

# **SNAPP Technologies Corporation's Processing Architecture**

## **A Combinatorial Architecture for Instruction-Level Parallelism**

### **Abstract**

The work presents a new principle for microprocessor design based on a pairwise balanced combinatorial arrangement of processing and memory elements. The proposed apparatus uses two operand instructions so that a set of executable machine instructions is partitioned by these address pairs. This partitioning allows concurrent processing of data independent instructions. Because the partitioning is done at compile-time, this design extracts substantial Instruction-Level Parallelism from executable code without the overhead of run-time methods. The sequential consistency of the concurrent execution of instructions, including indirect addressing and conditional jumps, is ensured by inserted directives and queues regulation. Generation of executable code requires minor adjustments to a standard compiler. The hardware is built of regular modular components. This design provides a straightforward division of labor among the different functional units. The suggested combinatorial architecture offers a family of constructions with various degrees of performance enhancement.

**Key words:** Computer Architecture, Instruction-Level Parallelism, Combinatorial Block Designs, Superscalar Microprocessors

## 1. Introduction

Uniprocessors have been a tremendous success in providing computational power. Designed to execute serial programs, performance of the uniprocessor has been improved by implementing superscalar and pipelining methods. An objective of these enhancements to uniprocessor design has been to extract as much Instruction-Level Parallelism (ILP) as possible from the sequential serial code. These enhancements have come at a cost of increasing processor complexity [6,8]. This paper builds on the ILP framework by offering a design for a processor which can take advantage of this fine-grained parallelism while reducing the overhead necessary in extracting the ILP from the serial program.

Typically, a superscalar processor works on a "window" of instructions in the instruction stream. It discovers the dependences among the instructions in the "window" and schedules execution of independent instructions to occur concurrently. The size of the window determines the upper bound amount of parallelism that can potentially be extracted from the code. Unfortunately, linearly increasing the size of the window quadratically increases the efforts of determining the independence among the instructions. Thus, performance improvements have been tempered by the extra complexity and irregularity of hardware which these constructions require [9].

The effectiveness of ILP can be improved by increasing the degree of concurrency for the considered "window" of instructions and by reducing the hardware needs for their run-time scheduling. A number of methods are available for this purpose: encoding the dependences of the instructions at compile time, implementing the Very Long Instruction Word (VLIW) technique, using predicated instructions, attempting speculative executions of conditional branches, and others. Additional efforts have been directed towards exploiting higher level parallelism through multithreading [4] and algorithmic enhancement [10].

This paper presents a computer organization that offers another approach to extract ILP in the framework of superscalar-style execution [2]. This design resembles a multiple-issue superscalar design in that independent instructions are issued and executed concurrently. The main conceptual difference between the typical superscalar and our design is that in our design the program is split into a number of interdependent parts before run-time and the dependences between the parts become explicitly exposed. The suggested construction is formally the project called REBUS which stands for Regulated Elements By Universal Scheme. The commercialization of the following architecture is SNAPP which stands for Shared Networked Acceleration Parallel Processor. The general structure of this computational scheme is depicted in Fig. 1. The SNAPP design offers compile-time scheduling as in VLIW while allowing execution of individual instructions to proceed more asynchronously as in a superscalar design. The instructions in Partitioned Instruction Streams are executed autonomously on Sliced Memory Hierarchies at their own pace determined by the availability of operands and resources.

In the suggested architecture, the whole memory system all along the memory hierarchy is partitioned into separate logical segments. Information exchange among the segments is performed only at the top of this hierarchy, at the level of Scratchpad registers. The Scratchpad registers are duplicated for several processing elements and combined with memory coordination units responsible for interfacing with the main memory. The program to be executed is partitioned among the processing elements at compile-time with some explicitly encoded control directives. Each processing element executes its portion of the program. In this way, the construction provides the superscalar-like out-of-order execution of independent instructions. The key point of the SNAPP construction is in arranging the information exchange through a pairwise balanced combinatorial design as will be explained below in Section 2.

Current superscalar designs are complex. Improving one aspect of performance often becomes detrimental to another. The suggested principle of information exchange presents a new approach to

constructing microprocessors that can simultaneously increase computational power while reducing structural complexity. In brief, the SNAPP architecture has the following indicative advantages:

1. The architecture provides an evolutionary change for computer design since it allows existing machine assembly code to be used as the instruction code after some straightforward transformation.
2. The size of the "window" of instructions to be scheduled with concurrency can be increased in a scaleable fashion. Thus, ILP can be exploited from the low levels found in normal serial code all the way up to multithreading and parallel instruction streams.
3. The segmentation of memory makes possible performance improvements due to parallelized memory access. Reduced latency in access of the memory system can be aided by more prefetching possibilities in the SNAPP design.
4. Utilization of future technology of ultra large-scale integration for microprocessor design encounters problems of internal timing which necessitates transition from synchronous to asynchronous systems [11]. The functioning of the SNAPP components does not require a centralized clock since components can operate coherently relying on their local clocks. Also, the burden of determining dependences in the instruction code occurs before run-time allowing for potentially shorter cycle times.
5. Verification of a processor constructed according to the proposed architecture would be simplified by the fact that it consists of relatively independent, modular components. The regularity of the design should allow for more efficient manufacturing.

The general description of the system is given in Section 3 which follows the explanation of the suggested principle of information exchange using pairwise balanced combinatorial designs. Sections 4 through 7 are related to diverse issues of a particular possible design of a computer system based on the suggested principle. Section 8 presents a review of our simulation program which was undertaken mostly to verify the sequential consistency of the unusual logic of out-of-order execution of the machine instructions.

## **2. Pairwise balanced combinatorial interconnection**

The functionality of computational work requires moving of an information item from one place to another: from a place where the item is stored to a place where it is needed for processing. Traditional methods of computer communications solve this problem either by applying interconnection links or through accesses to a shared memory. As has been suggested in [1], the desired processing functionality can be also implemented on a principle of information exchange through combinatorial replication. This allows data independent instructions allocated to different processing units to be executed separately. This organization to a certain extent resembles the approach of the Tomasulo algorithm which uses reservation stations in the processor to buffer operands and results freeing each instruction from the constraint of waiting for other instructions [8].

The considered principle of information exchange in computer systems is based on the arrangement of Processing Elements in the so-called combinatorial "block designs". The block design is a popular combinatorial structure which presents "selections of the subsets of a given set such that some prescribed conditions are satisfied"[7]. Initially, practical applications of combinatorial designs have been mostly related to planning of statistical experiments. Nowadays, this methodology acquires growing

importance in various areas of computer science (see, e.g. [5]). There are different ways of employing combinatorial designs to arrange computer interconnection facilities. The distinction of the suggested construction is that it provides direct interaction of arbitrary pairs of objects by replicating them in various blocks rather than by moving information items between the blocks. Realization of this construction is based on a particular type of combinatorial design, the balanced incomplete block (BIB) designs, which are characterized by a certain distribution of elements with a pairwise balanced property. A BIB design on a set of  $v$  elements is a collection of  $b$   $k$ -subsets such that each element appears exactly in  $r$  subsets and each pair of elements appears exactly in  $I$  subsets. Thus, this BIB design is also called a  $(b, v, r, k, I)$  - configuration. For the sake of illustration, consider an example of a collection of 12 3-element subsets of a 9-element set  $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9\}$  which constitute a  $(12, 9, 4, 3, 1)$  - configuration:  $B_1 = \{x_1, x_2, x_3\}$ ,  $B_2 = \{x_4, x_5, x_6\}$ ,  $B_3 = \{x_7, x_8, x_9\}$ ,  $B_4 = \{x_1, x_4, x_7\}$ ,  $B_5 = \{x_2, x_5, x_8\}$ ,  $B_6 = \{x_3, x_6, x_9\}$ ,  $B_7 = \{x_1, x_5, x_9\}$ ,  $B_8 = \{x_2, x_6, x_7\}$ ,  $B_9 = \{x_3, x_4, x_8\}$ ,  $B_{10} = \{x_1, x_6, x_8\}$ ,  $B_{11} = \{x_2, x_4, x_9\}$ ,  $B_{12} = \{x_3, x_5, x_7\}$ . The property of pairwise balanced property means that taking any pair of elements of  $X$ , say  $x_2$  and  $x_8$ , it is always possible to indicate a block containing this pair of elements, in this case it is  $B_5$ . There are well-established mathematical methods for creating BIBs with various parameters.

A scheme of interconnecting Processing Elements (PE) exploiting the pairwise balanced property is given in Fig. 2a which presents another example of one of the simplest BIB designs, a  $(7, 7, 3, 3, 1)$  - configuration. In this utilization of the BIB design, the PEs correspond to the subsets while the connection busses correspond to the elements. These elements are repeatedly represented in the PEs by the attached Registers which at the other side are linked to an MCU (Memory Coordination Unit). The MCUs are specific to the SNAPP microprocessor and are controlling the data movements.

The presented construction can be applied to a system of objects rendering an information exchange through their pairwise interactions. Thus, suppose we have 7 objects and we replicate them in 3 copies and place over the 7 PEs according to the bus numbers as indicated in Fig. 2a. Now, assume that there should be an interaction between two objects, say #2 and #7. The structure of the system ensures that for any given pair of objects there is always a PE which contains copies of these objects, in this case it is PE #3. Therefore, the interaction of the objects can occur immediately without moving any data across the system. As a result of interaction, objects change and all their copies can be updated simultaneously and independently through the busses which connect the PE where the corresponding objects have been replicated (in our example those PE are (#2, #5) and (#1, #7)). An important feature of this construction is that interactions of pairs of objects with different components do not interfere and can progress concurrently. This construction creates a distributed environment, which effectively supports the object-oriented paradigm; its performance characteristics have been investigated in [3].

The pairwise balanced combinatorial interactions can be arranged for different systems of communicating objects. In microprocessor design, it is convenient to introduce this technique considering as communicating objects the Scratchpad registers. The suggested mechanism of information exchange can be effectively exploited in a computer system which uses two-operand machine instructions.

A program can be partitioned amongst the PEs by having an instruction's operand pair determine the PE to which the instruction should be designated. An example illustrating the idea of this partitioning for simple arithmetic machine instructions is given in Fig 2b. Thus, the basic computational step--a transformation involving two operands--can be performed without delay at an appropriate processor. The result of an operation is put on the bus connecting all relevant processors (those with the register being written to) and becomes immediately available. Instructions using disjoint pairs of registers can run concurrently. Realization of this concurrence requires implementing a synchronization protocol. The synchronization information is pre-distributed to the processors in the form of control directives which are incorporated in the executable code.

Presenting operands of machine instructions by the Scratchpad registers allows a serial program

in this computer to be partitioned and distributed over the PEs containing these particular Scratchpad registers. The primary advantage of such an organization is that it clearly exposes an inherent concurrency in a sequence of machine instructions. The issues of how this structure provides sequential consistency and sustains splitting of instructions with indirect addressing and branching conditions are considered below.

### **3. General description of the system**

The key novel characteristics of the SNAPP architecture are: (1) a combinatorial arrangement for processing elements with unusual organization of their interactions, (2) a simple realization of sequential consistency of the execution of the instructions, and (3) an effective implementation of coordinated data movements between the different speed memory units all along the memory system hierarchy.

The SNAPP processor obtains speedups from the same source as do superscalar processors, that is from the concurrent execution of different instructions. Superscalar processors require an elaborated hardware to keep track of the various assignments for out-of order execution of the instructions. One advantage of the SNAPP processor is that each instruction is mapped uniquely to a processing element based on the instruction's operands, and therefore, no complicated run time assignment need take place.

The processing elements of the SNAPP computer operate with an instruction set of a two-operand machine. The SNAPP computer can accept ordinary application software in the form of source code written in a high level language. Generation of the object code can be performed with standard compiling procedures. Adjustments to the normal method of transforming the object code into executable modules are straightforward and require splitting the executable code into separate instruction blocks (each block to be assigned to a particulate PE) according to the BIB design.

From the point of view of each PE and its particular instruction stream, we have an ordinary sequential computer having a processor, cache, main memory, and secondary storage. A portion of the program for each PE is extracted from the main program according to references to the Scratchpad registers contained in this PE. The proper functioning of the portions of the program as a coherent whole is maintained by inserting supplementary Control Directives. Each PE refers to the memory system through MCUs. The structures of the MCUs and the PEs are given in Fig. 3 and Fig. 4. These operational blocks are interwoven in a combinatorial designed interconnection as illustrated by an example in Fig. 2a. A variety of combinatorial designs for the interconnection structure will work and provide flexibility to the suggested construction.

### **4. Organization of memory**

From a logical point of view, the memory of the system represents a unified entity. Physically, however, this system can be divided into a pre-established number of segments all along its hierarchical structure. Thus, in the scheme depicted in Fig. 2 there are 7 segments.

A slice of the memory system in a top-down cross-section includes the following storage units: (1) scratchpad registers, (2) cache memory, (3) a segment of main memory, and (4) access to secondary storage. Except for the top level of the Scratchpad registers, the storage units corresponding to different

segments are not connected, i.e., it is not possible to move data between these segments directly. The movements of data in the memory occur only through the top of the hierarchy--the Scratchpad registers which are interconnected.

The management of these data movements is performed by Memory Coordination Units (MCU) (Fig. 3). MCUs serve as mediators between processors and memory. An MCU combines scratchpad memory of some processors with the cache memory of a given memory slice. Each processor can also make use of its own private memory to handle intermediate data. Parts of the private memory can be maintained coherent in different Processing Elements in the same manner as the corresponding Scratchpad registers.

Besides local manipulations with data using private memory inside the PEs, all the data movements in the system are either "horizontal", among the Scratchpad registers, or "vertical", along the memory hierarchy. As a result, to ensure the sequential consistency of a program, it is sufficient to manage the dependences of the addresses of the interconnected Scratchpad registers and Cache in the MCU.

**5. Instruction set**

Normally, any set of machine instructions is algorithmically complete. Thus, the choice of a particular set of machine instructions is based on performance considerations. Most of the microprocessor systems nowadays are based on the RISC concept. The advantage of the RISC instructions is that they are composed of elementary operations of standard duration which enables their synchronous execution in a pipelined fashion. The SNAPP processor allows for concurrent execution of instructions in asynchronous fashion. Therefore, the system can benefit from the greater power of more diverse instructions, like CISC, without sacrificing the gain from concurrent execution.

With some minor modifications, the set of the instructions required for the proposed scheme corresponds to a regular set of the two-address instructions. These instructions have the following fields:

OpCode	Address 1	Address 2	Queues Control
--------	-----------	-----------	----------------

The additional component, the Queue Control field, accommodates the Write-Directives (**WD**) which are specific for the suggested design and will be introduced below. We will describe these instructions in most general form as they can be supported by the suggested structure. In concrete situations, the preference for a suitable subset of the instructions may be determined by the given configuration of the system and the application.

It should also be pointed out that the SNAPP design is not necessarily limited to two-address instructions. With some modifications, one can use the more standard three-address instructions which are of the following form: **OpCode Destination, Address1 and Address2**. The choice of the two-address instruction set was for developmental and explanatory purposes.

**Data instructions**

These instructions deal with data transformations and movements; they access memory and update its contents. The memory on which these instructions operate is of three types: Scratchpad registers (R), which are shared by all processing elements, Sectioned main memory (M) which is worked upon by a subset of processing elements, and private memory (P) which can be accessed only by a given processing element.

### *Operational instructions*

Instructions of this type deal with operands in the Scratchpad registers and/or private memory. Such an instruction takes operands from the two given locations, executes the indicated operation, and places the result in the second location. For example, an operation for addition will be: **ADD R1 R2**.

### *Data movement instructions*

These instructions are mostly required to supply the Scratchpad registers with processing information and to distribute the results. These instructions include direct and indirect addressing. Data movement instructions with direct addressing take a data item from the location specified by the first operand and place it in the location specified by the second operand, in essence, in the same way as in an ordinary processor. Indirect references have to take into account the particulars of the SNAPP architecture involving routing through the MCUs.

By virtue of the structure of the system, it is preferable to have indirect read instructions in a format which uses one register in the first and second position. For example, this instruction can be written as: **LOAD\_IND 'R1' R1** meaning that the contents of the storage specified in first position, register R1, are used as an address and that register R1 is provided with the contents of this address. This distinction does not impose any limitations on the algorithmic capabilities of the system. However, with this restraint the organization of the SNAPP processor is simplified. In formation of the executable code, these indirect read instructions are replicated in the PEs corresponding to the number of R1 allowing the overall the SNAPP processing structure to be sustained.

The instructions for indirect write, **STORE\_IND 'R1' R2**, take the contents of the first register R1 and using it as an address puts in this address the contents of the second register R2. In the SNAPP system, the indirect write functionality can be implemented in different variations. Importantly, realization of these instructions can benefit from the fact that the contents of the Scratchpad registers become known to the corresponding MCUs at run-time in advance.

## **Control instructions**

### *Unconditional and conditional branches*

These instructions change the contents of the program counter (PC) so that the order of execution of the program instructions is changed. The unconditional branch instruction forces the processor to execute the instruction specified by the operand PC: **JUMP PC**. The conditional branching instruction switches the execution of instructions to PC depending on the sign of a variable which is set processor-wide. Otherwise, by default, the standard order of instruction execution is preserved. The considered conditional branching instructions are in the format: **BNEG PC**, **BPOS PC**.

### *Interrupts and error trapping*

These instructions halt execution of the program temporarily or permanently. Also, these instructions would be used to report irregular situations within Processing Elements, like division by zero, overflow etc. In the case of the interrupt, the context switching can be done in each PE independently..

## 6. Interconnection structure and Processing Element functioning

In the suggested system, we employ a combinatorial arrangement in which the PEs (Fig. 4) are combined in subsets by the respective MCUs (Fig. 3). The created configuration minimizes latency and avoids conflicts in communications between the MCUs and PEs.

The major part of the Processing Element (Fig. 4) is a standard microprocessor with an instruction memory and private data memory. The PEs are connected to the central control of the system which is responsible for global management. The management signals from the central control are involved in delivery of program instructions, initiation of computations, and cessation of the computing process either for a final stop or for temporary interrupt. The feedback signals from a PE are issued when it determines an anomalous condition, like in arithmetic operations (overflow, division by zero) or in case of some hardware checks. Through the same pathway the PEs report the situation where they reach the STOP command.

The computations inside the processing node are based on the node's instruction stream which may be stored in its own instruction cache. As a working memory, this PE directly employs only the three Scratchpad registers which correspond to the three partitions to which this processing node is connected. The number of connections may vary depending on the utilized combinatorial design. The interactions with the whole memory system are done by using the MCUs as mediators. The operations of the processor involve Scratchpad registers, R1, R2, and R3. These registers form the MCU interface. This interface connects with Scratchpad registers in the corresponding MCUs. The MCUs are responsible for the supply of data from their slices of main memory and secondary storage. Therefore, the operations which involve accesses to main memory are organized through the corresponding Scratchpad registers. Since memory requests come from the Scratchpad registers, an MCU can monitor information passing through its scratchpad connection and prefetch the anticipated data. So, the cache memories in the suggested construction can be more efficient because they obtain certain information about future requests in advance.

The coherence of the contribution of a PE into the whole computational process is ensured by the operations of the Queues of Scratchpad Copies. These Queues enqueue all the information as it arrives in the Scratchpad registers. So, the Queues of Scratchpad Copies contain the various versions of the register contents as updated by the executed instructions which have that register as a destination. The information from the queues is removed according to special tokens - **WD** (Write Directives) - in the PE's program which are accommodated in the Queue Control field of the instructions during the formation of the executable code. The processor fetches an instruction and executes it with the following provisions. If the instruction contains only private memory operands, it is executed right away. If the instruction contains scratchpad memory operands, it takes them from the corresponding Queues of Scratchpad Copies. Thus, the necessary condition for this type of instruction to be executed is to have the corresponding Queue of Scratchpad Copies not empty. Otherwise, the PE has to stall. With this rule, the contents of the Queues of Scratchpad Copies are updated in such a way that an instruction within each PE can get only the proper operands. Writing a result back to the register may be done if the connecting bus is idle and the corresponding Queues of Scratchpad Copies in the adjacent PEs are not overflowed. The correctness of the presented protocol has been validated by the developed computer simulation model.

## 7. Formation of executable code

The computational process is performed as a collective activity of the processing elements. The source code is compiled into an object code with the above described set of machine instructions. The object code is then transformed into executable code modules for the processing elements. The execution modules distributed to the Processing Elements are to be linked independently.

The execution module for a given PE is derived from the main program according to simple rules. First of all, the instructions for a particular PE are selected from the list of machine instructions of the main program in such a way that their initial ordering is preserved. Instructions in which the operands pairs match the Scratchpad registers of a PE are uniquely assigned to this PE, as has been shown in the example in Fig. 2b. Instructions using only one Scratchpad register have to be placed in all PEs containing this register; thus, for the construction in Fig. 2a an instruction **LOAD\_IND 'R1' R1** have to be placed in all PEs containing register#1, i.e. PE#1, PE#2, and PE#4. Control instructions which do not have operands, like jumps and stop, have to go to every PE. Instructions, like in MACRO operations or subroutines, which use private memory rather than Scratchpad registers can be placed arbitrarily as long as inputs and outputs of corresponding program segments are made to match with the Scratchpad registers of a host PE..

Besides the distribution of the instructions, formation of an executable module for a PE requires additional control directives to coordinate the behavior of different PEs. These required control actions are aimed to remove the values from the Queues of Scratchpad Copies if they are inappropriate for a current instruction. The commands for this removal come from the above mentioned **WD** (Write Directive) instructions which are inserted in the executable module of a given PE in lieu of the instructions of the main program going to other PEs which contain a write request to the corresponding Scratchpad register. As a result, inside any PE the operands values will be consumed in a correct order irrespective of the progression of their creation in other PEs; if the needed value has not yet been created somewhere the corresponding Queue of Scratchpad Copies in the PE will be empty and the advancement of computations in this PE will be suspended. The **WD** tokens do not have to appear as special independent commands and may be incorporated in regular instruction cycle through the Queue Control field of the SNAPP instruction format as presented in section 5.

A conditional branch is based on the analysis of a certain Flag register which contains, for example, the sign of a number. The Flag register is treated as a register common to all PEs being accessed through a queue in the same mode as the Scratchpad registers with similar control directives. If the Flag queue is empty, the PE waits; if it is not empty, the PE branches based on the available control value.

As all the PEs in the SNAPP construction operate on their own executable modules following the described queue control protocol the sequential consistency of the whole program will be preserved. The accuracy of this assertion and the efficiency of the obtained computational process have been investigated by a software simulation model.

## 8. Software simulation model

To validate the SNAPP model, we developed a software simulation. The simulation was created primarily as a proof-of-concept to verify whether the suggested collective actions of a set of distinct processing units indeed produce the same output as a conventional serial uniprocessor. Also, this model also allows the evaluation of the gain in processing time which comes from the combinatorial overlapping of operations in different parts of the system.

The simulation program was developed in Visual Basic under Microsoft Windows (Fig. 5). The

program presents the simplest SNAPP structure based on the (7, 7, 3, 3, 1) - configuration. For comparison, the simulation package also includes a simulated UniProcessor. Both devices, the SNAPP and the UniProcessor, have a 512 location memory and seven Scratchpad registers. Each PE in SNAPP and the UniProcessor contain a Flag bit for branching. We use an instruction set similar to the one described in this paper. The simulation allows creating and running programs written in a limited assembly code.

The software model provides editors for initializing the data memory and writing assembly code programs. The same assembly program is written for both devices. To run the program, one must first load it into the instruction memories of the two simulated devices. The UniProcessor has the assembly program verbatim while for the SNAPP processor, the program is split among its PEs. The simulation program creates executable modules for each PE from machine instructions written directly for the UniProcessor.

The UniProcessor and each PE in the SNAPP processor have a program counter which is set to the start of the instructions to be executed. The course of the execution in the SNAPP system and in the UniProcessor are displayed simultaneously. At every step, the UniProcessor executes one instruction while each of the PEs in the SNAPP system is attempting to execute the current instruction of its instruction stream. Realization of the execution in a PE is subject to the availability of operands in their Queues of Scratchpad Values. If an operand is not yet available, the processing element enters a busy-wait state until it can be obtained. The SNAPP processor is considered to have finished the program once all the PEs have reached the STOP command. The correctness of the SNAPP design is established by the fact that the collective work of its processing units generates exactly the same state of the memory as the work of an ordinary sequential uniprocessor.

At this stage of simulation development, we can implement rather small programs as they are written directly in machine codes. A user-configurable time delay is associated with each operation (fetching, executing, memory access) to relate their contributions to the attainable speedup. According to our simulations, we find that on certain standard computing procedures, the SNAPP processor simulation tends to run up to 3 times faster than the sequential processor. It has to be noted that this speedup arises from the inherent Instruction-Level Parallelism in the suggested combinatorial partitioning of an arbitrary sequential program. We have not addressed the problems of further extraction of concurrency from separate processing streams by lifting the restrictions determined by global control and data dependences in instructions like conditional branching and indirect storing in the main memory. Resolution of these problems is associated with rearrangement of memory assignments at compile-time.

## **9. Conclusion**

The suggested technique and structural arrangements based on the combinatorial principle of information exchange represent a new approach to the organization of sequential computing. The major operational principles of this system have been introduced in the patent [2]. The gain in performance is expected from overlapping execution of the instructions in different processing units. This design addresses the bottleneck problems in the organization of computational processes, including fetching instructions, communication contention, branching hazards, and discordance in speed at different levels of the memory hierarchy. The partitioned memory hierarchy can be exploited for an efficient realization of I/O operations. It is important that this design removes all the bottlenecks simultaneously since leaving even one bottleneck would inhibit the advancement of a system's performance.

The SNAPP design offers flexibility in adjusting its structure and parameters. Besides the considered 7 processor scheme further developments can employ 12, 13, 15, 21, or 26 PE schemes and many other variations. This organization makes for a family of computer systems which may cover the

whole range of applications from personal computers and workstations to mini- and supercomputers. It can be also beneficial for real-time control systems presenting facilities for multitasking.

The SNAPP structure contains a certain degree of redundancy due to replication of processing elements, scratchpad registers, and communication busses. But, the modularity and uniformity of this structure allows for simpler testing of the microprocessor chips making the microelectronics production more economical with increasing yield and reliability. Hardware is built of well-developed types of elementary components driven by local clocks without complicated functional interactions. This can substantially decrease the cost of labor of microprocessor design.

The primary objective of this research was to show that the suggested combinatorial architecture can use the same machine instructions as an ordinary processor. It has been demonstrated that the system can be operational using practically the same software as an ordinary sequential processor. Translation of the source code into SNAPP executable modules would require minor adjustments to the compiler. Meanwhile, the SNAPP architecture is intrinsically capable of extracting parallelism from a much larger "window" of machine instructions than common superscalar processors.

One of the most promising directions in high performance computing is associated with multithreaded processing. This technology, however, has not yet materialized in successful commercial products because of "the extra cost and complexity of multithreaded hardware, coupled with the dearth of tools for extracting thread level parallelism from general applications" [4]. In the SNAPP processor, by virtue of the combinatorial structure of the information exchange the scope of the "window" of instructions is not limited to compact segments of program as in ordinary superscalar microprocessors and it can handle a "window" of instructions from a very large span. Therefore, for the SNAPP processor different program threads could be written to run disjointedly from other program threads. These program threads will automatically maintain the ability to interact through common registers without expensive context-switching requirements.

## References

- [1] Berkovich S Y, Multiprocessor Interconnection Network Using Pairwise Balanced Combinatorial Designs, *Information Processing Letters* 50 (1994) 217-222.
- [2] Berkovich S and Berkovich E, Methods and apparatus for concurrent execution of serial computing instructions using combinatorial architecture for program partitioning, US Patent No. 5,619,680, issued April 8, 1997.
- [3] Berkovich S Y and Lin-ching Chang, A Multiprocessor Network Arranging Replicated Objects in Pairwise Balanced Combinatorial Designs, *Journal of Circuits, Systems and Computers* 6 (1996) 85-91.
- [4] Byrd G T and Holliday M A, Multithread Processor Architectures, *IEEE Spectrum* (August 1995) 38-46.
- [5] Colbourn C J and van Oorschot P C, Applications of Combinatorial Designs in Computer Science, *ACM Computing Surveys*, 21 (1989) 223-250.
- [6] Johnson M, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.
- [7] Liu C L, *Introduction to Combinatorial Mathematics*, McGraw-Hill Book Company, New York, 1968.
- [8] Patterson D A and Hennessy J L, *Computer Architecture, A Quantitative Approach*, Morgan Kaufmann

Publishers, San Francisco, California, 1996.

[9] Tremblay M, Grohoski G, Burgess B, Killian E, Colwell R, and Rubinfeld P I, Challenges and Trends in Processor Design, Computer 31 (January 1998) 39-48

[10] Vishkin U, Can Parallel Algorithms Enhance Serial Implementation?, Communications ACM 39 (1996) 88-91.

[11] Werner T and Akella V, Asynchronous Processor Survey, Computer 30 (November 1997) 67-76

Authored By:

E. Berkovich  
Department of EE  
University of Maryland  
College Park, MD 20742  
USA

and

S. Berkovich  
Department of EE&CS  
The George Washington University  
Washington, DC 20052  
USA

#### BIOGRAPHICAL DATA

**Efraim Berkovich** received a BS in mathematics (*magna cum laude*) from Georgetown University in 1994. Since then, he has worked as a software engineer on information systems design and development. He is presently completing a graduate program in the Department of Electrical Engineering at the University of Maryland, College Park.

**Simon Berkovich** received an MS degree in Applied Physics from Moscow Physico-Technical Institute in 1960 and a Ph.D. degree in Computer Science from the Institute of Precision Mechanics and Computer Technology of the USSR Academy of Sciences in 1964. He participated in a number of R&D projects on the design of advanced hardware and software systems. Presently, he is a Professor of Engineering and Applied Science in the Department of Electrical Engineering and Computer Science at the George Washington University.

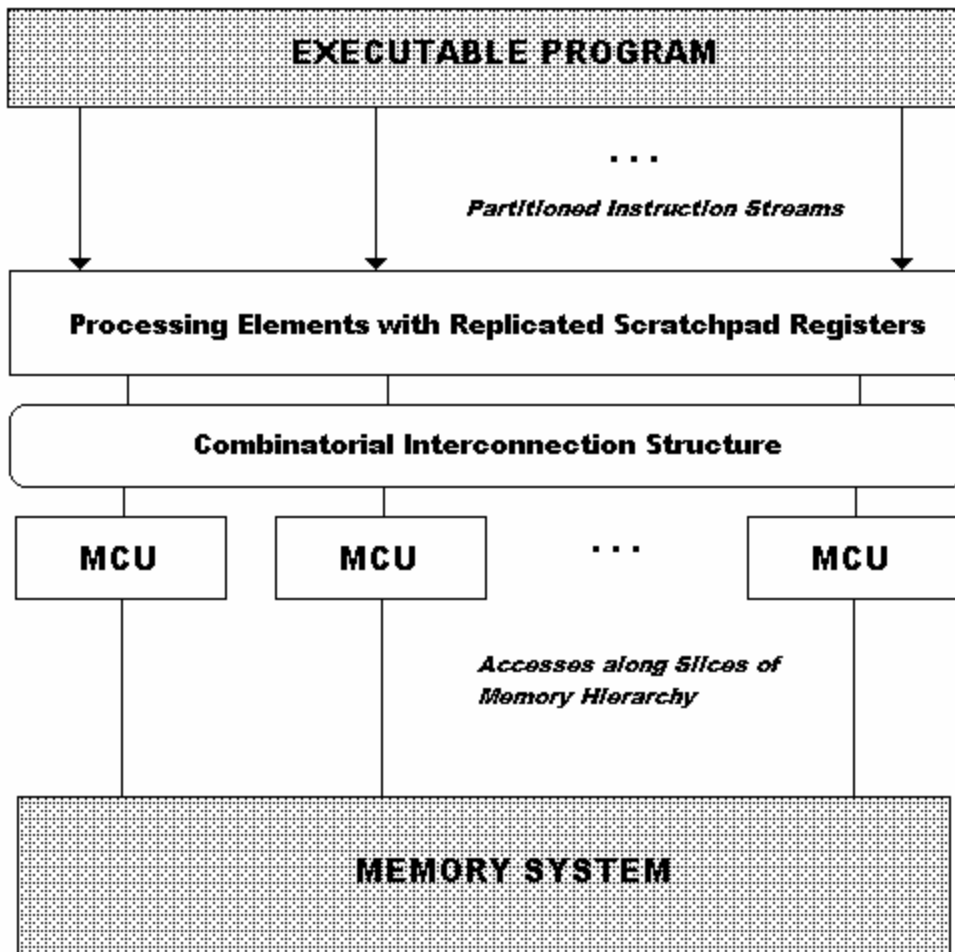


Fig. 1  
The architectural principle of a SNAPP Computer

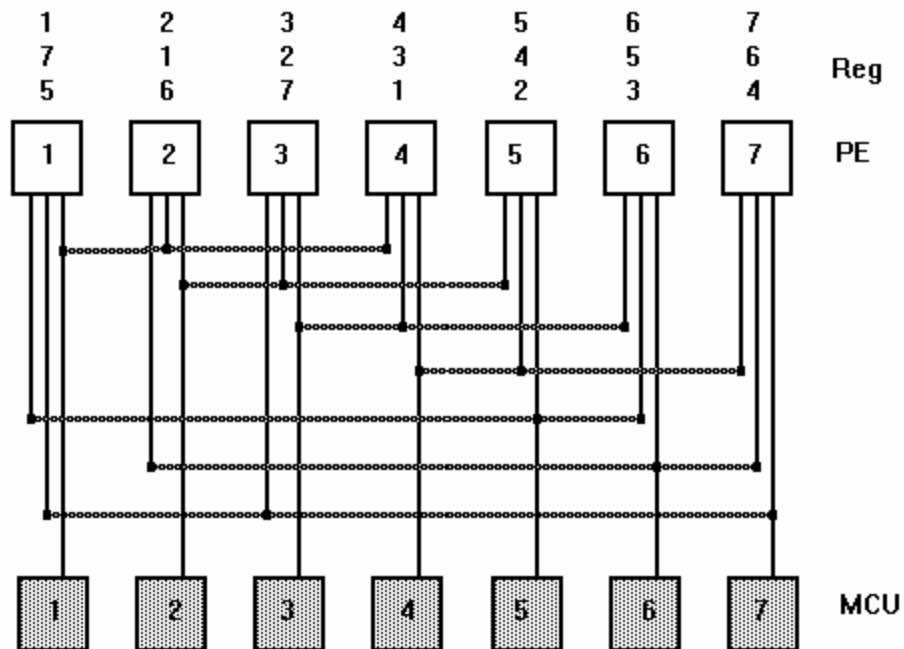


Fig. 2A  
Combinatorial Interconnection of Processing Elements and Memory Coordination Units in a (7, 7, 3, 3, 1) arrangement

Instructions are split according to the pairs of register operands:

Example:

<b>ADD</b>	<b>R1</b>	<b>R7</b>	<b>====&gt;</b>	<b>PE #1</b>
<b>MULT R2</b>	<b>R6</b>	<b>====&gt;</b>		<b>PE #2</b>
<b>DIV</b>	<b>R4</b>	<b>R5</b>	<b>====&gt;</b>	<b>PE #5</b>

Fig. 2B  
Partitioning instructions

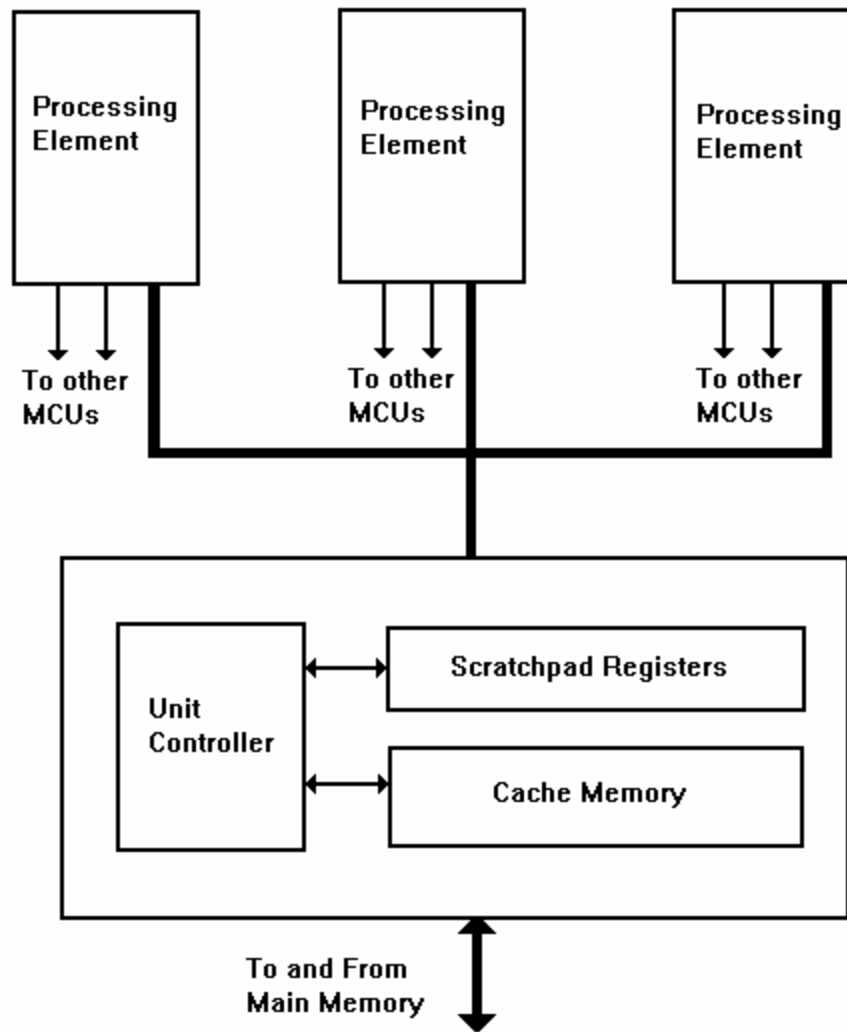


Fig. 3  
The Structure of the Memory Coordination Unit  
And Its Connections

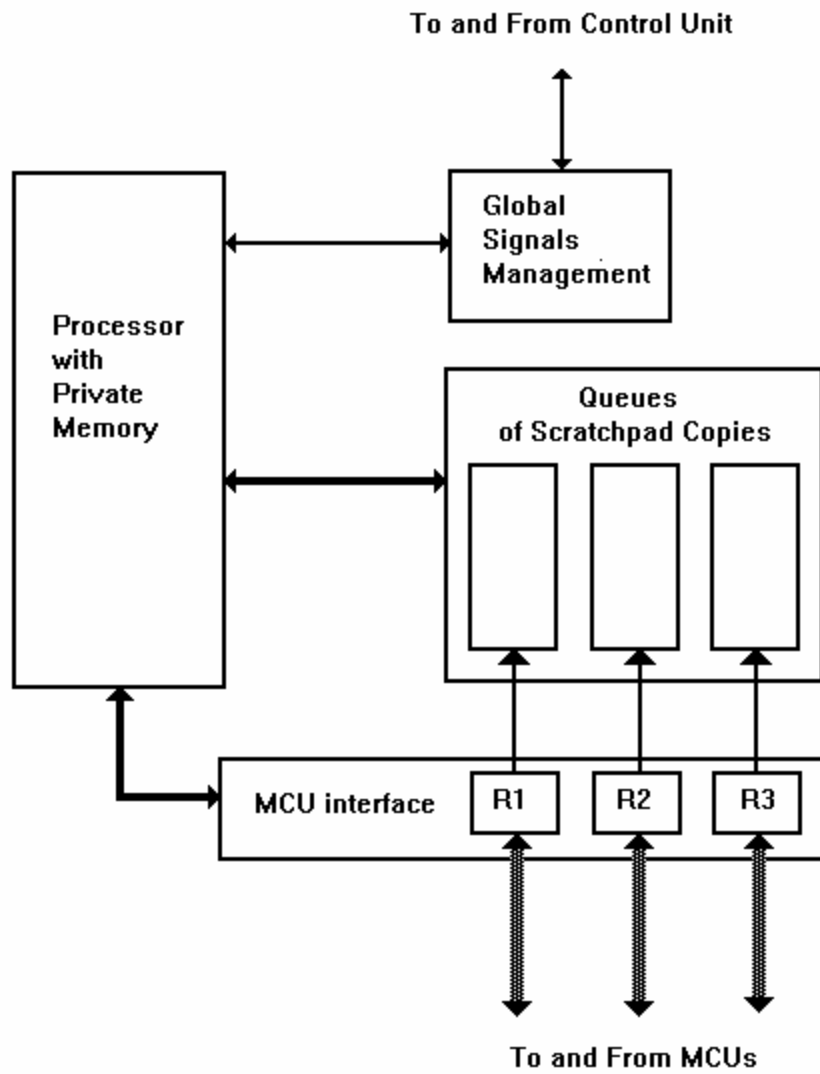


Fig. 4  
The Structure of a Processing Element

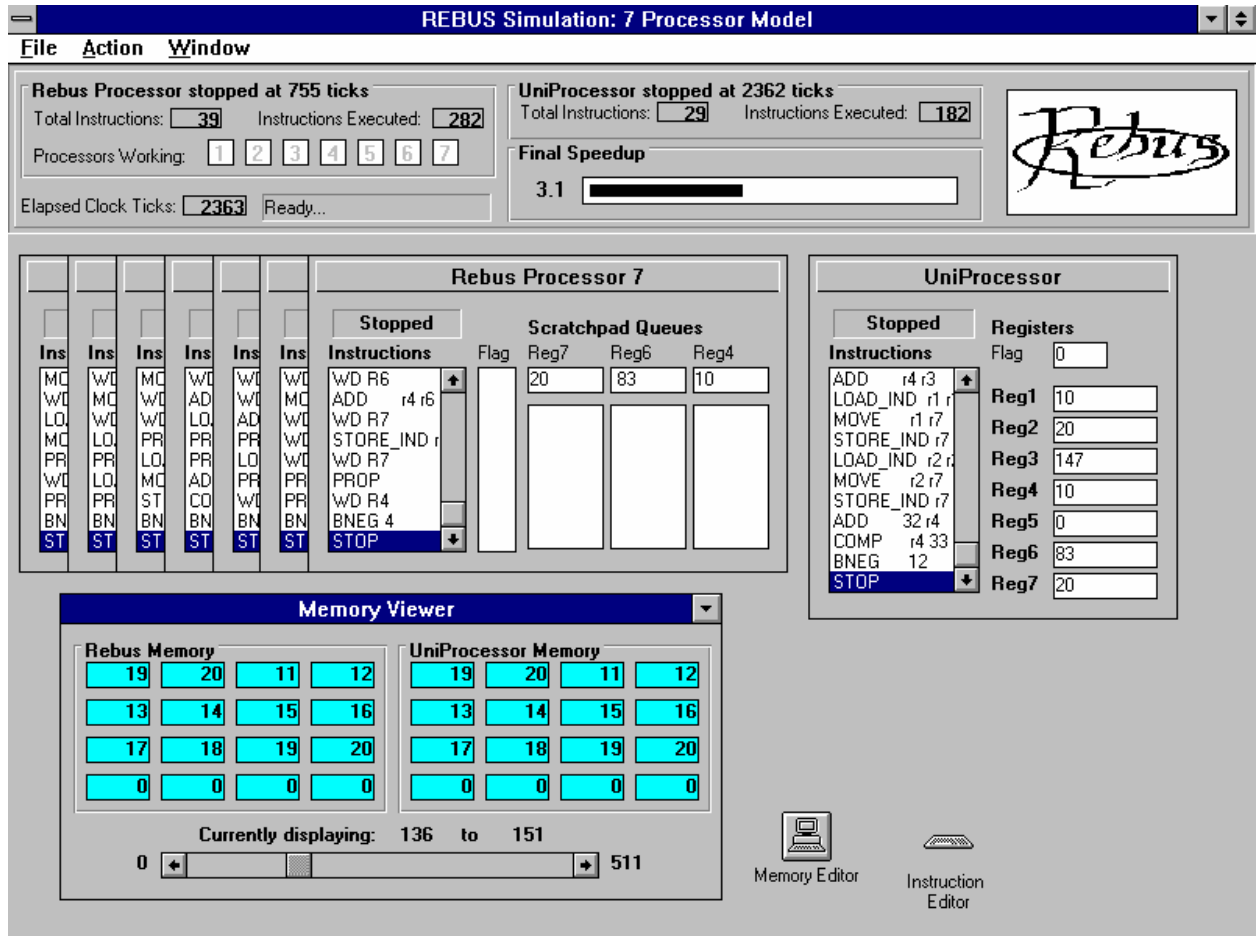


Fig. 5  
The SNAPP Processor Simulation