Chapter 6
Hardware Algorithms

Kentaro Sano and Hiroki Nakahara

Abstract Just implementing with hardware is almost nothing to contribute to achieve high performance. The performance of FPGA computing is depends on how to use efficient hardware algorithms for the target application. This chapter introduces various types of hardware algorithms useful for FPGA implementation. First, pipelining is the most popularly used technique. Recently, it is often automatically formed with HLS design tool. Then, general parallel processing techniques are introduced along Flynn's classic taxonomy. Systolic algorithms and data-flow models are also classic methods researched in 1970s' and 1980s', but they have been practically used after large-scale FPGAs are available for computation. Then, stream processing, simple but powerful framework, is introduced with a practical example. Next, cellular automaton, hardware sorting and pattern matching which are important in network processing a killer application of FPGAs are introduced.

Keywords Pipeline processing · SIMD processing · Systolic algorithm
Data-flow machines · Streaming processing · Cellular automaton · Hardware sorting · Pattern matching

A hardware algorithm is a procedure suitable for hardware implementation and the target hardware model. This chapter presents an outline of several hardware algorithms used for processing implementation in hardware, with specific emphasis on parallelism, control, and data-flow of processing.

K. Sano (✉)
RIKEN, Kobe, Japan
e-mail: kentaro.sano@riken.jp

H. Nakahara
Tokyo Institute of Technology, Tokyo, Japan
e-mail: nakahara@ict.e.titech.ac.jp

© Springer Nature Singapore Pte Ltd. 2018

H. Amano (ed.), *Principles and Structures of FPGAs*,
https://doi.org/10.1007/978-981-13-0824-6_6

6.1 Pipelining

6.1.1 Principle of Pipelining

Pipelining is a technique for speeding up many processing iterations done continuously. The flow production at a production plant is a typical example. Figure 6.1 presents the pipelining concept. Figure 6.1a depicts the non-pipelining case where processing iteration 2 is done sequentially after the completion of iteration 1. With the pipelining shown in Fig. 6.1b, we divide a processing iteration into n stages of uniform proportion. The processing iterations start without waiting for the completion of all stages of a preceding processing iteration. Figure 6.1b portrays an example of five-stage pipelining in which each processing iteration has five stages: n = 5. In the non-pipelining case, each processing iteration is completed within a time length L. In pipelining, a processing iteration is started in each time length of L/n after the processing of iteration 1 has finished. Therefore, throughput speedup is achieved by five times at most, where five is the number of processing iterations per unit time. In the five-stage example shown in Fig. 6.1b, six processing iterations are done in two time lengths of the non-pipelining case. Parallel processing with different parts of the entire sequential process is the principle of pipelining, which is understood as five stages processed in parallel when stage 5 of processing iteration 1 is started [1, 2].

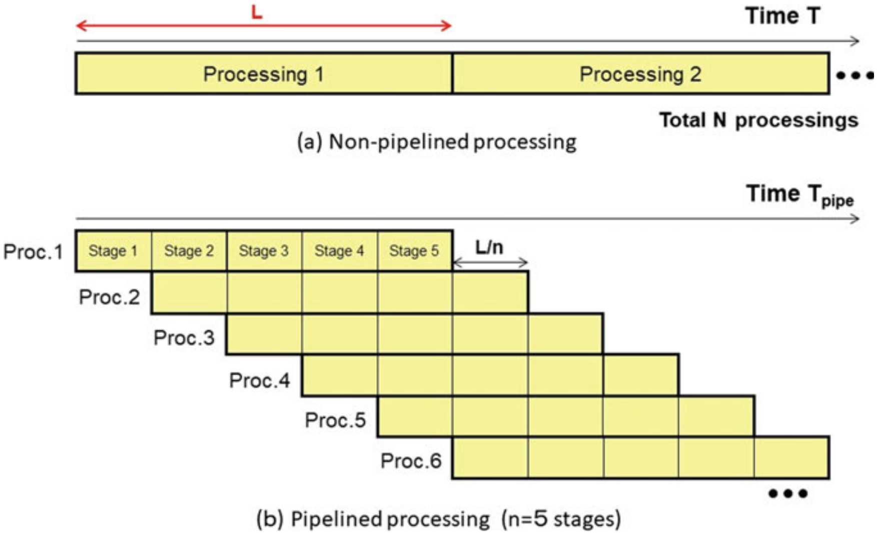


Fig. 6.1 Overview of pipelining

6.1.2 Performance Improvement by Pipelining

Actually, n stage pipelining does not necessarily mean five times speedup. Here, we develop the expression of the speedup factor, a measurement of performance improvement using pipelining, for a time length of one processing iteration L, N processing iterations to be done, and n stages in a single processing iteration.

Figure 6.1a shows that N processing iterations by a non-pipelining operation require $T(N) = LN$ time length for completion. We can derive the time length $T_{pipe}(N)$ for completion of N processing iterations for pipelining with n stages, as described below. First, the completion of processing iteration 1 requires L time length. However, the next processing iteration finishes L/n later after the completion of processing iteration 1. Consequently, $N - 1$ processing iterations, except for processing iteration 1, complete at intervals of L/n. These N operations end after $T_{pipe}(N) = L + (N - 1)L/n = (n + N - 1)L/n$ time length.

The speedup factor by pipelining described above, $S_{pipe}(N)$, is the ratio of $T(N)$ to $T_{pipe}(N)$, and can be calculated as:

$$S_{pipe}(N) = \frac{T(N)}{T_{pipe}(N)} = \frac{nN}{n + N - 1} = \frac{n}{1 + \frac{n-1}{N}}$$

If $n \ll N$, then $S_{pipe}(N) \cong n$ and the speedup factor over non-pipelining is n: the number of stages. However, this just means that n times higher throughput can be achieved while the latency of each processing iteration is not shortened. Although n times improvement of performance means the reduction of the whole time length for N processing iterations, it does not bring about the shortening of the time length for each processing iteration. If this is compared to the flow production in the automotive industry, then the car production per day (throughput) might increase by dividing a processing iteration into segments. However, the time between the car order and the delivery of the parts of that car to factories and the completion of production of that car (latency) does not change.

If the number of stages, n, is not much greater than that of processing iterations N, then the speedup by pipelining remains much smaller than n. For pipelining with $n = 6$ and $N = 5$, for example, $S_{pipe}(5)$ is $6/(1 + 1) = 3$ and the time of processing iteration is reduced to just one-third. Considering that the maximum speedup by pipelining with n stages is n, the pipelining efficiency, which is the percentage of the actually achieved speedup to the maximum, is given as:

$$E_{pipe}(n, N) = \frac{S_{pipe}(N)}{n} = \frac{1}{1 + \frac{n-1}{N}} = \frac{N}{N + n - 1}$$

For the example presented above, $E_{pipe}(6, 5)$ is $\frac{5}{5+6-1} = 0.5$. This means that the speeding up is just 50% at maximum. A long time length of insufficient parallelism has led to this small decrease in efficiency. In Fig. 6.1b, only one process is executed at the first stage of iteration 1. When the fifth processing stage of iteration 1 starts, the number of parallel processing iterations never attains 5, the maximum degree of parallelism. Consequently, at the beginning of pipeline processing, there exists a prologue period where perfect parallelism is not achieved. Quite similarly, at the end of pipeline processing, there exists an epilogue period during which the degree of parallelism decreases step by step. Perfect elimination of the prologue and epilogue periods is impossible. However, if the number of processing iterations, N, is sufficiently large, then the lengths of the prologue and epilogue periods are diminished in comparison to the total processing time. Consequently, the speedup factor approaches the maximum value of n. In contrast, if N is small, then the effects of prologue and epilogue periods are greater in a relative sense. Therefore, the speedup factor is smaller.

For other several reasons not mentioned above, hardware pipelining can intrinsically block the performance improvement. Thus, a lot of attention must be devoted during the pipelining design. The hardware configuration for pipelining is depicted in Fig. 6.2. Figure 6.2a presents the hardware configuration for a non-pipelining-based system, where the total processing is implemented by one combinational logic circuit. When the register of the preceding processing iteration is updated at the rising edge of the clock signal, a new data is outputted to the combinational logic circuit after the propagation delay of the register. Inputted data propagates through the combinational logic circuit. Then, the processed data arrive at the register of the next processing iteration after a certain delay in the critical path. Here, the critical path is that which provides the maximum delay in the circuit. After the period of setup time, which is necessary to secure correct latching of the arrived data by flip-flop, the processed data are written in the register by inputting a clock signal. Thereby, the processing iteration terminates. Consequently, one cycle time, which is equal to the time interval between two successive clock signal inputs, must be longer than the sum of the propagation delay, the delay in the critical path of the combinational logic circuit, and the setup time. This sum represents the upper bound of the maximum operation frequency.

Pipelining a circuit boosts the throughput by increasing the maximum operation frequency. Figure 6.2b depicts an example of a circuit with four pipeline stages: $n = 4$. A circuit can be divided into stages by inserting pipeline registers. With them, the combinational logic circuits on which the data must propagate are shortened. Even the critical paths of combinational logic circuit could be perfectly divided into uniform circuits, the cycle time might not be shortened to $1/n$ because of the existence of clock skew, which is the misalignment of clock signals supplied to each register, and because of the propagation delay and setup time in pipeline registers. Furthermore, dividing a combinational logic circuit into n uniform stages is sometimes impossible. Such a case is illustrated by the third pipeline stage in Fig. 6.2b. In this situation, one stage usually has a longer delay time than others and becomes the critical path, which happens quite often. Consequently, $n = 4$ might not increase the operation frequency by four times because the maximum delay is longer than one-fourth.

These difficulties generally become more deleterious with an increasing number of stages. Although an operation frequency improvement can be obtained by increasing the number of stages of shallow pipelining with a few stages, fine divisions such as

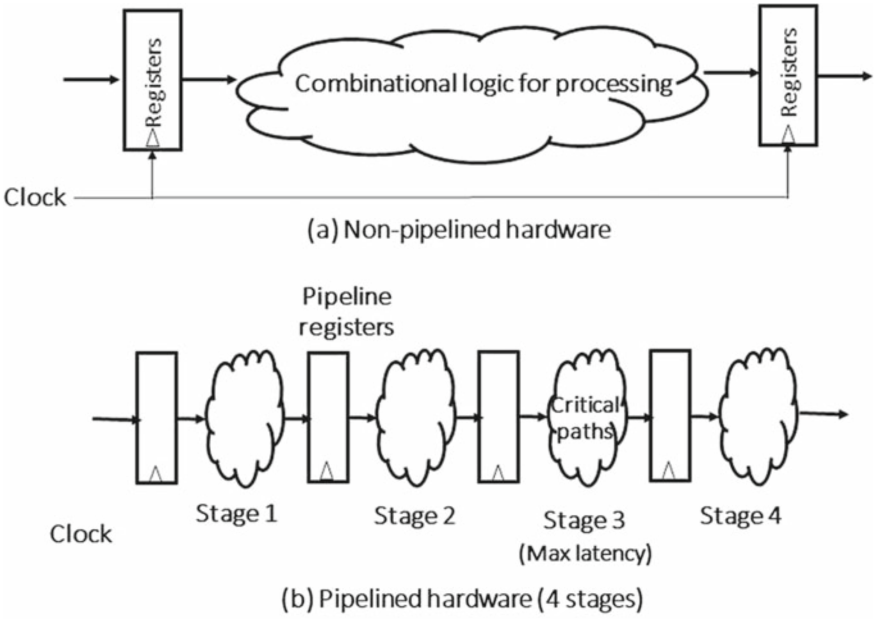


Fig. 6.2 Hardware structure for pipelining

dozens or hundreds might halt the operation frequency augmentation and might even degrade the performance due to clock skew and other factors. However, adding few pipeline stages instead of finely dividing the already existing ones is different. Actually, the increase in the entire circuit's latency by pipelining when compared to non-pipelining-based systems should be carefully investigated. Such delay can be caused by overhead related to pipeline registers or non-uniform staging, as shown in Fig. 6.2b.

6.2 Parallel Processing and Flynn's Taxonomy

6.2.1 Flynn's Taxonomy

To design high-performance hardware, we should consider processing parallelism. The taxonomy of architectures which Flynn proposed in 1965 is useful for considering parallelism for hardware [2, 3]. Hereafter, we refer to this as Flynn's taxonomy. Flynn's taxonomy classifies general-purpose computer architectures in terms of the concurrency degree in an instruction stream for control and a data stream to be processed (Fig. 6.3). It includes Single Instruction stream Single Data stream (SISD), Single Instruction stream Multiple Data stream (SIMD), Multiple Instruction stream

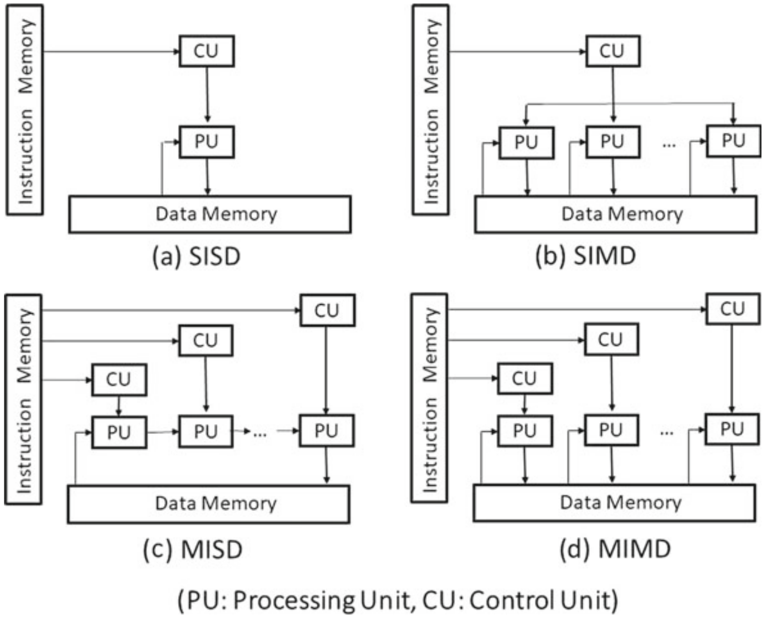


Fig. 6.3 Flynn's taxonomy

Single Data stream (MISD), and the Multiple Instruction stream Multiple Data stream (MIMD). Although this taxonomy is oriented to architectures of general-purpose processors that execute a sequence of instructions intrinsically, it is useful for classifying more general architectures of parallel processing hardware if we recognize the instruction stream as control in a broader sense.

In Flynn's taxonomy, computer architecture comprises the four components of the processing unit (PU), control unit (CU), data memory, and instruction memory. In SISD shown in Fig. 6.3a, one CU controls one PU based on the instruction stream read from the instruction memory. PU executes processing for a single data stream read from the data memory based on the controls directed by CU. Consequently, SISD does not perform parallel processing and represents the architecture of general and sequential processor without parallel processing. In the following, the other three items in the taxonomy are explained.

6.2.2 SIMD Architecture

In SIMD in Fig. 6.3b, a single CU reads out an instruction stream and controls multiple PUs simultaneously. Each PU executes common processing based on common controls but for different instruction streams. Consequently, SIMD is an architecture making use of data parallelism. The data memory can be accessed by all the PUs as local memories of the PUs or a single-shared memory common to all the memory. Because the SIMD architecture is suitable to process numerous data synchronously with a single sequence of instructions, it is used as the designated processor for image processing.

In addition, a microprocessor can incorporate SIMD instructions to provide itself with the function of parallel data processing. For example, Intel Corp., aiming at speeding up 3D graphics, designed a microprocessor, the Pentium MMX, with SIMD-type extended instructions. It was commercially produced in 1997 [4, 5]. The MMX Pentium can perform four 16-bit integer operations simultaneously based on one SIMD instruction. Furthermore, AMD Corp. introduced a new product of K6-2 processors equipped with 3DNow! Technology, which provides SIMD-type extended instructions for floating-point operations in 1998. Later, Intel processors were augmented with Streaming SIMD Extensions (SSE), an SIMD-type instruction set for floating-point operation. Then, Pentium III included SSE extended instructions and Pentium IV received extended instructions SSE2 and SSE3. Currently available microprocessors have been augmented with instructions of 128-bit integers and double precision floating-point operations in addition to other operations for compression of video images, as outlined above. They are prevailing as mainstream microprocessors. Now it is indispensable to make use of these SIMD-type extended instructions to take full advantage of the operational performance of microprocessors [5].

6.2.3 MISD Architecture

In the MISDs shown in Fig. 6.3c, multiple CUs read out instruction streams that differ from each other and which control multiple PUs. Although each PU works based on different controls, MISD processes a single data stream successively, regarding it as a whole. It is difficult to find a commercial product of this type in the market. A coarse-grained pipeline, in which in-line PUs work as stages of the pipeline and one provides each stage with different controls, might seem to be an MISD architecture. Because we recognize that CUs provide different functions to PUs in performing parallel processing with MISD, it earns the designation of architecture for functional parallelism. Application-specific hardware such as an image processor array that executes different processes of conversion of pixel values, edge detection, cluster classification, and others form each stage of pipeline corresponds to MISD-type architecture if each processing iteration is controlled by an instruction stream [6, 7].

6.2.4 MIMD Architecture

Regarding MIMD in Fig. 6.3d, multiple CUs read out instruction streams that differ from each other and which control multiple PUs independently. Different from MSID, each PU performs parallel processing based on different controls and for different data streams. Accordingly, MMID has an architecture simultaneously accommodating data parallelism and functional parallelism. It might perform different processes for multiple data based on different instruction streams. A tightly coupled multiprocessor like a symmetric multiprocessor (SMP), for which multiple processors and multiple processor-cores are connected on a common memory system, is an example of MIMD-type architectures. A cluster-type computer in which computation nodes made of memory and microprocessor are connected by the interconnection network is another example of MIMD-type architectures.

6.3 Systolic Algorithm

6.3.1 Systolic Algorithm and Systolic Array

Systolic algorithm is a general name for algorithms in which a systolic arrays [8, 9] are used to realize parallel processing. A systolic array is a regular array of many processing elements (PEs) for simple operations. It has the following characteristics:

1. PEs are arrayed in a regular fashion: they have the same structure or a few different structures.
2. The neighboring PEs are connected allowing the data movement to be localized with the connection. If a bus connection is used in addition to the local connection, then the array is designated as a semi-systolic array [8].
3. PE repeats simple operations and related data exchange.
4. All PEs perform operations in synchronization with a single clock signal.

Each PE performs its own operation in synchronization with the data exchange between neighboring PEs. Data to be computed flow into an array periodically, and the pipeline and parallel processing are performed while the data propagate in the array. Operations in the PE and the data stream caused by the data exchange between neighboring PEs resemble a bloodstream driven by the rhythmic systolic movement of the heart, hence the name systolic array. A PE is also designated as a cell.

Because the systolic array can scale the performance according to the array size by arranging PEs with simple structures and localized data movement, it is suitable for the implementation in an integrated circuit. Many applications were proposed for systolic arrays in 1980s and 1990s. Figure 6.4 portrays typical systolic arrays and systolic algorithms [9]. Systolic arrays have three types: 1D arrays having a linear array, 2D arrays having a lattice-like array, or a tree structure array with a tree-like connection. Many systolic algorithms for these arrays have been suggested, including signal processing, matrix operation, sorting, image processing, stencil calculation, and calculation in fluid dynamics.

Although the initial systolic array assumed hardwired implementation of fixed structures and functions, a general-purpose systolic array with a programmable or reconfigurable structure was proposed later [10]. Classification of systolic arrays in

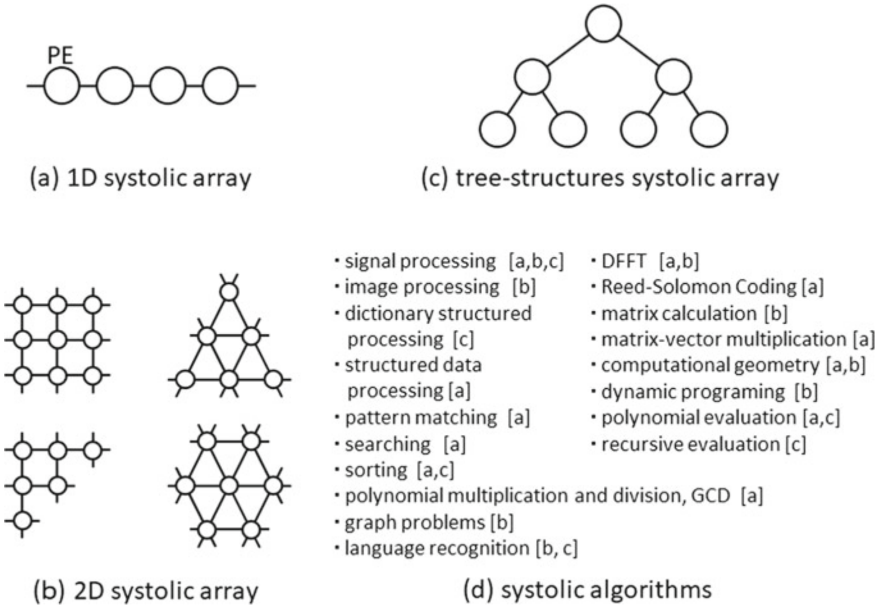

PE
(a) 1D systolic array
(c) tree-structures systolic array
• signal processing [a,b,c]
• image processing [b]
• dictionary structured processing [c]
• structured data processing [a]
• pattern matching [a]
• searching [a]
• sorting [a,c]
• polynomial multiplication and division, GCD [a]
• graph problems [b]
• language recognition [b, c]
• DFFT [a,b]
• Reed-Solomon Coding [a]
• matrix calculation [b]
• matrix-vector multiplication [a]
• computational geometry [a,b]
• dynamic programing [b]
• polynomial evaluation [a,c]
• recursive evaluation [c]
(b) 2D systolic array
(d) systolic algorithms

Fig. 6.4 Representative systolic arrays and systolic algorithms [9]

terms of general versatility is shown in Table 6.1 [10]. In this table, "Programmable" denotes the capability to dynamically change the function of the circuit by programming the fixed circuit. The "Reconfigurable" is the capability to statically change the circuit function by circuit reconfiguration. With the increasing scale of the systolic array, high-frequency operations with globally maintained synchronization become difficult sue to the propagation delay of the clock signal and other factors. Kung proposed the wave-front array, introducing data-flow to a systolic array to cope with this difficulty [11]. In his method, a PE, designed as an asynchronous circuit, operates with its own speed without the synchronization to a single clock signal. Furthermore, the data exchange between neighboring PEs is done using the handshake method.

Examples of systolic arrays and algorithms for 1D and 2D systolic arrays are introduced in the following sections.

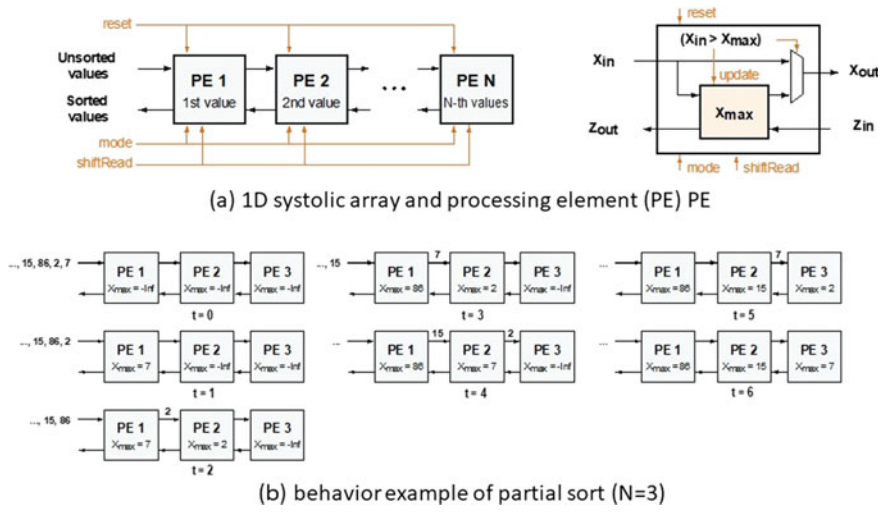
6.3.2 Partial Sorting by 1D Systolic Array

An operation of rearranging a given data row according to a given order is referred to as sorting. Sorting operations are important and are used in many applications. Here, we introduce a systolic algorithm that rearranges a given data row of n numerical elements in the descending order and returns the upper N data. Figure 6.5a depicts a 1D systolic array and its PEs, which partially rearrange the upper N data [12].

Table 6.1 Classification of systolic arrays for versatility [10]

Class	General-purpose									Special-purpose
Type	Programmable			Reconfigurable			Hybrid			Hardwired
Organization	SIMD or MIMD			VFIMD						VFIMD
Topology	Programmable		Fixed	Reconfigurable		Fixed	Hybrid		Fixed	Fixed
Intercon. Network	Static	Dynamic	Fixed	Static	Dynamic	Fixed	Static	Dynamic	Fixed	Fixed
Dimensions	n-dimensional ($n > 2$ is rare due to complexity)									n-dimensional

VFIMD Very-few-instruction stream, multiple data streams



(a) 1D systolic array and processing element (PE) PE

(b) behavior example of partial sort (N=3)

Fig. 6.5 Systolic algorithm of partial sort

Each of N PEs arrayed in the one-dimensional row has a register keeping the temporary maximum value, Xmax. Furthermore, if the input Xin is greater than Xmax, then Xmax is replaced with Xin. Consequently, the temporary maximum value is updated. If the update is made, the former temporary maximum value is sent to the right PE. If not, then Xin is sent to the right PE. When the last input data are sent to the Nth PE repeating the procedure described above, the upper maxima N are kept in the registers beginning with the left-end PE. An example of these operations is illustrated in Fig. 6.5b. Partial sorting of n data takes (N + n – 1) steps using a systolic array with N PEs.

Figure 6.5a shows that three control signals of reset, mode, and shiftRead are inputted to the systolic array. When the reset signal is asserted, the temporary maximum numbers in all PEs are reset to the possible maximum negative value. The mode control signal specifies a request: either the sorting operation or the reading out of the sorting result. In the former, 1 is inputted, whereas 0 is inputted in the latter. The shiftRead control signal asks to read out the result of sorting one by one in the descending order. As presented at t = 6 in Fig. 6.5b, the maximum values are lined up in a descending order starting with the left-end PE which has input and output ports. These values are read out by a systolic array used as the shift register through Zout and Zin connections. The control signal for shift is shiftRead.

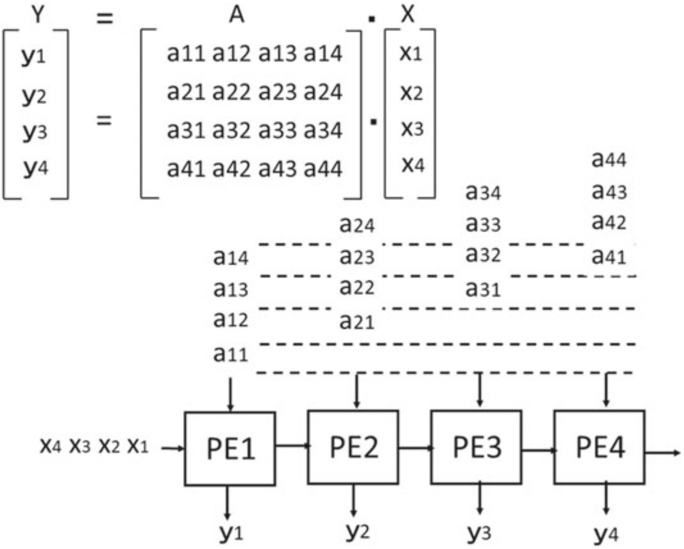


Fig. 6.6 Systolic algorithm of matrix-vector multiplication

6.3.3 Vector Product of Matrices by 1D Systolic Array

A 1D systolic array can perform the operation of vector product **Y = AX**. The operation of an $N \times N$ matrix requires N PEs. The systolic algorithm used for vector product of $N = 4$ matrices is depicted in Fig. 6.6. In PEs arrayed in the 1D row, the operation proceeds as follows: elements in **X** enter the left-end PE, whereas elements in matrix **A** enter each PE from above, both successively. Each PE has a register y_i to keep a temporary value of the element in the **Y** vector in addition to input x element of vector **X** and an element of matrix A. All PEs execute the calculation of $y_i = y_i + ax$ at each step and output x to the right neighbored PE.

At the beginning of the operation, y_i of each register is initialized to 0. Then, $y_1 = 0 + a_{11}x_1$ is calculated in PE1. In the next step, $y_1 = y1 + a_{12}x_2$ is executed in PE1 and $y_2 = 0 + a_{21}x_1$ in PE2, both being done in parallel. The inputs to PE1 through PE4 are done in a manner where the input to a PE occurs one step later than that to the left neighboring PE. This sends the data to the PE in a timely manner. When these operations are repeated until PE4 keeps the last matrix element, all elements of vector **Y**, from y_1 to y_4, are stored in the PE array. The columns of matrix **A** are inputted one element after another. Therefore, the total steps of operations become $2N - 1$.

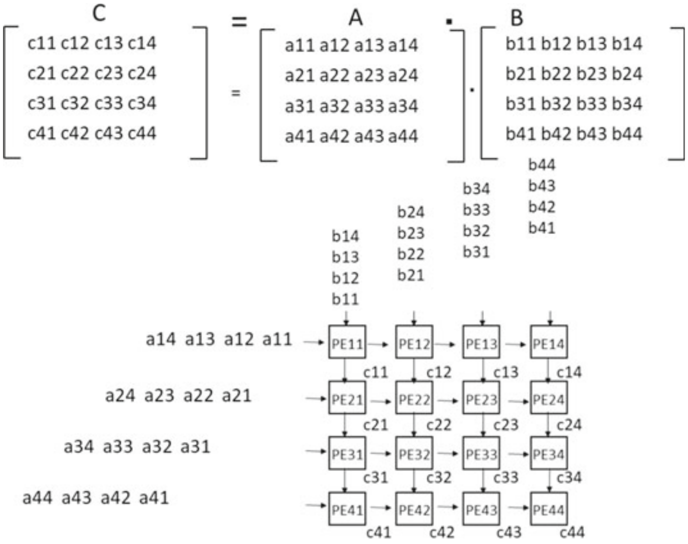


Fig. 6.7 Systolic algorithm of product of matrices

6.3.4 Product of Matrices by 2D Systolic Array

By extending the 1D systolic array discussed in the previous section to a 2D systolic array (with a lattice of PEs), it is possible to perform product operations of two matrices: **C = AB**. An $N \times N$ matrix multiplication requires an $N \times N$ systolic array with N^2 PEs. Figure 6.7 is an example of such an operation with $N = 4$. As in the vector product of matrices done by 1D systolic array, row and column elements are input from left and above top, shifting rows and columns. The function of a PE is the same as that in the previous section. All the internal registers are to be initialized to 0 at the beginning of the operation. When the last matrix elements of PE_{44}, a_{44}, and b_{44}, are input, all the resulting elements of Matrix C are in all the PEs. The number of required steps is $3N - 2$.

6.3.5 Programmable Systolic Array for Stencil Computation and Fluid Simulation

Although the examples introduced in the previous sections were those of simple PE operations, programmable systolic arrays oriented to many stencil computations, and applications for computational fluid dynamics (CFD) and others have been proposed [13–15].

The structure of a systolic computational-memory array and its PE designed for stencil computation is shown in Fig. 6.8. This array has a 2D scheme of vertically and

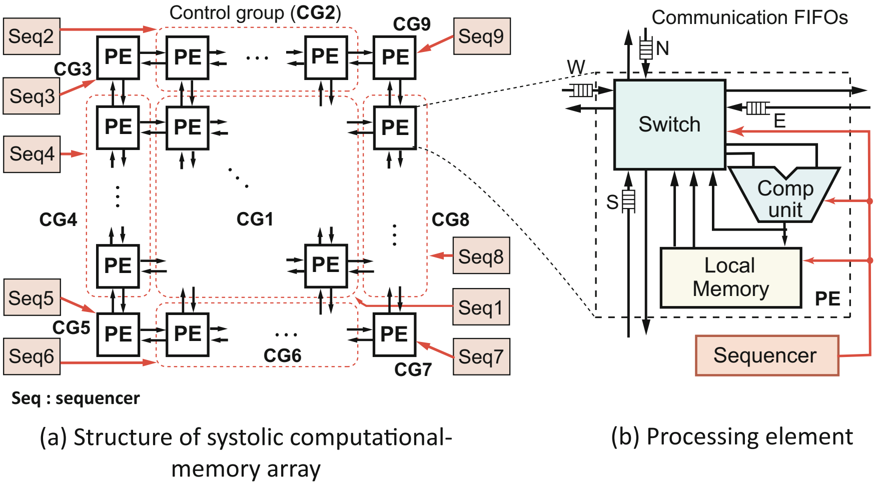


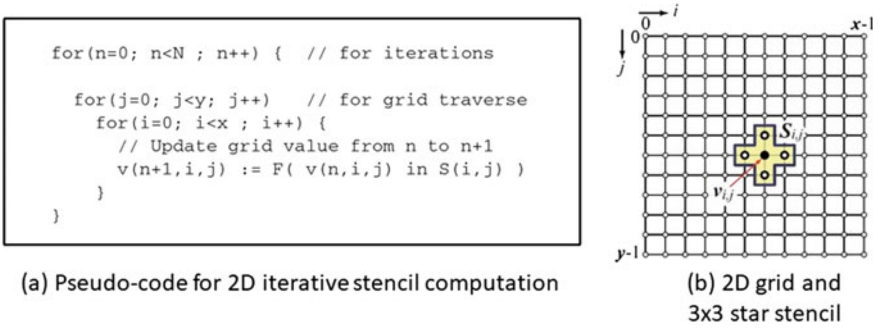
Fig. 6.8 Systolic computational-memory array and processing element

horizontally connected PEs. As shown in Fig. 6.8b, a PE comprises a computation unit, a local memory, a switch to send data to all directions of (W, E, S, and N) and a sequencer to control these elements using a microprogram. Because each PE has a large local memory and because the whole array is a memory not only for operations but also for data storage, this systolic array is considered as a computational memory. The computation unit can execute floating-point multiplications and additions. The computational sub-lattice data are stored in the local memory. A PE can perform various computations with a microprogram by repeating data read operations from the local memory or the neighboring PEs.

As shown in Fig. 6.8a, the systolic computational-memory array operation is made from many control groups (CGs). PEs in the same CG are controlled by a common sequence. They perform parallel processing of SIMD style. In this example, there are nine CGs inside the 2D array, with four sides and four corners because, in fluid dynamic computation and others, the computation in a regular pattern is done inside, whereas computation of different types is done for the boundary condition.

Figure 6.9 depicts pseudo-codes for stencil computation and an example of a 2D lattice with a 3 × 3 star-stencil computation of. As shown in Fig. 6.9b in stencil computation, an operation for a lattice point is done using data from neighboring points and the data on which the lattice point is renewed. The neighboring domain, where the data is referred, is designated as a stencil. The 3 × 3 star stencil shown in the figure, a fundamental one, is widely used. In 2D operations, the data at all lattice points are renewed after the same operations with the same stencil are performed over the lattice.

Figure 6.9a shows pseudo-codes for the 2D stencil computation. It has a triple loop structure consisting of loops for vertical and horizontal directions, and one for iterating them in a time step n. Function F(), the loop body, represents any



```
for(n=0; n<N ; n++) {   // for iterations

  for(j=0; j<y; j++)    // for grid traverse
    for(i=0; i<x ; i++) {
      // Update grid value from n to n+1
      v(n+1,i,j) := F( v(n,i,j) in S(i,j) )
    }
}
```

(a) Pseudo-code for 2D iterative stencil computation

(b) 2D grid and 3x3 star stencil

Fig. 6.9 2D stencil computation [13–15]

operation using data stored in a stencil. The product-sum operation using a weighting coefficient, shown below, is commonly used as F().

$$v(i, j) := c0 + c1v(i, j) + c2v(i - 1, j) + c3v(i + 1, j) + c4v(i, j - 1) + c5v(i, j + 1)$$

Here ":=" stands for the value update after the operation is written at the right-hand side. The PE in Fig. 6.8b performs the operation shown above for the partial lattice stored in the local memory, by using a microprogram in a sequencer. Sano et al. have derived the stencil algorithm by a fractional-step method for fluid dynamical phenomena. There, they repeated stencil calculations with different coefficients shown above and proved the calculations execution with the systolic computational-memory array. Further details can be found in [13–15].

6.3.6 Data-flow Machine

A data-flow machine [16, 17] is a computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. It does not have a program counter (at least conceptually), and the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable. Figure 6.10 compares the data-flow machine with the von Neumann one. Since the data-flow machine has no instructions, it can eliminate the bottleneck caused by the instruction memory fetch of the modern computer execution time.

Although no commercially successful general-purpose computer hardware has used a data-flow architecture, it has been successfully implemented in specialized hardware such as in digital signal processing, network routing, graphic processing, telemetry, and more recently in data warehousing.

A data-flow machine executes the data-flow graph for a given program as shown in Fig. 6.11. "Fork" copies the given data, "Primitive Operation" outputs the result of

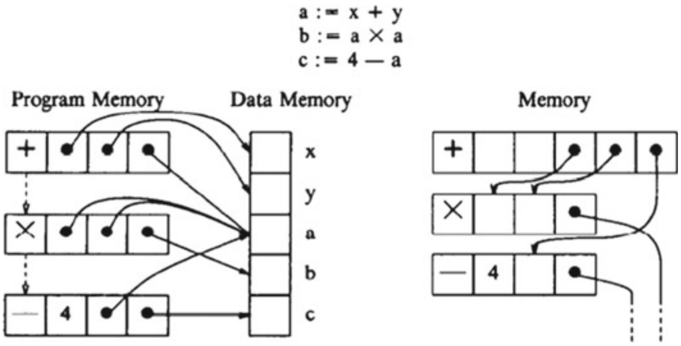


Fig. 6.10 A Comparison Von Neumann machine (Left) with a data flow machine (Referred by [16])

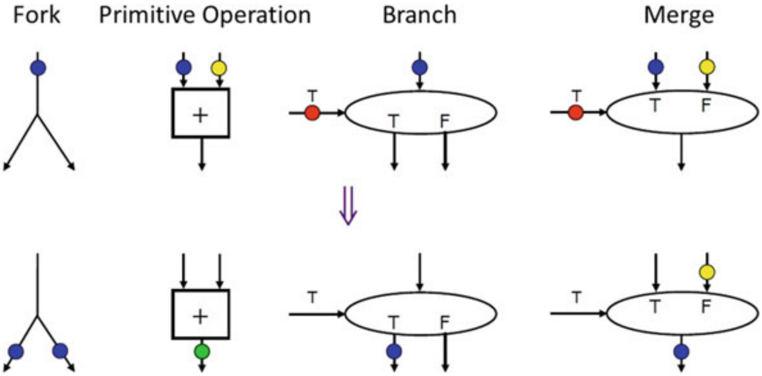


Fig. 6.11 Examples of a data flow node

the executed operation, “Branch” executes the conditional jump corresponding to the signal (True or False), and “Merge” selects the signal corresponding the conditional signal. Figure 6.12 shows the data-flow graph which executes the operations shown in Fig. 6.10. In Fig. 6.12, “◯” denotes the data to be executed, and it is called by the “Token.” Here we show the value of token in the circle. First, the adder starts the operation, since two tokens have their values. Then, it generates the execution result. Next, the latter multiplier and subtractor starts the operation since it has received the input token. In this manner, it is possible to easily know the inherent data parallelism in the program to be processed.

Similar to the von Neumann machine, the data-flow machine can realize the conditional jump and loop operations. Figure 6.13 shows an example of the conditional jump. It can be realized by the Branch and the Merge nodes. When the token arrives at the Branch node, it executes the branch operation, then, it sends the token to the selected operational node. Finally, the Merge node selects the output corresponding conditional signal. Figure 6.14 shows an example of the loop operation. While

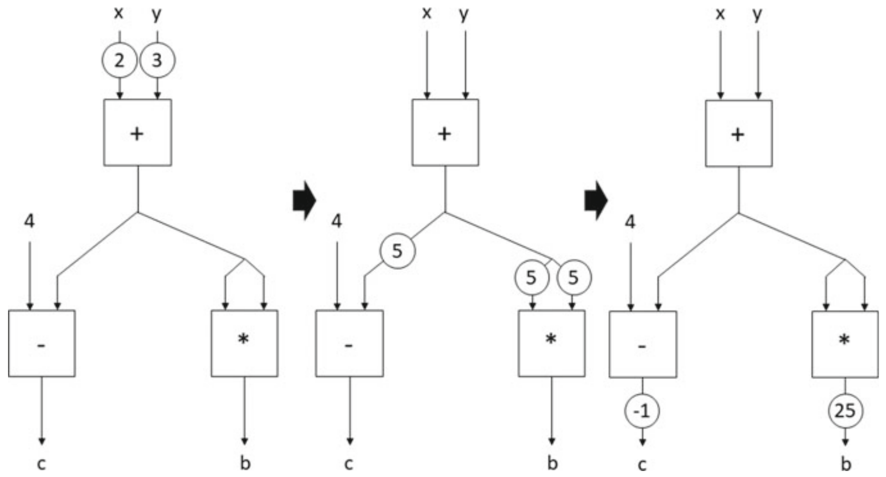
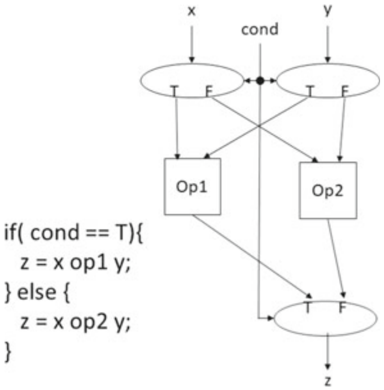


Fig. 6.12 An example of a data flow node

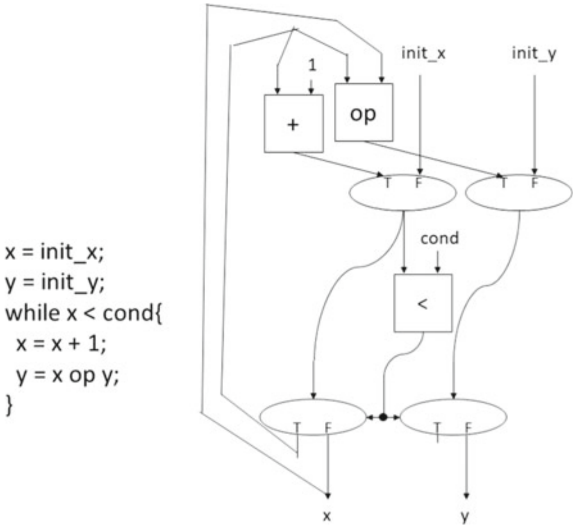
Fig. 6.13 A branch operation for a data flow graph



```
if( cond == T){
  z = x op1 y;
} else {
  z = x op2 y;
}
```

updating the initial value at the Merge node, it repeats the execution node through the Branch when the condition is satisfied. There are two types of data-flow machines that realize programs including loops. One is a static data-driven method, and the other is a dynamic data driven one. The static one expands all the loops and represents them as a flatten data-flow, while the dynamic one shares the executing unit with the processing of the subsequent loops using the data-flow of the loop body. The static data-driven method is based on a concept of a pure-driven method with a static data-flow graph. However, in many cases, since the size of the data-flow graph becomes too large to realize it with a data-flow machine which has a limited hardware resources. On the other hand, in the dynamic data-driven method, execution nodes in the loop body are shared by computing units, such as adders, subtractors, and multipliers. Thus, a control circuit must be provided. Otherwise, if multiple tokens between iterations exist, it is not possible to guarantee the computation result.

Fig. 6.14 A loop operation for a data flow graph



```
x = init_x;
y = init_y;
while x < cond{
  x = x + 1;
  y = x op y;
}
```

Figure 6.13 and 6.14 explains such a scenario. If y is updated before updating x in the loop, the computation result will differ from the expected one.

In the dynamic data-driven method, a tag is attached to the token in order to distinct present/next tokens in the iteration. It is called a tagged token method, or a colored (tokenized) one. By using tagged tokens, operations on tokens with the same tag can be guaranteed.

6.3.7 Static Data-Driven Machine

In the static data-driven method, it is often used to represent the operation and the operand of a node in a mixed manner. In the data-driven machine proposed by Dennis [18], the node information of the data-flow graph has the necessary data for the operation, the storage type of the calculation, and the destination for the storage. By using this node information as a token packet, the data-driven processing is realized. Note that, the token does not have tag information. It focuses on the static data-flow which does not include the loop processing. Figure 6.15 shows a hardware structure and each operand cell for the static data-driven architecture. In Fig. 6.15a, the operand cell stores the above information, and it represents the data-flow graph as an instruction of the data-driven machine in the whole instruction having valid information.

Hereafter, we explain the processing steps. When the operands are complete, the operation packet is sent through the arbitration network (ANET). This packet includes all information of the instruction cell, the operation type, and the storage destination for the result. The operation's result is transformed as a data token through the

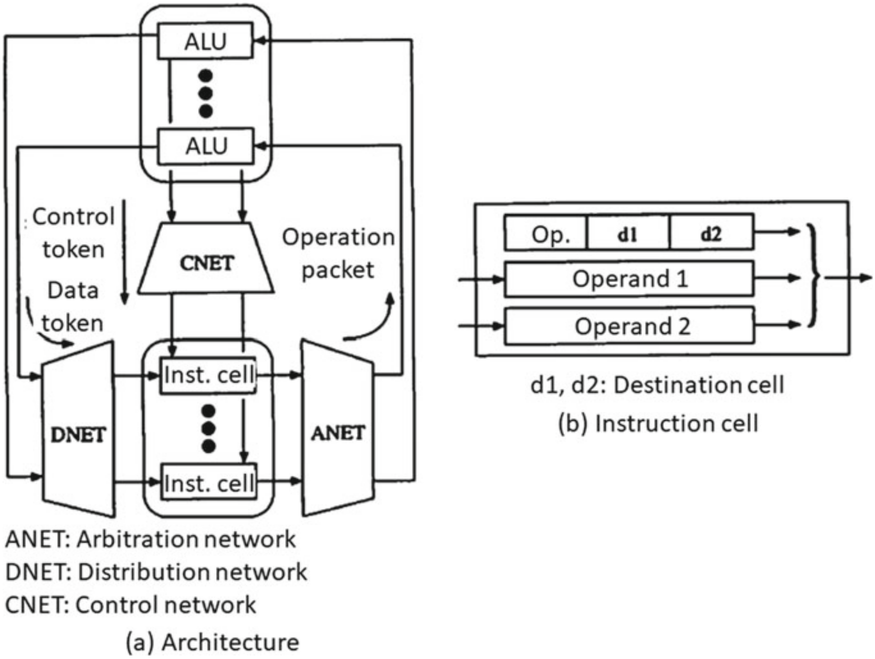


Fig. 6.15 A static data flow machine (Referred by [17])

distribution network (DNET). Then, it is written to the operand part in the instruction according to the storage destination (d1, d2, in Fig. 6.15b). Next, the instruction with the operands sends the operation packet at any time. By following these steps, a series of data driving driven processes is done.

6.3.8 Dynamic Data-Driven Machine

In the dynamic data-driven method, it separately represents the execution unit and the operation of the node. This representation can separate the flow graphs and data. Therefore, there is an advantage that loop processing can be considered by using a tagged token.

As shown in Fig. 6.16a, Arvind proposed a data-driven machine [19], which consists of N PEs with an $N \times N$ crossbar. Figure 6.16b shows the operand for the machine; “op” denotes an operation, “nc” denotes the number of constant to be stored, “nd” denotes the number to destinations, “constant 1” and “constant 2” denote the destination address, “s” denotes a statement number for the destination operand, “p” denotes the input port number for the destination operand, “nt” denotes the number of tokens for the destination operand, and “af” denotes an assignment function to be

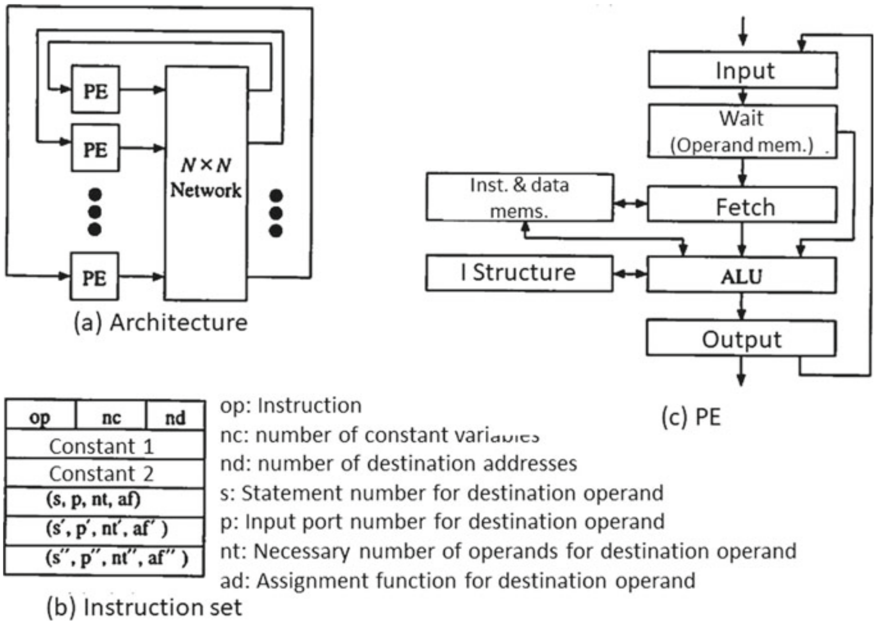

PE
PE
PE
N × N
Network
(a) Architecture
op
nc
nd
Constant 1
Constant 2
(s, p, nt, af)
(s′, p′, nt′, af′)
(s″, p″, nt″, af″)
op: Instruction
nc: number of constant variables
nd: number of destination addresses
s: Statement number for destination operand
p: Input port number for destination operand
nt: Necessary number of operands for destination operand
ad: Assignment function for destination operand
(b) Instruction set
Input
Wait
(Operand mem.)
Inst. & data
mems.
Fetch
I Structure
ALU
Output
(c) PE

Fig. 6.16 A dynamic data flow machine (Referred by [17])

used for the determination of a PE assignment. The af has four parameters. Operands only represent a data-flow, while the execution data are represented by data tokens. That is, the program (data-flow) and the data are separated. The data token consists of the statement number for the destination operand, the tag (color), the input port number, and the destination operand data. By changing the tag, the data-flow graph can execute the loop operation.

Figure 6.16c shows the structure of the PE and the execution steps are conducted as follows. The input unit receives the data token from the interconnection network or the output of its own PE. It is executed until receiving the operand data which are necessary for the execution of the operation. That is, associative search is performed on the operand memory for the statement number and tags of the data token. Two operands are necessary considering the case of performing binomial operation. If one operand has already been received, it is stored in the register. By associative search, it is possible to check whether the necessary operands for the operation are received. When the reception of the necessary operands is completed, the next instruction fetch unit reads the operation information from the instruction memory using the statement number of the storage destination. The newly arrived operand is read from the waiting part. At the same time, another received operand is read from the operand memory. As a result, the necessary information is completed. Then, the calculation is performed in the ALU. Finally, the operation result as a data token is sent to the destination storage following the instruction.

Note that, the “I structure,” which is similar to an array, provides a queuing function for a simple data structure. In array access in data-driven method, the data reading of certain array elements may occur before the data generation of ones. After data writing, a 1-bit existence (presence) bit is used for each element of the memory in order to guarantee reading. If its value is 1, it indicates that it has been written; otherwise, it is unwritten. If the presence bit is zero at the reading, it is suspended until the writing is completed. Thus, it is possible to guarantee synchronous data access with hardware.

The above is an overview of two data-driven methods and architectures, i.e., static and dynamic ones. Both of them are different when it comes to arithmetic operation control. This chapter outlined the earliest first-generation data-driven machines; however, regarding the second-generation machines, refer to [20].

6.4 Petrinet

A petrinet (also known as a place/transition (PT) net) is a directed bipartite graph, in which the nodes represent transitions and places. For example, events may occur, and they are represented by bars, while conditions are represented by circles. The directed arc denotes pre- and/or post-conditions for the transitions specified by arrows. The petrinets were introduced in 1939 by Carl Adam Petri for the purpose of describing chemical processes. A variation of the petrinet is a signal transition graph (STG), which is used to describe parallel or asynchronous systems.

By converting the data-flow graph shown in Fig. 6.17 to a petrinet, we have a new graph as shown in Fig. 6.18. The place corresponding to the input data has a token which represents the data. A condition to activate the transition, which represents the operation, is that there are at least more than one token on all the input places connecting the transition. After the activation, it generates a token to the place corresponding to the output. This means that the petrinet can easily represent both the data flow and its operation for the data-driven method. Figure 6.19 shows basic operations for the petrinet, e.g., parallel operation and synchronization. For more details about the petrinet and its parallel description, please refer to [21].

6.5 Stream Processing

6.5.1 Definition and Model

A processing method by which successive operations are done for successively inputted rows of data is referred to as stream processing [22–24]. The data element might be a single scalar data or a vector data including several words. Although the processing time is proportional to the number of elements (stream length), because

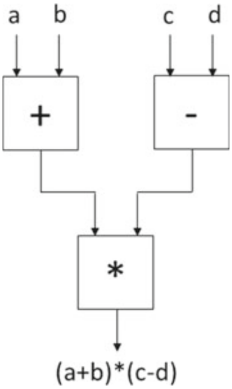


Fig. 6.17 An example of a data flow graph

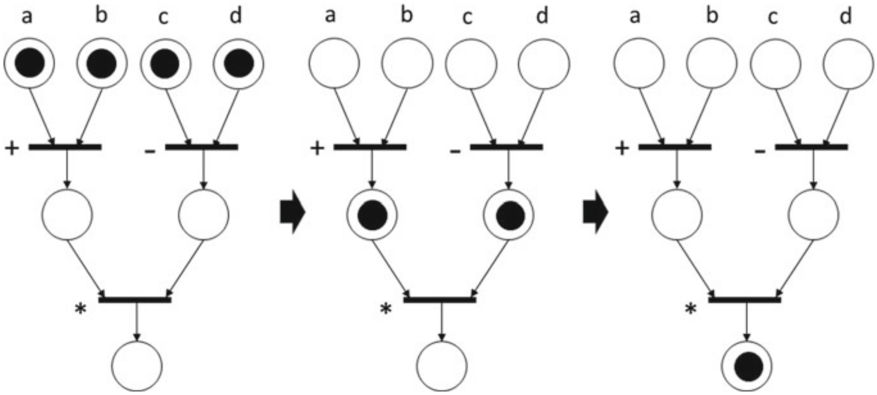


Fig. 6.18 An example of a Petrinet

only one iteration of the processing is executed at a given time, it can process any giant dataset in a sufficient time. The device for stream processing is not equipped with memory for all stream data. Data are supplied usually from an external memory, a network, or sensors. The device is used to process data that are too large to be stored in its own memory. A use case can be found in statistical information where inquiries are coming to a server through the Internet. In addition, when using stream data stored in external memory, an efficient use of data bandwidth by reading out data regularly with continuous addresses might be expected.

The processing of one element of stream data is designated as a processing kernel. Figure 6.20 depicts a model of stream processing by a single kernel. Here, the input is a data stream, but the output might be a data stream or might not be dependent on processing. Additionally, stream processing might be done by connecting processing

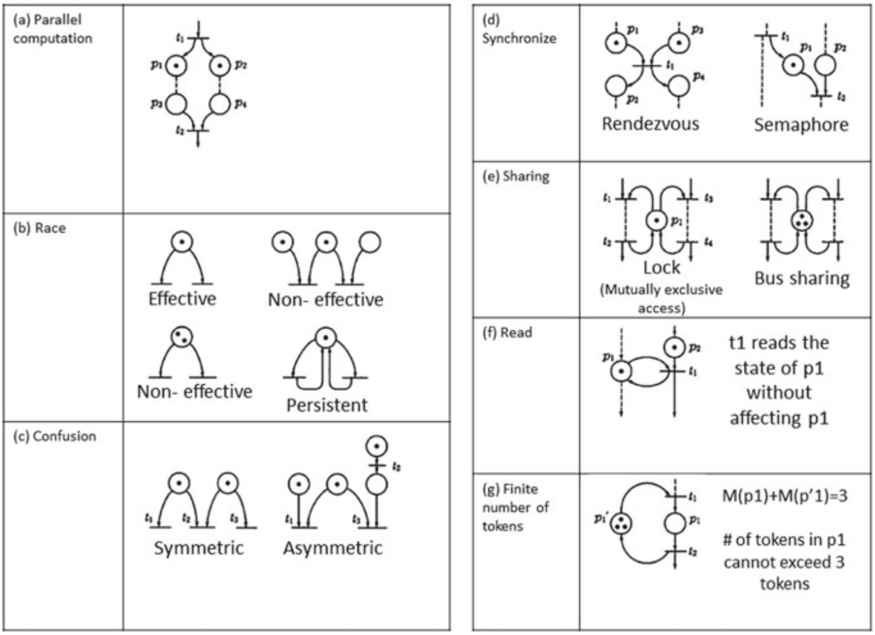
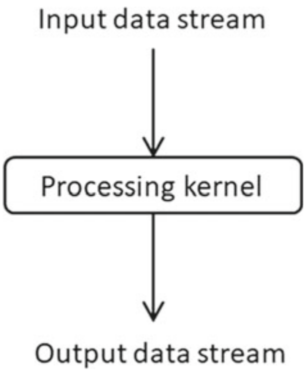


Fig. 6.19 Basic operations for a Petrinet (Referred by [21])

Fig. 6.20 Stream processing with a single kernel

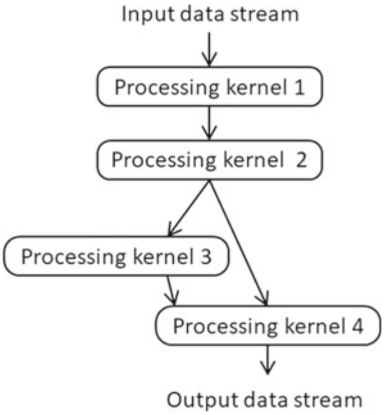


kernels according to their dependency, as shown in Fig. 6.21. This is an expression of stream processing by a data-flow graph where the processing kernel is a node in the graph.

6.5.2 Hardware Implementation

Several methods exist to realize the stream-processing concept. As a means to accomplish stream processing of high throughput by implementing software, it is possible to

Fig. 6.21 Stream processing with multiple kernels



incorporate vector instruction or SIMD instruction into a general-purpose microprocessor. These instructions can rapidly process vector data of definite length. However, a general-purpose microprocessor presents many limitations on parallelism and inefficient input and output data streaming caused by deep memory hierarchy. Consequently, high-performance stream processing generally requires hardware implementation.

Designing a high-throughput stream-processing hardware usually depends on a structure that performs many operations included in a unit processes in parallel. This structure requires a hardware design based on a parallel processing model such as pipelining, systolic algorithm, and data-driven approaches. As shown in Fig. 6.21, to realize stream processing for multiple kernels, given sufficient hardware resources, kernels designed as pipeline modules can be connected to each other and thereby statically implement a giant pipeline. If sufficient hardware resources are available to implement all necessary kernels simultaneously, high-speed stream computing can be achieved where we can input and output stream elements at each cycle.

What should be managed if sufficient hardware resources are not available? In such a case, not all the necessary processing kernels can be implemented. Hardware designers might opt for a design where different kernels share the same hardware resources and the processing of one stream data element is performed over a longer time. One discussing an example of such a case, shown in Fig. 6.21, the question that could be asked is: Where can we implement only half of the hardware resources for processing kernels? In this case, we can implement a module for kernel 1 and kernel 2 and another module for kernel 3 and kernel 4, and then we switch the mode, as shown in Fig. 6.22a. This resembles folding the original data-flow graph to get a smaller one and making the hardware mapping on it. Therefore, this method is referred to as folding. When using folding, the hardware works in time-sharing mode, as shown in Fig. 6.22b. First, the processing of kernel 1 for the input data occurs. Then, the mode is switched to execute kernel 2. Similar processing is done for kernels 3

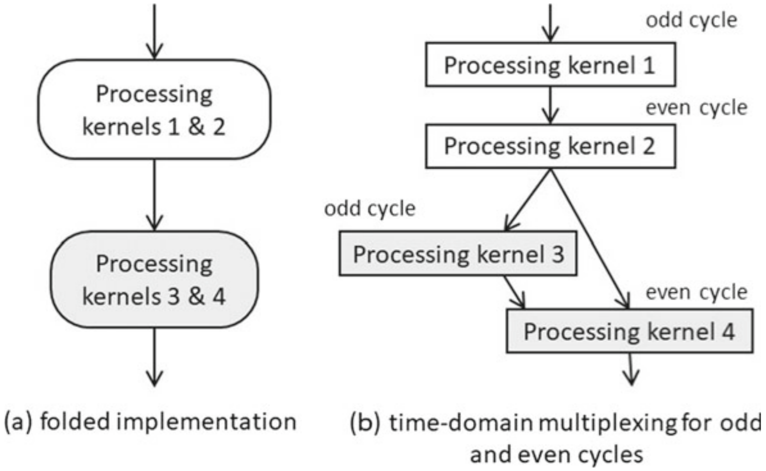


Fig. 6.22 Resource-saved implementation by folding kernels

and 4. Although the cycle time is doubled and the throughput decreases by half, with a small amount of hardware resources, stream processing can be accomplished.

In the example given above, all kernels work all the time and the operation rate is 100%. In this case, the operation rate of the folded processing kernel node is greater than 100%. The throughput decreases as a result. However, as in the case of conditional branching processing, the rate of operation is naturally less than 100%. Some kernels might have an operation rate less than 100%, even if aggregated with the operations of other kernels. Under these circumstances, the hardware resource consumption might be reduced without lowering the rate of operation in expense of a complicated control. Figure 6.23 demonstrates a simple example of folding with a rate of operation which is less than 100 Fig. 6.23a including conditional branching, two operations $x + y * z$ and $x * y + z$ are performed simultaneously. The signal selector outputs the results of the operation in two kernels with proportions of 90% and 10%, respectively. Accordingly, the real operation rates of both kernels are less than 100%. For this case, a module that can process both formulae, while considering common operations, can be implemented. The computation unit is shared in this design. Figure 6.23b illustrates an example. A multiplier and an adder are implemented for each formula, and one selector is inserted to switch the operations of the formulae. Only one operation is necessary for one formula. Therefore, the consumption of hardware resources can be reduced without increasing the number of cycles for one processing iteration of one element of stream data.

In addition, a stream-processing iteration, where some dependence relations exist between successive elementary processing iterations, might be implemented by inserting a delay buffer memory. This buffer sends intermediate data of stream

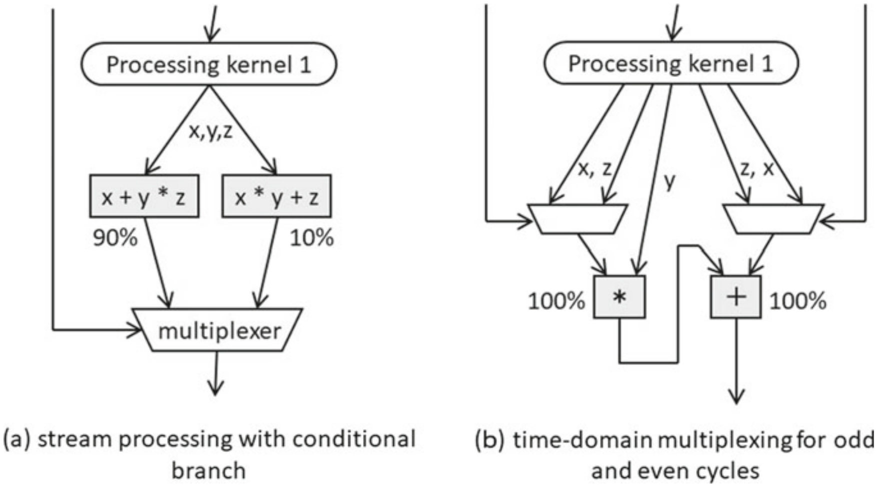
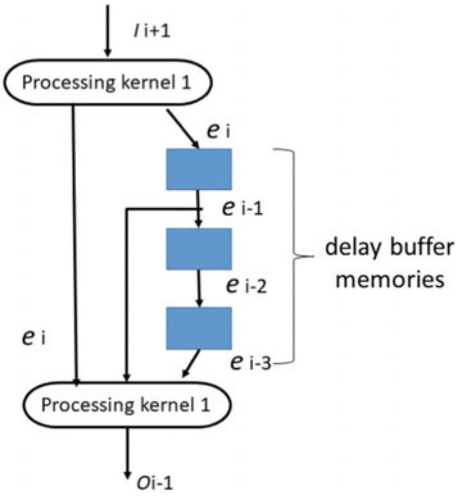

Processing kernel 1
x,y,z
x + y * z
x * y + z
90%
10%
multiplexer
(a) stream processing with conditional branch
Processing kernel 1
x, z
y
z, x
100%
*
+
100%
(b) time-domain multiplexing for odd and even cycles

Fig. 6.23 Folded implementation of underutilized stream processing

Fig. 6.24 Folded implementation of underutilized stream processing



processing to the next elementary processing iteration in a delayed fashion. In the example presented in Fig. 6.24, in addition to the current output result from processing kernel 1, delay buffer memories to send past data of to processing kernel 2 are inserted. Stream processing of stencil calculation applying delay buffer memory is introduced in the next section.

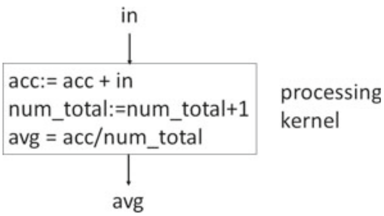


Fig. 6.25 Stream-processing hardware for averaging a series of scalars

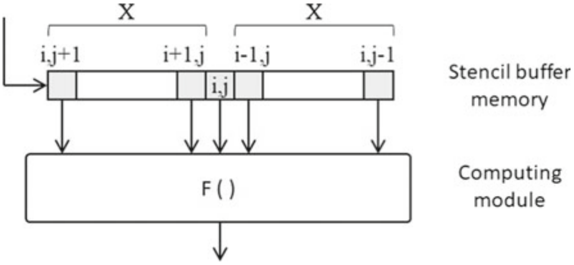


Fig. 6.26 Stream-processing hardware for 3 × 3 star-stencil computation (Fig. 6.9)

6.5.3 *Examples of Stream Processing*

An example of stream processing to find the average of a scalar array is depicted in Fig. 6.25. The processing kernel has two registers: acc and num_total. They are initialized to zero at the beginning of the operation. Whenever scalar data are inputted, it is added to acc and num_total is incremented by 1. At the end of one cycle, acc is divided by num_total to get avg; it is outputted as the average of the input data up to that moment.

Figure 6.26 displays an example of hardware for stream processing of 2D iterative stencil computation. This example is for the 2D iterative stencil computation shown in Fig. 6.9. It generates data stream of lattice data $v_{i,j}$ by traversing the 2D computational lattice, as shown in Fig. 6.9b in the x direction. In the processing kernel, the value at lattice point (i, j) is evaluated by function $F()$ using 5 data of $v_{i,j+1}$, $v_{i+1,j}$, $v_{i-1,j}$, $v_{i,j-1}$ in a 3 × 3 star stencil. In this case, a delay buffer memory is used because the previous and subsequent elements are necessary in addition to the current input element of input data [25]. This buffer memory is referred to as the stencil buffer memory.

Let X be the width of a 2D computational lattice. The stencil buffer is a 2X + 1 long shift register with a five readout ports for $v_{i,j+1}$, $v_{i+1,j}$, $v_{i-1,j}$, $v_{i,j-1}$ as shown in Fig. 6.26. After X cycles, after inputting the data at the current lattice point (i, j), the data appear exactly at the center of the stencil buffer. The five data in the star-stencil become readable simultaneously. The operation module makes use of this data and outputs calculated values at lattice point (i, j). A stencil computation for 2D lattice requires buffers that are proportional to the lattice width. Although the 3D lattice requires buffers that are proportional to the cross-sectional area of the lattice, it needs more buffer memory than the 2D one does. Therefore, on-chip memory cannot

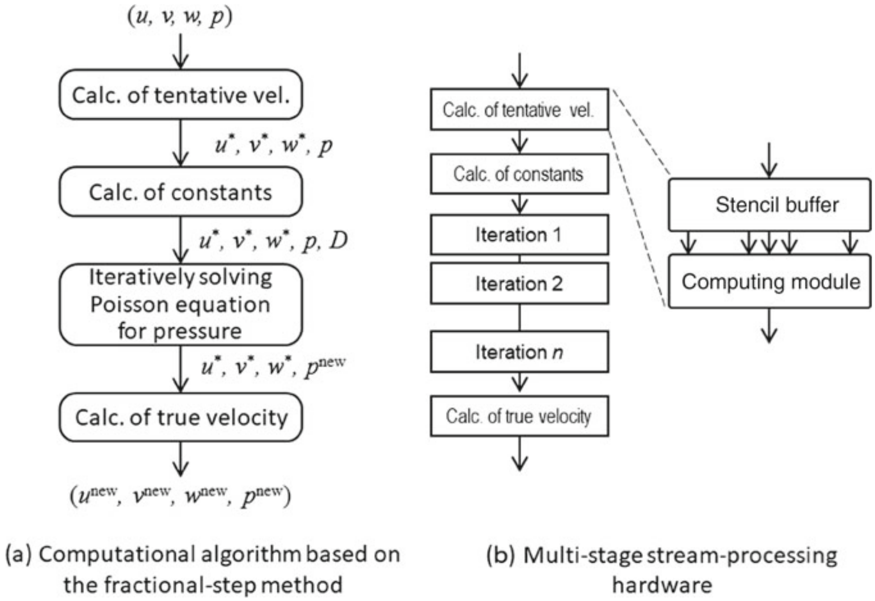
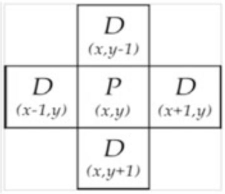


Fig. 6.27 Computational algorithm of incompressible fluid dynamics and its stream-processing hardware

provide sufficient capacity for a large 3D lattice. In this case, the calculation lattice can be divided into smaller partial lattices which are then computed one by one.

A hardware example of multiple computing stages for stream processing of incompressible fluid dynamics is shown in Fig. 6.27. As depicted in Fig. 6.27a, the algorithm, based on the fractional-step method used in CFD, comprises four stages of operations [13, 26]. It is known that each stage is of stencil computation, which refers to neighbors of each lattice point in the orthogonal lattice. For this reason, each step is implemented by stream-processing hardware which is composed of the stencil buffer memory and the computation module, as shown in Fig. 6.26. It is possible to construct stream-processing hardware to perform fluid computation for a single time step by the in-line connection of hardware for each computing stage [26]. Part of the iterative solution of a Poisson equation is implemented using a fixed construction with an *n* element array of hardware for one-iteration stencil calculation. This result is based on the experience that, although sufficient implementation has been done with conventional iteration methods to reduce the residual to a sufficiently small magnitude, the iterations were normally less than a definite number. The description above is a proposed solution to the case in which there is no definite number of iterations. However, the method and conditions should be thoroughly examined to address any practical problem.

Fig. 6.28 Von Neumann neighborhood



6.6 Cellular Automaton

A cellular automaton is a discrete model studied in computability theory, mathematics, physics, complexity science, theoretical biology, and microstructure modeling. The concept was originally discovered in the 1940s by von Neumann [27]. It consists of a regular grid of cells, each in one of a finite number of states, such as *on* and off [28]. The grid can be in any finite number of dimensions. For each cell, a set of cells (referred to as neighborhood) is relatively defined to the specified cell. An initial state (time t = 0) is elected by assigning a state for each cell. A new generation is created according to some fixed rules that determine the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. Typically, the rule for updating the state of cells is the same for each cell and does not change over time and is applied to the whole grid simultaneously, though exceptions are known.

Figure 6.28 shows an example of the neighborhood. Let K be the number of states for each cell. Then, five cells including the center have K5 states. Thus, there are KK^5 rules of cellular automaton based on the neighborhood. The well-known cellular automaton is a life game which can be specified with the following rules for K = 2.

- Born: If there are more than or equal to three living cells around the dead cell, then it lives in the next step.
- Living: If there are two or three living cells around a living cell, then it survives in the next step.
- Dead: Otherwise, it dies in the next step.

A finite cell is widely used to simulate the cellular automaton. Generally, although it is implemented assuming a finite rectangle, an implementation of the boundary becomes a problem. There is a method of treating all boundaries as a constant; however, the disadvantage is that the number of rules increases. Another way is to make it as a torus [29], which simulates an infinite rectangle by connecting upper, lower, left, and right respectively, and filling an infinite plane with the same rectangle in the same plane. Figure 6.29 shows a cellular automaton circuit using a torus connection by placing PEs in a 3 × 3 grid pattern. In the case of the game of life, the PE has the state of each cell, and it executes the above rules for each step. Then, it updates the state of the cell in the next step.

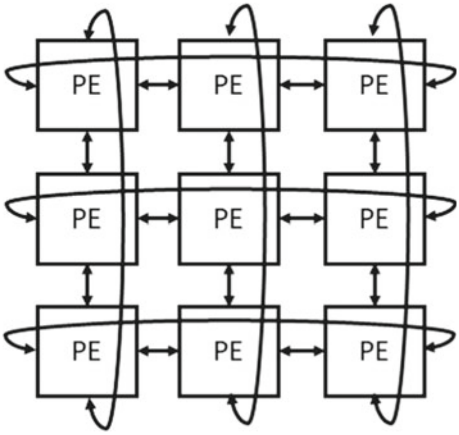


Fig. 6.29 Cellular automata circuit using a 3 × 3 PE

Since all the inputs and outputs in a cellular automaton circuit are parallel, it is highly compatible with FPGA implementation. Also, it has the possibility to surpass the existing von Neumann architecture. In practice, as the number of rules to be computed increases, the throughput can be improved using pipeline circuits. In recent years, an attempt has been applied to build more physical cellular automata from the viewpoint of materials, rather than circuits or devices [30]. It is expected to replace the existing von Neumann-type architecture by applying these realizations to FPGAs.

6.7 Hardware Sorting

In computer science, a sorting is an algorithm that puts *n* elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. FPGA-based hardware accelerations for sorting are widely used for database, image processing, and data compression. Here, we introduce a sorting network and a merge sort tree that are suitable for hardware implementation.

The simplest sorting algorithm for hardware implementation is the sorting network [31], based on the bubble sort. This algorithm sorts the neighboring elements in parallel. Figure 6.30 shows the sorting network of four elements. It consists of the exchange units (EUs) which sort the neighboring elements. In this circuit, the number of wires is n. Each element passes at most n − 1 EUs. In this case, since it can be performed in parallel, we can realize a fully pipelined EU circuit to increase the throughput. Note that, since it requires n-parallel wires, the amount of hardware tends to be large. The known sorting network on FPGA is the Batcherfs odd–even merge sort [32].

The other type of hardware sorting algorithms is the merge sort tree [33], which is based on the binary tree structure where each vertex is realized by the EU. This circuit has many FIFOs in the input and output, and performs the sorting in parallel.

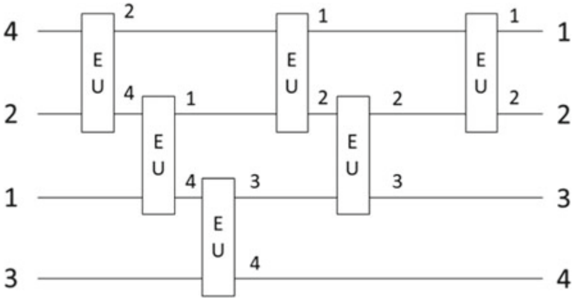


Fig. 6.30 A sorting network for four elements

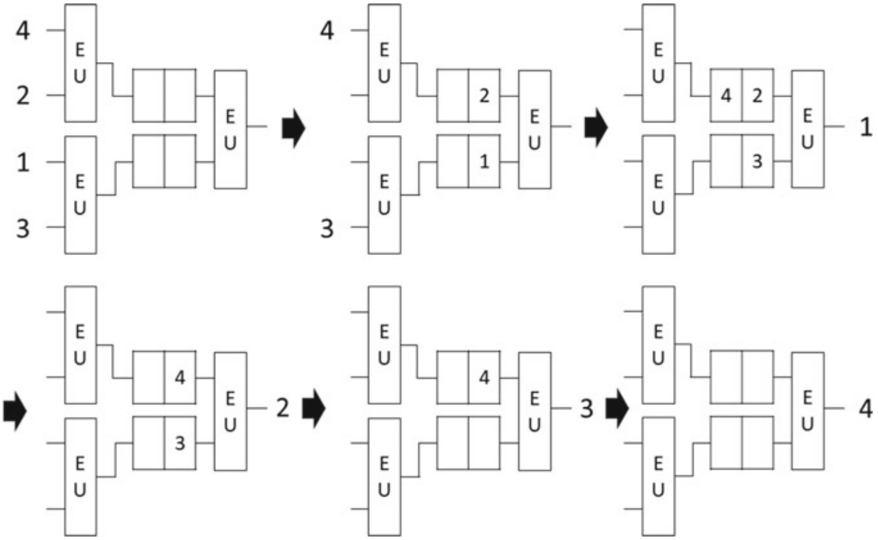


Fig. 6.31 An example of a marge sort tree for four elements

Figure 6.30 shows an example of the merge sort tree for four elements. In the merge sort tree, the input is a sequence to be sorted, and in each level of the tree, the sorted sequence is sent to the next level through the FIFO. Therefore, it is possible to increase the throughput by inserting a pipeline register for each level. To realize a high speed and small area on the FPGA, a combination of sorting network and merge sort tree shown in Fig. 6.31 has been proposed [34].

6.8 Pattern Matching

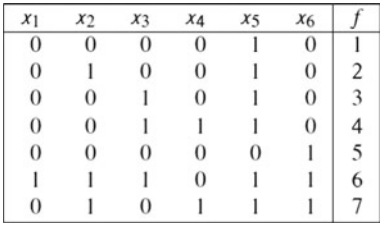
One of the killer applications for FPGA is the pattern matching which finds a given pattern in the data. Pattern matching algorithms are roughly categorized to an exact matching, a regular expression matching, and an approximate matching. In this section, we introduce the different algorithms for these matchings.

6.8.1 Exact Matching

An exact matching finds a fixed pattern; however, each element of the pattern takes three values, i.e., one, zero, and don't-care which can take both zero and one. Typically, the exact matching can be realized by a content addressable memory (CAM) [35]. Here, we introduce the index generation unit (IGU) N6.8-3 which is a CAM emulator, then we implement it on FPGA.

Let us suppose that the index generation function f is as shown in Figs. 6.32 and 6.34 shows the decomposition chart for f. In Fig. 6.34, the label in the right side denotes X_1 = (x2, x3, x4, x5), the label in the left side denotes X_2 = (x1, x6), and the entry denotes the function value. Note that, each column has at most one nonzero element. Thus, f can be realized by a main memory whose input is only X_1. The main memory maps a 2^n sets to k + 1 sets. This is an ideal case; however, in most cases, we must check X_2 since f may cause mismatch. To do this, first, we store the correct X_2 to an auxiliary (AUX) memory. Then, we use a comparator to generate a correct f when f is equal to X_2; otherwise, it generates zero. Figure 6.33 shows the IGU. First, we read the q from the main memory corresponding to p bit X_1. Then, $X0_2$ is read from the AUX memory corresponding to q. Next, q is generated if $X0_2$ is equal to X_2; otherwise, it generates zero.

Figure 6.35 shows an example of the IGU realizing the index generation function shown in Fig. 6.32. When (x1, x2, x3, x4, x5, x6) = (1, 1, 1, 0, 1, 1), the index "g6" corresponding to X_1 = (x2, x3, x4, x5) = (1, 1, 0, 1) is read. Then, $X0_2$ = (x1, x6) = (1, 1) corresponding to $X0_2$ = (x1, x6) = (1, 1) is read. Next, the correct signal is sent



x_1	x_2	x_3	x_4	x_5	x_6	f
0	0	0	0	1	0	1
0	1	0	0	1	0	2
0	0	1	0	1	0	3
0	0	1	1	1	0	4
0	0	0	0	0	1	5
1	1	1	0	1	1	6
0	1	0	1	1	1	7

Fig. 6.32 An example of an index generation function

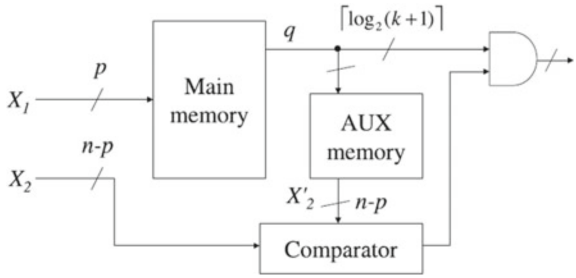


Fig. 6.33 An index generation unit (IGU)

	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	x_5
	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	x_4 X_1
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	x_3
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	x_2
00	0	0	0	0	0	0	0	0	1	2	3	0	0	0	4	0	
01	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
10	5	0	0	0	0	0	0	0	0	0	0	0	0	7	0	0	
11	0	0	0	0	0	0	0	0	0	0	0	6	0	0	0	0	
x_6, x_1 X_2																	

Fig. 6.34 An example of a decomposition chart for an index generation function

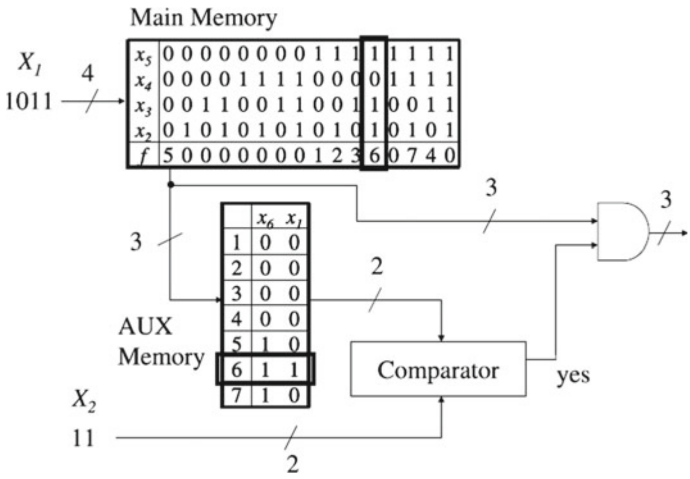

Main Memory
X_1
1011
4
x_5 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
x_4 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
x_3 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
x_2 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
f 5 0 0 0 0 0 0 0 1 2 3 6 0 7 4 0
3
3
3
x_6 x_1
1 0 0
2 0 0
3 0 0
4 0 0
5 1 0
6 1 1
7 1 0
AUX Memory
2
Comparator
yes
X_2
11
2

Fig. 6.35 Operation for an IGU

to the AND gate, then "g6" is generated. Since the IGU realizes a mapping which generates $k + 1$ sets from given $2n$ sets, its memory size is drastically reduced from $O(2^n)$ to $O(2^p)$. The theoretical analysis of the IGU is introduced in [36], and the applications for the IGU are presented in [37, 38].

6.8.2 Regular Expression Matching

A regular expression consists of a character and a metacharacter which represents a set of a strings. Various network applications use regular expression matchings to detect malicious data in incoming packets. Regular expression matchings spend a considerable fraction of the total computation time for these applications. The throughput using the Perl compatible regular expression (PCRE) library on a general-purpose MPU is up to hundreds of megabits per second (Mbps), which is too slow for most applications. Thus, a dedicated circuit for regular expression matchings is required. For network applications, since the high-mix low-volume production and the frequent update for new protocols are required, FPGAs are widely used. With the advent of FPGAs embedding dedicated high-speed transceivers for high-speed networks, we expect extensive use of FPGAs in the future.

Regular expressions are detected by finite automata (FA). In a deterministic finite automaton (DFA), for each state and each input, there is a unique transition. While in a non-deterministic finite automaton (NFA), for each state and each input, multiple transitions may exist. In an NFA, there exist ϵ-transitions to other states without consuming input characters.

Most of the proposed regular expression matching circuits are based on finite automata. An Aho–Corasick DFA (AC-DFA) [39] is a known algorithm. A combination of the bit-partitioned AC-DFA and the MPU is proposed [40]. Also, Baeza-Yates proposed the NFA algorithm based on a shift and bitwise AND operations [41], and its hardware realization on an FPGA is proposed [42]. A resource-efficient FPGA realization by prefix and postfix sharing of regular expressions [43], and mapping repeatedly appearance parts of regular expressions into a Xilinx FPGA primitive (SRL16) [44] have been also published.

Hereafter, we introduce the NFA-based regular expression matching algorithm which is suitable for FPGA realization. Figure 6.36 shows a conversion from a regular expression to NFA. In Fig. 6.36, ϵ denotes the ϵ-transition, and the gray circle denotes the accept state. Figure 6.37 shows the NFA representing the regular expression "abc(ab) * a", and shows the state transition for the input string "abca". For each element in the vector corresponding to the state in the NFA, and '1' denotes the active state. Figure 6.38 shows the circuit for the NFA shown in Fig. 6.37. To emulate this NFA, a memory is used to detect a corresponding character, and the detection signal is sent to the matching element (ME). The ME emulates the state transition, and generates the match signal. In the ME, the FFs store the vector shown in Fig. 6.37, where i denotes the transition signal from the previous state; *o* denotes the transition signal to next state; *c* denotes the character detection signal from the memory; *ei* and *eo* denote the in/out signals from the ϵ-transition.

Figure 6.39 compares the NFA [42] with the DFA [40] with respect to the complexity of the parallel execution hardware. Even if we apply the bit-partition which reduces the amount of hardware, the complexity still holds $O(\Sigma^{sm})$. When the number of rules for the regular expression increases, the amount of memory tends to be exponentially large for the DFA based realization, while the NFA-based one does

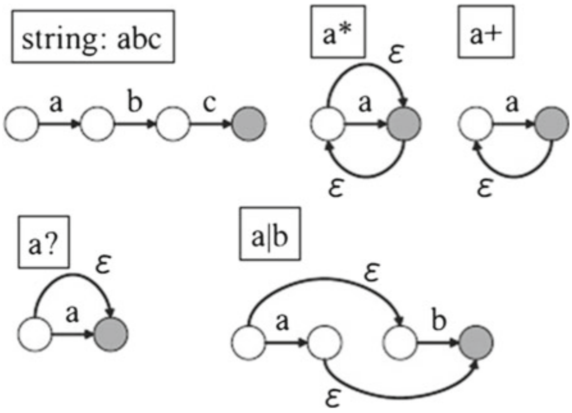
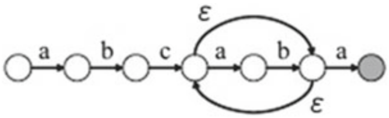


Fig. 6.36 Conversion from a regular expression to an NFA



```
initial
           (1,  0,  0,  0,  0,  0,  0)
input 'a'
           (1,  1,  0,  0,  0,  0,  0)
ε -transition
           (1,  1,  0,  0,  0,  0,  0)
input 'b'
           (1,  0,  1,  0,  0,  0,  0)
ε -transition
           (1,  0,  1,  0,  0,  0,  0)
input 'c'
           (1,  0,  0,  1,  0,  0,  0)
ε -transition
           (1,  0,  0,  1,  0,  1,  0)
input 'a'
           (1,  1,  0,  0,  1,  0,  1)
ε -transition
           (1,  1,  0,  1,  1,  1,  1) Accept 'abca'
```

Fig. 6.37 An NFA accepting 'abc(ab) * a'

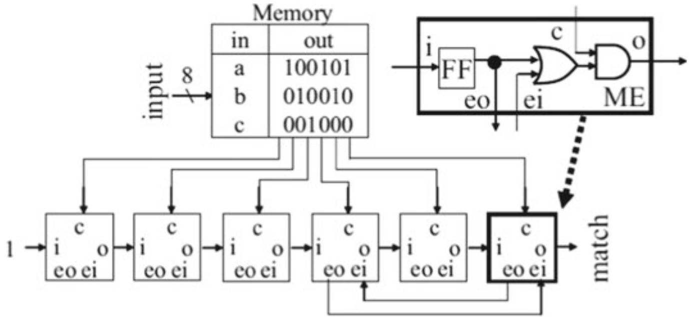
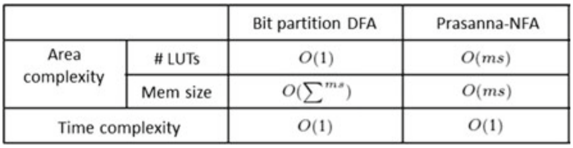


Fig. 6.38 Circuit for an NFA



		Bit partition DFA	Prasanna-NFA
Area complexity	# LUTs	$O(1)$	$O(ms)$
	Mem size	$O(\sum^{ms})$	$O(ms)$
Time complexity		$O(1)$	$O(1)$

Fig. 6.39 Comparison of complexities

not increase the amount of memory. Thus, the NFA-based one is suitable for FPGA implementations.

6.8.3 Approximate Matching

An approximate matching consists of finding an edited pattern in a text. It finds a corresponding pattern in the text while deleting, replacing, and inserting a character. Many approximate matchings are based on dynamic programming. Approximate matching is used in bioinformatics to evaluate similarity between the DNA sequences.

Let "ACG" be a text, and "TGG" be a pattern. Then, we compute the edit distance:

1. Delete "A" from the text "ACG," then we have "CG."
2. Delete "C" from "CG," then we have "G."
3. Insert "G" to "G," then we have "GG."
4. Insert "T" to "GG," then we have the text "TGG," which corresponds to the given pattern.
5. Terminate.

In this example, we set the editing score for both the insertion and deletion to 1. Since the replacement includes insertion and deletion, its score is 2. The above example showed that the editing score between "ACG" and "TGG" is 4.

Figure 6.40 shows a system for an approximate matching. The host PC sends a text and a pattern. The matching system reads the text from the buffer memory, and

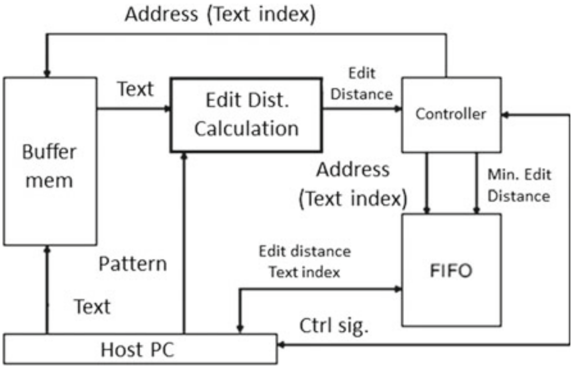


Fig. 6.40 System for an approximate matching

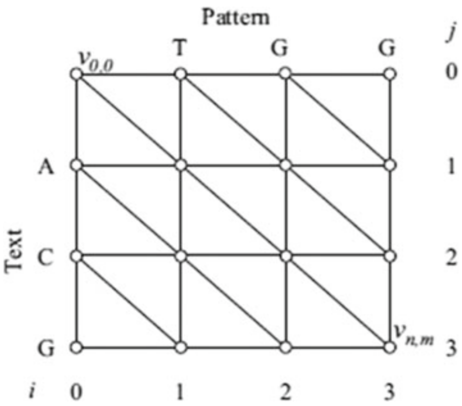


Fig. 6.41 An example of an approximate matching graph

then the editing calculating circuit computes the editing score for a part of the text and the pattern. The controller stores the address whose position of the text and its minimum editing score to the FIFO. The system shifts the text for calculating the score. When all the text is matched, the host PC reads the position of the matched pattern and its minimum score from the FIFO. Then, the edited pattern is computed. For approximate matching, since the calculation time for the editing score is dominant, FPGA accelerators are desired.

The editing score between two strings can be calculated using dynamic programming. The Needleman–Wunsch (NW) algorithm [45] computes the minimum value of the editing score of the entire text and a pattern, while the Smith–Waterman (SW) algorithm [46] computes the editing score of a part of text and a pattern. Here, we introduce the basic algorithm for calculating the minimum value of the editing score between two strings using dynamic programming.

Let P = (p1, ..., pn) be a pattern, and T = (t1, ..., tm) is a text. Suppose that the matching graph for an approximate string, which has $(n + 1) \times (m + 1)$ vertices labeled by each column and row. For a coordinate (i, j), a vertex v_i, g_j is placed. We assume that the upper-left vertex is set to (0,0), and the coordinates (i, j) increase toward the lower-right vertex (n, m). For $0 <= i <= n - 1$ and $0 <= j <= m - 1$, there are edges connecting $v_{i,j}$ to $v_{i+1,j}$ and others connect $v_{i,j}$ to $v_{i,j+1}$. Also, there are diagonal edges connecting $v_{i,j}$ to $v_{i+1,j+1}$. Figure 6.41 shows an example of the approximate matching graph for the text ACG and the pattern TGG.

Let s_{del} be an editing score for deletion, sins the score for insertion, and s_{sub} the score for replacement. Here, we set $s_{del} = 1$, $s_{ins} = 1$, and $s_{sub} = 2$. For each vertex $v_{i,j}$, we must make sure to edit score of a subpattern $P^t = (p_1, p_2, \ldots, p_i)$ and subtext $T^t = (t_1, t_2, \ldots, t_j)$. We define that the vertex score denotes the editing score for each vertex. The minimum vertex score for $v_{i,j}$ is obtained by the following expression:

$$v_{i,j} = min \begin{cases} v_{i-1,j-1} \\ v_{i-1,j} + s_{ins} + \begin{cases} 0 & if \;\; p_i = t_j \\ s_{sub} & otherwise \end{cases} \\ v_{i,j-1} + s_{del} \end{cases} \tag{6.1}$$

By applying the above expression recursively from the vertex $v_{0,0}$ to $v_{n,m}$, we can obtain the minimum editing score. The following algorithm shows how to obtain the minimum editing score:

Algorithm

Input: Text T with length m, and pattern P with length n.
Output: Minimum editing score at vertex $v_{n,m}$.

```
1: $v_{i,0} \leftarrow i$, $(i = 0, 1, \ldots, n)$, $v_{0,j} \leftarrow j$, $|(j = 0, 1, \ldots, m)$
2: **for** $j \leftarrow 1$ **until** $j \leq m + n - 1$ **begin**
3:     **for** $i \leftarrow 1$ **until** $i \leq n$ **begin**
4:         **if** $0 < j - i + 1 \leq m$, compute $v_{i,j-i+1}$ with Eq. 6.1.
5:         $i \leftarrow i + 1$
6:     **end**
7:     $j \leftarrow j + 1$
8: **end**
9: Let $v_{n,m}$ be edit distance and stop.
```

Here, we assume that $n \ll m$. For example, for an alignment in bioinformatics, $n = 10^3$ and $m = 10^9$. An algorithm, which calculates the minimum editing score by dynamic algorithm, is called the Naive method. It uses a processing element (PE) [47] to calculate each column of the approximate matching in parallel. Figure 6.42 shows the architecture of the PE for the Naive method. In this figure, s denotes the number of bits for each character, n denotes the number of characters in the pattern. To directly perform recurrent expression, the naive method receives a text (tin) and a pattern (p in). Then, it selects the replacement score or not through the corresponding detection circuit. At that time, for each vertex, it calculates the editing score. Then, it selects the minimum score selector to generate the minimum one.

Each PE calculates the score corresponding vertex, then outputs it in parallel. Note that, t denotes the time stamp. We consider the data dependency to compute the vertex $v_{i,j}$ by the PEi. To compute $v_{i,j}$, three pieces of data for $v_{i,j-1}$, $v_{i-1,j}$, and $v_{i-1,j-1}$ are necessary. At time (t – 1), since $v_{i,j-1}$ is the output value of the PE_i, it is obtained by the feedback loop. Also, $v_{i,j-1}$ is obtained by the output of PE_{i-1}. At time (t – 2), $v_{i-1,j-1}$ is obtained by PE_{i-1} and is retained by a register. The cascaded PEs shown in Fig. 6.42 computes the approximate matching graph in parallel. In other words, it performs the naive method shown in steps 3–6 of the above algorithm. Thus, its computation complexity becomes $O(m)$. More details of the circuit on FPGA have been demonstrated in [48].

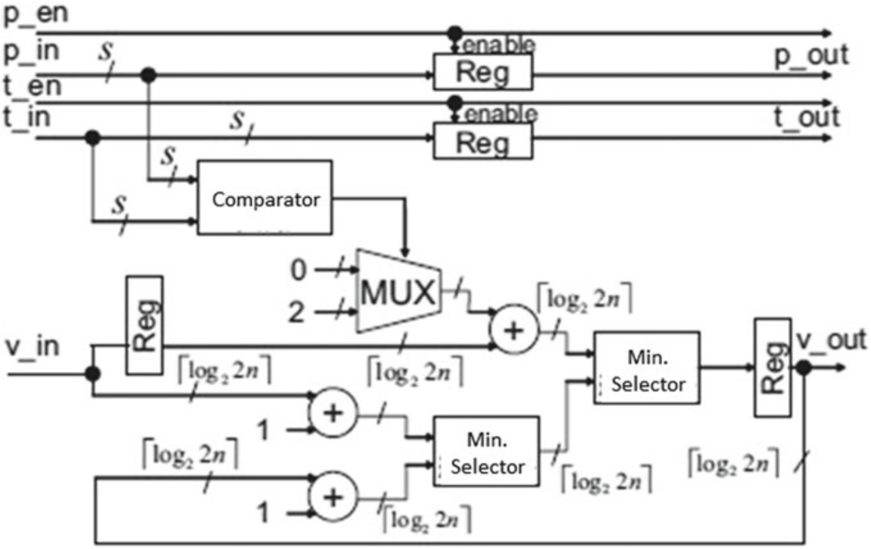


Fig. 6.42 A processing element (PE)

References

1. D.A. Patterson, J.L. Hennessy, *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface* (Morgan Kaufmann Publishers Inc., 2008)
2. H.S. Stone, *High-Performance Computer Architecture* (Addison-Wesley Publishing Company, 1990)
3. M.J. Flynn, Some computer organizations and their effectiveness. IEEE Trans. Comput. **21**(9), 948–960 (1972)
4. A. Peleg, U. Weiser, MMX technology extension to the intel architecture. IEEE Micro **16**(4), 42–50 (1996)
5. M. Hassaballah, S. Omran, Y.B. Mahdy, A review of SIMD multimedia extensions and their usage in scientific and engineering applications. Comput. J. **51**(6) 630–649 (2008)
6. A. Downton, D. Crookes, Parallel architectures for image processing. Electron. Commun. Eng. J. **10**(3), 139–151 (1998)
7. A.P. Reeves, Parallel computer architectures for image processing. Comput. Vis. Gr. Image Process. **25**(1), 68–88 (1984)
8. H.T. Kung, Why systolic architecture? IEEE Comput. **15**(1), 37–46 (1982)
9. J. MaCanny, *Systolic Array Processors* (Prentice Hall, 1989)
10. K.T. Johnson, A.R. Hurson, B. Shirazi, General-purpose systolic arrays. IEEE Comput. **26**(11), 20–31 (1993)
11. S.-Y. Kung, K.S. Arun, R.J. Gal-Ezer, D.V. Bhaskar Rao, Wavefront array processor: language, architecture, and applications. IEEE Trans. Comput. **C-31**(11), 1054–1066 (1982)
12. K. Sano, Y. Kono, FPGA-based connect6 solver with hardware-accelerated move refinement. Comput. Archit. News **40**(5), 4–9 (2012)
13. K. Sano, T. Iizuka, S. Yamamoto, Systolic architecture for computational fluid dynamics on FPGAs, in *Proceeding of IEEE Symposium on Field-Programmable Custom Computing Machines* (2007), pp. 107–116
14. K. Sano, W. Luzhou, Y. Hatsuda, T. Iizuka, S. Yamamoto, FPGA-array with bandwidth-reduction mechanism for scalable and power-efficient numerical simulations based on finite

difference methods. ACM Trans. Reconfig. Technol. Syst. **3**(4), Article No. 21, (2010), https://doi.org/10.1145/1862648.1862651

15. K. Sano, FPGA-based systolic computational-memory array for scalable stencil computations, in *High-Performance Computing Using FPGAs* (Springer, 2013), pp. 279–304
16. A.H. Veen, Dataflow machine architecture. ACM Comput. Surv. **18**(4), 365–396 (1986)
17. K. Hwang, F.A. Briggs, *Computer Architecture and Parallel Processing* (McGraw-Hill, Inc., 1984)
18. J.B. Dennis, Dataflow supercomputer. IEEE Comput. **13**(4), 48–56 (1980)
19. A.V. Kathail, A multiple processor dataflow machine that supports generalized procedures, in *Proceeding of ISCA81*, pp. 291–296, May 1981
20. G.L. Gaudiot, *Advanced Dataflow Computing* (Prentice Hall, 1991)
21. J.L. Perterson, *Petrinet Theory and the Modeling of Systems* (Prentice Hall, 1981)
22. S. Hauck, A. DeHon, *Reconfigurable Computing* (Morgan Kaufmann Publishers Inc., 2008)
23. R. Stephens, A survey of stream processing. Acta Inform. **34**(7), 491–541 (1997)
24. A. Das, W.J. Dally, P. Mattson, Compiling for stream processing, in *Proceeding International Conference on Parallel Architectures and Compilation Techniques* (2006), pp. 33–42
25. K. Sano, Y. Hatsuda, S. Yamamoto, Multi-FPGA accelerator for scalable stencil computation with constant memory-bandwidth. IEEE Trans. Parallel Distrib. Syst. **25**(3), 695–705 (2014)
26. K. Sano, R. Chiba, T. Ueno, H. Suzuki, R. Ito, S. Yamamoto, FPGA-based custom computing architecture for large-scale fluid simulation with building cube method. Comput. Archit. News **42**(4), 45–50 (2014)
27. J. von Neumann, The general and logical theory of automata, in *Cerebral Mechanisms in Behavior? The Hixon Symposium* ed. by L.A. Jeffress (Wiley, New York, 1951), pp. 1–31
28. S. Wolfram, Statistical mechanics of cellular automata. Rev. Mod. Phys. **55**(3), 601–644 (1983)
29. J. von Neumann, A.W. Burks, *Theory of Self Reproducing Automata* (University of Illinois Press, 1966)
30. A. Bandyopadhyay et al., Massively parallel computing on an organic molecular layer. Nat. Phys. **6**, 369–375 (2010)
31. D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Addison Wesley Longman Publishting Co., Inc., Redwood City, CA, USA, 1998)
32. K.E. Batcher et al., *Sorting Networks and Their Applications* (Spring Joint Computer Conference, AFIPS, 1968), pp. 307–314
33. D. Koch et al., FPGA sort, in *Proceedings of FPGA* (2011), pp. 45–54
34. J. Casper, K. Olukotun, Hardware acceleration of database operations, in *Proceedings of FPGA* (2014), pp. 151–160
35. T. Kohonen, *Content-Addressable Memories*, vol. 1, Springer Series in Information Sciences (Springer, Berlin Heidelberg, 1987)
36. H. Nakahara, T. Sasao, M. Matsuura, A regular expression matching circuit: decomposed non-deterministic realization with prefix sharing and multi-character transition. Microprocess. Microsyst. **36**(8), 644–664 (2012)
37. H. Nakahara, T. Sasao, M. Matsuura, H. Iwamoto, Y. Terao, A memory-based IPv6 lookup architecture using parallel index generation units. IEICE Trans. Inf. Syst. **E98-D**(2), 262–271 (2015)
38. H. Nakahara, T. Sasao, M. Matsuura, A virus scanning engine using an MPU and an IGU based on row-shift decomposition. IEICE Trans. Inf. Syst. **E96-D**(8), 1667–1675 (2013)
39. A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
40. L. Tan, T. Sherwood, A high throughput string matching architecture for intrusion detection and prevention, in *Proceedings of 32nd Int'l Symposium on Computer Architecture (ISCA 2005)* (2005), pp. 112–122
41. R. Baeza-Yates, G.H. Gonnet, A new approach to text searching. Commun. ACM **35**(10), 74–82 (1992)
42. R. Sidhu, V.K. Prasanna, Fast regular expression matching using FPGA, in *Proceedings of the 9th Annual IEEE Symposium on Field-programmable Custom Computing Machines (FCCM 2001)* (2001), pp. 227–238

43. C. Lin, C. Huang, C. Jiang, S. Chang, Optimization of regular expression pattern matching circuits on FPGA, in *Proceeding of the Conference on Design, Automation and Test in Europe (DATE 2006)* (2006), pp. 12–17
44. J. Bispo, I. Sourdis, J.M.P. Cardoso, S. Vassiliadis, Regular expression matching for reconfigurable packet inspection, in *Proceeding IEEE International Conference on Field Programmable Technology (FPT 2006)* (2006), pp. 119–126
45. T.F. Smith, M.S. Waterman, Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
46. S.B. Needleman, C.D. Wunsch, A general method applicable to the search for similarities in the Amino-Acid sequence of two Proteins. J. Mol. Biol. **48**, 443–453 (1970)
47. L.J. Guibas, H.T. Kung, C.D. Thompson, Direct VLSI implementation of combinatorial algorithms, in *Proceedings of the Conference VLSI: Architecture, Design, Fabrication* (1979), pp. 509–525
48. Y. Yamaguchi, T. Maruyama, A. Konagaya, High speed homology search with FPGAs, in *Proceedings of Pacific Symposium on Biocomputing* (2002), pp. 271–282