FPGA Implementation of Data Flow Graphs for Digital Signal Processing Applications

Hala Al-Zu'bi, Osama Al-Khaleel and Ali Shatnawi

Department of Computer Engineering, Jordan University of Science and Technology, Irbid 22110, Jordan

***Abstract*:** A rapid growth in digital signal processing applications has increased the requirement for high-speed digital systems. Multiprocessor systems are the best choice for these applications. A prior sequence of operations should be applied to the operations that describe the nature of these applications before hardware implementation is produced. These operations should be time scheduled and hardware allocated. This paper proposes a new scheduling technique for digital signal processing (DSP) applications that are represented by data flow graphs (DFGs). In addition, hardware allocation is implemented in the form of embedded system. The proposed scheduling technique also achieves the optimal scheduling of a DFG at design time. The optimality criterion considered in this algorithm is the maximum throughput within the available hardware resources. The hardware system is composed of one or multiple homogenous pipelined processing elements and designed to meet the maximum-rate schedule. Two implementations are proposed of the system architecture according to the number of the processing elements. These are the serial system and the parallel system. The serial system comprises one processing element where all tasks are processed sequentially. On the other hand, the parallel system has four processing elements to execute tasks concurrently. The hardware systems have been described, functionally tested, and synthesized using the Verilog HDL and Xilinx ISE. Synthesis results show that the parallel system outperforms the serial one by 25.5% in terms of performance with extra area penalty. The relationship between the number of instructions that are executed in both systems, the system area, and the system performance as represented by the system frequency, have been studied. The proposed scheduling technique is shown to outperform the retiming technique, which we have chosen to compare with. The serial system has better performance with 19.3% higher system frequency than that of the retiming technique. The parallel system also outperforms the retiming technique in terms of the system frequency by 51.2%.

***Keywords*:** FPGAs, DFG, DSP Applications, Scheduling, Hardware, Architecture**.**

1. Introduction

Digital signal processing (DSP), image processing, and communications tasks are computationally intensive and require high-power consumption. Therefore, they demand very high-speed computations and high throughput systems. Multiprocessor systems are the best choice to implement DSP applications since they have parallelism nature [1].

Field-programmable gate arrays (FPGAs) are manufactured integrated circuits that are designed to be configured by the designer. It contains programmable logic blocks which have RAM blocks and large resources for implementing logic gates. Wide range of applications can be implemented on FPGAs by being specified using any hardware description language (HDL). Hardware implementation is created with the aid of logic synthesis tools.

Logic synthesis [2] is the process of designing digital systems that are described in HDL to get an optimized hardware implementation. Logic synthesis uses standard cell libraries such as the basic logic gates (AND, OR, NOR) library, and the library of the macro cells (adder, MUXs, memory, flip-flops) to produce a design.

There are many reasons to perform logic synthesis in designing digital systems [3]. Among these reasons are shorter design cycle, fewer bugs, ability to find design space, and the ability to resynthesize targeting different chip technologies as an FPGA or ASIC. The synthesis process mainly involves Description, Scheduling, and Resource Allocation [2].

The behavior of any digital system can be described using one of the hardware description language (HDL). These languages can be used to describe the system behavior specifications. The graphic representations that contain data flow and control flow are also used to represent the system behavior specifications. The data flow graph (DFGs) model has proven to be an efficient model in showing the data dependencies between the tasks of a given algorithm and exposing their inherent parallelism.

A data flow graph is a directed graph that consists of nodes and edges. The mathematical notation to represent graph G is $G = (V, E)$; where V is the set of the nodes that represent the operations and E is the set of the edges of the graph that represent the communication paths. Each node v has input and output data ports that connect it to other nodes through edges. A node has a computational delay due to the operation represented by this. Each edge e is associated with a pair of nodes and it has a non-negative number called an ideal delay [4].

The nodes of a graph are represented by $v_1, v_2, v_3, \dots, v_N$, and its edges are represented by $e_1, e_2 \dots$. A directed edge which is referred to as an arc is represented by $e = (v_m, v_n)$, where v_m is the source node and v_n is the destination node. The source and destination nodes are called the end nodes. This edge represents the precedence constraints between these end nodes.

A path is a sub graph which contains a set of finite number of nodes and a set of edges that connect between nodes. Within the path, if the source node is the same as the destination node, this path is called a loop or a directed circuit. In the loop, there should be at least one ideal delay token (node input or output) to be computable. If there is at least one loop in a DFG, then the DFG is called cyclic, otherwise it is acyclic.

Each time a node receives data tokens as inputs via its input arcs and processes them due to its operation, it produces output tokens into its output arcs. More than one node can be executed concurrently. Therefore, a data flow model is successful in exhibiting parallelism in digital algorithms. This model considers only data dependency between nodes.

DSP applications are commonly represented using signal flow graph and DSP block-diagram [5]. In a DFG model, operations of a DSP application are represented by nodes and the data dependency (signal path) between nodes are represented by arcs (edges) [6] as shown in Figure 1.

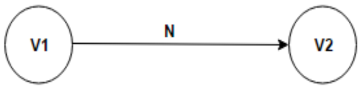

Figure 1: DFG model to represent a DSP application.

The number associated with a node identifies the index of a corresponding operation (V1 and V2 in Figure 1). There is a non-negative number associated with each edge that represents its ideal delay (N in Figure 1). A non-zero ideal delay N connecting node V1 with V2 means that node V2 depends on the output of V1 that is produced N iterations back. In this case, an edge (represents a FIFO buffer) is used to store the produced results from different iterations. The number associated with each edge is omitted if the ideal delay is zero.

For example, Figure 2 shows a block-diagram of the second order infinite impulse response (IIR) filter, and Figure 3 shows its signal flow graph. In the filter representation, symbols a, b, c and d are the filter coefficients. In Figure 4, a DFG representation of the second order IIR filter is depicted. As shown, the nodes of the graph are addition operation nodes (the "+" symbol) and multiplication operation nodes (the "*" symbol). There is a computational delay for every node due to its operation and a positive ideal delay for some edges representing the inter-iteration dependencies.

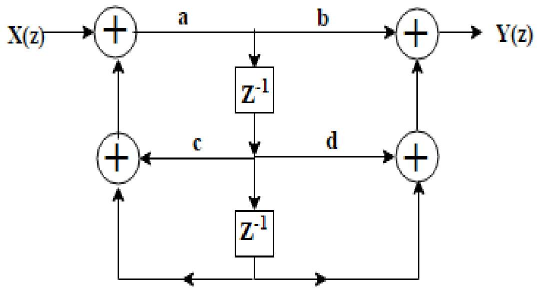

Figure 2: Block diagram representation of a second-order IIR filter.

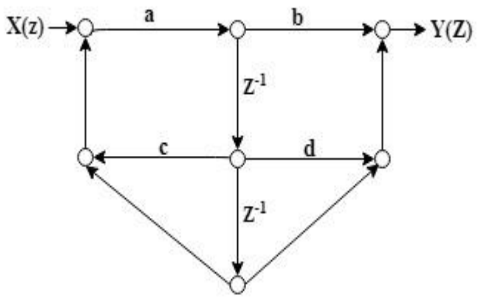

Figure 3: Signal flow graph representation of a second-order IIR filter.

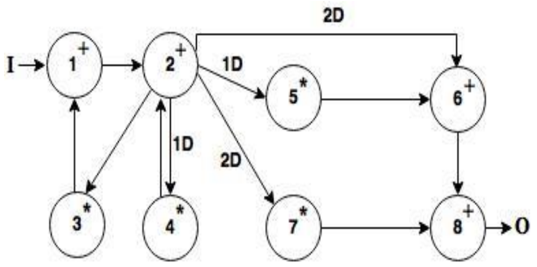

Figure 4: DFG representation of a second-order IIR filter

2. Scheduling and Resource Allocation

In this process, each operation in a DFG is assigned to control steps. These control steps are clock cycles in the synchronous system. A scheduling process must preserve precedence constraints between tasks. A program presented by a graph may be executed once or repeated many times.

The target from the scheduling in this case is to minimize the finishing time, which increases the speed to meet the timing constraints. It can also be designed to minimize the area to meet the resource constraints. In the case of timing constraints, the scheduling algorithm parallelizes operations. However, in the case of resource constraints, it serializes operations scheduling. As DSP algorithms are generally periodic, the scheduling process for them are periodically repeated. This period is referred to as the iteration period. The iteration period in a cyclic graph has a lower bound (called the iteration bound), which is the minimum possible time between successive outputs.

Precedence constraints should be met to preserve the algorithm input-output behavior specifications. There are two types of precedence constraints*:* intra-iteration precedence constraints and inter-iteration precedence constraints [7].

Intra-iteration precedence constraints are represented by arcs with zero ideal delays [7]. This means that within the arc $v \to u$, node v produces a token at the nth iteration and node u consumes this token at the same nth iteration. Node v should be scheduled and executed before node u within the same iteration. If we remove all arcs with non-zero ideal delays (inter-iteration precedence constrains) from a cyclic DFG, an acyclic DFG is produced. This graph will have only intra-iteration precedence constrains which is normally used for constructing the multiprocessor scheduling.

In the inter-iteration precedence constraints, nodes v and u are connected by an arc $v \to u$ with an ideal delay i. Node u depends on a token produced by node v in some previous iteration. Hence, node u can be scheduled and executed before node v. This constraint is used in constructing multiprocessor scheduling using inter-iteration parallelism to achieve minimum iteration period.

Many scheduling and assignment techniques are used in multiprocessor systems [7] [8]. Multiprocessor scheduling techniques are classified into different types depending on different criteria.

The scheduling process that is performed at compile time (before executing) is known as static scheduling. While the scheduling process that is performed at run time (during execution time) is known as dynamic scheduling. Static scheduling is used when the scheduling behavior is known at design time. This type of scheduling is used to minimize the execution overhead. The result of this scheduling can achieve the required optimality criteria. An algorithm whose behavior is represented using a synchronous DFG (SDF graph) can be scheduled statically. On the other hand, the dynamic scheduling is performed during the execution of operations. An example of this type is the conditional operation. All algorithms except SDF graph representation are scheduled at runtime [9]. In this work, the proposed technique uses a static scheduling algorithm.

Other types of scheduling techniques are based on exploring the precedence constraints between nodes for getting better results. There are two types of scheduling in this case: overlapped and non-overlapped scheduling [7]. In overlapped scheduling, some tasks in the *n*th iteration can be scheduled to execute before some tasks in a preceding iterations. On the other hand, overlapped scheduling usually achieves lower iteration period compared with non-overlapped scheduling. However, in non-overlapped scheduling, the execution of tasks within two consecutive iterations is not overlapped. So, the execution of tasks in (*n+1*)th iteration begins after completion the execution of all tasks in the *n*th iteration. The minimum achievable iteration period using this technique is equal to the critical (longest) path of the precedence graph.

Moreover, there are synchronous and asynchronous scheduling techniques depending on the system state changes and the computational and transmission delays between system elements. The synchronous scheduling technique is represented by a global clock. It is divided into two types: static and non-static. A schedule is called static if each operation is allocated to the same processor at all iterations. Therefore, each processor executes a unique set of nodes with a total computational delay that is not exceeding the iteration period. There are some schedules that produce a static schedule. Examples of such schedules are critical path method (CPM) schedule and schedules for systolic arrays [10].

In non-static scheduling techniques, an operation can be scheduled to run on different processors at different iterations depending on the hardware allocation algorithm. This type of scheduling includes cyclo-static periodic multiprocessor schedules. This technique depends on the time and the processor displacement. When an operation *v* is scheduled at iteration *n* in processor P_k at time t, then at iteration (*n*+1), operation *v* is scheduled on processor $P_{(K+k) \text{ modulo } N}$ at time (T+t). T is the iteration period and N is the number of the processors. If the processor displacement K is zero, a cyclo-static will be a static schedule. This schedule achieves the critical path input-output delay; thus, it is referred to as delay-optimal schedule. The drawback of this way of scheduling is the high complexity of the hardware implementation.

The scheduling techniques can be classified as either iterative/constructive or transformational.

In the iterative/constructive, the schedule of the DFG nodes is built by adding one node at a time, until all nodes of the graph are scheduled. There are many methods used for choosing nodes to be scheduled and to which processor they will be assigned. The types of iterative/constructive scheduling techniques are: As soon as possible (ASAP) scheduling [7], As late as possible (ALAP) scheduling [7], List scheduling [11], Freedom-based scheduling [12], and Force-directed scheduling [13].

In the ASAP, the nodes in a DFG are scheduled step by step from the first control step to the last control step. The node is said to be ready if all of its predecessors are scheduled. This preserves the precedence constraints. A ready node is scheduled to the earliest control step. This procedure is repeated until all nodes are scheduled. Ready nodes are executed at the current control step if there are available resources, or some ready nodes may be delayed to the next control step if there are no available resources.

The ALAP is like ASAP scheduling technique. But in ALAP, nodes of a DFG are scheduled from the last control step to the first. A node is scheduled if all of its successors are scheduled. In ASAP and ALAP scheduling techniques, no priority to the critical section of the graph is given. So, a non-optimal outcome may result.

In list scheduling, nodes of a DFG are arranged in a list depending on the precedence constraints and some priority rules. Nodes whose resources are available, are selected to be scheduled to the current control step. Otherwise, they are delayed to the next control step. In the first stage of the proposed technique in this paper, we use list scheduling to order the DFG nodes into nodes and tasks queues. This order is used to prevent processors from entering a deadlock state.

Freedom-based scheduling detects the node to be scheduled for freedom or mobility. Mobility represents the time in which the node can start execution. So, the nodes on the critical path or the loop are scheduled first. Then other nodes are scheduled one at a time based on their mobilities.

A new metric, called the force, is calculated for each token in the force-directed scheduling. It is used in selecting a token to be scheduled and in selecting a control step. The force between a token and a specific control step is proportional to the number of nodes of the same type that can be scheduled to that control step. The schedule is built by giving priority to the minimum force value for a pair of token-control steps. Then the forces are updated, and the process is repeated. This method tends to achieve a maximum utilization of resources which results in using a minimum number of hardware resources. However, its time complexity is higher than the list scheduling technique.

Most of iterative scheduling techniques use heuristics to look for efficient schedules, but they do not necessarily produce optimal ones. A transformational scheduling algorithm starts with an initial schedule that is either maximally serial or maximally parallel. Then a transformation is applied to the initial schedule to produce another schedule. Transformation algorithms differ in the operations they perform. These operations are: Exhaustive search, Exhaustive search with branch-and-bound, and Transformations with heuristics.

In Exhaustive search, all combinations (serial and parallel) are examined until getting the best schedule. This has the advantage of exploring all possible designs and tradeoffs. But it requires heavy computations and large size design.

Exhaustive search with branch-and-bound is an improvement of exhaustive search, which cuts off the search along a sub-optimal path [14]. As a result, it requires less computations and it is suitable for complicated designs.

In the transformations with heuristics, rules are used to achieve the best transformation which possibly meets the specified constraints.

The optimality criteria are related to the completion time of the scheduling process and the number of resources that have been used in the hardware implementation. A time of the scheduling process is taken to be the iteration period or the average number of control steps per cycle. A schedule must meet the constraint that the execution of all tasks with one iteration should not exceed the iteration period. The chip area is determined based on the number of hardware units used. This number is lower bounded by the processor bound. There are other objectives to be met in optimal schedules. Examples of such objectives are power dissipation, library of cells, clock skew, and package selection.

Bounds of the optimal schedules are: Iteration period bound, Processor bound, and Input-output delay bound.

In a cyclic DFG, a node cannot start execution unless all tokens are available. The time elapsed between two

consecutive firings of a node in the loop is known as the loop bound.
The loop bound of a loop L is defined by:

$$\frac{D_l}{N_l} \quad (1)$$

Where D_l is the total computational delay of all nodes in the loop L, and N_l is the total number of ideal delay elements in the loop L.
In the hypothetical case of having infinite resources, the loop bound of some loops can reach the minimum iteration period which is known as the iteration period bound [15] (the minimum average time between successive outputs). When the schedule iteration period is equal to the iteration period bound, this schedule is said to be rate optimal. The iteration period bound T_0 is given by the maximum value of the iteration bound among all loops in the DFG and is mathematically presented by Equation 2.

$$T_0 = Max\left[\frac{D_l}{N_l}\right] \quad (2)$$

If the loop bound of a given loop is equal to the iteration period bound, this loop is said to be a critical loop. In a non-critical loop, there is a time difference between the iteration period bound and the loop bound that is called the slack time [16]. The *slack time* of the loop *C* is defined by

$$ST(C) = T_0 N_C - D_C \quad (3)$$

Where N_C is the total ideal delays of the loop C and D_C is the sum of the total computational delays. The slack time of the circuit is equal to the negative of the length of this circuit, that is

$$ST(C) = -len(C) \quad (4)$$

A loop is more critical than another loop if it has a lower slack time. For acyclic DFG and in case of using unlimited resources, the iteration period is limited by the longest operation computational delay.
When using the minimum number of processors in the scheduling process of a DFG (Processor-Optimal Schedule), the lower bound of the number of processors is defined by:

$$P_0 = \left\lceil \frac{D}{T} \right\rceil \quad (5)$$

Where P_0 is the Processor Bound [8], D is a total computational delay of nodes, and T is the iteration period. From [17], in the pipeline execution, the processor bound is given by

$$P_{i0} = \left\lceil \frac{n_i * ti}{p_i * T} \right\rceil \quad (6)$$

Where i is the type of the processor, P_i is the pipelining level of the processor, n_i is the number of nodes of type i and t_i is the computational delay of processor i.
Because of precedence constraints, a processor bound cannot be achieved for some iteration period. That is the precedence constraints may prevent the optimization scheduling process.
The input-output delay is the time consumed when the input node of the graph that takes the data token to the output node of the graph produces the output token. The minimum input-output delay is the longest path from the input node to the output node. This path is called the Delay Bound (L_0) and is defined by:

$$L_0 = \max_{P \in I/O\ path} len\ P \quad (7)$$

In resource allocation, the function units are assigned to execute nodes (operations), storage units (registers) to save tokens, and wires to establish communication paths (buses and multiplexers) that are used for data transfer. While assigning the tasks of a DFG to hardware resources, minimizing the number of hardware resources is sought. This can be achieved by grouping the software elements (nodes and data values) that have the same lifetime into groups to share a single hardware element. This type of allocation is affected by the type of resources and is called folding [18].
There are two types of hardware functional units: homogenous and heterogeneous. The homogenous functional units are similar and perform the same operations. They are suitable for the synthesis of Arithmetic Logic Units (ALUs) and general-purpose DSP tasks. Systems with homogenous processing elements have higher flexibility and scalability compared with those with heterogeneous processing elements. A heterogeneous system uses more than one type of functional units. Different functional units perform different operations. Generally, heterogeneous systems have higher power efficiency and smaller area than homogeneous systems.
Variables and results that will be used later are stored in storage units (memories or registers). Each variable has a life time which represents the time difference between the variable production and the variable latest consumption. Storage units that are used in the system may be single or multiport memory or registers
Communication paths are used to connect hardware units with each other to allow data transfer between system units as per the precedence constraints. These paths are generally formed from wires or multiplexers. Shard communication is a more complicated design and slower, but it requires less wiring than multiplexers which are better in system design. Communication allocation should minimize communication paths as much as possible.
The goal of the hardware allocation techniques is to reduce the number of processing elements, memories, registers, multiplexers, and the size of the interconnection network [19]. Allocation techniques are grouped into two categories: global and iterative/constructive. In the global techniques, exhaustive search is used to find the number of allocations simultaneously. The iterative/constructive techniques allocate one resource at a time. It is more likely to reach optimal solutions by using iterative/constructive techniques.
Global techniques include graph-theoretic formulation, mathematical programming, and branch-and-bound algorithms. These techniques can achieve optimal solutions; yet they demand complex computations and require high processing power. Thus, they may be limited to small-size problems.
One of the graph theoretic formulation performs hardware allocation by constructing a graph in which nodes represent operations, data, and interconnections. This technique attempts to find the set of connected subgraphs. To minimize the number of hardware resources, the algorithm should minimize the number of connected subgraphs. This algorithm is called clique partitioning [20]. The disadvantage of this technique is that it requires exponential time to compute.

Branch and bound technique explores a design space of the data path and searches for the optimal result. It tries each element with all possible corresponding hardware elements, keeps the best solution and cuts off the others. This technique needs high computation and long processing time. To overcome these difficulties, near-optimal solutions are sought [21].

The Iterative/Constructive techniques are used to find acceptable solutions that consume less processing power and computational time than global techniques. In these techniques, an iterative one-task assignment at a time is performed [19].

There should be some knowledge in advance about the hardware resources that are needed by each operation to achieve the optimized results. This knowledge includes resource computational delay, access time, and the interconnection structure and delays. The allocation process needs to know the control step at which the operations are executed and the variables produced. The goal is to use the minimum of used hardware resources. There is interdependence between the allocation operation and the time scheduling. System Synthesis might be done by hardware allocation followed by scheduling, by scheduling followed by hardware allocation or by combining and performing them simultaneously.

In this work, we propose a new scheduling technique that is targeting DSP applications represented by a DFG. It achieves the optimal scheduling in terms of the maximum throughput of a DFG using the available hardware resources. This technique is composed of two parts: software analysis of the DFG and hardware assignment of the tasks to achieve the desired algorithmic behavior. The two parts are combined to form a complete system. In the software part, the DFG nodes are ordered into a node queue depending on their inter-related data dependencies. This queue is then used to create a compound task queue. Each compound task is created by clustering two nodes together into one task. These tasks are represented by special purpose instructions to be used in a hardware system. The second part of the scheduling technique is the allocation of the tasks of the DFG on a general pipelined hardware architecture. This system stores the DFG operations as instructions into the system memory. These operations are stored in a sequence to be executed in every iteration. Two hardware systems are proposed: a serial system and a parallel system. The serial system has one processing element where all tasks are processed sequentially, whereas the parallel system uses four homogenous parallel processing elements for concurrent task execution.

The rest of this paper is organized as follows: Section 3 presents the related work. Section 4 presents the proposed scheduling technique. The hardware implementation of the proposed systems is presented in Section 5. Section 6 provides the experimental results. Finally, the conclusion is given in Section 7.

3. Related Work

Many algorithms for static scheduling of iterative data flow graphs on multiprocessing systems exist in literature.

In [22], the algorithm introduces an optimal scheduling of a cyclic data flow graph into multi homogenous processing system. It depends on fixing the iteration period and keeping the number of the processors variable until obtaining a solution. This algorithm is known as the range chart technique. A reference node is selected, then the earliest firing time and the latest firing time for every node are computed. The flexibility for each node is computed by taking the difference between these two quantities. It means the range in which the node can be executed. The node with the minimum flexibility is scheduled and chosen as a reference node. The range chart is updated until all nodes in the DFG are scheduled. In this algorithm, tasks are assigned to homogenous processing elements according to their computational delays. The output of the algorithm is a scheduling matrix that presents the allocation of the tasks to each processor and the iteration indices of the given tasks. Here, the optimality criteria include throughput optimality, delay optimality, and hardware resources optimality.

An improvement to the previous algorithm is presented in [23]. An efficient hardware implementation of an optimal scheduling algorithm of a DFG into pipelined heterogeneous processing elements is proposed. The iteration period has been decreased and less hardware resources are needed. This leads to lower time complexity. This algorithm uses the earliest firing time and the latest firing time to all nodes in a DFG. As in [22] the reference node is picked, then nodes are scheduled into heterogeneous processing elements regardless of their computational delays.

Other popular scheduling algorithms are the list scheduling algorithms which sort tasks of a graph in a list based on their priorities, followed by selecting a resource to achieve a better schedule. Some list scheduling algorithms are mentioned next.

An Incremental Subgraph Earliest Finish Time (INCSEFT) strategy is proposed in [24]. It produces a schedule for directed acyclic graph (DAG) tasks on a heterogeneous platform. The ranks of the graph tasks are calculated in a bottom-up way. Starting from a sub graph and growing it incrementally by adding the critical paths that minimize the finish time by assigning it to processors. This low complexity strategy produces effective near-optimal schedules. This approach reduces the scheduling time; it performs better for graphs with large number of nodes.

The work in [25] proposes improved predict earliest finish time (IPEFT) algorithm for list-based scheduling at compile time in a heterogeneous platform. This algorithm has two phases in scheduling DAG tasks: task prioritization phase and processor selection phase. In the first phase, a rank is calculated for each task depending on the length of the longest path from this task to the exit task. The task of a higher rank has a higher priority. In the second phase, the earliest finishing time of the task on each processor is calculated. The task is scheduled in the earliest time slot between two scheduled tasks. The scheduling time is reduced without increasing the algorithm complexity.

In [26], a scheduling algorithm for a dependency graph on multi computing heterogeneous computing system is proposed. This algorithm consists of two phases: prioritization phase and processor selection phase. In the first phase, tasks are ordered to be scheduled in the processor selection phase. The computation costs of the heterogeneous processors and the communication cost of the heterogeneous links are used for calculating the priority in acyclic DFG. Since the computational cost of task on different processors is different, the task's earliest finish time is computed by calculating the mean value of the computation and the communication costs in finding the upward rank (earliest startup time) of the tasks. The proposed algorithm takes the standard deviation of the expected computation and communication costs as a significant attribute while calculating the upward rank of the task. In the processor selection phase, the algorithm uses a list scheduling technique and duplication technique. The duplication technique is used for the entry task only, and it is replicated for all processing elements. Tasks are arranged in a priority queue in a decreasing order of the upward ranks. The task which has a higher priority is scheduled first. But a duplication of the entry task leads to redundant duplication. This would waste the resources and adversely affect the case of a bounded number of processors.

In [27], an optimal scheduling of parallel tasks in heterogeneous system is achieved. It presents a recursive task scheduling algorithm for a limited number of heterogeneous processors. This algorithm has three phases: task prioritizing, processor selection, and moving phase. In the first phase, an accumulative rank is computed, and the priority is assigned to all tasks. Then, in the processor selection phase, tasks are scheduled on processors which ensures that the latest start time for the task is satisfied. At the moving phase, all possible tasks are moved until the entry task is zero. The algorithm decreases the final schedule length. The performance of the algorithm is shown using heterogeneous earliest finish time, iterative list scheduling, and scheduling length.

There are scheduling algorithms that try to reach better performance in scheduling a DSP application by shortening the execution time and achieving parallel scheduling. In [28] the scheduling algorithm uses a clustering technique based on a chain cluster. It gathers a chain of nodes (actors) into virtual actor. Then the main graph is scheduled using priority-based scheduling algorithm (PBS). The chain of actors is scheduled by HFS algorithm on the processor that executes the virtual actor.

In [29], simple DSP applications such as FIR and IIR filters expressed as DFGs are directly mapped into FPGA implementation. To improve the performance, a retiming technique is used to achieve the required levels of pipelining. This technique is used to move delays in a DFG without making changes to the input/output characteristics or the iteration bound. It is aimed to reduce the clock period and the power consumption. Retiming is applied using the cut theorem which cuts the DFG into sub-sets and moves delays between them until obtaining the best pipelined DFG. The final implementation has been synthesized using the Xilinx FPGA devices.

4. The Proposed Scheduling Technique

4.1 Technique Overview

A list of ordered tasks is created from a cyclic DFG based on the data dependencies between its nodes at compile time. In this stage, we prepare a DFG to be executed on the implemented systems. The goal is to reduce the execution time by minimizing the number of tasks to be executed. In the first stage of the proposed technique, the nodes of the DFG are arranged into a node queue based on their data dependencies. Then, tasks are created by combining nodes in the node queue under some conditions. The created tasks are ready to be executed on a hardware system. This phase has been implemented using C++ programming language.

4.2 Cyclic to Acyclic Conversion

The first step in constructing queues is to convert a cyclic DFG into a directed acyclic graph (DAG). Figure 4 shows a DFG representation for a second order IIR filter. In this DFG, each of the edges that connect node 2 to node 3 and node 2 to node 7 has an ideal delay of two, whereas each of the edges that connect node 2 to node 5 and node 2 to node 4 has an ideal delay of one. These edges which have non-zero ideal delays are broken and replaced by their values from the previous iterations, represented by empty flags in the DAG as shown in Figure 5. The input stream is represented by a storage unit.

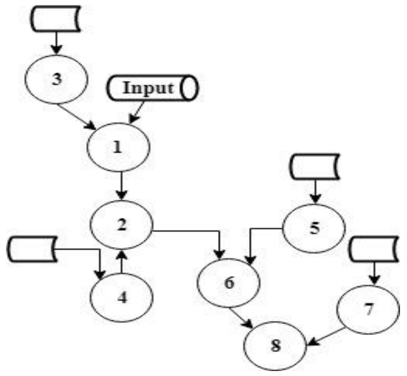

Figure 5: DAG of a cyclic DFG in Figure 4

4.3 Node Queue Constructing

Once a cyclic DFG is converted into a DAG, the nodes of the DAG are arranged into a queue depending on interrelated data dependencies between graph's nodes as follows:

- The node queue construction stage starts from node number 1 in the graph and continues until all the nodes are scheduled and placed in the queue.
- Each node in the DAG is checked for being ready to be inserted in the node queue. The node is considered ready if its operands are available (from previous iterations or from a user input). A ready node is directly inserted into the queue.
- If one or more of the node's operands are not available, their sources are checked for availability. This is repeated until a ready node is found.
- The whole process is repeated until all nodes of the DFG are scheduled.

Figure 6 shows the flowchart for construction the node queue.

4.4 Task Queue Constructing

Tasks Creation: To minimize the number of nodes in a DFG, some nodes are clustered into one compound task. Multiplication accumulation operation (MAC) is derived from merging a multiplication node that is followed by an addition node into one task. This compound operation requires less computational time than the computational time of the two nodes summed up. This reduces the overall execution time by minimizing the number of tasks to be executed.

Compound Task Queue Constructing: Some development processes are applied at this stage to improve the throughput of the system. A task creation process is applied to the node queue and the results are placed in the task queue. If there exists a multiplication node followed by an addition node in the node queue, these nodes are clustered into a compound task. This algorithm works as follow:

- After constructing the node queue and all nodes are scheduled, this stage starts.
- A node is loaded from the node queue.
- The operation of the loaded node is checked, if it is a multiplication operation the next node from the queue is loaded. Otherwise, if the node's operation is addition, the node is inserted into the task queue.
- The second node's operation is checked and if it is addition then go to the next step. Otherwise skip the next two steps.
- The first loaded node (which will be multiplication operation in this case) is checked. If any node in the DFG depends on it, a copy of this node is inserted as a single operation task into the task queue.
- The first loaded node is merged with the second node into a task and inserted as a compound task into the task queue.
- If the second node's operation is not an addition, the first loaded node is inserted into the task queue, and the second node is checked with its next node in the node queue.
- All remaining nodes in the node queue are inserted in the task queue as single nodes or as compound tasks.

Figure 7 shows the flowchart for constructing the compound task queue.

4.5 Examples

Four benchmarks of DSP filters are discussed: The second order IIR filter, all-pole lattice filter, fourth-order Jaumann wave digital filter, and the fifth-order wave elliptic filter.
A DFG representation, a node queue, and a task queue are presented for each filter. There might be one or more input and output nodes. Without loss of generality, the input stream is considered as always available at the start of the execution phase.

4.5.1 The second order IIR filter

The node queue and the compound task queue of the second order IIR filter are shown in Table 1. The scheduling is obtained based on the following steps:

- The node queue creation process starts by vising node 1.
- The inputs of node 1 are checked for availability. Since the first operand is provided by the user, the input is considered available.
- The second operand of node 1 is node 3 which is not available.
- The operands of node 3 are checked. As in a DAG, the input of node 3 is available from the previous iteration.
- Node 3 is inserted into the queue and node 1 is checked again for readiness.
- Repeat all the steps until all nodes are scheduled.

Following the flowchart in Figure 7, to merge two nodes into a compound tasks, the operations of the nodes should be a multiplication followed by an addition. For example, task 1 combines node 3 (multiplication node) and node 1 (addition node). Similarly, nodes 4 and 2, 5 and 6, 7 and 8 are also clustered into compound tasks. There are no single operation tasks in this case.

4.5.2 All-Pole Lattice Filter

The DFG of the all-pole lattice filter is shown in Figure 8 which is taken from [30]. The node queue and the compound task queue are shown in Table 1. The node queue shows the node order according to their data dependencies. Based on the flowchart in Figure 6, node ordering starts from node 1 which is inserted directly in the node queue because its operands are ready. Nodes 2, 3, 4, 5, 6, 7, 8, 9, 10, and 11 are directly scheduled in order because all of them have previously scheduled operands. After that, nodes 12, 13, 14, and 15 are scheduled.
Some nodes from the node list are merged to create the compound task queue which lists the task ID and the corresponding nodes that belong to the task as shown in Table 1.
Following the flowchart in Figure 7, to merge two nodes into a compound tasks, the operations of the nodes should be a multiplication followed by an addition. For example, task 2 combines node 2 (multiplication node) and node 3 (addition node). Similarly, nodes 5 and 6, 8 and 9, 11 and 12 are also clustered into compound tasks. Node 5 is duplicated to be in task 4 as a single operation task and as a merged node with node 6 to create the compound task 5. This is because node 5 is needed as an operand for other tasks. Nodes 8 and 11 have a similar case to node 5. Nodes 13, 14, and 15 are inserted into the task queue as single operation tasks. The total number of tasks is reduced by 6.7% in comparison with the original number of nodes in the DFG.

4.5.3 Fourth-Order Jaumann Wave Digital Filter

The DFG of the fourth-order Jaumann wave digital filter is shown in Figure 9 [30]. Similarly, the node queue and the compound task queue are constructed according to the flowcharts in Figures 8 and 9, respectively. It is clear that node 1 is not ready because its operand from node 6 is not ready or node 6 has not been scheduled yet. Therefore, node 1 is replaced by node 6 to check its readiness. From Figure 9, node 6 can be scheduled because its operands are ready (from previous iterations node 7 and node 16). The scheduling process is then repeated again starting from node 1 and the process is replicated until all nodes are scheduled.
Task 2, 5, 9 and 12 are compound tasks that are resulted from clustering 8 and 7, 12 and 11, 14 and 13, and 9 and 10, respectively. All other tasks have single nodes. The number of tasks is reduced by 18% compared with the node count in the node queue and the DFG. The node queue and the compound task queue for this filter are listed in Table 1.

4.5.4 Fifth-Order Wave Elliptic Digital Filter

Figure 10 shows the DFG for the fifth-order wave elliptic filter [30]. This DFG have more nodes than the previously mentioned filters. Table 2 contains the node queue and the compound task queue after applying the approaches presented by the flowcharts in Figures 8 and 9. In this case, constructing the node queue starts from node 2, because node 1 represents the user input. Operands of node 2 are ready from previous iterations and from user input. Thus, it is scheduled first. From Table 2, the percentage of the compound tasks is higher. In fact, the task count is reduced by 25.7% compared with the node count in the original DFG.

Table 1. The node queue and the compound task queue of the IIR, the all-pole lattice filter, and the 4th-order jaumann Wave digital

IIR filter			all-pole lattice filter			4th-order jaumann Wave digital filter		
Node Queue	Compound Task Queue		Node Queue	Compound Task Queue		Node Queue	Compound Task Queue	
Node ID	Task ID	Nodes Within Task	Node ID	Task ID	Nodes Within Task	Node ID	Task ID	Nodes Within Task
3	1	3&1	1	1	1	6	1	6
1	2	4&2	2	2	2&3	8	2	8&7
4	3	5&6	3	3	4	7	3	1
2	4	7&8	4	4	5	1	4	17
5			5	5	5&6	17	5	12&11
6			6	6	7	12	6	2
7			7	7	8	11	7	3
8			8	8	8&9	2	8	15
			9	9	10	3	9	14&13
			10	10	11	15	10	4
			11	11	11&12	14	11	5
			12	12	13	13	12	9&10
			13	13	14	4	13	16
			14	14	15	5		
			15			9		
						10		
						16		

Table 2. The node queue and the compound task queue of the 5th-order wave elliptic filter

5th-order wave elliptic filter			5th-order wave elliptic filter (continue)		
Node Queue	Compound Task Queue		Node Queue	Compound Task Queue	
Node ID	Task ID	Nodes Within Task	Node ID	Task ID	Nodes Within Task
2	1	2	12	19	25
11	2	11	19	20	26&27
17	3	17	21	21	24
28	4	28	23	22	30
22	5	22	20	23	31&32
18	6	18&16	25	24	29
16	7	10	26	25	34
10	8	8&7	27	26	33&35
8	9	6	24		
7	10	5&3	30		
6	11	4	31		
5	12	9	32		
3	13	13	29		
4	14	14&15	34		
9	15	12	33		
13	16	19	35		
14	17	21&23			
15	18	20			

5. Hardware Implementation of the Proposed Systems

Two hardware systems have been proposed and implemented in this work. The first system processes the data serially using a single pipelined processor, whereas the other system uses multiple pipelined and homogenous processors to process the data in parallel. Both systems are used for processing DSP applications that are represented by a DFG. The hardware implementations of the systems have been carried out using Verilog HDL and targeting different Xilinx FPGAs.

5.1 Instruction Format

Each task in the task queue is represented in the form of an instruction that is presented in Figure 11. The instruction is 41-bit wide and contains all of the data needed for execution. There are eight different fields in the instruction. These are:

- The task's identifier (6 bit).
- Operand1 identifier: Represents the ID of the task that produces operand1. (6 bit).
- Operand2 identifier: Represents the ID of the task that produces operand2. (6 bit).
- Operand1 iteration number: The number for the iteration in which operand1 is produced. (2 bit).
- Operand2 iteration number: The number for the iteration in which operand2 is produced. (2 bit).
- The multiplication factors: 16-bit immediate data that is used in the multiplication process. (16 bit).
- Operation code: represents the operation in the task. (addition, multiplication or add-multiply) (2 bit).
- Last_task bit, which is set to 1 for the last task in the task queue of the DFG.

5.2 Hardware Implementation of the Serial System

The pipelined serial system consumes less resources in comparison with the parallel one. In this system, only one task is executed every clock cycle because only one processing element exists. The system consists of seven main units: processing element, address generator, main memory, instructions buffer, state table, multiway function buffer, and execution array. All units are driven by an external clock. Figure 12 shows a block diagram for the serial system. The processing element executes tasks sequentially. It supports addition, multiplication, and multiplication-addition operations. Three stages pipelined multiplier and a single stage adder Intellectual Property (IP) cores are used. Once the processing element is free the *ALU_Ready* signal is set and a new task is loaded from the execution array for execution. The constant operands are embedded within the instruction (the task). The operation performed by the processing element varies according to the task itself. The result from the processing element *ALU_Result* is forwarded to other units in the system to be used by other tasks. The block diagram for the processing element is shown in Figure 13.

The address generator is shown in Figure 14. The output of the address generator is connected directly to the address lines of the main memory. The address is incremented sequentially at each clock cycle. The output of the address generator is initially set to 0. Therefore, the instructions are stored sequentially starting from location 0 in the main memory. In addition, the output of the address generator is connected to the address lines of the state table and the multiway function buffer through MUXs to allow clearing all locations in these two units. Once all instructions are stored in the main memory, the *Reset* signal is used to clear the output of the address generator to 0.

Based on the address from the address generator, an instruction is fetched from the main memory and stored in the instructions buffer. The *EC* signal is used to enable or disable the unit. It is controlled by the *Buffer_full* signal and bit number 6 in the instruction. If the instruction buffer is full, the *Buffer_full* signal is asserted in order to stop incrementing the address. Similarly, when reaching the last instruction in the iteration, the address generator stops incrementing until a new iteration starts. At the beginning of each iteration, two control signals are used to clear the address to 0 and the next iteration is started. These two signals are the *Rb* and the *Rd* signals which come from the multiway function buffer and the state table, respectively. More discussion about these two signals is provided later.

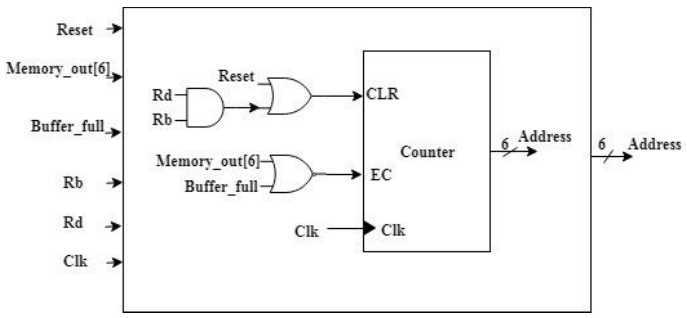

Figure 14: The address generator in the serial system

There is a single shared main memory in the system that is used to store instructions. The instructions are initially loaded from an external file *Data_In*. At each clock cycle, based on the address generated by the address generator, an instruction is fetched from the main memory into the instructions buffer *Memory_out*. The main memory is shown in Figure 15.

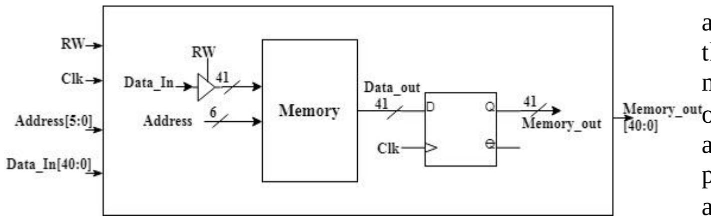

Figure 15: A block diagram for the main memory in the serial system.

The system has one instructions buffer which is shown in Figure 16. It holds the tasks (instructions) that have been fetched from the memory until the execution array is ready to receive them. The first four instructions in each iteration are loaded directly to the execution array. Once the execution array is free, it sends a ready signal *Ready* to the instructions buffer and one instruction *Buffer_out* is loaded to the bus that is connected to the execution array and the state table. The *Buffer_full* signal alerts the address generator that the buffer is full. The buffer is cleared at the beginning of any new iteration when receiving the *Rb* and the *Rd* signals.

As presented in Figure 17, the state table is a dual output one-dimensional 1-bit wide memory array that is addressed by the IDs of the operands in the task. It provides the readiness state of all tasks for the current iteration. Initially, each cell in the state table is cleared to 0 using the *clear* signal. An operand is considered ready if it belongs to a previous iteration or to the current iteration and its corresponding cell in the state table is 1. In our case, we save the operands produced by any task for the most recent four iterations. If the operand is ready, its ID and the iteration number are written into the outputs (*S1* for operand1 ID, *S2* for operand2 ID, *CTR [1:0]* for iteration number of operand1 and *CTR [3:2]* for iteration number of operand2) that are connected to the multiway function buffer. If the operand is not ready, 0 values are placed on its corresponding outputs. The state of each task is updated directly after the execution and its content is cleared at the end of each iteration.

The hardware implementation of the multiway function buffer in the serial system is shown in Figure 18. The multiway function buffer holds the ready operands of each task for the most recent four iterations. Initially, the buffer is cleared to 0's; it then starts receiving ready operands from the processing element. The ready operands are read from this unit using the operand IDs and the iteration number that come from the state table (*S1*, *S2*, and *CTR*). The operands along with their IDs and iteration numbers are sent to the execution array unit through the *Src1* and *Src2* outputs. During each iteration, the multiway function buffer is updated with the new result produced by each task, *ALU_Result*. At the end of the iteration, the buffer content is updated for the next iteration and the *Rb* signal is generated.

Figure 19 shows the hardware implementation of the execution array unit in the serial system. The execution array consists of four cascaded stages (EA1, EA2, EA3, and EA4). The inputs to this unit come from the instructions buffer and the outputs are connected to the processing element. Each task enters this unit through stage EA4 and passes through the other stages until it reaches the processing element. During this journey, the execution array guarantees that all operands of the task are ready before starting the execution. This is done by comparing the IDs of the operands in the task with the IDs of the operands that are produced by the task that has just finished execution over the processing element and has produced the *ALU_Result*. Also, by comparing with the IDs of the operands that have been read from the multiway function buffer through the *Src1* and the *Src2* outputs. At any time, four tasks can exist in the execution array (one task in each stage). Once the task that holds the processing element is done, the *ALU_ready* signal is asserted and the task in the EA1 stage starts running on the processing element. At the same time, the *Ready* signal is used to shift up the tasks in the other stages from one stage to the other and a new task is inserted into stage EA4. The first four tasks are directly shifted up without watching the ready signal.

In the EA4 stage, two 16-bit registers R1, and R2 are used to hold the two operands of the task when they are ready. These two registers are initially cleared to 0. If any of the operands is not ready, the value of the corresponding register is kept at 0. The two registers are concatenated with the instruction and the resulting 73 bits are forwarded to the other stages. In any of the stages, if an operand becomes ready after being not ready, its value is copied to its corresponding field in the 73 bits.

There are some differences in the circuit design of the registers in the execution array. In Figure 19, the dotted frame selects part of a register design. Differences are based on the register position in the execution array. This guarantees correctness at the beginning of each iteration. The register that is directly connected to the instructions buffer allows the first four tasks of every iteration to pass this stage to the next one without the need of the *Ready* signal be set. The register in the following stage allows the first three tasks move to the next stage without the *Ready* signal be set. The register in the second stage allows the first and the second tasks only to proceed to stage EA1 of the unit.

The last stage of this unit (EA1 in Figure 12) has slightly different design. It allows the first task only to move to the processing unit. After moving the task to the processing element, it generates the *Ready* signal to the other stages and to the instructions buffer. The *Ready* signal triggers the instructions buffer to load a new instruction and to pass tasks between stages. Figure 20 shows the hardware that generates the *Ready* signal in stage EA1.

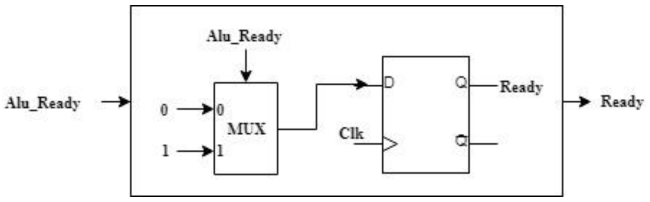

Figure 20: Generation of the *Ready* in stage EA1

5.2.1 Serial System Functionality

The serial system executes task as follows:

- Initially, the address counter is cleared to 0. Then, a descriptive information of the DFG is loaded into the main memory, and the state table and the function buffer are cleared.
- The address generator produces sequential addresses to the memory. Tasks are loaded into the instructions buffer.
- The instructions buffer checks the IDs of the tasks and directly loads the first four tasks of every iteration without saving them in its queue. (Other tasks are saved).
- Once the instruction buffer is full, it sends the *buffer_full* signal to the address generator to stop counting.
- When the instructions buffer receives the *Ready* signal from the execution array unit, it sends one task to the state table and the execution array register.
- Task's operands are checked for availability using the state table. Ready operands are sent to the multiway function buffer.
- The multiway function buffer receives the ID and iteration index of each operand, fetches the corresponding values and sends them to all stages of the execution array.
- At every clock cycle and when a stage receives a ready signal from the first stage in this unit a task is shifted up to the other stages.
- The execution array stages wait for any values coming from the multiway function buffer or from the processing element and match them with the task's operands which they hold.
- Operands values are stored within a task as soon as the *Alu_Ready* signal is asserted. Then a task is loaded into the processing element.
- After execution, the result is sent to the state table, to the multiway function buffer, and to the execution array.
- The state table and the multiway function buffer update their content according to the executed task. These units keep checking the last_task bit of the task for the end of iteration.
- At the end of each iteration, the state table and the instructions buffer are cleared, and the buffer contents are updated to be compatible with the next iteration. The *Rb* signal and the *Rd* signal from the multiway function buffer and the state table are activated to reset the address generator and a new iteration is started.

5.3 Hardware Implementation of the Parallel System

Figure 21 shows a block diagram for the parallel system. The parallel system consists of: Processing elements, address generator, main memory, instructions buffers, state table, multiway function buffer, and execution arrays. All units have similarities with the units of the serial system with some expansions or duplications in order to support parallelism. They are driven by the same external clock.

There are four homogenous pipelined processing elements to execute tasks concurrently. Up to four tasks can be executed at every clock cycle. Each processing element supports all required operations (addition, multiplication and multiplication-accumulation). The processing element in the parallel system is identical to that in the serial system, shown in Figure 13.

The address generator, which is shown in Figure 22, is responsible for generating the addresses in the system. It has similar circuit design as the address generator in the serial system with minor differences. There are four instructions buffers in the parallel system, if at least one of them is full, its *Buffer_full* signal (*Buffer_full1*, *Buffer_full2*, *Buffer_full3* or *Buffer_full4*) is asserted in order to stop the address generator. Four tasks are concurrently fetched from the memory. There are four enable signals that represent the last_task bit in each instruction. These are bits *Memory_out1[6]*, *Memory_out2[6]*, *Memory_out3[6]* and *Memory_out4[6]*. After the last instruction in the iteration, the address generator stops counting until a new iteration is started.

The main memory, which is shown in Figure 23, concurrently loads four instructions, through the *Memory_out1*, the *Memory_out2*, the *Memory_out3* and the *Memory_out4* output ports, into the instructions buffers; each instructions buffer receives one instruction.

The parallel system has four instructions buffers that are identical to the one used in the serial system.

As shown in Figure 24, the state table in the parallel system deals with four instructions that are read from the main memory. If the operand is ready, its ID and the iteration number are written into the outputs (*S1* for operand1 ID, *S2* for operand2 ID, *S3* for operand3 ID, *S4* for operand4 ID, *S5* for operand5 ID, *S6* for operand6 ID, *S7* for operand7 ID, *S8* for operand8 ID, *C1* for iteration number of operand1, *C2* for iteration number of operand2, *C3* for iteration number of operand3, *C4* for iteration number of operand4, *C5* for iteration number of operand5, *C6* for iteration number of operand6, *C7* for iteration number of operand7 and *C8* for iteration number of operand8). If the operand is not ready, 0 values are placed on its corresponding output. The state tasks are updated directly after the execution (*ALU_Result1*, *ALU_Result2*, *ALU_Result3 and ALU_Result4*). The table content is cleared at the end of each iteration.

The multiway function buffer is an expanded version of the multiway function buffer used in the serial system. The multiway function buffer receives eight operands that are outputs from the state table (*S1*, *S2*, *S3*, *S4*, *S5*, *S6*, *S7*, *S8*, *C1*, *C2*, *C3*, *C4*, *C5*, *C6*, *C7* and *C8*). Then the operands, their IDs, and their iteration number are sent to the execution array units through ports *Src1*, *Src2*, *Src3*, *Src4*, *Src5*, *Src6*, *Src7* and *Src8*). This unit receives ready operands from the processing elements through ports *ALU_Result1*, *ALU_Result2*, *ALU_Result3 and ALU_Result4*. At the end of the iteration, the bits *ALU_Result1[22]*, *ALU_Result2[22]*, *ALU_Result3[22]* and *ALU_Result4[22]* along with the buffer content are updated for the next iteration and the *Rb* signal is asserted. Figure 25 shows the hardware implementation of the multiway function buffer used in the parallel system. The execution array consists of four parallel stages (EA1, EA2, EA3, and EA4). The inputs of each stage come from the corresponding instructions buffer input (*Memory_out*) and the outputs of the multiway function buffer. The output *Array_out* of each stage is connected to one of the processing elements. Similar to the execution array unit in the serial system, this unit holds tasks in its stages until all operands of the task are ready. This is done by using the same comparison operations as in the serial system. But in the parallel system there are four operands that are produced by the tasks that have just finished execution. These operands are placed on ports *ALU_result1, ALU_result2, ALU_result3 and ALU_result4*. Once the task that holds a processing element is done, the *ALU_ready* signal for that processing element is asserted and the task in its corresponding stage starts running on the processing element. At the same time, the *Ready* signal triggers the corresponding instructions buffer to load a new instruction and to pass task into the execution array stage. Figure 26 shows the hardware implementation of the execution array unit in the parallel system.

5.3.1 Parallel System Functionality

The parallel system works as follows:

- The address is cleared to 0. A descriptive information of the DFG is loaded into the main memory and the state table and the multiway function buffer are cleared.
- The address generator produces four sequential addresses to the memory and four tasks are fetched into the instructions buffers.
- Each instructions buffer checks the task's ID and directly loads the first four tasks of every iteration without saving them in its queue. (Other tasks are saved).
- If one of the instructions buffers is full, it sends a *buffer_full* signal to the address generator to stop counting.
- Once an instructions buffer receives a *ready* signal from its execution array register, a task is sent to the state table and to the execution array stage.
- Tasks operands are checked for availability using the state table. Ready operands are sent to the multiway function buffer.
- The multiway function buffer receives operands IDs and their iteration numbers, fetches their corresponding values and sends them to their stage in the execution array unit.
- Every execution array stage waits for any values coming from the multiway function buffer or from the processing element and matches them with the task's operands which it holds.
- Operands values are stored within a task, as soon as the *Alu_Ready* signal, of the stage of the execution array, is asserted. Then a task is loaded to the processing element.
- After execution, the result is sent to the state table, to the multiway function buffer, and to the execution array.
- The state table and the multiway function buffer update their contents according to the executed tasks. These units check task's last_task bit for end of iteration.

At the end of each iteration, the state table and the instructions buffers are cleared, and the multiway function buffer contents are updated to be compatible with the next iterations. The *Rb* signal and the *Rd* signal from the multiway function buffer and the state table respectively are used to reset the address generator and a new iteration is started.

6. Experimental Results and Comparisons

The serial system and the parallel system have been synthesized targeting Virtex-7 XC7VX690T-FFG1157 FPGA device from Xilinx. The implementation has been carried out for 64 instructions (tasks), 64-bit state table, and 64 X 64 bit buffer in the multiway function buffer to maintain tasks information for four iterations. The multiplier size is 16-bit X 16-bit. Also, a 16-bit adder is used.

FPGAs nowadays are widely used to implement vast number of applications. Examples of such applications are presented in [31] and [32]. FPGAs contain a matrix of configurable logic blocks (CLBs). Each CLB has two slices. A slice has four LUTs. A LUT stores a predefined list of outputs for every combination of inputs and provides a fast way to retrieve the output of a logic function. It produces two outputs: one is registered (Using Flip Flop) and the other is not registered (combinational). When using a registered output, a slice is counted as a slice register. On the other hand, if a combinational output is used then the slice is counted as a slice LUTs. The number of used slice registers and slice LUTs in any implemented system reflects the area of that system.

6.1 Serial System Implementation Results

Synthesis results show that the design of the serial system runs at a frequency of 355.29 MHz. The area in terms of LUTs is 8867 and it consumes 4818 slice registers. Table 3 summaries the serial system FPGA implementation results.

Table 3. The serial system FPGA implementation results.

Number of Slice Registers Utilized	Number of Slice LUTs Utilized	Minimum Period Time
4818	8867	2.815 ns

6.2 Parallel System Implementation Results

Synthesis results show that the design of the parallel system runs at a frequency of 476.554 MHz. The area in terms of LUTs is 21229 and the number of slice registers is 12223. Table 4 shows the parallel system FPGA implementation results.

Table 4. The parallel system FPGA implementation results.

Number of Slice Registers Utilized	Number of Slice LUTs Utilized	Minimum Period Time
12223	21229	2.098ns

6.3 The Serial System versus the Parallel System

Both systems are compared in terms of the number of used slice registers, the number of used LUTs, and the frequency. Figure 27 shows the comparison between the serial system and the parallel system in terms of the number of slice registers that are used in the circuit design. The comparison is done for different number of instructions. When the systems are dealing with 64 instructions, the parallel system occupies 12223 slice registers in comparison to 4818 slice registers in the case of the serial system. As shown in Figure 27 the number of occupied slice registers (for both systems) increases with the number of instructions. The increase in the serial system is approximately 50% and it is doubled in the parallel system. Moreover, the gap between the number of slice registers occupied by the parallel system and that of the serial system also increases with the number of tasks or instructions. Table 5 shows the number of occupied slice registers by the serial system and by the parallel system and the difference between them for different number of instructions.

Table 5: The number of occupied slice registers by both systems for different number of instructions.

Number of instructions	Number of occupied slice registers		
	Serial System	Parallel System	Difference between both
64	4818	12223	7405
128	9113	18360	9247
256	17520	32264	14744
512	34537	66164	31627

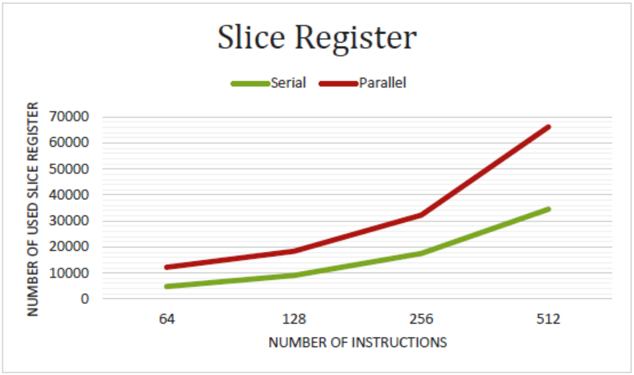

Figure 27: The number of occupied slice registers by both systems for different number of instructions.

The number of Slice LUTs gives an idea about the system area. As presented in Figure 28, starting from 64 instructions, the serial system consumes 8867 LUTs. When the number of instructions is increased to 128, the serial system consumes 17207 LUTs, and it consumes 31753 LUTs in the case of using 256 instructions. When the number of instructions reaches 512, 50428 slice LUTs are consumed.

On the other hand, in the parallel system, 21229 LUTs are consumed for the case of 64 instructions. This number increases with the number of instructions. For example, in the case of 512 instructions, 170671 LUTs are needed to build the system. Obviously if the number of instructions in the systems is increased, more slice LUTs are needed. This is due to the increase of size of the main memory, the multiway function buffer and the state table. Table 6 shows the number of occupied slice LUTs by both systems and the difference between them for different number of instructions.

Table 6: The number of occupied slice LUTs by both systems for different number of instructions.

Number Of instructions	Number of occupied slice LUTs		
	Serial System	Parallel System	Difference between both
64	8867	21229	12362
128	17207	44521	27314
256	31753	72888	41135
512	50428	170671	120243

Figure 29 shows the frequencies for both the serial and the parallel systems for different instructions counts. It can be said that there is almost no variation in the frequency for the serial system as the number of instructions increases. For example, it is 355MHz for the 64 instructions case and 351MHz for the 512 instructions case. The frequency in the parallel system slightly changes as the number of instructions increases. For the case of 64 instructions, the frequency is 477 MHz and it is 441MHz for the case of the 512 instructions. In general, the frequency in the parallel system is higher than that of the serial system.

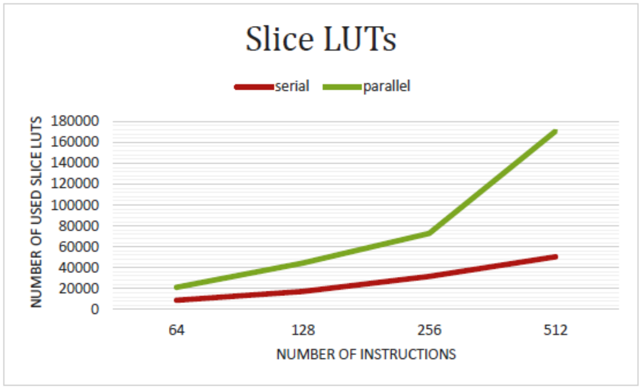

Figure 28: The number of occupied slice LUTs by both systems for different number of instructions.

Also, system frequencies decrease as the size increases, because the system becomes bigger, and it consumes more routing time and needs more time to update memories. The serial system performance is almost constant but in the parallel system the decrease in the system frequency can be up to 4.5%.

Table 7: Frequency in MHz for both systems for different number of instructions.

Number of instructions	Frequency in MHz		
	Serial System	Parallel System	Difference between both
64	355	477	122
128	355	470	115
256	355	462	107
512	351	441	90

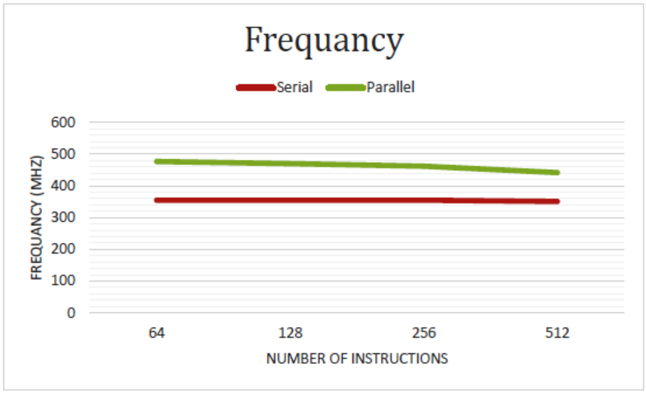

Figure 29: Frequency in MHz for both systems for different number of instructions.

6.4 The Proposed Systems versus the Retiming Technique in [29]

Table 8 shows a comparison between the two systems that are presented in this paper and the retiming technique presented in [29]. This technique uses the retiming scheduling algorithm to improve the performance of DSP applications represented by DFGs. Retiming is applied using the cut theorem which cuts a DFG into sub-sets and moves delays between them until obtaining the best pipelined DFG. All of the systems have been synthesized using the Xilinx Virtex-5 FPGA.

Table 8 shows the system clock frequency for the three designs. The proposed parallel system achieves the best performance with a clock frequency of 266 MHz as compared to 210 MHz and 176 MHz in the proposed serial

system and the retiming technique, respectively. The parallel system achieves 51.2% higher performance than the retiming technique, while the serial system achieves 19.3% higher performance than the retiming technique.
The comparison in term of DSP48 devices that are used in the system implementation shows that both the serial system and retiming technique require two DSP48 devices each, while the parallel system requires three DSP48 devices.

Table 8: Comparison between the proposed systems and the retiming technique [29].

Circuit/System	CLK (MHz)	DSP48
Serial System	210	2
Parallel System	266	3
Technique of [29]	176	2

7. Conclusions

This paper proposes a new scheduling technique that is targeting DSP applications represented by a data flow graphs (DFGs). The technique is composed of two parts: software analysis of the data flow graph and hardware assignment of the tasks to achieve the desired algorithmic behavior. The scheduling technique is designed to minimize the execution time of a single iteration of the DSP application and thus maximizing system performance. In the software part, the nodes are arranged in a queue of nodes such that when the tasks associated with these nodes are executed in order, data dependencies are preserved. Two or more tasks may be combined to form a compound task. In many cases, the compound task can be executed in a time that is less than the sum of the execution times of the individual tasks. For example, when a multiplication operation is followed by an addition, which is very common in DSP applications, the resulting three-operand operation can be executed in less time than the time needed to perform the two operations separately. This process eventually decreases the number of execution cycles for each iteration of the iterative DFG.
The second part of the scheduling technique is the allocation of the tasks of the DFG on a general-purpose pipelined hardware architecture. This stores the DFG operation as instructions into the main memory. These operations are stored in a sequence to be executed as many iterations as needed. We have proposed two implementations of the system architecture: a serial system and a parallel system. The serial system comprises one processing element where all tasks are processed sequentially. This system is characterized by a small area size; it requires less number of slice registers and less number of slice LUTs than the parallel one. The parallel system, however, focuses on performance rather than system area. . This system uses four homogenous parallel processing elements for concurrent task execution. The relationship between the number of instructions that are loaded and executed, and the system area is studied. Similarly, the performance for different problem sizes is analyzed.
Since there are no many previous contributions in this specific field, we have chosen to compare our results with the retiming technique presented in [29]. It has been shown that our system outperforms the retiming technique in terms of the iteration time. In terms of the system area, the retiming technique and the serial version of our system require the same number of DSP slices when FPGA hardware implementation is used.

8. Acknowledgement

The authors acknowledge the support of XILINX to Jordan University of Science and Technology (JUST). Through their university program, XILINX has been donating free licenses for their CAD tools to be used for teaching and research purposes at JUST. All of the experimental results presented in this paper have been obtained using the XILINX CAD tools.

References

[1] D. DeFatta, J. Lucas, W. Hadgkiss, “Digital signal processing, a system design approach,” John Wiley & Sons. Vol.2, pp.10-660 1988.
[2] L. Trevillyan, “An overview of logic synthesis systems,” ACM/IEEE Conference on Design Automation, Miami Beach, USA, pp. 166-172, 1987.
[3] M. McFarland, A. Parker, R. Camposano, “The high-level synthesis of digital systems,” Proceedings of the IEEE, Vol. 78, No. 2, pp. 301-318, 1990.
[4] K. Thulasiraman, M. N. S. Swamy, “Graphs, Networks and Algorithms,” Wiley-Interscience publication, Vol.1, pp. 95-592, 1981.
[5] R. Schafer, A. Oppenheim, “Digital Signal Processing,” 1st ed. Englewood Cliffe, New Jersey: Prentice Hall, pp. 31-32, 1975.
[6] A. Shatnawi, “Compile-time scheduling of digital signal processing data flow graphs onto homogeneous multiprocessor systems,” Ph.D. Thesis Department of Electrical and Computer Engineer, Concordia University, Montreal Canada, 1996.
[7] K. Parhi, D. Messerschmitt, “Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding,” IEEE Transactions on Computers, Vol. 40, No. 2, pp. 178-195, 1991.
[8] P. Gelabert, T. Barnwell, “Optimal automatic periodic multiprocessor scheduler for fully specified flow graphs,” IEEE Transactions on Signal Processing, Vol. 41, No. 2, pp. 858-888, 1993.
[9] Y. Kwok, I. Ahmad, “Static scheduling algorithms for allocating directed task graphs to multiprocessors,” ACM Computing Surveys, Vol. 31, No. 4, pp. 406-471, 1999.
[10] E. Lee, D. Messerschmitt, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” IEEE Transactions on Computers, Vol. 36, No. 1, pp. 24-35, 1987.
[11] S. Davidson, D. Landskov, B. Shriver, P. Mallett, “Some Experiments in Local Microcode Compaction for Horizontal Machines,” IEEE Transactions on Computers, Vol. 30, No. 7, pp. 460-477, 1981.
[12] E. Girczyc, J. Knight, “An ADA to standard cell hardware compiler based on graph grammers and scheduling,” Proc IEEE Int Conf Computer Design, Nevada, Las Vegas, USA, pp. 726-731, 1984.
[13] P. Paulin, J. Knight, “Force-directed scheduling for the behavioral synthesis of ASICs,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 8, No. 6, pp. 661-679, 1989.
[14] L. Hafer, A. Parker, “A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic,’ IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 2, No.1, pp. 4-18, 1983.
[15] M. Renfors, Y. Neuvo, “The maximum sampling rate of digital filters under hardware speed constraints,” IEEE Transactions on Circuits and Systems, Vol. 28, No. 3, pp. 196-202, 1981.
[16] A. Shatnawi, M. Ahmad, M. Swamy, “Scheduling of DSP data flow graphs onto multiprocessors for maximum throughput,” IEEE International Symposium on Circuits and Systems, Orlando, FL, USA, pp. 386-389, 1999.

[17] C. Y. Wang, K. Parhi, “High-level DSP synthesis using concurrent transformations, scheduling, and allocation,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 14, No. 3, pp. 274-295, 1995.

[18] K. Parhi, C. Wang, A. Brown, “Synthesis of control circuits in folded pipelined DSP architectures,” IEEE Journal of Solid-State Circuits, Vol. 27, No. 1, pp. 29-43, 1992.

[19] M. McFarland, A. Parker, R. Camposano, “Tutorial on high-level synthesis,” 25th Design Automat, New Jersey, Atlantic City, USA, pp. 330-336, 1988.

[20] M. Balakrishnan, A. Majumdar, D. Banerji, J. Linders, J. Majithia, “Allocation of multiport memories in data path synthesis,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 7, No. 4, pp. 536-540, 1988.

[21] E. Demeulemeester, W. Herroelen, “A Branch-and-Bound Procedure for the Multiple Resource-Constrained Project Scheduling Problem,” Management Science, Vol. 38, No. 12, pp. 1803-1818, 1992.

[22] A. Shatnawi, “Optimal Scheduling of Digital Signal Processing Data-flow Graphs using Shortest-path Algorithms,” The Computer Journal, Vol. 45, No. 1, pp. 88-100, 2002.

[23] A. Shatnawi, J. Ghanim, M. Ahmad, “High level synthesis of integrated heterogeneous pipelined processing elements for DSP applications,” Computers & Electrical Engineering, Vol 30, No. 8, pp. 543-562, 2004.

[24] G. Wang, Y. Wang, H. Liu, H. Guo, “HSIP: A Novel Task Scheduling Algorithm for Heterogeneous Computing,” Scientific Programming, Vol. 2016, pp.1-11, 2016.

[25] N. Zhou, D. Qi, X. Wang, Z. Zheng, W. Lin, “A list scheduling algorithm for heterogeneous systems based on a critical node cost table and pessimistic cost table,” Concurrency and Computation: Practice and Experience, Vol. 29. No.5, pp. 1-11, 2016.

[26] E. Munir, S. Mohsin, A. Hussain, “SDBATS: A Novel Algorithm for Task Scheduling in Heterogeneous Computing Systems,” Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), Cambridge, MA, USA, pp. 43-53, 2013.

[27] Y. Kang, Y. Lin, “A Recursive Algorithm for Scheduling of Tasks in a Heterogeneous Distributed Environment,” 2011 4th International Conference on Biomedical Engineering and Informatics (BMEI), Shanghai, China, pp. 15-17, 2011.

[28] G. Liu, Y. He, L. Guo, “Static Scheduling of Synchronous Data Flow onto Multiprocessors for Embedded DSP Systems,” Third International Conference on Measuring Technology and Mechatronics Automation, Shanghai, China, pp. 338–341, 2011.

[29] R. Woods, J. McAllister, G. Lightbody, Y. Yi, “FPGA-Based Implementation of Signal Processing Systems,” Chichester, United Kingdom: John Wiley & Sons, pp. 145-169, 2009.

[30] S. de Groot, S. Gerez, O. Herrmann “Range-chart-guided iterative data-flow graph scheduling,” IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, Vol. 39, No. 5, pp. 351-364 ,1992.

[31] R. Kadu, D. Adane, “Hardware Implementation of Efficient Elliptic Curve Scalar Multiplication using Vedic Multiplier,” International Journal of Communication Networks and Information Security (IJCNIS), Vol. 11, No. 2, pp. 270-277, 2019.

[32] M. Zeeshan, S. Khan, “Robust Sampling Clock Recovery Algorithm for Wideband Networking Waveform of SDR,” International Journal of Communication Networks and Information Security (IJCNIS), Vol. 5, No. 1, pp. 10-18, 2013.

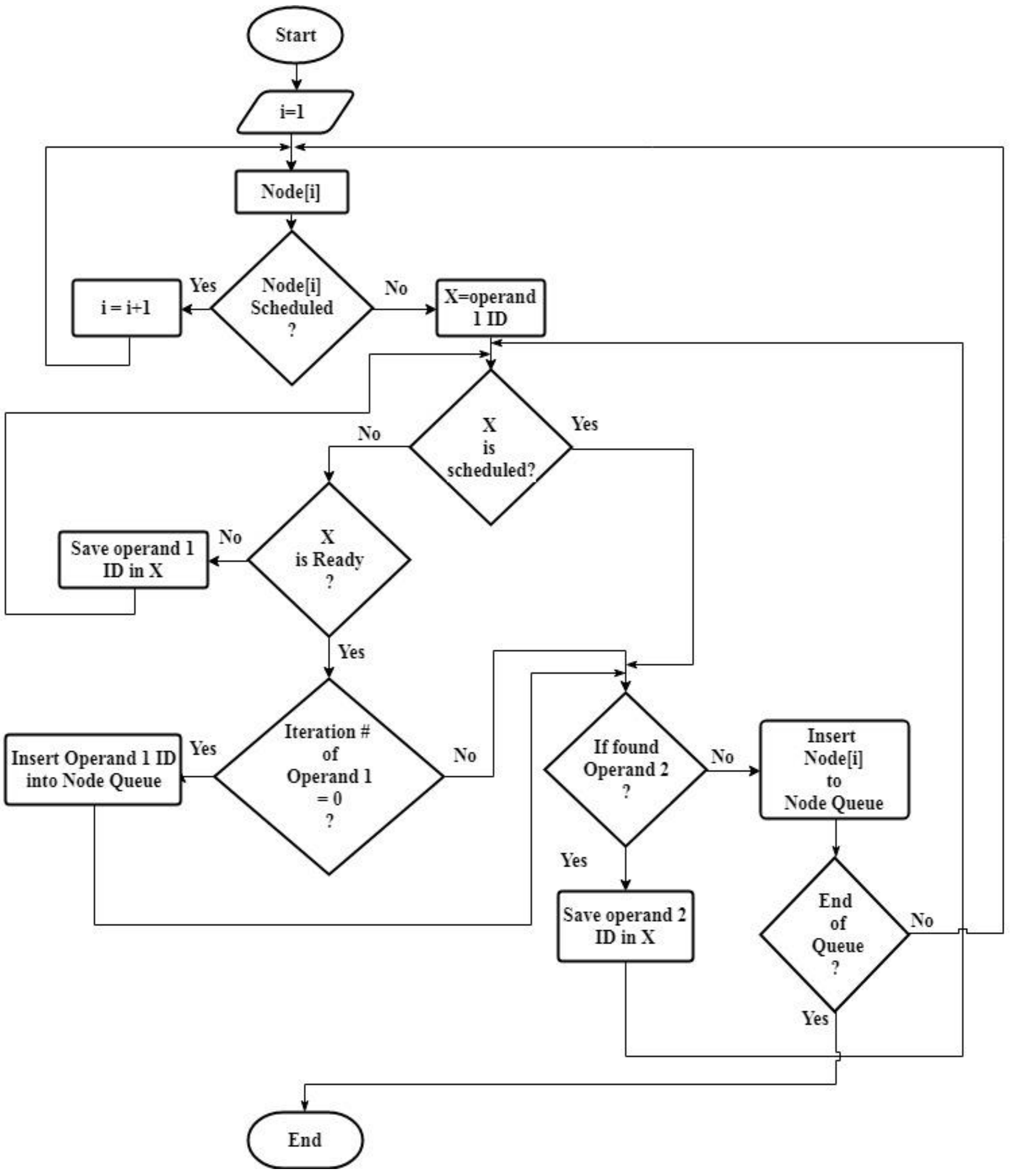

Figure 6: Flow chart of node queue constructing.

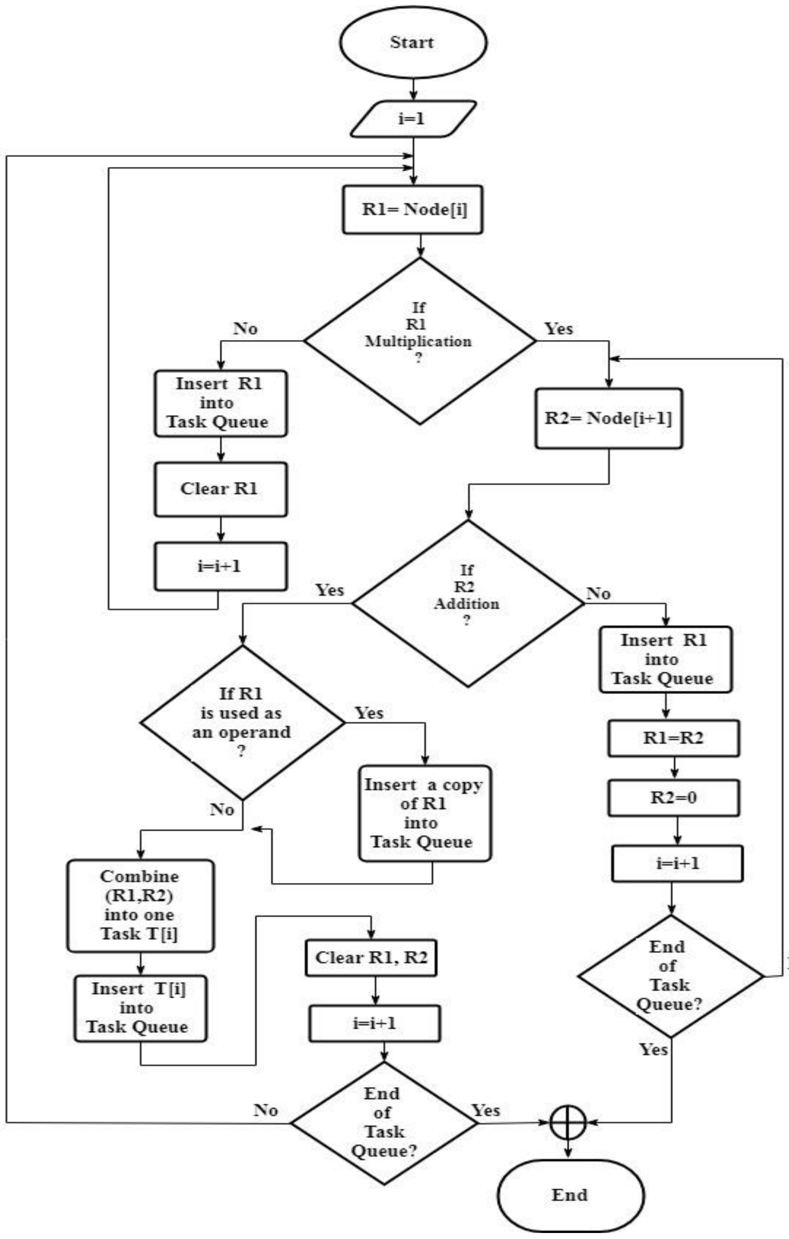

Figure 7: Flow chart of compound task queue constructing.

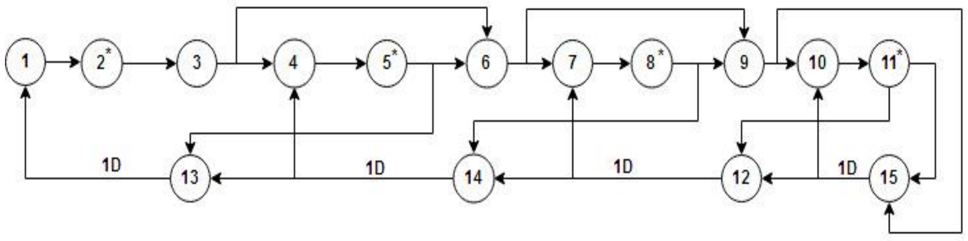

Figure 8: The DFG of the all-pole lattice filter [30].

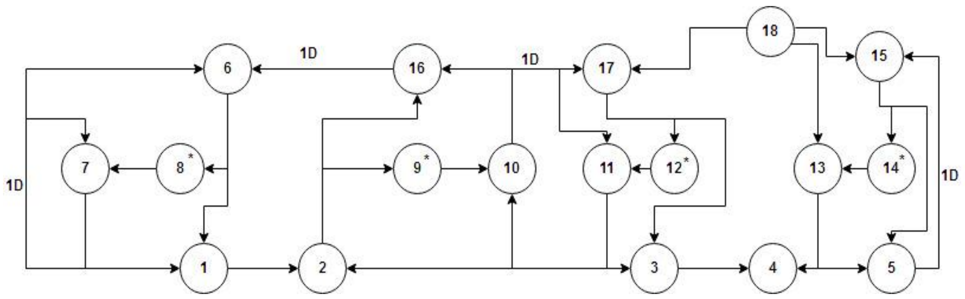

Figure 9: The DFG of the fourth-order jaumann wave digital filter [30].

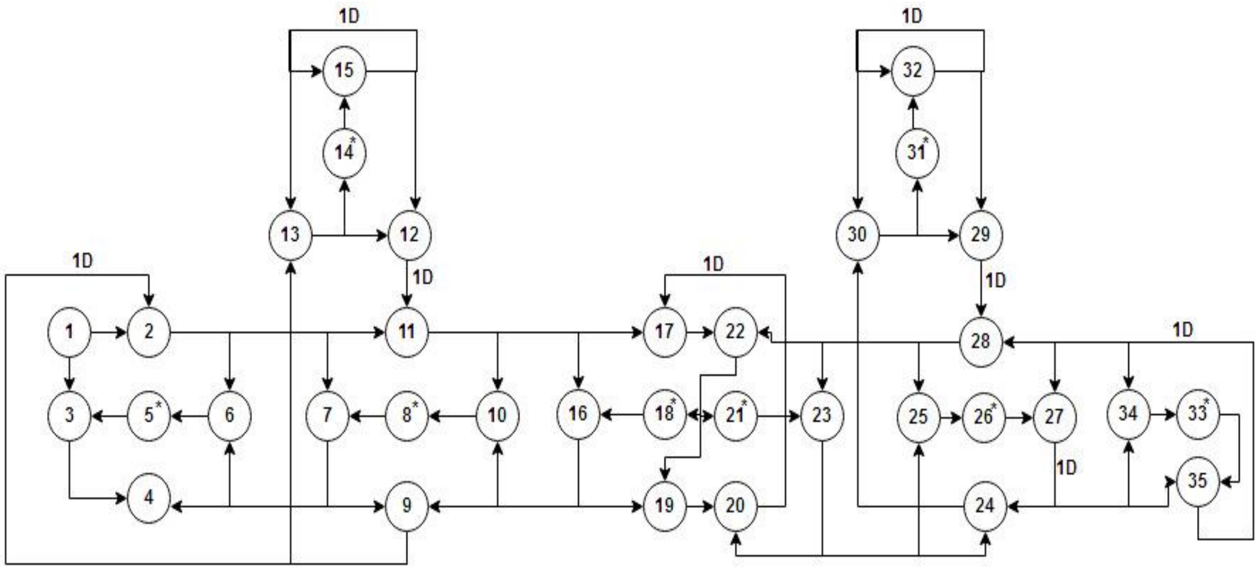

Figure 10: The DFG of the fifth-order wave elliptic filter.

40 25	24 19	18 13	12 7	6	5 4	3 2	1 0
Multiplication factor	Operand1 identifier	Operand2 identifier	Task identifier	Last_task bit	Operation Code	Operand1 iteration number	Operand2 iteration number

Figure 11: The instruction format of a DFG tasks.

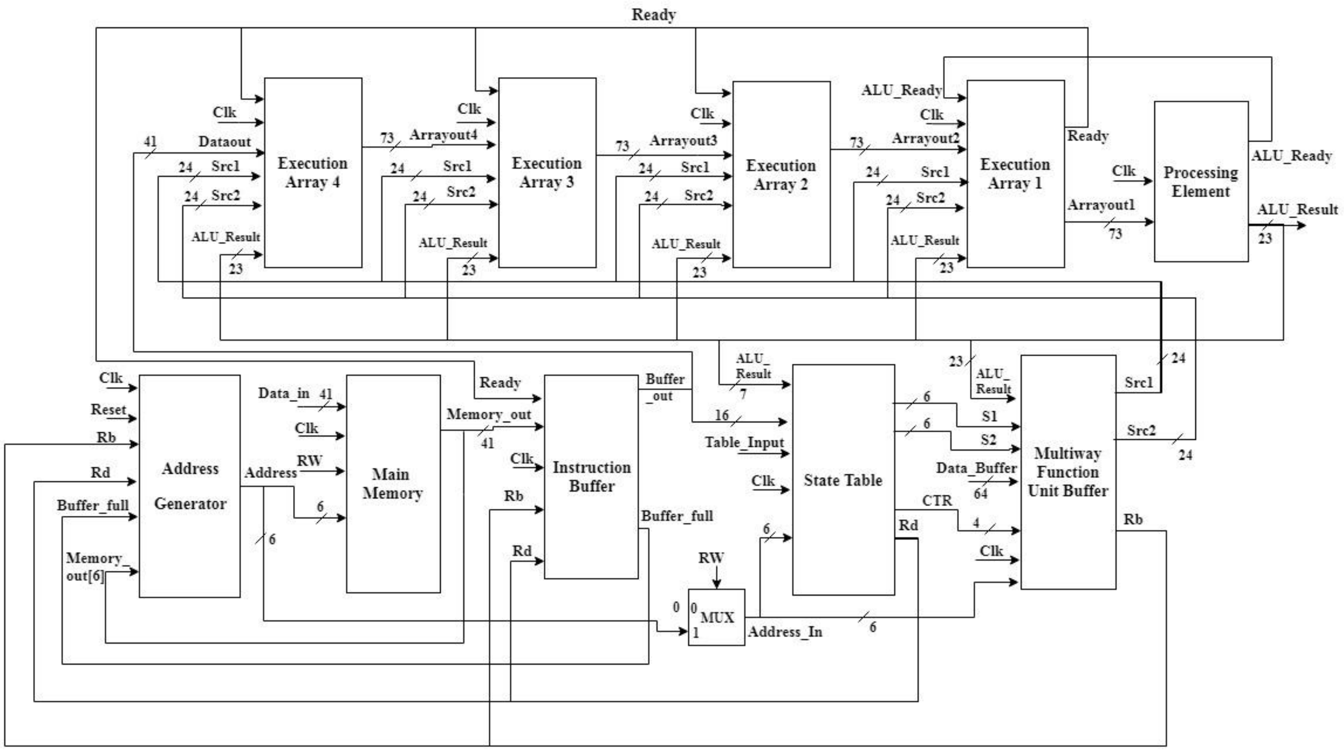

Figure 12: The block diagram for the serial system.

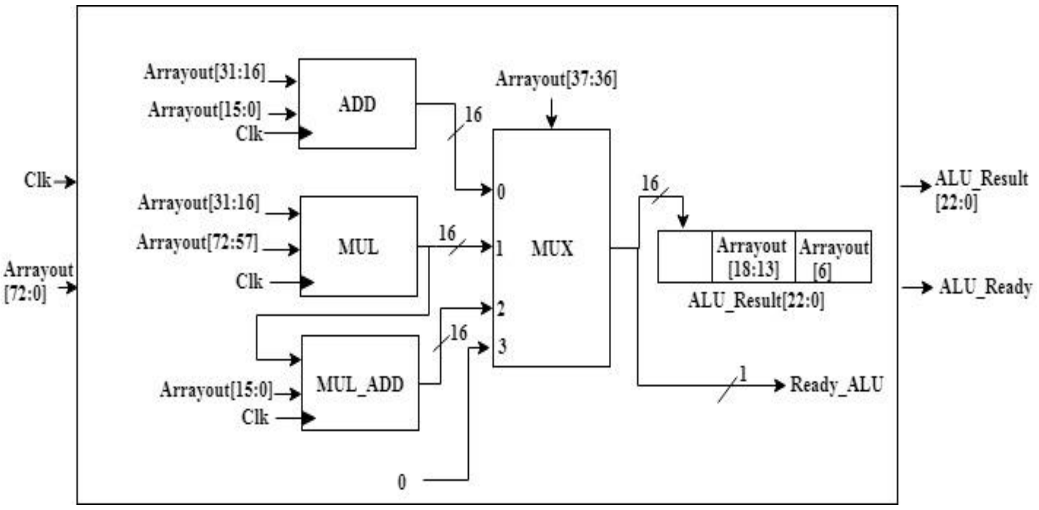

Figure 13: The block diagram for the processing element.

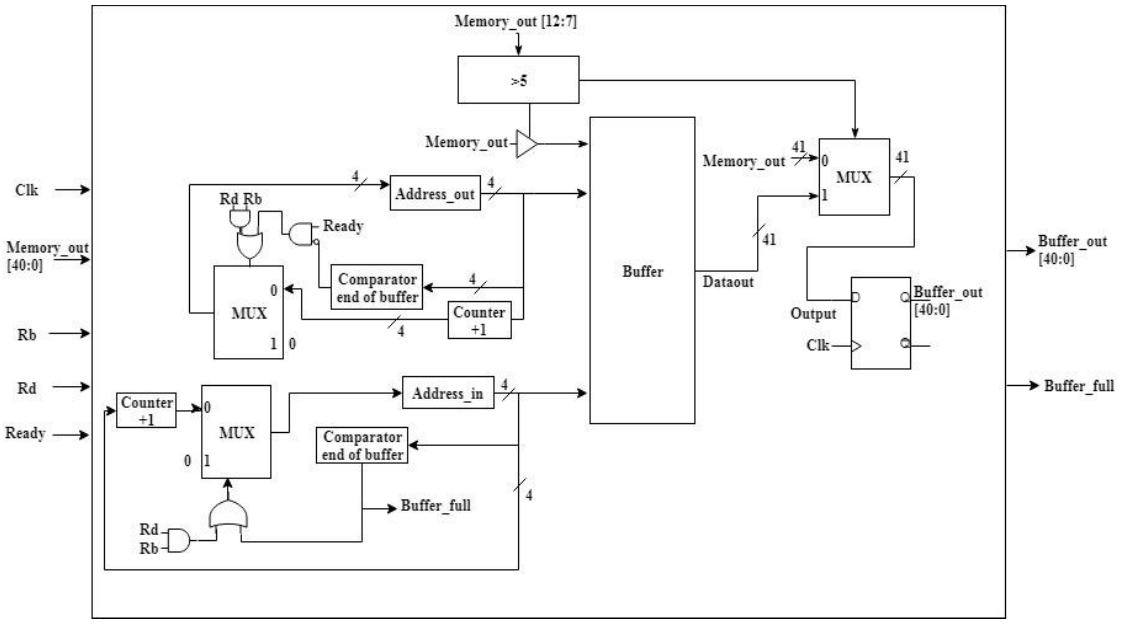

Figure 16: The hardware implementation of the instructions buffer.

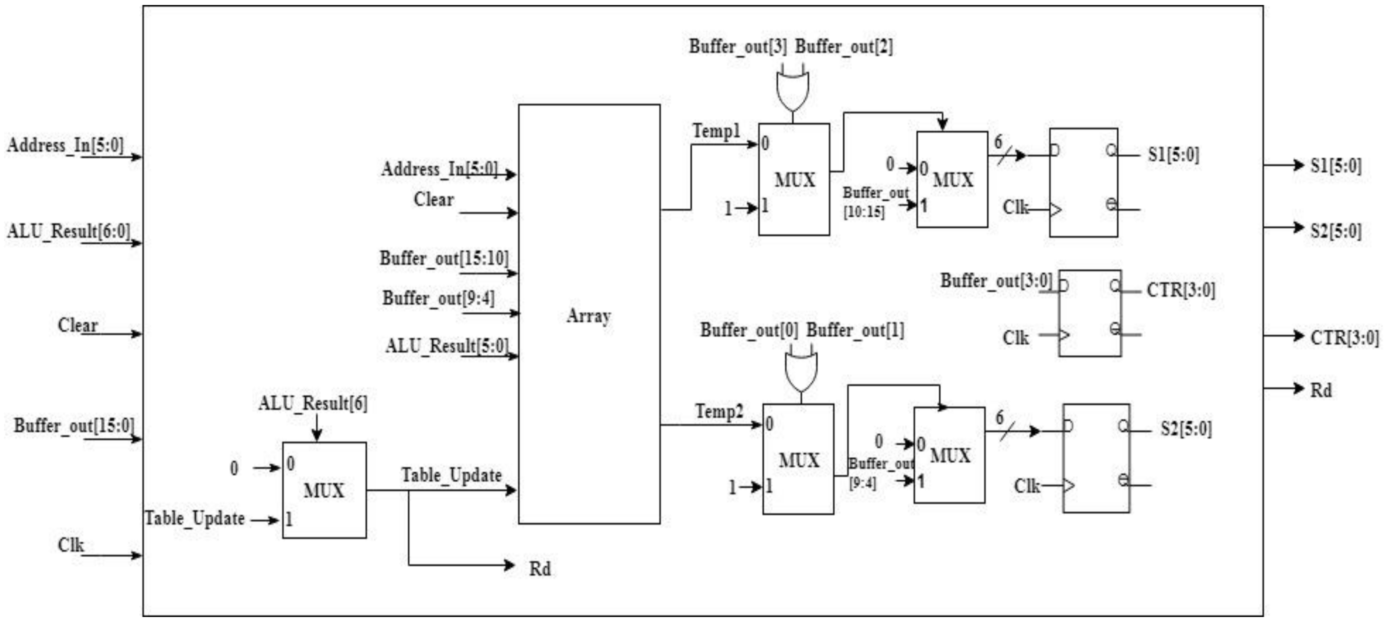

Figure 17: The hardware implementation of the state table in the serial system.

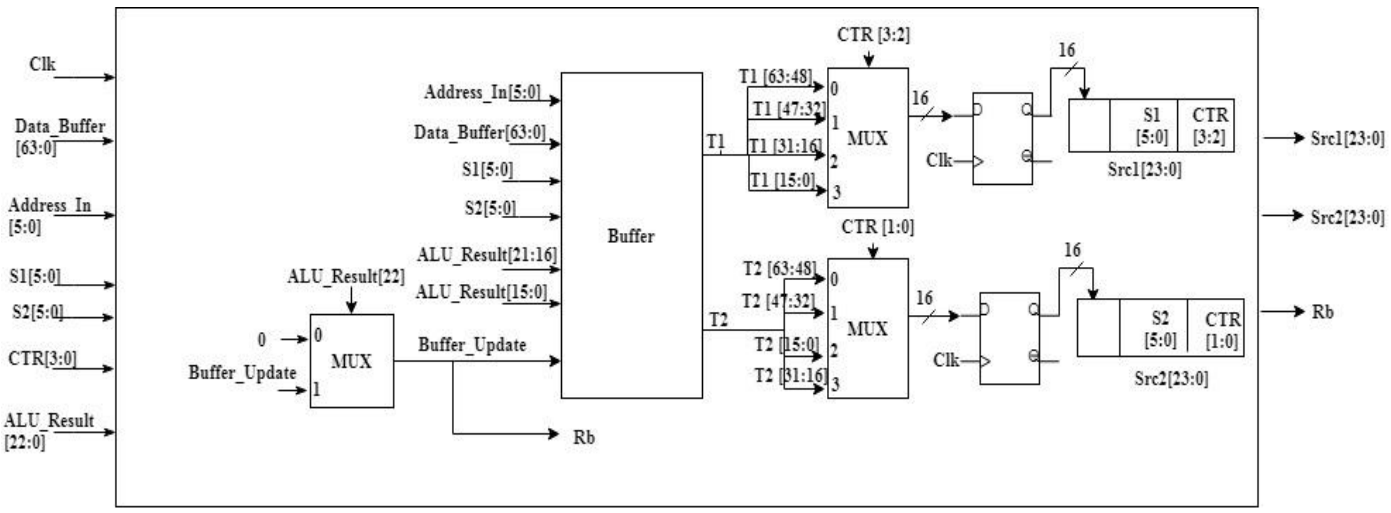

Figure 18: The hardware implementation of a multiway function buffer in the serial system.

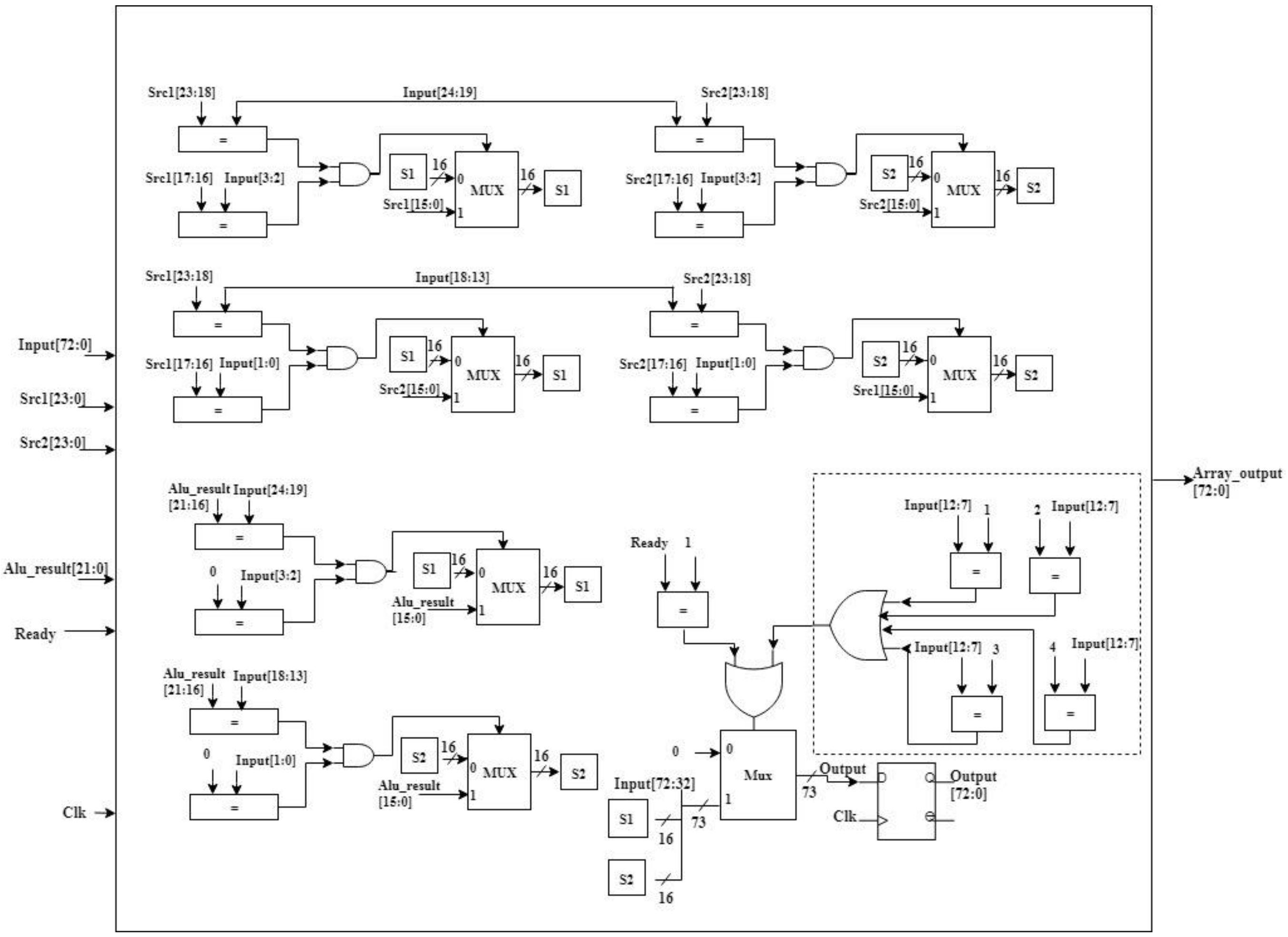

Figure 19: The hardware implementation of the execution array unit in the serial system.

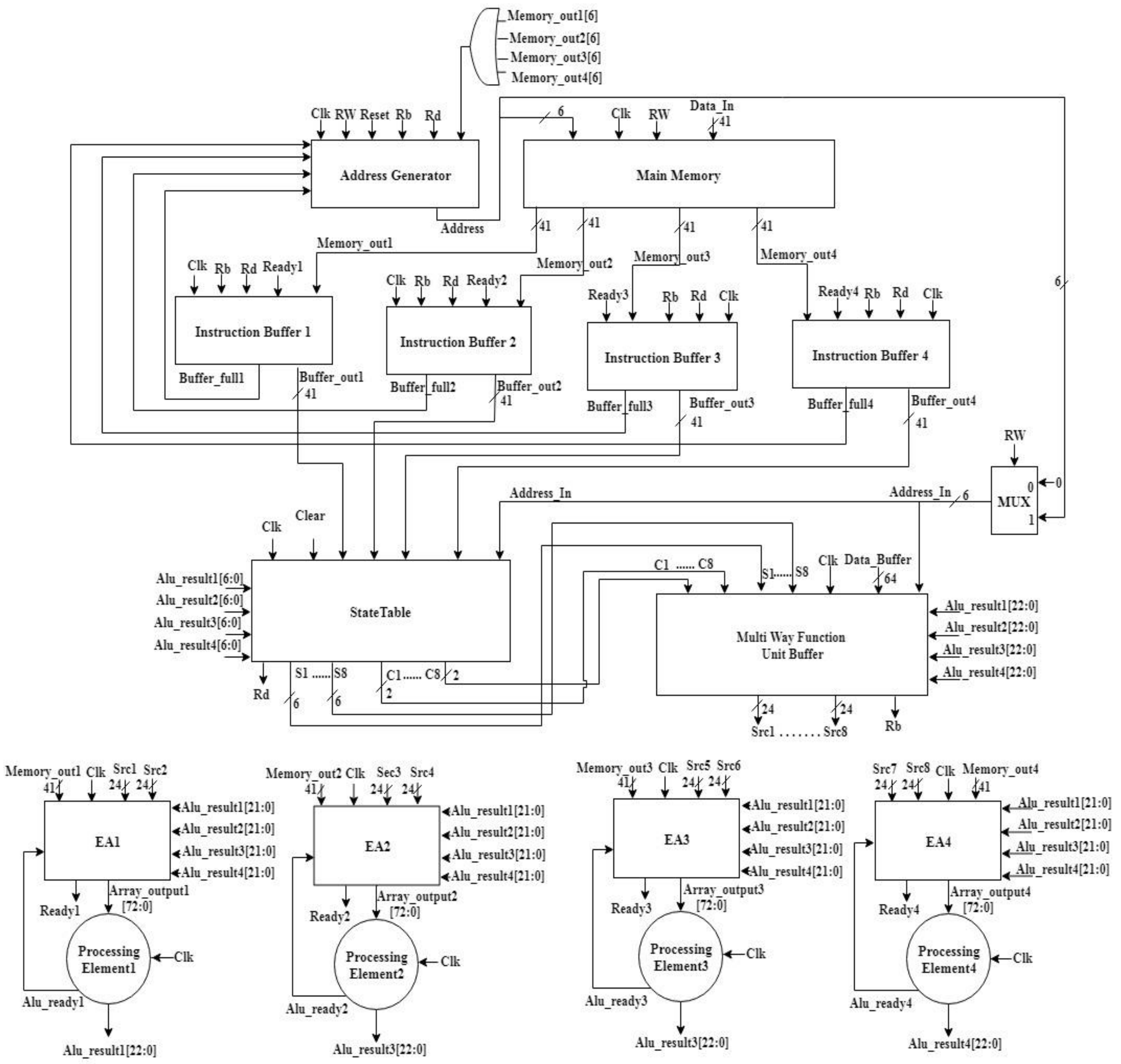

Figure 21: The block diagram of the parallel system.

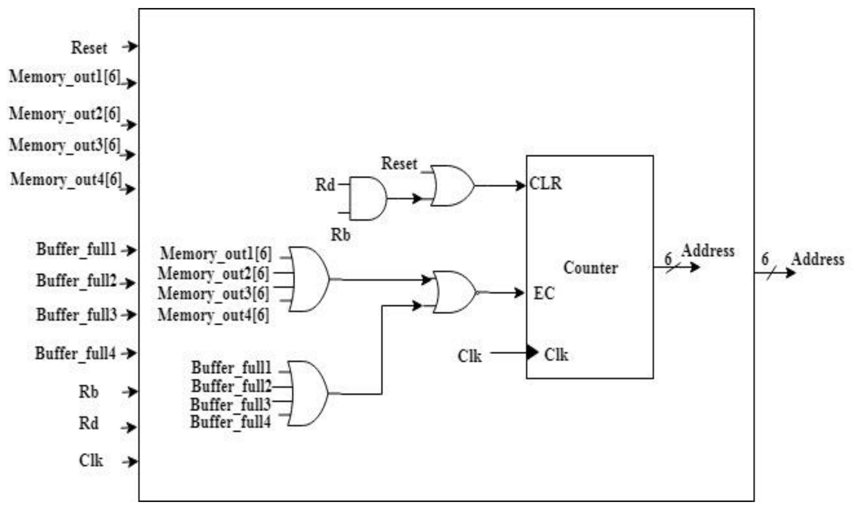

Figure 22: The hardware implementation of the address generator in a parallel system.

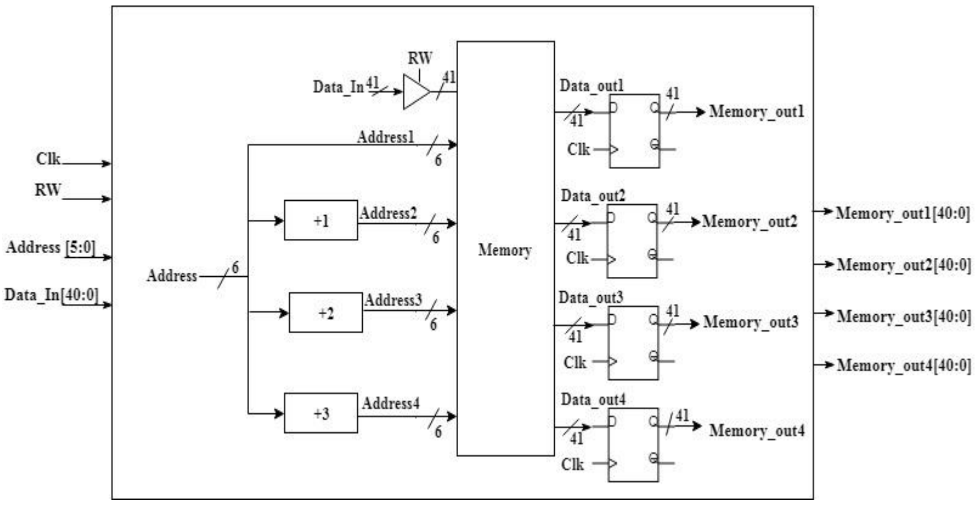

Figure 23: The hardware implementation of the main memory unit in a parallel system.

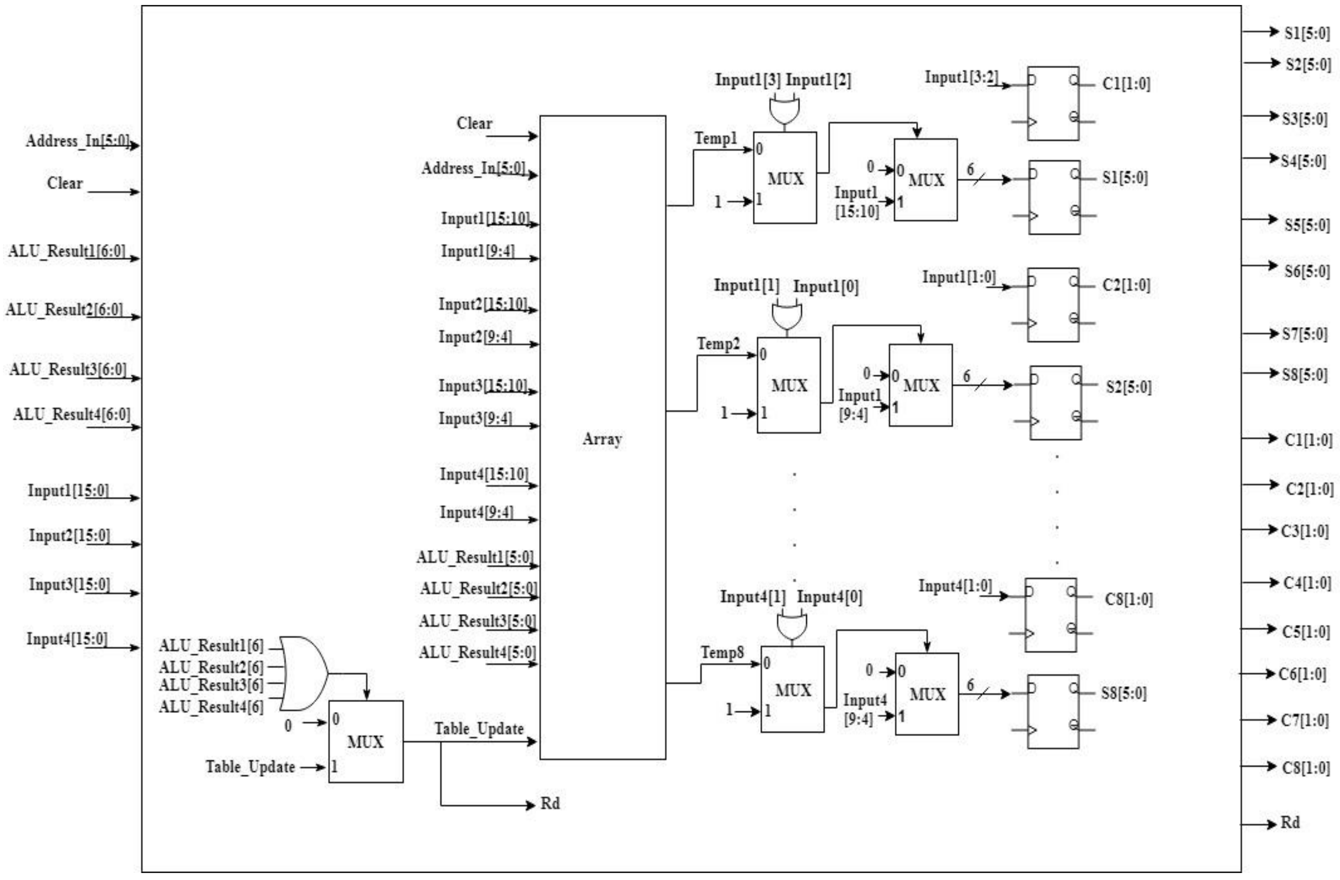

Figure 24: The hardware implementation of the state table unit in a parallel system.

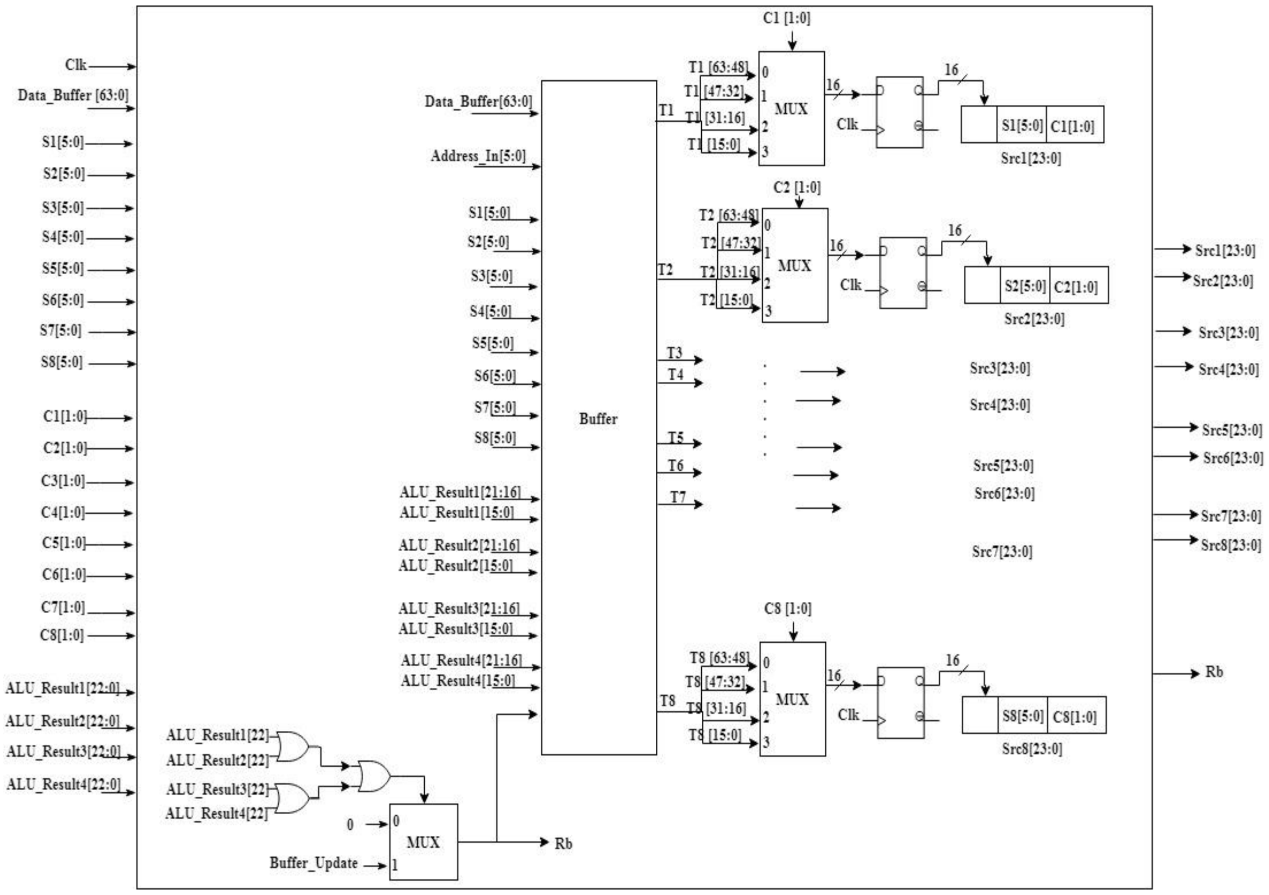

Figure 25: The hardware implementation of the multiway function buffer at a parallel system.

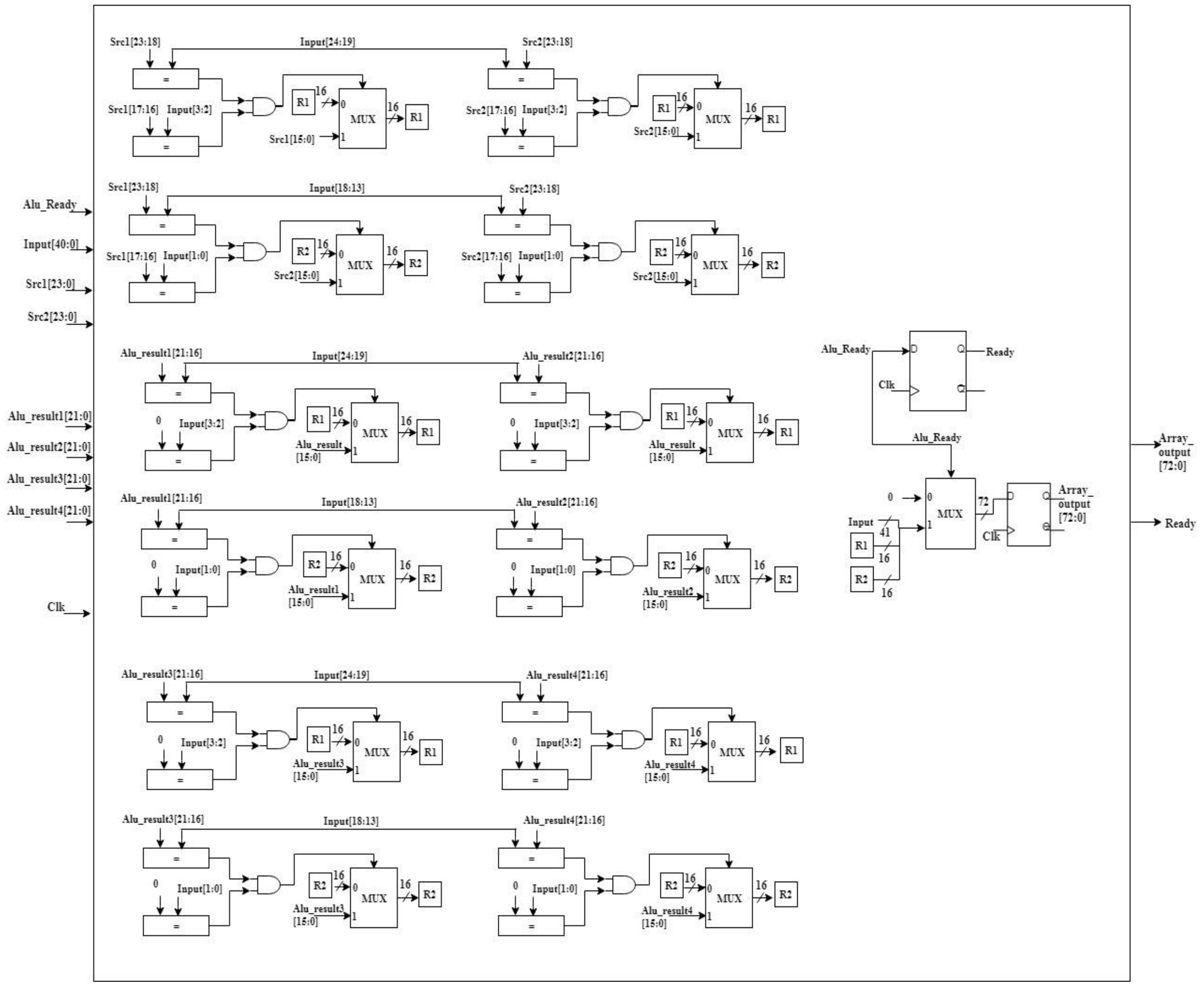

Figure 26: The hardware implementation of the execution array unit in the parallel system.