stages. When the nack token arrives, the valid token and the word are captured in another register. The selection of another register is done by changing the write-register number. The captured valid signal and word are selected to output by changing the read-select number when the nack signal is in a state of low.

Figure3.7 shows an example corner case. If the valid's register is empty then the nack token can be canceled. This idea removes unnecessary retiming. The number of duration cycles of the nack assertion is $max(0, P_{stage} - C_{nack})$, where P_{stage} stages before and C_{nack} cycle-length of nack assertion. The contiguous proposed pipeline forms a queue, this idea bursts unnecessary bubbles on the path (series of pipeline stages).

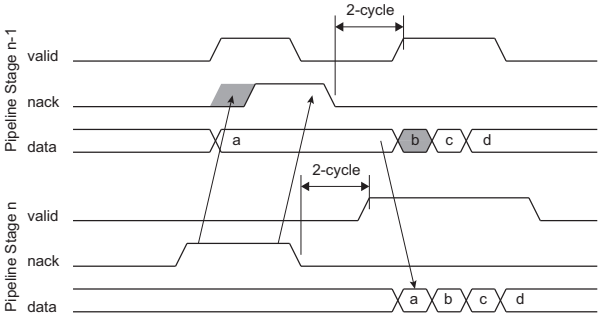

Figure 3.7: Timing Chart on Pipeline

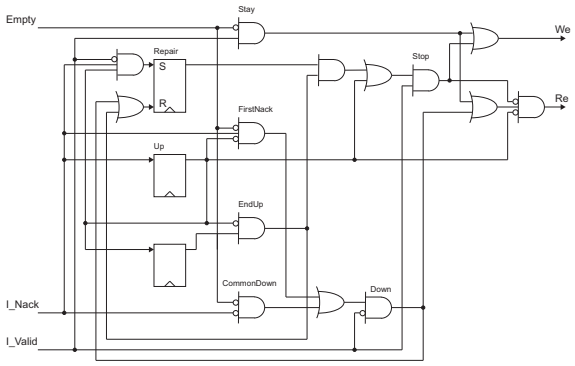

Figure 3.8: Buffer Control

3.3 Buffer Control

The FIFO (Buffer) works as a retiming (pipeline) register. Figure3.8 shows a buffer control signal generation. The Rn_Nack is a nack token with n-cycle delay. The I_Stop and I_SendID come from the FanOut controller. The I_Stop makes the buffering as a signal of Stop. The Down signal sends buffered data words. The Stop and Down signal increments and decrements a counter which indicates a buffer pointer. The Stop, Down, and Stay signals make write enable and clear signals with the counter value of "level" for every entry in the buffer.

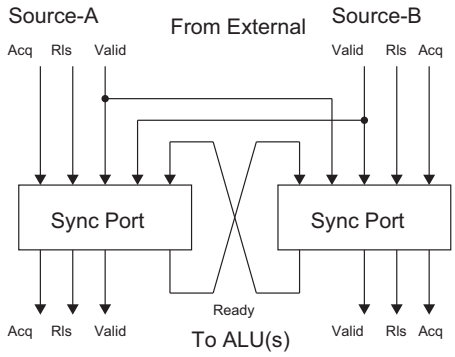

Figure 3.9: Synchronizing Logic Circuit on Port

3.4 Synchronization on Port

One source data block (a message) must wait for arriving of another source data block (message). We can not know which sources arrive first, and it might be the same timing. The synchronization logic circuit attached to the port works for the synchronization as shown in Figure3.9.

The sync-port unit checks the acquirement token for the first input detection and the release token for the last input detection. It captures a block length embedded in attribute word, and uses it to detect next attribute word. The unit detects the data block, or waits for arriving first data of its port, and generates a ready signal when it arrives. The unit also checks other ports' read signal. The unit continues to generate nack token ultil all first data arrive.

The sync-port unit also controls bypassing the input to an output port. The bypassing is necessary to route (configure) a follower path.

3.5 Wait Unit: Waiting for Operands

All source operands should arrive at the same appropriate timing. Simply use of valid and nack tokens leads to failed timing as shown in Figure3.10. This example shows that the nack token is asserted when other source operands are not coming (no valid). This very simple approach can not correctly synchronize

State	Input					Next State
	CHECK_B	FILL_B	Cancel	Valid_B	Valid_A	
Empty	X	X	X	T	T	Fill
	X	X	X	F	T	Wait
	X	X	X	F	F	Empty
	X	X	X	T	F	Empty
Fill	X	X	X	F	F	Empty
	X	X	X	T	T	Fill
	T	X	X	F	T	Check
	F	X	X	F	T	Wait
	X	X	X	T	F	Empty
	X	X	X	F	T	Fill
Wait	X	X	X	T	F	Check
	X	X	T	X	X	Check
	X	X	F	X	X	Wait
Check	X	X	X	F	F	Empty
	X	X	X	T	T	Fill
	X	T	X	F	T	Check
	X	F	X	F	T	Wait
	X	X	X	T	F	Wait

Table 3.2: FSM Transition for Wait Unit Controller.

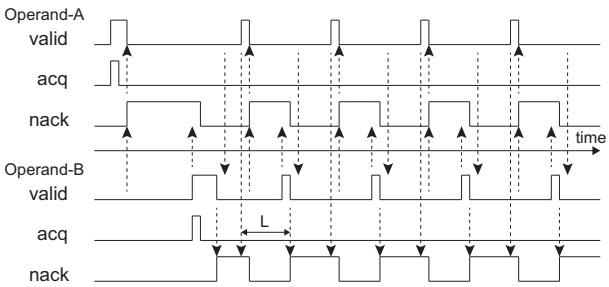

Figure 3.10: Typical Timing Problem for Synchronization

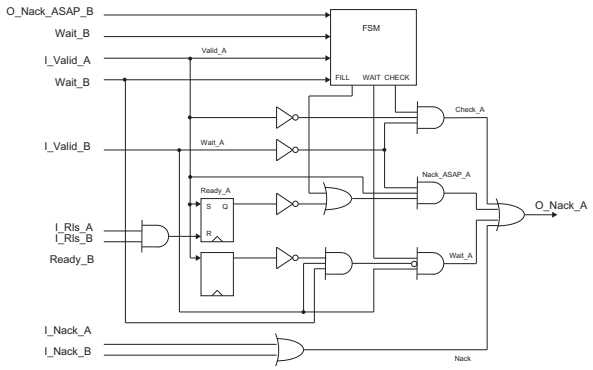

Figure 3.11: Wait Unit Logic Circuit

feeding the source operands. One approach is to have a buffer, however, this approach also limits the timing by the buffer size. To synchronize among source operands, we apply the nack token.

3.5.1 Wait Unit Architecture

Figure3.11 shows a synchronization logic circuit for two source operands, A and B. A ready register indicates that the source operand is in a ready state to operate. It is set by the actual valid signal and cleared by a firing of the release token. There are four sources for the nack token generation. A nack signal is an assertion of a valid or nack token. A wait signal is an assertion when waiting for another source operand. A nack_asap is an assertion when an incoming nack token arrives. A check signal is an assertion after synchronization completion.

3.5.2 ALU Configuration

Figure3.12 shows an example ALU configuration using SyncPort unit and wait unit. The PortSyc unit is located after the input register of the ALU. Every input port for the source operand should have the unit. When all sources have arrived, firing by operands triggers starting the execution.

Wait unit after the PortSync unit checks tokens

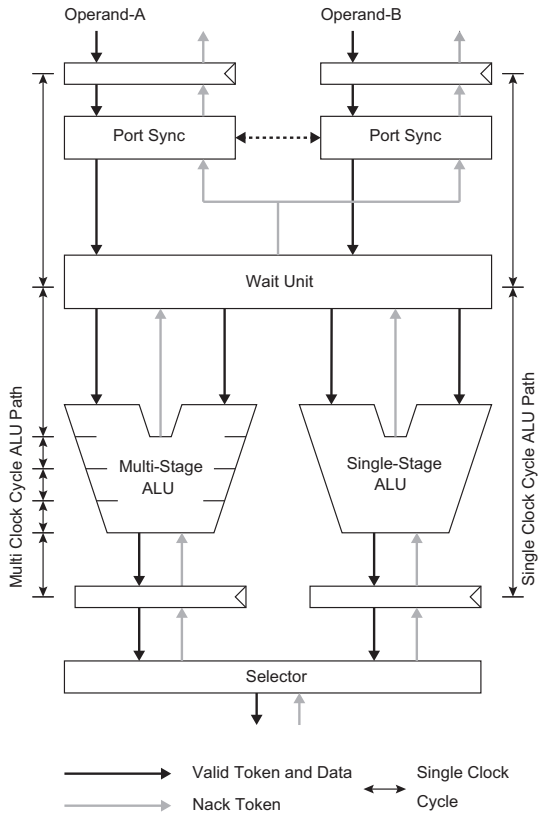

Figure 3.12: ALU Configuration

and issues a nack token when the inputs do not make firing for operation. After the wait unit, a single-stage or multi-stage arithmetic, logic, shift, etc, feeds synchronized source operands, and executes their own operation.

3.6 Summary

This chapter explained the proposal pipeline register avoiding data-dropping caused by handshaking. The method removes huge efforts for placement and routing of the user's graph onto the grid array because of no longer needs adjustment of paths in order to retime for appropriately feeding source operands. Thus, the compiler's workload can be reduced. In addition, source operand data must wait for another source data, PortSync unit serves the waiting state before execution by using the nack token. Wait unit makes stalling for the retiming at execution.

Chapter 4

Dynamic Routing for Indirect Memory Access

Software often needs random memory access. Random access is based on indirect memory access that address is obtained by another memory access. Typical CGRAs have an assumption that data is packed into one data set as dense data. Therefore, The CGRAs can not support the software. In order to support it, we propose the indirect memory access method on CGRA with dynamic routing.

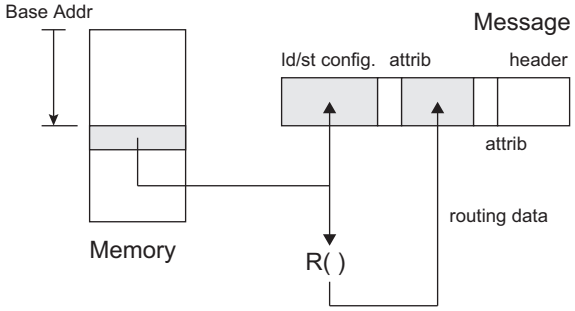

Figure 4.1: Concept of Indirect Memory Access

4.1 Concept of Indirect Memory Access

4.1.1 Common Indirect Access

Figure4.1 shows typical indirect memory access. In order to access data in Array_B, its access address is obtained from Array_A. CGRA needs to support indirect memory access on distributed on-chip memories.

4.1.2 Indirect Access on Distributed Memories on a Chip

We translate the indirect access to a dynamic routing problem as shown in Figure 4.2. Path length on a right and a left direction are positive and negative integer numbers, respectively. The message tries to pull data from RE in where unknown location and push the data to another RE already known in this example. The message runs to RE having location information, and queries to the RE. The message gets the address, and RE composes an appropriate routing data set including routing data after the pulling.

Figure 4.3 shows our indirect memory access method on the distributed on-chip memories.

The indirect memory access is translated to a routing. Index for Array_B is translated to routing data and stored in Array_A memory.

We assume that memory address space is uniform, but the space is subdivided into multiple memory banks. The bit field in the address is subdivided into three fields as shown in Figure4.3. The least significant two bit-fields can be used for access to a memory bank. The most significant bit field can be assigned to the memory bank ID as a tag. In addition, the

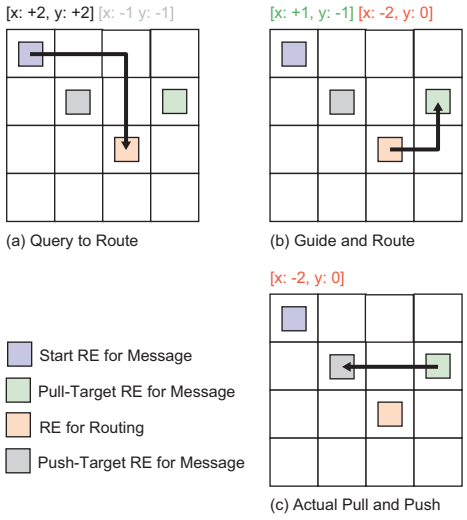

Figure 4.2: Example Dynamic Routing

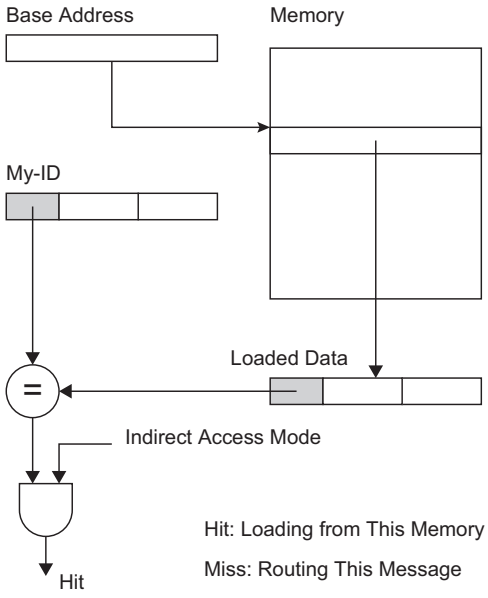

Figure 4.3: Idea of the Dynamic Routing

My-ID field can also be subdivided by the same way.

The tag field is compared, and if it is matched (a hit) then the memory bank is the target destination bank. Otherwise, the message requesting the loading is routed to another memory bank by inserting routing data. This is equivalent to load data from Array_A by the base address, the data word is used for the routing as shown in Figure 4.4. The Array_A has a role in the lookup table for the routing data. By preliminary storing the routing data in the on-chip memories, the program obtains data in the Array_B.

Case of our microarchitecture, the indirect memory access can be implemented by loading an ID from the memory bank, the ID is used for memory access for the array (ex. Array_B), and for generating routing data to the memory bank having target data. When the tags are not matched, routing data (and its attribute word) is generated. The generating routing data involves an adjustment for routing data and its attribute word after the configuration data used for the memory access (loading). We call the work a recovery. The recovering generates routing data and its attribute word for intermediate routing to the target. The recovering block is inserted into the message as shown in Figure4.4 where $R(*)$ is a function of the routing data generation. $R(*)$ function feeds the two least significant bit fields in the IDs. The one field is the corresponding physical address on the x- or y-axis.

After the sending configuration data for the loading, all routing data must be adjusted because the routing path was changed. The routing adjustment has the same work as the recovering but checks the length of the routing block and checks direction of the routing. When the direction is matched to the direction in recovering, the length must be adjusted

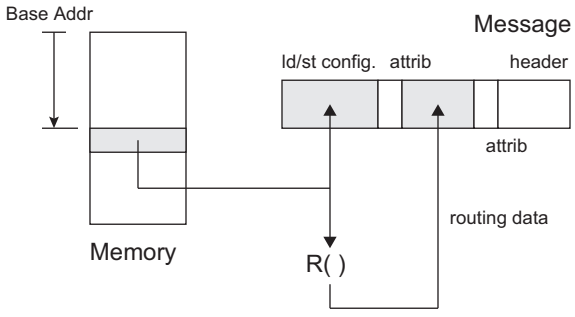

Figure 4.4: Indirect Access with Dynamic Routing

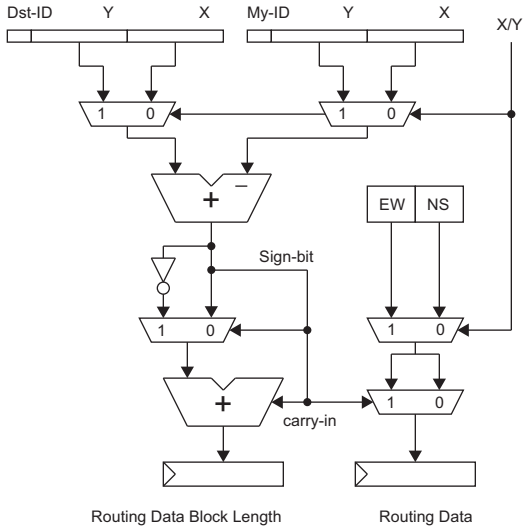

Figure 4.5: Attribute Word and Routing Data Generations

by subtraction of the original length minus the recovered length.

4.2 Dynamic Routing Architecture

We assume that a routing method is X-Y routing. First, the x-direction is routed and the y-direction is routed at the next.

4.2.1 Routing Data Generation

Routing data is obtained by subtraction of destination address (Dst-ID) from its address (My-ID). Its sign flag indicates an inverse direction on the axis. For example, the north and south directions are assigned to zero and one on the sign flag, respectively.

Figure4.5 shows an extension of the load/store unit to have dynamic routing. Up-most IDs are fed into the extension logic. The ID is subdivided into an element select field and x and y address fields. The extension logic circuit generates routing data and its data size (length) embedded in an attribute word attached to the routing data (block). Absolute value

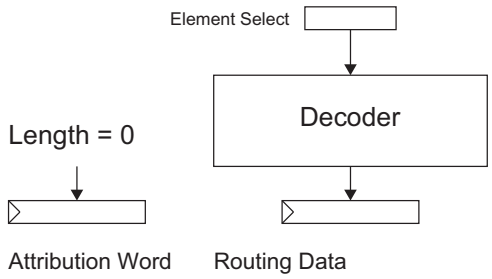

Figure 4.6: Routing to CRAM

is generated by an adder for the length. The X/Y signal selects the x-axis or y-axis.

4.2.2 Routing Data for Element Selection

When a message reaches a target retiming element having multiple memories (CRAMs), the message must enter into a particular CRAM by its routing data. The routing data and its attribute word are generated by a decoder and a constant, respectively, as shown in Figure 4.6. The decoder generates a one-hot code that bit in the field is corresponding to the particular CRAM.

4.2.3 Multi-Level Indirect Access

By our method, multi-level indirect memory access is possible as shown in Figure4.7. The multi-level access is to replace the routing data and load configuration data with the extended load/store unit. It does not affect follower attribution block(s). By pulling a block from the CRAM, the block can be appended to the message.

4.3 Control Indirect Access

Figure4.8 shows FSM for controlling the indirect access. The controller forces to kick-start loading data word located on the base address in the memory.After the loading, The controller terminates its work when the part of it and My-ID is matched. Otherwise, the controller starts its sequence.

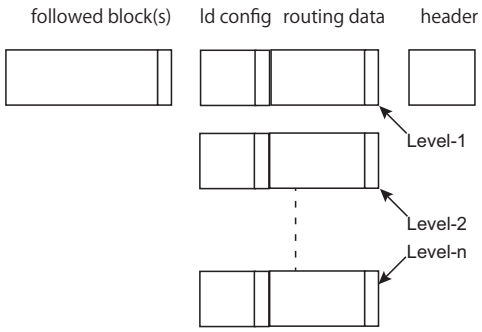

Figure 4.7: Multi-Level Indirect Access

There are two cases of that straight routing belonging to vertical or horizontal directions from the source location. In such cases, routing data generation for the x- or y- direction must be skipped. The controller also attempts to insert attribute word needed to the generated routing data.

4.4 Summary

Software sometimes has indirect memory access. The addressing pointer in an array is dynamically changed. We translated the indirect access to the dynamic routing. It takes a tiny extension module in CRAM, which detects tag-match, inserts routing data and its attribute word, and an adjusting followed routing block(s).

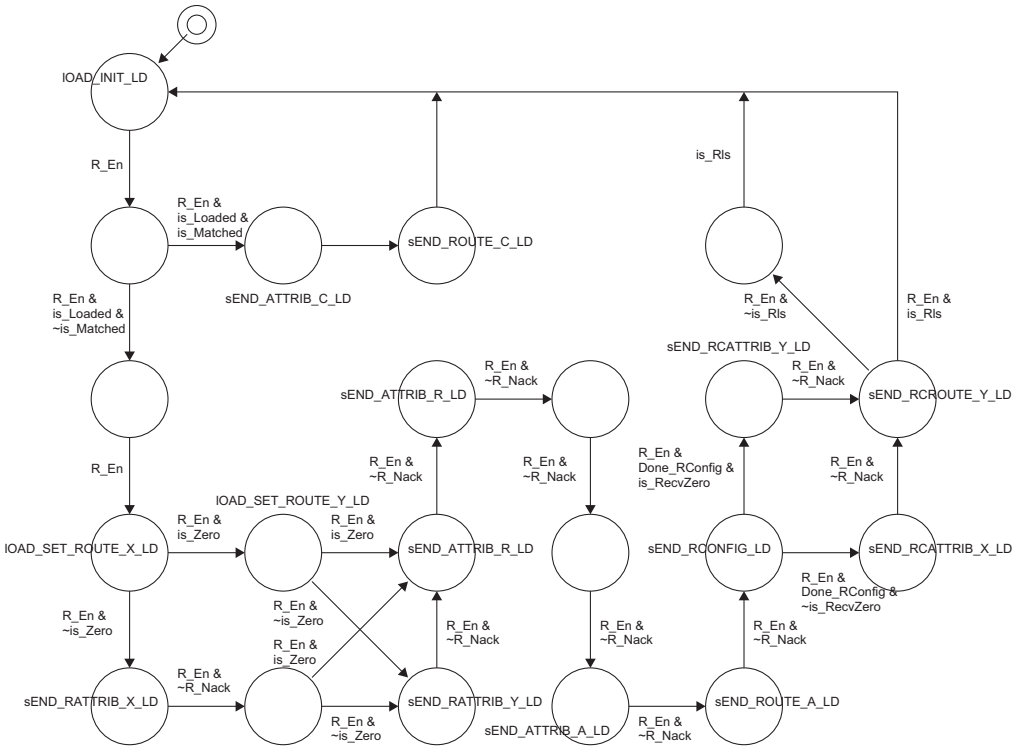

Figure 4.8: FSM Diagram for COntrolling Indirect Access

Chapter 5

Compressed Operations

5.1 Motivation

5.1.1 Sparseness in Applications

Recent applications use sparseness in data, which improves inference accuracy in the case of deep learning tasks for example. Graph processing also takes such sparseness to represent a topology of the graph. These sparseness are focused on not only execution performance but also energy consumption reduction, thus enhancing the energy-efficiency.

The sparse data includes many zeros. Sometimes, more than 90% of the data are zeros in deep learning tasks. Therefore, the execution with such data has the opportunity to have a lesser number of execution steps and also smaller storage requirements. Regarding the execution step reduction, multiplication with zero can skip its operation because we know that the result is zero. Almost all applications handling digital signal processing consist mainly of so many multiply-add (or multiply-accumulate) operations. In the case of addition with zero, the operation can send non-zero data on another operand, thus the addition can also be skipped. By arranging sparse data to have only non-zero data word(s), the necessary storage size can be reduced. This smaller footprint has a positive side-effect of the reduction of external memory accesses. Therefore, by use of sparseness, this technique introduces energy-efficiency improvement.

In addition, we extend the idea to have a shared value which is most frequently appeared (MFA) in a data block. It is a zero-skipping execution when the MFA value is zero.

5.1.2 Summary of Benefits from Sparse Data

Sparse data introduces the energy-efficiency, which can be summarized as follows;

- Skippng of Operations by zeros

  The number of execution steps can potentially be reduced by skipping the operation involving zero-operand. Regarding spatially mapped user's graph, simply skipping one operation can not reduce the execution steps, because the critical path on the graph is the same as a conventional non-skipping operation.

- Storage Access Reduction by rearranging data structure

  Reducing the number of execution steps needs to change or rearrange the data structure. We focus on sparse data, so compression is the most effective approach that does not need to change the data structure but needs only to remove zeros.

In order to obtain higher energy-efficiency, we combine zero-skipping and data compression.

5.1.3 Concept of Compressed Operation

Our technique called compressed operation combines both operation-skipping and compression.

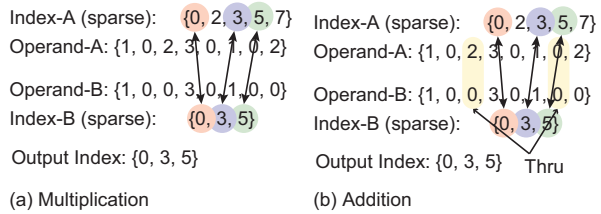

Figure 5.1: Concept of Indexing the Data Words to Non-Skip Detection

Concept of Operation-Skipping

Our skipping focuses on skipping opportunity enhancement, namely, all operands are dynamically compressed and have indexes for each.

Table5.1 shows the summary of zero-skipping on operations having two operands (A and B). When the index on both operands is matched, means non-zeros have the same address on the original data, and thus should perform the operation (Op), otherwise, the operation can be skipped. This idea does not need a decompression before the operation for all operands and simply performs on compressed data.

Figure5.1 shows a relationship between indexes and their operands for a data block. We need to check non-zero values to perform an operation. The index is to show the actual address in the data block and data elements of zero-value can be removed from the data block. Indexes for this example should have indexes of {0, 3, 5}.

General-Purpose Operation-Skipping

Case of our approach, we focus on data word sharing by compression too, which includes compression for zeros. We focus on a data block having a fixed length on an original data structure. In order to compress (remove) such shared data word(s), we focus on to detection of the most frequently appeared (MFA) data value which can be omitted in the compressed data block. Zero data words can be compressed if that zero is the MFA value in the data block. The shared value is attached to the compressed data as a header word.

The indexes take additional storage, up to the length of the data block. We can reduce the size by half because we know which size of shared data or non-shared data is greater than the other when we detect the MFA value, thus we can select one for indexing. Case of zeros to share, the number of zeros is greater and lesser are equivalent to sparse and dense data, respectively. By preparing a one-bit flag to indicate whether the data block is sparse or dense, we can compress the data block with indexing efficiently.

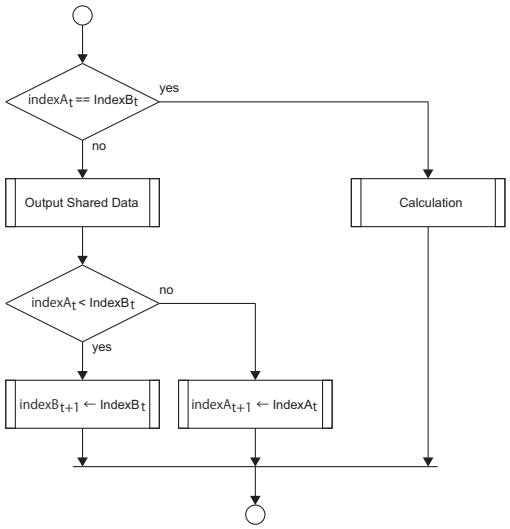

Figure 5.2: Baseline Compressed Operation

5.2 Sequence of Compressed Operation

Let us look into the zero-sharing meaning of compressed with zero-value before exploring sharing non-zeros. Figure5.2 shows a flow-chart for the compressed operation. Index addresses non-zero data words in case of sparse compression. Thus, a dedicated operation is necessary when all operands have the same index value, otherwise, the operation has an opportunity to skip.

In case of an addition, all source operands having the same index value have to operate. Next, check the next index if at least one index value is smaller than another one. Then its indexed value is non-

Index	Output Result								Skip	Output
	ADD	SUB	MLT	DIV	OR	AND	XOR	SFT		Index
$I_A > I_B$	B	-B	0	0	B	0	B	0	Yes	I_B
$I_A < I_B$	A	B	0	∞	A	0	A	A	Yes	I_A
$I_A = I_B$	Op	Op	Op	Op	Op	Op	Op	Op	No	I_A or I_B

Table 5.1: Summary of Zero-Skipping.

zero, however, another one in the same position of the data word is zero. Thus, the addition can skip the operation and make a pass-through of the non-zero operand data word. Case of multiplication, all indices having the same value have to operate. Next, check the next indices and if the indices are not the same, the operation can be skipped. At the skipping with index-mismatch, one data word having a bigger index value should be succeeded to the next operation and its data word should be reused. Other smallest indexes should be sent with their operation result to follow the operation.

On a temporal dot-product on a MAC unit, simple skipping is sufficient to reduce both execution steps and storage requirements. On a spatial dot-product on the arithmetic unit array, simply skipping multiplication and addition independently takes N-step operation, where N is a number of array elements on the critical path. Therefore, it contributes only to footprint reduction in this case, and can not reduce execution steps. In order to speed up the execution, we prepare a re-order buffer attached to every ALU. The re-order buffer is equivalent to one used in out-of-order processors. When indices are not matched, one source having a larger index value takes a shared data word, and another source operand takes the non-zero data word. The result from ALU is stored in the buffer.

Our approach takes into account a pipeline stall. The proposed series of pipeline registers form a queue and can cause stalling sometimes. Thus, by sending the data words in-order as soon as possible, the queue buffers the data words.

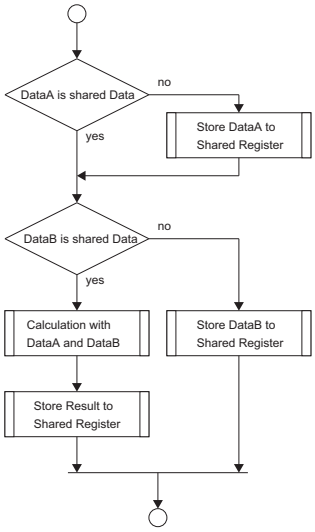

Figure 5.3: Initialization of Compressed Operation

5.2.1 Initialization of Compressed Operation

The compressed operation must calculate new shared data word before operations and send it to the next operator(s). There are three combinatorial cases for the two-operand operator. The first case is all data blocks have a shared data word. The second case is all data blocks do have not a shared data word. Another case is that one or more data blocks do have not a shared data word. The last two cases should operate for all elements.

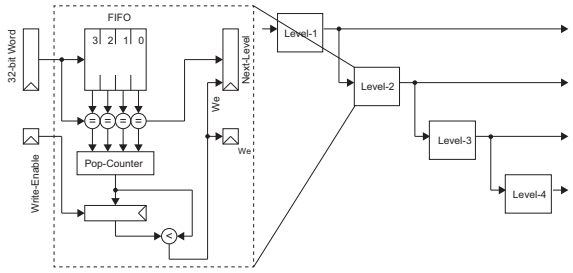

Figure 5.4: Most Frequently Appeared Value Detector

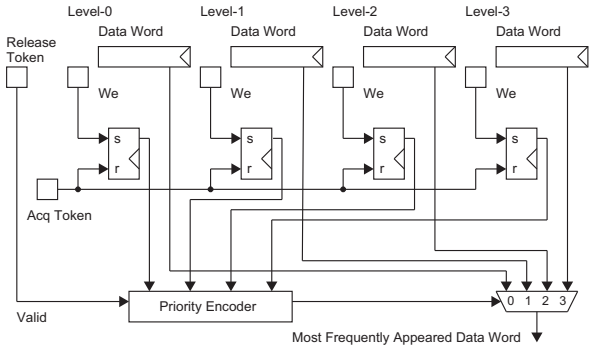

Figure 5.5: Selection of Most Significant Appeared Data Word

5.3 Compression Architecture

The compression must seek the most frequently appeared (MFA) data value in the data block. This needs to count the appearance frequency for every data value.

One approach is to take FIFO having a full-size data block, count for every word on every FIFO entry, and also need to shift the count value. However, it needs a huge number of counters and comparators, it can not use SRAM for the FIFO. In addition, the need for registers takes a bigger area and thus takes huge energy consumption. This is not suitable for our architecture focusing on the energy-efficiency.

5.3.1 Most Frequently Appeared Value Detection Architecture

We propose a multi-level FIFO-based detector. FIFO length is four in the example. Figure5.4 shows detector architecture, 4-ary tournament based on four FIFOs ($= \log_2 256/2^4$, where 256-length of block data). The data word is captured on the left side of the unit. At the last comparison in the detector unit, a valid data word in the most high-level is the MFA data value. This technique reduces the comparator requirement, and we can design with parameters of data block size and FIFO size, these decide the number of levels of a tournament (the number of FIFOs).

Figure5.5 shows the selection logic circuit for detected candidates of MFA data word at the last phase of the detection. There are four input pairs of a write enable and candidate data word at every level. The last data word signal enables to validation of a selection number. The first word's acquirement token signal clears the write-enable holder. The set of write-enables is fed into a priority encoder that generates the selection number. The selection number selects the actual MFA data word from the candidates.

5.3.2 Index Compression

Concept of Index Compression

Index storage needs a size of $(FD/16)\log_2 FD$-Bytes, where FD is the size of block data. Case of the 256-word data block, the storage should have up to 256 bytes, thus the indexed compression needs relatively larger storage for the indices.

In order to shrink the index storage, we use bit-map encoding for indices that every bit-field indicates a position of shared data word (case of dense compression) or non-shared data words (case of sparse compression).

Index Compression Architecture

Figure5.6 shows how indices are compressed to a set of one-bit flags and its baseline architecture. Let us see a case of a 256-word data block and a 32-bit data word.

Index is 8-bit ($= \log_2 256$). The most significant 3-bit is used for addressing flag storage. The flag storage consists of 8-entry ($= 2^3$), and every entry

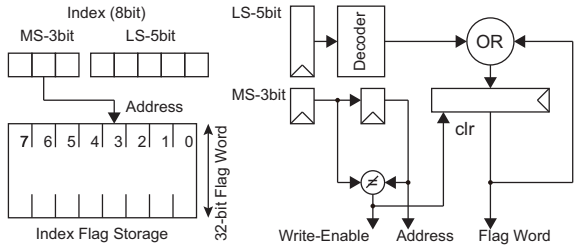

Figure 5.6: Index Compression Logic Circuit

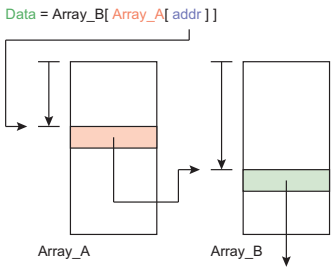

Figure 5.7: Index Decompression Logic Circuit

consists of a 32-bit word. The least significant 5-bit is decoded to 32-bit data word having a one-hot flag. The data word is ORed with a previous value which register is cleared by changing the most significant 3-bit, which means changing the address of index storage. This technique compresses from 256 bytes to 32 bytes.

Index decompression can be implemented by a counter and a number leading zero (NLZ) unit. The counter value is equivalent to the most significant bits addressing the storage. The NLZ unit generates the index within the compressed word. The position of the number detected by the NLZ unit should be cleared after the use of the value. The counter value and the number are concatenated and output as an actual index value.

5.3.3 Index Decompression

Decompression can be done by a counter to address a single word in storage. Firstly, if the compression takes dense data, then indexes are shared values. Therefore, the index set for the data block should be

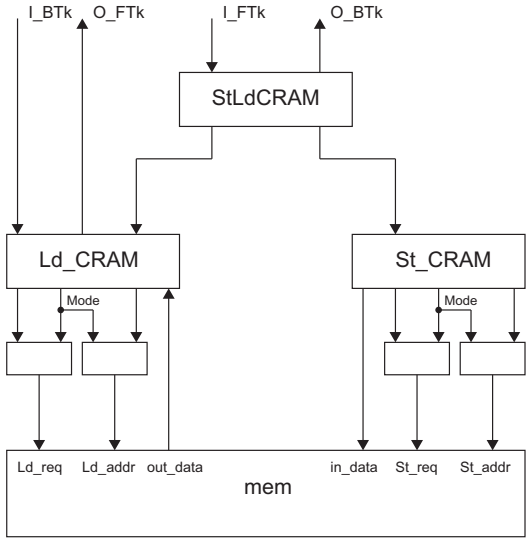

Figure 5.8: Baseline CRAM Block Diagram

transformed to sparse data formation. After that, if the counter value is the same as the index then fill a non-shared value in order, otherwise fill the shared value. The case of sparse data has an opposite policy to fill.

5.4 Extension in RE

After the compressed operations, the number of words and the indices having non-shared data word in a data block can be changed. In order to restore the data block to minimize its length, RE works to do the restoring.

5.4.1 Baseline CRAM

Figure5.8 shows a baseline CRAM block diagram that is equivalent to Figure **??**. Load and store requests enter into StLdCRAM (FanOut Link). The storing holds its path until its release, follower requests must wait for the release. The loading releases the path after entering the request into Ld_CRAM, a follower store request can enter the St_CRAM and starts its storing.

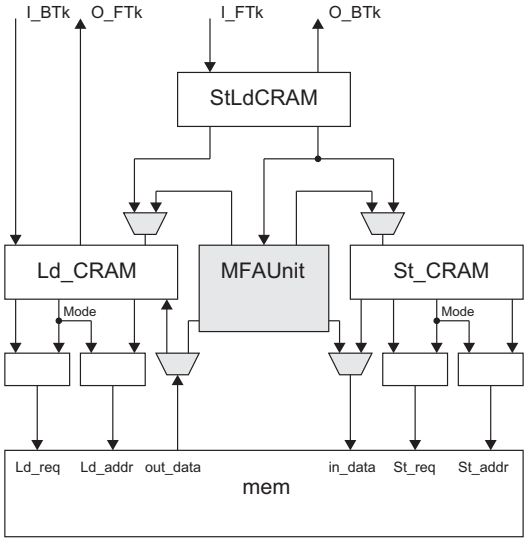

Figure 5.9: Extended CRAM Block Diagram

5.4.2 Extended CRAM

Figure5.9 shows the extended CRAM block diagram, the gray parts are the extension. MFA unit (MFAUnit) works for the control.

MFA unit snoops the store request and its configuration data. After the storing, the MFA starts the restoring when a flag of sharing is asserted in the configuration data. The MFA unit works to detect the MFA value at the same time as storing the data block in the memory. MFA unit keeps the detected value and the number of shared words in the data block.

The restoring issues load request and store request to the Ld_CRAM and St_CRAM, respectively. The loaded data words are fed into the MFA unit and compared their value with the detected shared value. When these match, the MFA unit does not store the word in the memory. At the same time, the MFA unit counts the number of stores.

During the restoration, a follower request coming from the StLdCRAM module must be blocked, therefore, the MFA unit sends the nack token to the module.

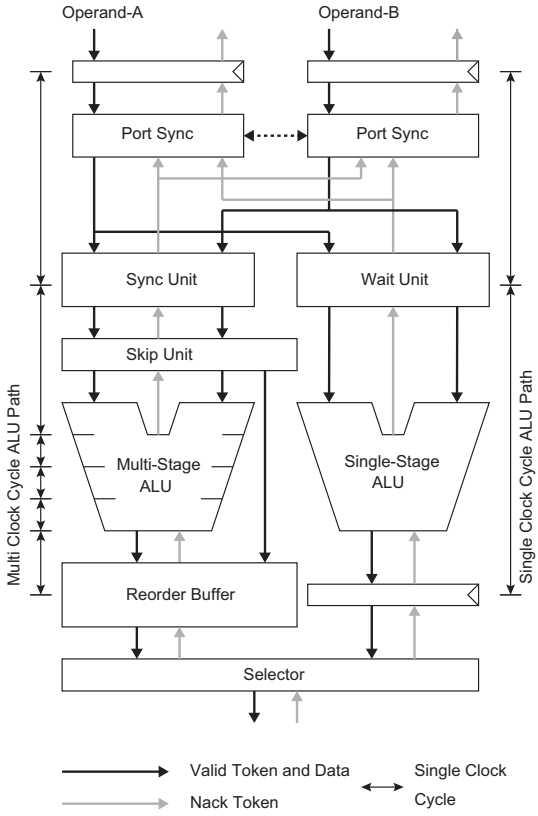

Figure 5.10: ALU with Skip Extension

5.4.3 MFA Unit

The MFA unit consists of several modules. One module is used for the snooping and captures the configuration data for storing. The module sends the configuration data to Ld_CRAM and St_CRAM under the control of the MFA controller.

The MFA controller manages not only the snooping but also detecting the MFA value. In addition, the controller decides to work on the calculation of the actual block length after the restoration.

When loading data words through Ld_CRAM for the restoring, the data words are bypassed to the St_CRAM. The storing is done only when a mismatch is occurred by between the MFA value and the loaded data value. The storing also involves storing its index.

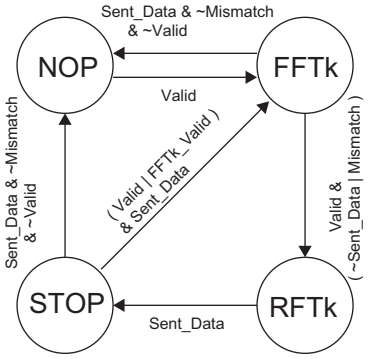

Figure 5.11: Sync Control FSM

5.5 Extension in PE

The previous section explained about RE side extension. PE must also be tested to do the skipping. This idea can be applied to series of a multi-stage pipelines. Figure shows the entire datapath consisting of a single-stage ALU and multi-stage ALU. The multi-stage ALU has a skip unit and a reorder buffer.

5.5.1 Sync Unit

In order to synchronize the operation on ALU with appropriate source data with the compressed operation, we propose a sync unit that manages to send source operands corresponding index value, synchronizes the source operands when the indices are not the same and sends calculated new shared value at first when skipping is possible. The sync unit is located before the ALU.

5.5.2 Skip Unit

Every multi-stage pipeline must detect its skipping and or pass-through. The skip unit does not send the source operands to the pipeline when the skipping and or the pass-through is possible. In addition, the unit sends data words having a value for skipping and or pass-through.

For example, in the case of skipping on a multiplication, multiplication with zero that is not shared data value also has opportunities to skip. Then, the operation is skipped, sources are removed from the pipeline, and a zero value is sent to a reorder buffer. Multiplication with one that is not shared data value also has opportunities for the pass-through. The other non-zero data word of the source operand is sent to the re-order buffer. In the case of addition with zero that is not shared data value also has opportunities for the pass through the pipeline. Then, another source operand data word is sent to re-order buffer.

5.5.3 Re-order Buffer

The multi-stage pipeline can send results out-of-order when a skipping and or a pass-through is detected. The compressed operation must guarantee an in-order arrangement (having increasing order in the indices) in the result data block. In order to support the in-order sending from the ALU, the re-order buffer is used in the ALU. The buffer is located after the ALU.

The re-order buffer is equivalent to the one used in an out-of-order execution in modern processors. It has a buffer unit and a tag buffer. The buffer unit has write- and read-pointers to store in and read from buffer memory. The write pointer is stored in the tag buffer when source operands are synchronized (fired by a valid token). When a data word arrives from the pipeline, a tag (write pointer) is read from the tag buffer, and used for storing the arrived data word to the appropriate address in the buffer. Sending data words from the buffer unit uses the read pointer.

5.6 Summary

Array data has opportunities to skip its operation(s). We use a series of indices that is increase-order. The index addresses a particular data word in a data block. Removing redundant data words for the data block makes a small footprint.

If indices for source operands are matched, then the operation for these operands must be performed. Otherwise, one source should be performed in the future, and operation for source operand(s) having the smallest index value must be performed. Thus syn-

chronization for indices is equivalent to computation operation and issuing a nack token.

In addition, we also prepared a skipping unit to skip or a pass-through of the operations. The skipping supports not only an operation using shared data but also some constant value defined by the ALU. For example, multiplication can have zero and one constant for skipping and pass-through, respectively.

Chapter 6

External Access Subsystem

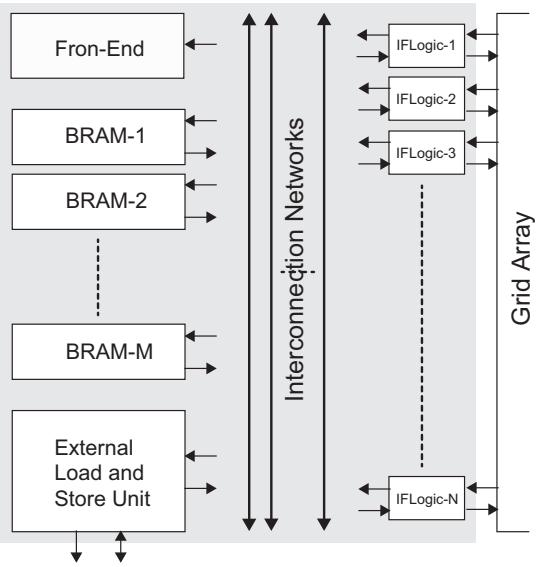

Figure 6.1: ERAM: External Access Subsystem

Previous chapters explained a core part of Electron Nest architecture. In order to execute with external memory, we need an external access subsystem. This chapter explains the subsystem called ERAM.

6.1 Block RAM and Front-End

Figure6.1 shows the ERAM architecture. There are global buffers, interconnection networks, interface logic (IFLogic), and front-end units. The global buffer stores coarse-grained data blocks. The data blocks in the buffer are used to reuse in order to suppress external memory accesses. One interconnection network connects between one element on an edge of the grid array and a buffer, between buffers, between one element and external memory, and between a buffer and external memory. IFLogic severe connection between an element on an edge of the grid array and interconnection networks. The front-end unit works for the connections.

6.2 Front-End Unit

Figure shows a block diagram of the front-end unit. It consists of a FanIn tree unit, an instruction controller, a decoder logic, a rename unit, a port map unit, and a commit unit.

The FanIn tree unit selects one of requests coming from an external load unit and grid array. The FanIn unit consists of FanIn links. The front-end unit is triggered by an acquirement token coming from the FanIn tree unit.

The instruction controller unit feeds a header of a message, and also feeds routing data to connect through the interconnection network. In addition, the controller serves a configuration data for BRAM to store or load, thus, the controller also feeds the attribute word and BRAM's configuration data. The decode logic decodes the fetched instruction. The sets of decoded signals are sent to the rename unit.

The connection is steered by identification (ID) called port-ID. The rename unit renames a destina-

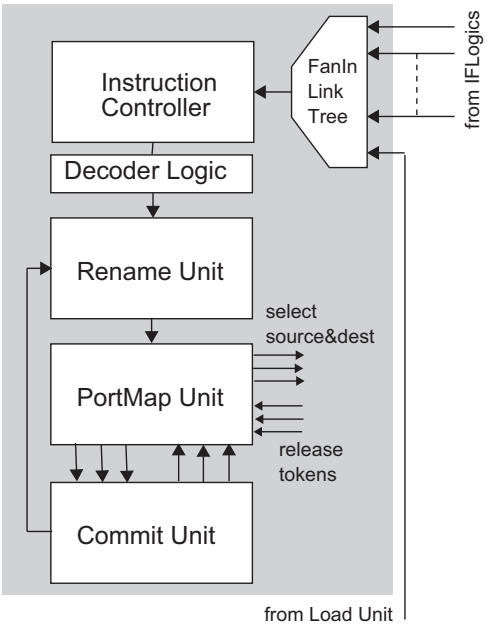

Figure 6.2: Front-End Unit

tion port-ID that is defined in the instruction-set as an architecture-dependent ID with an actual port-ID. This is equivalent to a register renaming on the register file and its register rename unit in traditional microprocessors, which decouples instruction-set architecture and actual hardware configuration. Therefore, our subsystem can have multiple global buffers that the number of global buffers can exceed the maximum numbers defined in instruction-set. In addition, the rename unit resolves hazards, of RAW (read-after-write), WAR (write-after-read) and WAW (write-after-write) hazards. Thus, the rename unit provides chances to access in parallel.

The port-map unit generates a set of select signals for destination and source ports. The signal is used for gating one interconnection network for the destination and source. The port-map unit receives release tokens from source elements and requests releasing of the connection to the commit unit.

The commit unit receives the release requests and serves in-order releases. The commit unit is equivalent to the reorder buffer used in traditional out-of-order microprocessors. The commit unit notifies the commitments to the rename unit and notified entries do releasing the maintaining. At the same time, the commit unit acknowledges to the port-map unit that releases the map entries.

6.2.1 Instruction Controller

The instruction connects among the grid array, global buffers, and load and store unit for external memory access. One request coming from the load unit or the grid array serves a set of connections through the FanIn tree unit. The set consists of several instructions.

6.2.2 Instruction-Set and Decoder Logic

The instruction-set architecture is very simple because that represents one connection. A port on every I/O has a unique ID called a port-ID. Bit-field in the instruction is separated into an opcode, a destination ID, and a source ID. Therefore, there is a tiny encoding logic circuit only for the opcode.

6.2.3 Rename Unit

The rename unit serves to rename the port-ID from architecture to implementation. Thus, the implemented EN can have many global buffers greater than the maximum number defined in instruction-set. In addition, the renaming removes RAW (read-after-write), WAR (write-after-read) and WAW (write-after-write) hazards. Therefore, the rename unit improves the number of connections.

6.2.4 Port Map Unit

The port map unit serves to map between a port-ID and an actual interconnection network, thus generating select signals for source and destination elements. After storing the mapping flag in a table, the flag is not updated until receiving a release token from a source element.

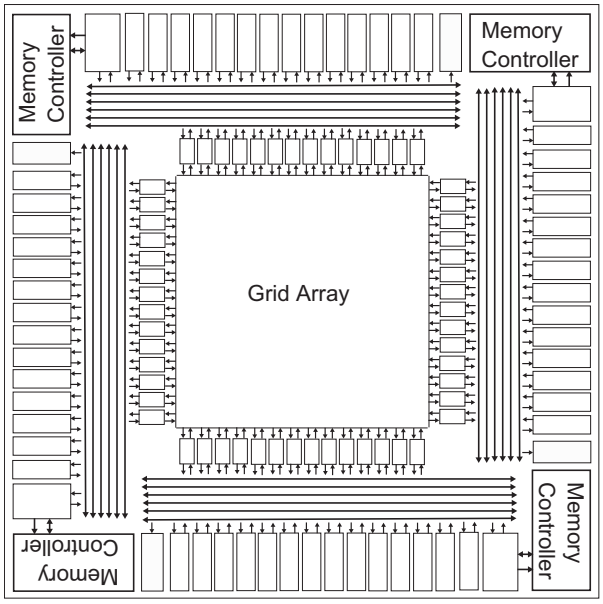

Figure 6.3: Single Chip Electron Nest

6.2.5 Commit Unit

The commit units serve in-order retirement of instructions. The unit consists of a ring buffer controller and a buffer-like set of registers. The controller addresses a head and a tail of the register file. Commit requests coming from a port map unit are stored in the register file, and tail entry has priority for the commitment. The entry notifies its commitment to the rename unit, and at the same time, sends acknowledgement to the port map unit.

6.3 Single Chip Electron Nest

Figure6.3 shows a single-chip Electron Nest system. The grid array is the center of the chip. Four ERAM subsystems surround the grid array.