PROCESSOR DESIGN BASED ON DATAFLOW CONCURRENCY

Sotirios G. Ziavras

Department of Electrical and Computer Engineering

New Jersey Institute of Technology

University Heights

Newark, New Jersey 07102, USA

Email: ziavras@njit.edu

Telephone: (973) 596-5651

Fax: (973) 596-5680

Abstract

This paper presents new architectural concepts for uniprocessor system designs. They result in a uniprocessor design that conforms to the data-driven (i.e., dataflow) computation paradigm. It is shown that usage of this, namely *D^2-CPU* (Data-Driven) processor, follows the natural flow of programs, minimizes redundant (micro)operations, lowers the hardware cost, and reduces the power consumption. We assume that programs are developed naturally using a graphical or equivalent language that can explicitly show all data dependencies. Instead of giving the CPU the privileged right of deciding what instructions to fetch in each cycle (as is the case for CPUs with a program counter), instructions are entering the CPU when they are ready to execute or when all their operand(s) are to be available within a few clock cycles. This way, the application-knowledgeable algorithm, rather than the application-ignorant CPU, is in control. The CPU is used just as a resource, the way it should normally be. This approach results in outstanding performance and elimination of large numbers of redundant operations that plague current processor designs. The latter, conventional CPUs are characterized by numerous redundant operations, such as the first memory cycle in instruction fetching that is part of any instruction cycle, and instruction and data prefetchings for instructions that are not always needed. A comparative analysis of our design with conventional designs proves that it is capable of better performance and simpler programming. Finally, VHDL implementation is used to prove the viability of this approach.

Keywords -- Processor design, data-driven (dataflow) model of computation, comparative analysis, concurrent operations, distributed CPU.

1. INTRODUCTION

PC(Program Counter)-driven CPUs are very widely used in all kinds of application environments. Current designs are the result of more than 25 years of evolution. The incremental performance improvements during this time

period have been primarily the direct or indirect result of higher transistor densities and larger chip sizes. Transistor densities and single-chip processor performance double approximately every eighteen months (Moore's law). Impressed by these facts, CPU designers have rarely attempted to circumvent this evolutionary path. Probably, they believe in the saying "If it does not break, do not fix it." RISC (Reduced Instruction Set Computers) CPUs have basically the same structure with CISC (Complex Instruction Set Computers) CPUs; they differ only in the complexity of individual components and instruction sets. Also, VLIW (Very Large Instruction Word) CPUs are similar to RISCs while relying heavily on the compiler and their wide data buses for good performance. Also, general-purpose processors exhibit 100 to 1,000 times worse energy-delay product than ASICs. Now is the right time to carry out a sincere evaluation of current CPU design practices and start an earnest investigation of alternative design philosophies, before we probably reach the silicon technology's limits around the year 2010.

Unfortunately, the trend in CPU design has been to take advantage of increases in transistor densities to include additional features. However, this is the wrong path. According to Hennessy “The performance increases over the past 15 years have been truly amazing, but it will be hard to continue these trends by sticking to the basically evolutionary path we are on” [16]. He reminds us that all basic ideas built into CPUs have been around for more than 20 years! In a more dramatic tone he says, “Occasionally, we have processors that get so big they roll backwards down the hill, rolling over the design team. So we can keep on pushing our ever larger rocks up these hills, but it’s going to get harder and harder, and it may not be the smartest use of all our design talent.” Our research effort presented here stems from this and other observations. To summarize, we strongly believe that current CPU designs are characterized by numerous redundant operations, do not match well with the natural execution of programs, have unreasonably high complexity, and consume significant power. In the remaining part of this section, our main purpose is to justify these observations by investigating current, advanced CPU design practices. We also summarize work related to the data-driven computation paradigm that underlies our design.

Current high-end microprocessors implement wide instruction issue, out-of-order instruction execution, aggressive speculation, and in-order retirement of instructions [6]. They generally implement on the fly only a small, dynamically changing window of dataflow execution. Under the pure *data-driven (dataflow) computation paradigm (model)*, an instruction is executed as soon as its operands become available [4]. Instructions are not ordered and carry with them their operands. As soon as an instruction produces a result, the result is copied into all other instructions that need it as input; different tokens with the same result are propagated to these instructions. The data-driven execution model matches well with the natural flow of program execution. However, we do not yet know how to implement this paradigm efficiently with current hardware design practices and silicon technologies. [5] showed that the dataflow instruction firing rule (that is, the instruction departure for the execution unit) can be implemented easily through state-dependent instruction completion. Therefore, they treated every memory reference as multiple instructions. Evaluating simple arithmetic expressions was rather slow because it required two operand fetch operations from the *activation frame*. An activation frame is located in the memory and is allocated

to each instruction just before it starts executing; it facilitates the storage of *token*s. A token includes a value, a pointer to the instruction that must be executed, and a pointer to an activation frame. The instruction contains an opcode (i.e., operation code), the activation frame offset where the token address match will occur, and the addresses of one or more instructions that need the result of this instruction. Therefore, this implementation of the dataflow computation paradigm is characterized by very significant computation and storage overheads. Monsoon, a multiprocessor based on these dataflow-based techniques was also implemented.

The data-driven computation paradigm has been primarily employed in the implementation of parallel computers, where this paradigm is basically applied among instructions running on different processors; the latter processors are PC-driven. The majority of the groups that have designed dataflow multiprocessors and multicomputers have made many compromises because they constrained themselves into developing systems with COTS (Commercial Off-The-Shelf) processors [5, 21]. In contrast, a data-driven processor was introduced in [22] that utilizes a self-timed pipeline scheme to achieve distributed control. This design is based on the observation that the data-driven paradigm can accommodate very long pipelines that are controlled independently, because packets flowing through them always contain enough information and data on the operations to be applied. However, this processor design also suffers from several constraints imposed by current design practices. Several data-driven architectures have been introduced for the design of high performance ASIC devices [23, 24]. In addition, several techniques have been developed for the implementation of ASICs in VLSI when the dataflow graphs of application algorithms are given. However, these techniques employ straightforward one-to-one mapping of nodes from the dataflow graph onto distinct functional units in the chip. An exception is the recently proposed implementation of dataflow computation on FPGAs [25].

Multithreading is widely employed in CPU designs. For multithreaded processors, each program is partitioned into a collection of instructions. Such a collection is called a *thread*. Instructions in a thread are issued according to the von Neumann (i.e., PC-driven) model of computation, that is they are run sequentially. Similarly to the dataflow model, instructions among threads are run based on data availability [11]. A large degree of thread-level parallelism is derived through a combination of programmer, compiler, and hardware efforts (e.g., aggressive speculation). COTS processors can implement non-preemptive multithreading, where a thread is left to run until completion. However, the compiler must make sure that all data is available to the thread before it is activated. For this reason, the compiler must identify instructions that can be implemented with *split-phase operations*. Such an instruction is a load from remote memory. Two distinct phases are used for its implementation. The load operation is actually initiated in the first phase (within the thread where the load instruction appears). The instruction that requires the returned value as input then resides in a different thread. This split-phase technique guarantees the completion of the first thread without extra memory-access delay.

Since multithreading supports dataflow execution among threads, it is worth also looking at other significant designs that adhere to multithreading. The Tera MTA (Multi-Threaded Architecture) implements a DSM (Distributed Shared-Memory) parallel machine with multithreaded computational processors (CPs) and interleaved memory modules connected via a packet-switched interconnection network [13]. In a DSM parallel computer, the local memories of individual processors are collectively combined to form a global address space that is accessible by all processors. MTA does not contain data caches. Each CP implements 16 protection domains (that is, it can execute 16 distinct applications in parallel) and 128 instruction streams [12]. Each instruction stream uses its own register set and is hardware scheduled. A user program is called a *task* and consists of a collection of *teams*. Each team utilizes a protection domain. Most often, teams from the same task are assigned to protection domains in different processors. Each team assumes a set of virtual processors, each of which is a process assigned to a hardware stream.

A thread may generally *spawn* (i.e., create) other threads. Contrary to subroutine calls, a spawned thread can operate concurrently with the spawning thread [15]. A new thread can be spawned at any time (that is, not only at the end) during the execution of the spawning thread. Cilk is a C-based runtime system for multithreaded parallel programming. [14] shows the efficiency of the Cilk work-stealing scheduler. A processor running out of work is allowed to steal work from another processor chosen randomly. It steals the shallowest ready thread in the latter's spawn area because it is likely to spawn more work than other deeper threads.

EARTH (Efficient Architecture for Running THreads) is a multiprocessor that contains multithreaded nodes [17]. Each node contains a COTS RISC processor (EU: Execution Unit) for executing threads sequentially and an ASIC synchronization unit (SU) that supports dataflow-like thread synchronizations, scheduling, and remote memory requests. A *ready queue* contains the IDs of threads ready to execute and the EU chooses a thread to run from this queue. The EU executes the thread to completion and then chooses the next one from the ready queue. The EU's interface with the network and the SU is implemented with an event queue that stores messages. The SU manages the latter queue. The local memory shared by the EU and SU in the local node is part of the global address space; that is, a DSM machine is implemented. A thread is activated when all its input data become available. The SU is in charge of finding out that all of its data is available and sends its ID to the ready queue. A sync(hronization) signal is sent by the producer of a value to each of the corresponding consumers. The sync signal is directed to a specific *sync slot.* Three fields constitute the sync slot, namely *reset count, sync count,* and *thread ID*. The reset count is the total number of sync signals required to activate the thread. The sync count is the current number of sync signals still needed to activate the thread (this count is decremented with each arriving sync signal). When it reaches zero, it is set back to its original value (i.e., the count is reset) and the thread ID is placed in the ready queue. Frames are allocated dynamically using a heap structure. Each frame contains local variables and sync slots. The thread ID is actually a pair containing the starting address of the corresponding frame and a pointer to the first instruction in the thread. The code can explicitly interlink frames by passing frame pointers from one function to another. User

instructions can access only the EU, not the SU. The implementation of the EU with a COTS processor implies that its communication with the SU is made via loads and stores to special addresses recognized by both units.

However, multithreading is not necessarily the most promising high-performance solution because it does not implement the dataflow model at the lowest possible, that is instruction, level and for the entire program. Also, multithreading and prefetching significantly increase the memory bandwidth requirements [18]. The *Processing-In-Memory (PIM)* technique [9] integrates logic into memory chips to potentially reduce the required memory bandwidth; this is achieved through operations that reduce the size of the transmitted data.

Reconfigurable computing has become a major research area, primarily due to the recent success of FPGAs (Field-Programmable Gate Arrays). For example, the architecture of a custom computing machine that integrates into a single chip a fixed-logic processor, a reconfigurable logic array, and memory was presented in [1]. The reconfigurable logic serves as a standard co-processor that, however, can change configuration based on a reconfiguration table that can be programmed at run time. Tomasulo's algorithm that assumes resource reservation stations is applied to fit the FPGA to the main processor's pipeline. Data stored in the main memory, the processor cache and the local memory of the FPGA must be coherent. A universal and reconfigurable message router for the implementation of interconnection networks in parallel computers was introduced in [8]. This router also can adapt statically and dynamically based on information loaded into its configuration table. Its adaptation is with regard to the number of I/O channels, their width, and I/O channel mappings. However, reconfigurable CPUs with powerful instruction sets are yet to appear because of their increased complexity.

Multiple-issue processors are an alternative to vector processing units [20]. Dynamic multiple-issue machines apply superscaling. On the other hand, VLIW processors are static multiple-issue machines (that is, the instruction execution schedule is determined at static, or compile, time) [19]. In the latter case, the compiler combines many independent instructions together to be sent simultaneously to the CPU. Each component instruction is to use its own execution unit in the CPU.

To match the ILP (instruction-level parallelism) of applications, current microprocessor designs apply *resource duplication* (or *superscaling*). More specifically, several copies of commonly used functional units are implemented in the CPU. This approach permits a smoother flow of instructions in CPU pipelines, with reduced stalls. An improved approach, namely *resource widening*, increases the width of resources to avoid bottlenecks at resource interfaces [3]. For example, whereas duplication of an ALU results in duplication of the I/O buses and the ALU resources only, widening forces also duplication of the register file. The Intel IA-64 will be the first commercially available microprocessor based on the *EPIC (Explicit Parallel Instruction Computing)* design philosophy [10]. Contrary to superscalar processors that yield a high degree of ILP at the expense of increased hardware complexity, the EPIC approach retains VLIW's philosophy of statically generating the execution schedule (for reduced

hardware complexity), while, however, improving slightly the processor's capabilities to better cope with dynamic factors.

The high complexity of individual processors has a dramatically negative effect on the overall complexity and performance of parallel computers. As a result, the design of parallel computers with scalable architecture and outstanding performance becomes a Herculean task [2, 7]. It is expected that about one million threads will be active simultaneously in PetaFlops (10^{15} floating-point operations per second) computers that may be built in the next 10-15 years. If current design practices perpetuate into the future, we can only imagine the amount of redundant operations and their share in power consumption for PetaFlops capable systems [7]!

To summarize, a close look at the current RISC, CISC, and VLIW processor design philosophies (these are the prevalent PC-driven designs) shows several deficiencies. They are characterized by large amounts of redundant operations and low utilization of resources directly related to the implementation of application algorithms. The program counter, which resides within the CPU, is used to fetch instructions from the memory. The first phase of the instruction fetch operation, however, is required only because of the chosen computation model (i.e., PC-driven); that is, this CPU request to the memory is not part of any application algorithm but is the result of centralized control during program execution. To alleviate the corresponding time overhead, current implementations use instruction prefetching with an instruction cache; many hardware resources (i.e., silicon transistors) are then wasted that could, otherwise, be used in more productive (i.e., direct, application-related) tasks. Another problem with current designs is the fact that the operands do not often follow their instructions to the CPU; the only exceptions are with instructions that either use immediate data or their operands reside in CPU registers. Additional fetch cycles may then be needed to fetch these operands from either the main memory or the attached cache. However, these fetch cycles also should be avoided, if possible. These fetch cycles are even unavoidable with current dataflow designs that use activation frames. Again, to mitigate this problem current designs choose data cache memories; the corresponding transistors could, otherwise, be used in more productive tasks. In contrast, in pure dataflow computing the instructions go to the execution unit(s) on their own, along with their operand(s), as soon as they are ready to execute.

Our D^2-CPU processor design philosophy is based on the pure data-driven computation paradigm and, like RISC and VLIW designs, promotes a small set of rather simple instructions, for ease and efficiency of instruction decoding and implementation. Also, large "register files" within the processing unit contain each time "active instructions" with their operands. To conclude, it is now widely accepted that the procedure applied within many advanced microprocessors for the execution of an ensemble of CPU-resident instructions resembles closely the data-driven computation paradigm. This is due to the fact that advanced microprocessors apply (super)pipelining techniques with resource reservation stations and keep track of data interdependencies between instructions in the CPU. However, we can definitely argue that more direct hardware support is needed if our ultimate objective is to

eventually fully integrate this computation paradigm into mainstream computing. For this reason, the data-driven execution model must be applied simultaneously to all instructions in the program. This can be achieved only if we depart completely from current CPU design practices. This paper dwells on this thesis/argument and proposes a new philosophy for CPU design that comes very close to this objective.

The paper is organized as follows. Section 2 discusses in more detail the advantages of the data-driven computation paradigm, and introduces new design concepts that can lead to D^2-CPU implementations for higher performance, ease of programmability, and low power consumption. Justifications are also presented. Section 3 presents the details of our proposed D^2-CPU implementation. Section 4 presents a comparative analysis with conventional designs. Section 5 presents a VHDL implementation of the program memory to demonstrate the viability of the approach. Finally, Section 6 presents conclusions and future objectives related to our new CPU design concepts.

2. INTRODUCTION OF THE D^2-CPU

Let us begin this section by introducing briefly the semantics of the data-driven computation paradigm, followed by a list of its advantages and disadvantages. The following scenario describes the sequence of steps for the implementation of an instruction under the data-driven computation paradigm.

- *Instruction issuance/firing*: it is the departure of the instruction for the execution unit. An instruction is issued just after all of its operands become available to it.
- *Token propagation*: it is the propagation of an instruction's result to other instructions that need it. As soon as an instruction completes execution, it makes copies of its result for all other instructions, if any, that need it. Different tokens that contain the same result are then forwarded to different instructions.
- *Instruction dissolvement*: it is the destruction of the instruction just after it produces all of its tokens for other receiving instructions. Loop instructions can be treated differently, as discussed later.

The main *advantages* of data-driven computation are:

- Instructions are executed according to the natural flow of data propagated in the program.
- Most often, there is a high degree of embedded parallelism in programs and, therefore, very high performance is possible.
- It is free of any side effects (because of the natural flow of data that guides the execution of instructions).
- It reduces the effect of memory-access latency because all operands are attached to their instructions.
- It naturally supports very long, distributed, and autonomous superpipelining because all instruction packets flowing through execution units are accompanied by all required information (including their operands).
- Based on the last observation, clock skewing is not an issue and, therefore, there is no need to synchronize all functional units.

The main *disadvantages* of data-driven computation are:

- Increased communication (or memory access) overhead because of explicit token passing.
- Instructions are forced to use their data (in incoming tokens) when they arrive, even if they are not needed at that time.
- The manipulation of large data structures becomes cumbersome because of the token-driven approach. Data access by-reference only for such structures may be needed to achieve high performance.
- The hardware for matching recipient instruction addresses in the memory with tokens may be complex and expensive.

Advances in current CPU designs lack the potential for dramatic performance improvements because they do not match well with the natural execution of program codes. In efforts to alleviate this critical problem, which inadvertently affects all aspects of program execution, designers often apply expensive techniques. However, the relevant hardware is not used to directly run part of the original program. Instead, it is used in "unproductive" (i.e., not directly related to the application) tasks that result in small "productive" utilization of the overall system resources. For example, the time overhead of fetching instructions and/or operands into PC-driven CPUs is reduced with expensive means, such as preprocessing (i.e., software program analysis), instruction and data prefetching, cache, internal data forwarding, etc. However, resulting processors have the following drawbacks:

- In our effort to hide the mismatch between the application's needs and the PC-driven execution model, we waste numerous, otherwise precious, on-chip resources. Many hundreds of thousands or millions of transistors are needed to implement some of the above techniques within a single CPU, whereas the "productive" utilization of these resources is rather small.
- Power consumption increases for two reasons. Firstly, the overhead of the instruction fetch cycle, which is not explicit in any application program, appears for each individual instruction in the program; too much an overhead to pay for centralized control during program execution. Since these are interchip data transfers that are quite expensive, this cost is very substantial. Secondly, unnecessary power consumption results from prefetching unneeded instructions and data into caches. Mobile computing has recently become popular and will probably dominate the computing field in the near future. For a relevant system to be successful, it needs to be power efficient.
- Numerous cycles are wasted when a hardware exception (interrupt) occurs. This is because after the CPU gets informed about the external event, it has to store the current state of the machine and then fetch code to run the corresponding interrupt service routine. If the appropriate context switching is instead selected outside of the CPU, then the appropriate instructions can arrive promptly.

It is now time to identify/derive the major requirements for single CPU designs that could avoid old pitfalls of the type discussed above:

- Programs are developed using a fine-grain graphical, or equivalent, language that shows explicitly all data dependencies among the instructions. Libraries of existing routines can further aid programming, as long as they

are developed in this manner. Also, usage of a graphical language simplifies code development and facilitates better assignment of tasks to a parallel computer containing many D^2-driven CPUs.

- Instructions contain all their operand fields, as in pure data-driven computation.
- A software preprocessor finds all instructions in the program that can run in the beginning because of non-existent data dependencies. These head instructions are to be sent first to the execution unit.
- Following the head instructions to the execution unit are instructions that are to receive all their (input) operands from one or more head instructions. These instructions can proceed for execution just after they receive their operand(s). This rule is applied to all instructions during execution, so that they follow to the execution unit the instructions that are to produce values for all of their missing operands.
- Instructions that are to receive one or more operands from instructions that are ready to execute, but they will still be missing one or more operands leave for an external cache, namely the EXT-CACHE, where they wait to receive their tokens. To reduce the traffic, instructions that will receive the same result are grouped together in the cache in an effort to collectively receive a single token that can be used to write all relevant operand fields. If not all of the token receiving instructions can fit in the EXT-CACHE, then a linked list is created in the memory for instructions that do not fit.
- Only one copy of each instruction, including its operand(s), resides at any given time within the entire (i.e., memory, cache, and CPU) machine. This is in contrast to the wide redundancy of instructions and data present in the cache, memory, and CPU of conventional machines.
- Instructions do not keep pointers to their parent instructions (that produce values for their input operands). Therefore, they sleep until they are forced into the EXT-CACHE or the execution unit, in order to receive their token(s). This lack of backward pointers, however, does not seem to permit instruction relocation which is needed to implement multiprogramming and/or virtual memory. Without its parent pointer(s), a relocated instruction cannot inform its parent(s) about its new location, for appropriate token propagation. Despite this observation, we have devised a technique that permits token forwarding inexpensively for relocated instructions without parent pointers. Our technique is described in the next section.
- After an instruction is executed, it is dissolved. However, special care is needed for instructions that have to be reused in software loops. A relevant technique that permits instruction reuse is presented in the next section.
- Instructions have unique IDs for token passing only while they reside outside of the execution unit. These IDs are used to find instructions and force them into the EXT-CACHE or the execution unit. In the latter case, an interface actually keeps track of these IDs, so that minimal information is manipulated or stored in precious execution unit resources.

These requirements are taken into account in the next section to derive an innovative design, namely the D^2-CPU design, that satisfies the following objectives:

- A radical uniprocessor design that implements the data-driven computation paradigm in its pure form.

- A processor design with distributed control that minimizes the amount of redundant operations and maximizes performance.
- High utilization of resources in productive work (that is, work not associated with redundant operations).
- Low hardware complexity for high performance.
- Low cost and low power consumption.

3. D^2-CPU DESIGN

We propose in this section an innovative uniprocessor design that takes advantage of advances in PIM, cache-memory, and integrated-circuit technologies to implement very efficiently the data-driven computation paradigm. We call the proposed design the *D^2-CPU* design, where D^2 stands for Data-Driven. The specifications for this design were produced earlier. Figure 1 shows the main components of the new processor. The EXT-CACHE (EXTernal to the execution unit CACHE) is distributed; it is formed as a collection of $DSRAM_i$s (Distributed SRAM caches), for $i = 0, 1, 2, \ldots, 2^d-1$, one DSRAM module per DRAM module. The DRAMs originally contain the program code (that is, all instructions with their filled or unfilled input data). More information about the individual units in Figure 1, and their modes of interoperability, are presented in this section. Let us first present some basic assumptions.

We assume that each instruction comprises an opcode field and, without loss of generality, up to two operand fields. It has one or two operand fields for a unary or binary operation, respectively. If the instruction is to produce a result (since an instruction may just change the machine "status"), then a variable-length sequential list follows. This list contains pairs of recipient instruction IDs and single-bit operand locators, so that data produced by the former instruction (that will be transmitted in tokens) can be propagated to the latter instructions; the operand locator bit is *0* or *1* for the first or second operand field, respectively. We actually assume that the instruction's context ID is appended to the actual instruction ID to form a new ID. The context ID is not needed for single-process systems. When the instruction goes to the ERU for execution, these pairs go to their associated DSRAMs and wait there to receive their result tokens; the ID also of the result producing instruction is kept in each DSRAM to facilitate the identification of the incoming token for data replication.

The core of this design is the *Execution-Ready Unit (ERU)* that comprises the following components:

- The *Processing Unit (PU)* where the operations specified by the instructions are executed. It contains several functional units that can be used by a single, or simultaneously by multiple, instructions. Its design follows the RISC model. The PU contains at least one copy of an adder, a multiplier, and a logic unit. It is recommended to also include a vector unit for multimedia and engineering applications.
- The *ERU-SRAM (static RAM in the ERU)* contains instructions ready to execute (i.e., all of their operand fields are filled). However, these instructions cannot proceed to the PU yet because the functional units that they require are currently in use. When a functional unit becomes available, its ID is sent to this cache in order to find an

instruction that requires this unit for implementation. Therefore, resource matches are seeked based on unit IDs. This cache storage of ready-to-execute instructions guarantees very high performance.

- The *SRAM** (which is implemented in SRAM technology as well) contains instructions with one or more unfilled operand fields that are all to be written by one or more instructions currently residing in the PU and/or ERU-SRAM. Therefore, the former instructions will execute in the very near future. As discussed earlier, when a specific functional unit in the PU becomes available, it broadcasts a signal to the ERU-SRAM and instructions there requiring the particular unit compete for it. As a rule of thumb, the instruction with the larger number of recipient instructions in the SRAM* wins. Thus, for each instruction in the ERU-SRAM there is a hardware count of its result recipient instructions in the SRAM*.

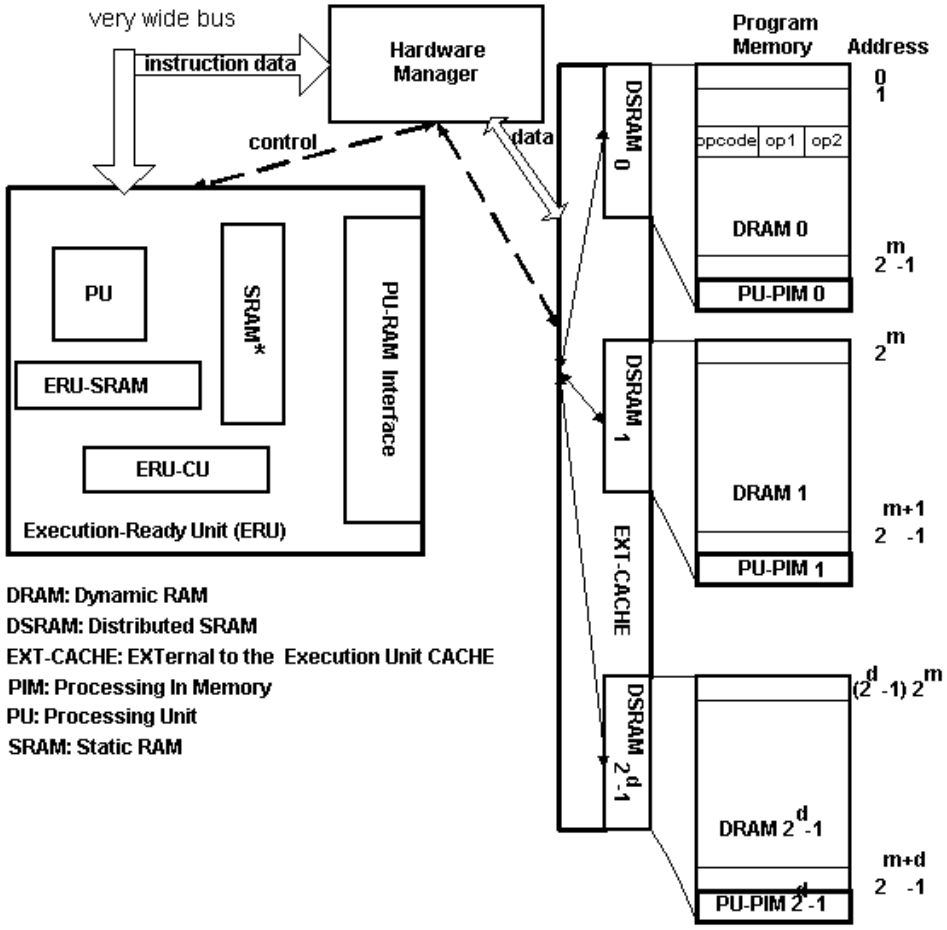

Figure 1. Architecture of the D^2-CPU and its interface.

- The *ERU-CU* is the control unit of the ERU. It keeps track each time of the total number of result recipient instructions in the SRAM* for each instruction currently in the ERU-SRAM, as discussed in the last paragraph. It also facilitates data forwarding within the ERU, for recipient instructions in the SRAM*.
- The *PU-RAM interface* receives from the hardware manager all instructions entering the ERU (that is, all instructions eventually going to the PU for execution). It distributes the instructions accordingly to the PU, ERU-SRAM, and SRAM*. When an instruction produces a result, the PU-RAM interface forwards to the hardware manager the *virtual ID* of this instruction followed by the result. The virtual ID of an instruction is a temporary ID assigned to it by the hardware manager just before it enters the ERU.

The *hardware manager* performs the following tasks:

- It initially sends the head instructions of the program to the ERU for execution.
- Whenever one of the remaining instructions proceeds to the ERU on its own, it first makes a request to the hardware manager for a virtual ID. This ID will uniquely identify the instruction during its residency in the ERU. This ID is a small number in the range *0* to *n-1*, where *n* is the maximum number of instructions that can reside simultaneously in the ERU. These IDs are recycled. The reasons for assigning new IDs to instructions (instead of keeping their long external/original IDs) are: (1) to minimize the required bandwidth between the ERU and external components; (2) to minimize the size of ERU-internal resources storing the instructions' IDs; and (3) to minimize the required resources that process ERU-resident information.
- It maintains a table that can be accessed to quickly translate on-the-fly short (virtual) IDs into long (original) instruction IDs for results broadcast by the PU-RAM interface in the ERU. The modified tokens are then forwarded to the EXT-CACHE where they are consumed by their recipient instructions, as described in the next paragraph.

The *EXT-CACHE* contains at any time instructions that are to receive tokens from instructions residing at that time in the ERU. Actually, several classes of instructions may reside in the EXT-CACHE at any time during program execution. These classes are:

- **Class A**. Instructions with two unfilled operand fields. One of these fields is to be filled with data that will arrive from an instruction currently in the ERU.
- **Class B**. Instructions with one unfilled operand field for which the token is to arrive from an instruction currently residing in the ERU. These instructions cannot fit in the ERU because the SRAM* is fully occupied.
- **Class C**. Instructions that are not missing any operands but they cannot fit in the ERU-SRAM because it is fully occupied. Of course, we should like all such instructions to reside in the ERU.

Only one copy of each not-yet-executed instruction is present in the system at any time during program execution. The part of the program that still needs to be executed is distributed among the off-chip DRAM and EXT-CACHE, and the on-chip ERU-SRAM, SRAM*, and PU. The currently achievable transistor density for chips allows the implementation of large enough memories to realize the ERU-SRAM, SRAM*, and DSRAM components so that

they very rarely overflow. Without hardware faults, there is no possibility for the appearance of deadlocks in this design. Even if the ERU-SRAM is fully occupied at some time, the instructions in it will definitely execute in the near future because the PU will be released soon by the currently executing instructions. If one or more instructions outside of the ERU are ready to execute but cannot enter the ERU because the ERU-SRAM is fully occupied, then they wait in an external queue until space is released in the ERU-SRAM. A similar technique is applied if the SRAM* is fully occupied. In fact, we could combine the ERU-SRAM and SRAM* memories into a single component. However, for the sake of simplicity we have chosen to separate them in this presentation.

The virtual IDs assigned by the hardware manager to instructions departing for the ERU are q-bit numbers, where $q=log_2 n$ (n is the maximum number of instructions that can reside simultaneously in the ERU, and we assume that it is a power of 2). These IDs are recycled. For each *program memory module* $DRAM_i$, there is a distinct cache $DSRAM_i$ in the EXT-CACHE, for $i=0, 1, 2, ..., 2^d-1$, containing instructions that are to receive tokens from instructions currently residing in the ERU. Entries in this cache have the format shown in Figure 2, and the fields in each entry are:

- *IID*: the *token-sending instruction ID* assigned originally by the compiler-loader combination.
- *IAD*: the *token-receiving instruction ID*. It is the ID of an instruction that came from the corresponding $DRAM_i$. This instruction will consume a token produced by the instruction with ID stored in the *IID* field of this entry. More than one entries may have the same value in their *IID* field (if many instructions are to receive data produced by the same instruction).
- *OPFL*: *operand field locator*. A value of zero in this one-bit field means that the data from the received token is for the first operand field of the instruction with ID *IAD*. A value of one implies that the second operand field must be written with the data in the received token.
- INSTR: the *token receiving (recipient) instruction*. It is the instruction with original ID stored in the IAD field of this entry.

We assume that each token leaving the ERU with the result also contains the ERU-related (i.e., virtual) ID of the instruction that produced the result. The hardware manager changes on the fly the virtual ID to the original ID of this sending instruction and broadcasts this token to all DSRAMs in the EXT-CACHE. We assume up to 2^s original instruction IDs and r-bit instructions. We need to point out here that these DSRAM entries are created dynamically by the hardware manager, have a very short life span, and exist only inside the DSRAMs; they are created only when instructions leave for the ERU. That is, we do not load into the computer system pointers to parent instructions.

IID	IAD	OPFL	INSTR
Size of the field (bits): *s*	*s*	*1*	*r*

Figure 2: The format of entries in DSRAMs.

The ERU receives instructions for execution from the hardware manager; the latter can also choose in the EXT-CACHE an active context ID (for example, this process is required for exceptions that force the execution of interrupt service routines). The hardware manager forces instructions into the ERU by first storing them into FIFO buffers and then prompting the ERU to read from these buffers using a very wide bus. Asynchronous communications with appropriate acknowledgments between these two units can achieve this task. Ideally, each individual DSRAM in the EXT-CACHE should interface the ERU with its own FIFO buffer and corresponding control signals, and the ERU should have separate buses to access separate FIFO buffers. Therefore, it is not the ERU that fetches intructions for execution; it is fed with instructions directly by the EXT-CACHE. In addition, the amount of ERU outward data transfers is minimal because produced data tokens are often forwarded internally to the local SRAM*. When the PU produces the result of an instruction, this result is forwarded to the ERU-CU (Control Unit in the ERU) and PU-RAM interface units. The former unit stores the result appropriately into recipient instructions, if any, in the SRAM*. The latter unit sends a special token to the hardware manager, so that the instructions outside of the ERU that are supposed to be recipients of the produced data can receive the result. A single token is produced by the PU-RAM interface unit in order to minimize the amount of traffic and, therefore, maximize the effective bandwidth. As emphasized earlier, the special token comprises two fields that contain the produced data and the ERU-resident ID of the instruction that produced the result, respectively. The hardware manager replaces this short ID with the instruction's original ID and forwards the token to all DSRAMs. Fetching short IDs (assigned by the hardware manager) is not a heavy penalty to pay for the elimination of the PC. Let us not forget that a PC-driven CPU requires the implementation of a wide address bus and appropriate control lines. Our ERU needs fewer pins to fetch (in one memory bus access) this ID, whereas PC-driven CPUs need more pins to access instructions (in two memory bus accesses, by first sending out a RAM address and then receiving the data).

For a fixed value of *i*, the tasks carried out by the *PU-PIM*$_i$ are:

- It loads the corresponding DSRAM$_i$ with all those instructions from the DRAM$_i$ that are to receive tokens from instructions leaving for the ERU and are also missing data for two operand fields. Also, it always updates appropriately the DSRAM$_i$ directory.
- It removes instructions from the DRAM$_i$ that are not to be executed further because of loop exiting. The reuse of instructions for the implementation of program loops is addressed later in this section.

- It maintains three distinct lists of addresses for instructions in the $DRAM_i$, if any, that do not fit in the EXT-CACHE, ERU-SRAM, and SRAM*, respectively. These lists are kept in the local $DRAM_i$ for instructions that do not fit in one of these three units because of a respective overflow.
- It copies data from tokens broadcast by the ERU (via the hardware manager) into the appropriate operand fields of instructions appearing in the aforementioned overflow lists for the EXT-CACHE and SRAM* units.
- It carries out garbage collection in the $DRAM_i$ since the data-driven model of computation necessitates deallocation of memory space dynamically through instruction dissolvement.
- It finds instructions in the $DRAM_i$ and $DSRAM_i$ that are to receive their last operand from instructions leaving for the ERU and forwards them to the hardware manager that finally stores them into the SRAM*.
- It services requests by the program loader and the operating system for instruction loading and relocation in the $DRAM_i$. Justification of the need to support instruction relocation and a technique to solve this problem for the D^2-CPU are presented in the next subsection.

In order to speed up the overall process, the $PU\text{-}PIM_i$ unit may be replicated many times for each module (or block of modules) $DRAM_i$. The ensemble of all components that form the ($PU\text{-}PIM_i$, $DRAM_i$) pair is then looked upon as a parallel shared-memory machine. These operations are summarized in Table I.

Table I. Set of required operations for a PU-PIM.

OPERATION	ACTION
FETCH	Instruction fetch into the DSRAM
REMOVE	Removes an instruction from the DRAM because of loop exiting
MAINTAIN EXT-CACHE	Maintains a list of recipient instruction addresses when the EXT-CACHE overflows
MAINTAIN ERU-SRAM	Maintains a list of recipient instruction addresses when the ERU-SRAM overflows
MAINTAIN SRAM*	Maintains a list of recipient instruction addresses when the SRAM* in the ERU overflows
TOKEN PASSING	Copies data in tokens broadcast by the ERU to instructions in these maintainance lists
GARBAGE COLLECTION	Garbage collection in the attached DRAM
LOOKAHEAD INSTRUCTION DISPATCH	Finds and forwards to the hardware manager instructions in the DSRAM or DRAM that are to receive their last input operand(s) from instruction(s) leaving for the ERU
INSTRUCTION LOADING	Instruction loading into the DRAM

INSTRUCTION RELOCATION	Instruction relocation in the DRAM

Let us now justify the incorporation of the *DRAM program memory* in our design. We strongly believe that RAM memory (where data is accessed using distinct addresses) is absolutely essential in any computer system, including the D^2-CPU that implements the data-driven computation paradigm. The justification is as follows:

- Accessing data with distinct addresses is quite natural. Even humans are accessed uniquely and efficiently with home addresses, names in ordered directories, telephone numbers, email addresses, or any combination of them.
- There exist many devices that work extremely efficiently using strict memory addressing schemes (e.g., scanners, PC monitors, HDTV sets, etc.). For example, PC monitors use sequential addressing, often with bank switching and/or line interlacing.
- It is generally a good design practice to assign distinct memory addresses to I/O devices. It simplifies significantly related operations (i.e., data exchanges between I/O devices and program instructions). Some of the tokens produced in the D^2-CPU are stored into buffers that interface I/O devices.

3.1. Support for Instruction Relocation

If data-driven computation is to compete with, and surpass in success, the PC-driven paradigm, then it has to provide for *multiprogramming* and *virtual memory*. Both of them, in turn, require support for instruction relocation. Instruction relocation in data-driven computation seems to be a very difficult problem to solve because of the need for token passing with ever-changing instruction (target) addresses. We have devised the following solution for the implementation of instruction relocation in a way that token passing using original instruction IDs is still possible. More specifically, the following procedures are applied for memory assignment and instruction relocation:

- The compiler-loader combination assign the original instruction IDs to correspond to absolute memory addresses. If a memory location is free at that time, then the corresponding instruction, if any, is loaded there. The instruction's context ID (that is, the program number) is also stored in the memory along with that instruction. If that memory location is occupied by another instruction, then the former instruction is relocated early according to the method described below.
- For the sake of simplicity, we assume *instruction relocation* limited to a single program module $DRAM_i$. A distinct *ID memory* module ID_MEM_i, implemented in DRAM technology, is associated with each program memory module $DRAM_i$. The two memory modules have the same number of locations. The j^{th} entry in the ID_MEM_i contains the starting address of a hash table containing pointers to all instructions with original ID equal to j, but with different context IDs, for $j = 0, 1, 2, ..., 2^m-1$. When an instruction with original ID F relocates in the $DRAM_i$, then the respective PU-PIM_i unit stores in the hash table pointed at by the value in address F of the ID_MEM_i the context ID and the new address of this instruction. The hashing technique applied to locate the new address of an instruction uses the program/context ID as the key. Figure 3 shows the ID memory.

- The PU-PIM$_i$ unit keeps track of the location of all instructions in the DRAM$_i$. More specifically, it updates the hash tables to point to the new location of instructions. Each instruction carries its original ID and context ID. Whenever it is moved to a new location, the corresponding pointer in the associated hash table is updated by the PU-PIM$_i$. This scheme eventually implements memory indirect addressing for token propagation with maximum flexibility.
- As described earlier, if an instruction with context ID Q goes to the ERU for execution, then the instructions that are to receive its tokens must go to the SRAM* or EXT-CACHE. To find the current location of an instruction in the latter group, with original ID F and context ID Q, the PU-PIM$_i$ performs the following tasks:
  - If the instruction at location F of the DRAM$_i$ has ID F and context ID Q, then the recipient instruction has been located.
  - Otherwise, the hash function with key Q is applied for the table pointed at by the entry at location F of the ID_MEM$_i$, in order to find the new location in the DRAM$_i$ of the token recipient instruction.

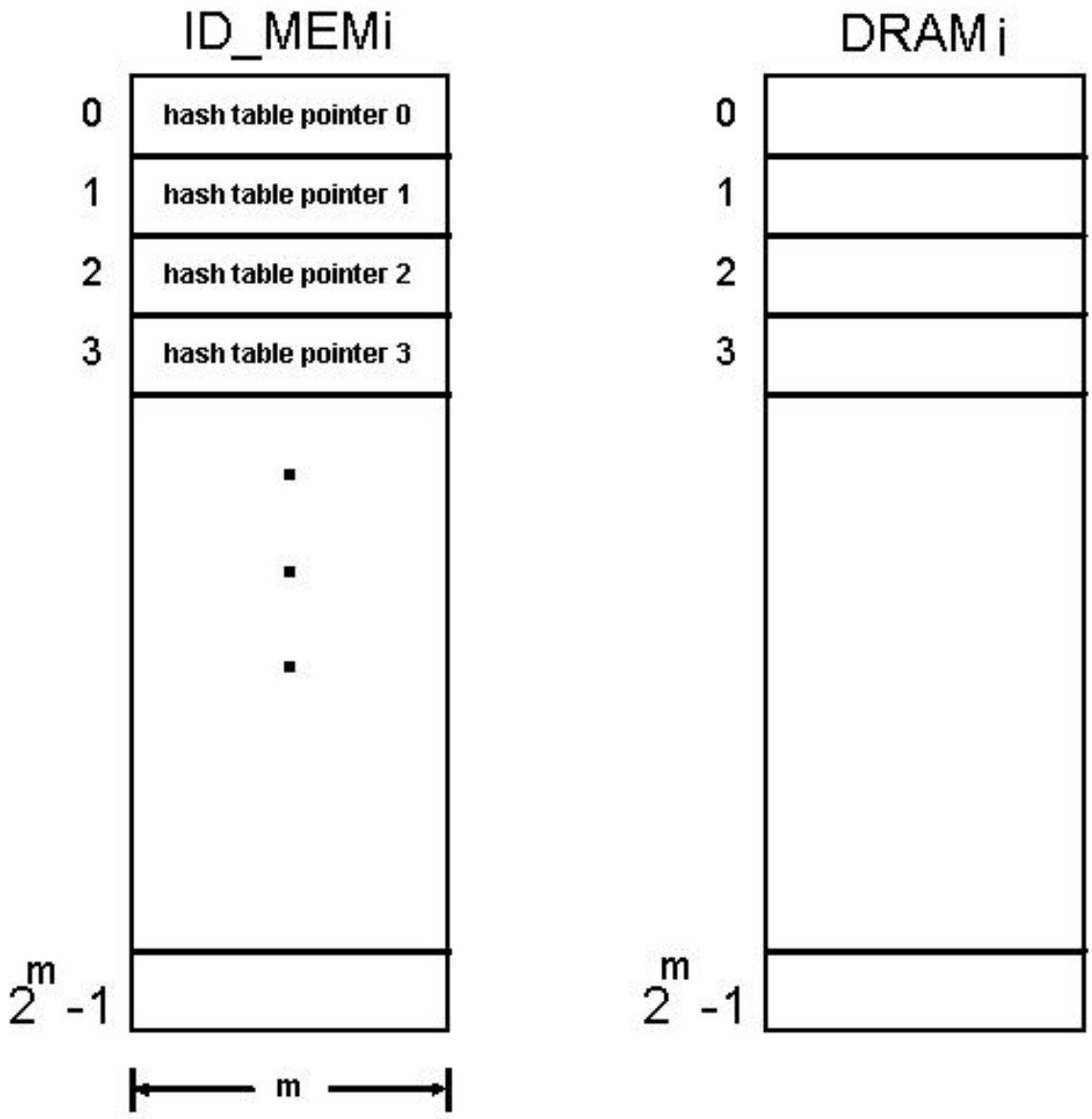

Figure 3. The i[th] pair of the ID_MEM and DRAM memory modules.

3.2. Handling Exceptions

Let us now describe the process followed by our D^2-CPU in handling exceptions. We must distinguish between software and hardware exceptions (interrupts). For example, a software exception will occur if an erroneous result is to be produced by an instruction (e.g., division by zero). An appropriate set of instructions are normally executed before the instruction that may produce the erroneous result. If their code determines that the erroneous result will show up with the execution of the latter instruction, then a thread of instructions (i.e., an exception routine) are activated (by passing an *activation token*) to deal with this problem. Among other tasks, this thread removes from the system the "faulty" instruction; although this instruction will never be executed because its full set of operands will never become available to it, it must still be removed to free memory/system space. Of course, some software exception routines (like for arithmetic overflow) must always be available at run time. Replication of the exception routine code an unknown number of times may then be needed. The hardware manager is instructed (by the

instruction that makes the determination about the erroneous result occurrence) to use the PU-PIMs in replication of code stored in the DRAM memory. The hardware manager sets as context ID of the replicated code the context ID of the determining instruction. It is not necessary to halt the execution of instructions that do not belong to the exception routine. If required to do so, however, because of high priority, then the hardware manager can temporarily ignore all instructions in the EXT-CACHE with context ID different from that of the exception determining instruction.

Hardware exceptions can be dealt with in the following manner. The instructions for exception routines are initially stored in the DRAM memory. The hardware manager receives the exception request along with an exception ID. This ID is used to uniquely determine the address of the first instruction in the exception routine. The hardware manager forces the PU-PIMs to make a copy of the exception routine code and also sends an activation token to the first (i.e., head) instruction. The instruction is eventually fetched from a DRAM into the ERU via the hardware manager. From this point on, the hardware manager disables all transfers to the ERU of instructions that have different context ID than this exception ID. It also sends this exception ID to the ERU to disable the execution of instructions with different context IDs. Every (software or hardware) exception routine contains a *last instruction* that upon execution forces (by passing a token) the hardware manager to enable all context IDs for the resumption of program execution. Our design can easily deal with different priorities for exceptions by keeping active, each time an exception occurs, all contexts with ID greater than or equal to the ID (priority) of the exception.

It is worth mentioning here that, sometimes the ERU-SRAM and SRAM* memories may be almost full and, depending on the amount of code needed to run when an interrupt/exception occurs or a new process starts executing, space has to be made in these memories for the new instructions. The applied technique will be similar to “space replacement” techniques currently applied to the memory hierarchies of conventional computer systems. Nevertheless, space is allocated on a demand basis, as needed, so the overhead is not significant; there is no need to relocate all instructions residing in these memories. If we also consider the facts that exception handling rarely occurs in computer systems and exception handling routines are often small in size, then we can conclude that this overhead is negligible compared to the total amount of time needed to execute application programs.

3.3. Program Loop Implementation

To facilitate efficient memory management, loop instructions should not be repeated unwisely in the memory. A bit in each instruction can indicate its inclusion in a loop, so that the instruction can be preserved at the end of its execution for future executions. Only its operand fields are emptied, if needed, after each execution. Upon exiting a loop, the last instruction sends a special *dissolve token* to the first instruction in a special routine that removes all loop instructions from the memory; only the PU-PIMs are involved in this process. It is not necessary to have a single loop exit. As for conditional branching, instructions that are not executed (because other execution paths

were followed) are dissolved similarly by special routines. However, we do not cover all details here because more work may be needed for a better solution with higher performance and lower cost.

3.4. Similarities and Differences Between the D^2-CPU and VLIW Architectures

Similarly to VLIW architectures, our ERU requires a wide instruction bus because each DSRAM memory can send one instruction (that also contains up to two operands) in a single memory cycle. Also, we assume asynchronous transfers between the ERU and individual DSRAMs, whereas the simultaneous transfer of all components (i.e., simple instructions) of a long instruction is a prerequisite for the execution of instructions on VLIW architectures. Therefore, the D^2-CPU design has these obvious advantages over VLIWs:

- Dramatic reduction in redundant operations (because VLIWs are PC-driven).
- It is faster because of asynchronous instruction and data transfers (when a DSRAM is ready, it just sends its instruction to the ERU without waiting for the other DSRAMs to also become ready).
- Any code is portable since there is no need for a compiler to group together simple instructions into large ones that are executable on the specific machine. In contrast, simple instructions are dispatched to the ERU.
- Therefore, there is no need for an expensive compiler.

4. COMPARATIVE ANALYSIS

Let us now carry out a comparative evaluation of our D^2-CPU with conventional PC-driven CPUs. Comparisons will be made under various criteria. For the sake of simplicity, we assume non-pipelined units.

4.1. Criterion: Operation Redundancy

The main purpose for the introduction of the D^2-CPU design was to dramatically reduce the number of redundant operations during program execution and to use the corresponding resources for the efficient implementation of operations explicitly requested by the executing code. The first step was to completely eliminate the instruction fetch cycle, which is implemented for every instruction in the PC-driven model, by adopting the data-driven model of execution. Assume the following parameters:

- k: the number of clock cycles in that part of the memory fetch cycle that begins with the initiation of the cycle by the microprocessor and ends when the memory starts sending the data on the memory bus.
- f: the probability for a page fault to appear (that is, data still reside in secondary devices).
- p: the penalty in number of clock cycles resulting from a page fault.
- N: the total number of instructions in the program that have to be executed.

We can assume that our D^2-CPU has a zero probability for page faults because: (a) there can always be enough free memory space to load code since instructions are dissolved just after their execution and (b) pages are created based on the data dependence graphs of programs that show explicit data forwarding and therefore imply known snapshots for future instruction executions. The data-driven computation paradigm, and therefore our CPU, requires

to write the result of each operation into instructions that will use it as an input operand. The recipient instructions will arrive later for execution, along with their operands. Under the PC-driven paradigm, the result is written once but it is fetched separately for individual instructions by implementing memory fetch cycles that use both directions of the memory bus in two consecutive phases.

Therefore, the redundancy in operations for the PC-driven design results in an overhead of $T_{operations} = O(N*(k+f*p))$. In fact, this number may be higher with pipelining because of unnecessary prefetch operations for failed branch predictions. It also increases further for cache memory.

4.2. Criterion: Storage Redundancy

Contrary to PC-driven CPUs that attempt to hide memory latency by keeping duplicates of the "most probable to use in the near future" information in caches attached to the CPU, our design keeps only a single copy of each "yet to be executed" instruction in the system. Of course, the D^2-CPU does not need (cache to keep) duplicates of instructions because the execution path is known at any time from the data dependence graph. Also, instructions are inherently dissolved for our design just after they are executed, and therefore they free space, and simplify numerous compiler and operating system tasks. Therefore, the PC-driven design has storage overhead *O(N)* words because it does not dissolve instructions after they are executed and also keeps multiple copies of many instructions. Assuming a fixed number of operands per instruction, the storage redundancy associated with the D^2-CPU due to having multiple copies of the operands (i.e., one copy per instruction) is *O(N)*. However, the PC-driven design requires to store the operand addresses for instructions; these addresses also occupy space *O(N)*. The redundancy associated with storing on the D^2-CPU the addresses of instructions that are to receive data tokens is also *O(N)*. Therefore, all these counts of redundancy cancel each other out in comparative analysis.

4.3. Criterion: Programmability and Program Translation

The D^2-CPU does not require advanced compilers and powerful programming languages because the programs are written in graphical languages that relate directly to the execution model of the processor. PC-driven CPUs, on the other hand, require careful programming and incorporation of advanced compiler techniques. Although the majority of applications are parallel in nature, the programmer is forced to put very substantial order in the code because the target machine executes all programs sequentially (i.e., it is PC-driven). Secondly, the compiler traces and translates the program sequentially. Thirdly, the program counter (which is a single resource) actually initiates all major CPU activities in a way that implies each time an active segment in the sequential source code.

4.4. Criterion: Required Resources

We assume the following:

- The two designs have identical ALU execution units in their CPUs.
- The two designs have identical instruction (opcode) decoding units in their CPUs.

The PC-driven CPU requires on-chip cache for $O(m*s)$ words of instructions and data, where m is often the chosen number of the most frequently used instructions and data, and s is the number of words in cache blocks chosen to conform to spatial locality in program execution and data referencing. The D^2-CPU requires on-chip cache of $O(n)$ words for the ERU-SRAM and SRAM*, where n is the number of functional units in the PU. We should expect that $O(m*s)>O(n)$.

As for the CPU-external cache, the PC-driven design requires a second-level cache with more than $O(m*s)$ words. Assume that t is the node degree for outgoing edges in the data dependence graph of programs. The D^2-CPU design requires $O(n*t)$ words for the DSRAM because $O(n)$ instructions may reside in the ERU simultaneously, and each such instruction may have to send its result to $O(t)$ other instructions. Again, we should expect the PC-driven design to require more external cache memory than the D^2-CPU design.

The D^2-CPU design requires PU-PIM units for the DRAMs; they are not present in conventional PC-driven designs. However, these units are external to the ERU and are distributed among the DRAMs. Therefore, they do not complicate the CPU design. Also, they have relatively small complexity since they are required to be capable of implementing only the small set of operations shown in Table I.

4.5. Criterion: Turnaround Time

The turnaround time of a program depends on several parameters. Our objective is to find the expected peak performance of each CPU. We make the following assumptions for an analytical evaluation of performance:

- N: the total number of instructions in the data-dependence graph of the program.
- The program contains only zero-, one-, and two-operand instructions.
- N': the total number of zero-operand instructions when we start the execution of the program.
- q: the ratio of one-operand instructions in the remaining code (of N-N' instructions). These instructions are uniformly distributed in the program.
- 1-q: the ratio of two-operand instructions in the remaining code (of N-N' instructions). These instructions are uniformly distributed in the program.
- Every instruction in the program produces a one-word result.
- The entire program is originally stored in the DRAM (that is, there are not any page faults) and there is no need for relocation of instructions.
- The program does not contain any software loops.
- t: the fixed number of outgoing edges for each instruction in the data-dependence graph. Leaf instructions also in this graph copy their result.

- Every instruction opcode fits in a single word for the PC-driven CPU. Every instruction opcode and the corresponding instruction's ERU-resident ID are stored in a single word for the D^2-CPU.
- *B*: the maximum bandwidth of the processor-DRAM bus in words per cycle.
- Every instruction operand or result fits in a single word.
- The PU-PIMs do not consume any extra cycles for garbage collection.
- The hardware manager for the D^2-CPU replaces original instruction IDs with ERU-resident (virtual) IDs, and vice versa, on the fly without any overhead.
- Memory addresses are one-word long.
- The order chosen for writing data from a produced token into the *t* recipient instructions does not affect the performance of the D^2-CPU.
- For any executed instruction, its token recipient instructions are uniformly distributed among the DRAMs.
- PU-PIMs write data from a produced token at the collective rate of *B* recipient instructions per cycle (since each datum is a single word).
- *c*: the total capacity, expressed in number of words, of registers in the PC-driven CPU. It is the same as the total cumulative capacity of the ERU-SRAM and SRAM* units in the D^2-CPU. It is important to note, however, that the capacity of the latter CPU can be much larger because of its much lower complexity (that is, low-utilization per-word caches on the PC-driven CPU are replaced with high-utilization ERU-SRAM and SRAM* units on the D^2-CPU).
- *E*: the peak execution rate, in number of instructions per cycle, for ALU operations in both CPUs.
- Instructions in the program do not have any resource dependencies.
- The CPUs do not contain cache where instructions or data could be prefetched.
- The PU-PIMs fetch instructions into their DSRAMs at a rate that usage of the DRAMs is transparent to the ERU. It is a reasonable assumption because this data-driven process can overlap the transferring of instructions from the DSRAMs to the ERU.

4.5.1. Turnaround Time on the PC-Driven CPU

There are several components in the derivation of the total turnaround time of a program run on the PC-driven CPU. The first component is the total amount of time required to fetch all instruction opcodes into the CPU. The unidirectional effective bandwidth of the CPU-DRAM bus is *B/2* words per cycle, because it is divided equally between the address and data buses (each datum or address fits in a single word); also, two transfer phases of opposite direction are implemented for each instruction because the instruction's address must be first sent out using the memory address register (MAR) in the CPU. Thus, the total number of cycles needed to fetch all instruction opcodes is

$$T^{PC}_{instr-fetch} = \frac{N}{B/4} = \frac{4N}{B}.$$

The next component is the total time required to fetch all instruction operands into the CPU. The total number of cycles is given as the summation of operand fetch times for one- and two-operand instructions, as shown in

$$T^{PC}_{oper-fetch-memory} = \frac{4q(N-N')}{B} + \frac{8(1-q)(N-N')}{B} = 4(2-q)\frac{N-N'}{B}.$$

The next component is the total execution time for all instructions, in number of cycles. Instructions can be executed in parallel at the execution rate of E instructions per cycle. Thus, the total execution time is given by

$$T^{PC}_{execute} = \frac{N}{E}.$$

Another component is the total time required for instructions to store their result into the memory. Every instruction in the program produces a single result which is stored into the memory in a two-phase memory cycle (each phase uses a different direction on the bus). Therefore, the total number of cycles is

$$T^{PC}_{res-store} = \frac{4N}{B}.$$

However, we also need to take into account the fact that some instructions may find one or both of their operands in a CPU register; we still assume that all instructions also store their result into the memory. We have assumed above that the CPU space devoted to the implementation of registers in the PC-driven CPU is identical to the space used by the ERU-SRAM and SRAM* units in the D^2-CPU. This space is c words. Therefore, the PC-driven CPU comprises c general-purpose registers (GPRs), where each register contains a single word. We assume that an instruction always stores its result into a GPR, and it is kept there for other instructions to use for the largest possible number of consecutive instruction cycles. Also, the instructions in the program uniformly utilize the registers. The expected maximum number of instruction cycles for which a result will occupy a given GPR is

$$a = \frac{cT^{PC}_{execute}}{N} = \frac{c}{E}$$

where the numerator represents the cumulative number of registers available for the entire program execution and the denominator represents the total number of results produced by instructions. This number of cycles corresponds to having $a/(2-q)$ complete instructions that use these registers at any time (if x and y are the numbers of one- and

two-operand instructions, respectively, at any time, then $x+2y=a$ and $x/y=q/(1-q)$). Therefore, we have to subtract from the total operand fetch time the value of

$$T^{PC}_{oper-fetch-reduce}=\begin{cases}4\dfrac{a}{2-q}\dfrac{N-N'}{B}, & if \quad t>\dfrac{a}{2-q}\\ T^{PC}_{oper-fetch-memory}, & if \quad t\le\dfrac{a}{2-q}\end{cases}.$$

Therefore, the actual value of the expected total operand fetch time is given by

$$T^{PC}_{oper-fetch}=T^{PC}_{oper-fetch-memory}-T^{PC}_{oper-fetch-reduce}.$$

Thus, the expected total turnaround time for the program is given by

$$T^{PC}=T^{PC}_{instr-fetch}+T^{PC}_{oper-fetch}+T^{PC}_{execute}+T^{PC}_{res-store}.$$

4.5.2. Turnaround Time on the D^2-CPU

The first instructions to arrive at the ERU are the N' zero-operand instructions. This phase requires time equal to N'/B cycles because each such instruction occupies a single word and all transfers are unidirectional (from the hardware manager to the ERU). These instructions then consume N'/E cycles to execute, and t copies are made of every instruction's result for token receiving instructions. The expected maximum number of instructions that can be transferred in any other single cycle from the hardware manager to the ERU is

$$\Delta=\frac{B}{3-q}$$

where the bandwidth B of the channel is divided by the expected average length of an instruction in words (from the ensemble of one- and two-operand $N-N'$ instructions); there exist $q(N-N')$ and $(1-q)(N-N')$ one- and two-operand instructions that consist of two and three words, respectively.

During the execution of the first N' (zero-operand) instructions, $\varepsilon=DN'/E$ new instructions enter the ERU (let N' be a multiple of E). Since our ultimate objective here is to compare the performance of the two designs under the assumption of optimality for program runs, all of these e instructions will receive their tokens from the former instructions and will then be ready to execute; we generally assume that the space for the ERU-SRAM and SRAM*

is collectively optimized for the highest performance. We assume that $B=c$ and $D=E$, so that units do not overflow. Assuming that the e new instructions are good representatives of the instruction mix in the program, they collectively consume $(2-q)e$ of the produced tokens; this number is found from $qe+2(1-q)e$. Thus, the total number of cycles consumed in step 1 is

$$T_1^{D2} = \frac{N'}{B} + \frac{N'}{E}.$$

In the next step (i.e., step 2), all N' results produced earlier are also transmitted outside of the ERU and consume N'/B cycles, and the latter e instructions are executed in the PU and consume e/E cycles. In fact, not all N' results may have to be transmitted, but we ignore this fact here for the sake of simplicity. Therefore, the number of cycles consumed in this step is

$$T_2^{D2} = \max\{\frac{N'}{B}, \frac{e}{E}\}.$$

Let us now assume that $N'=E$, for the sake of simplicity. If $N>>N'$, this assumption has negligible effect on the result. In each remaining step, D new instructions enter the ERU and consume $(2-q)D$ tokens produced in the preceding step. All D results are also transmitted outside of the ERU. The total number of tokens produced in the program by all instructions will be Nt, while $(2-q)(N-N')$ will be consumed by instructions that utilize the ERU; the remaining tokens, we assume, are used to change the state of the machine and/or communicate with peripheral devices. In practice, an average value of t is between 1 and 3. The total number of such steps is $p=é(N-N')/Dù-1$. To summarize, in each remaining step the following operations take place:

- D tokens from the preceding execution leave the ERU, consuming 1 cycle.
- D instructions are executed utilizing $(2-q)D$ tokens, and consume 1 (or D/E) cycle.
- At the same time, D new instructions enter the ERU and consume 1 cycle.

The instruction execution overlaps the transfer of new instructions, and therefore two cycles are consumed in step i, where $i=3, 4, 5, ..., p+2$. In the final step, D tokens leave the ERU consuming a single cycle.

Thus, the total turnaround time for the D^2-CPU is

$$T^{D2} = T_1^{D2} + T_2^{D2} + 2p + 1 = T_1^{D2} + T_2^{D2} + 2|-\frac{N-N'}{\Delta}-| - 1.$$

4.5.3. Comparative Analysis of Turnaround Time

Let us now carry out a comparative analysis of the two CPU designs for reasonable values of the parameters appearing earlier. We assume, as earlier, that $N'=E$ and $B=c$ for the values of the parameters involved, and that the capacity c is identical for both designs. It is also reasonable to always fit in the space of capacity c words a number

of instructions equal to the value of E, for smooth program execution on the D^2-CPU. If x and y represent the numbers of one- and two-operand instructions, respectively, resident in the ERU-SRAM at any given time (we assume no need for an SRAM*), then $2x+3y=c$, $x/y=q/(1-q)$, and $x+y=E$. We then obtain $c=(3-q)E$.

Figures 4 and 5 show relevant results for the speedup achieved by the D^2-CPU when compared to the PC-driven design; this speedup is given as the ratio of the execution times. The speedup in all of the corresponding cases is greater than two. It becomes obvious that our D^2-CPU design results in much better performance at a much lower cost. In reality, the performance improvement should be even larger than that shown in Figures 4 and 5 because: (a) the capacity c in the ERU of the D^2-CPU should be larger than the capacity in the PC-driven CPU because the former design has lower hardware complexity; and (b) the D^2-CPU can always take much better advantage of higher degrees of parallelism in application algorithms.

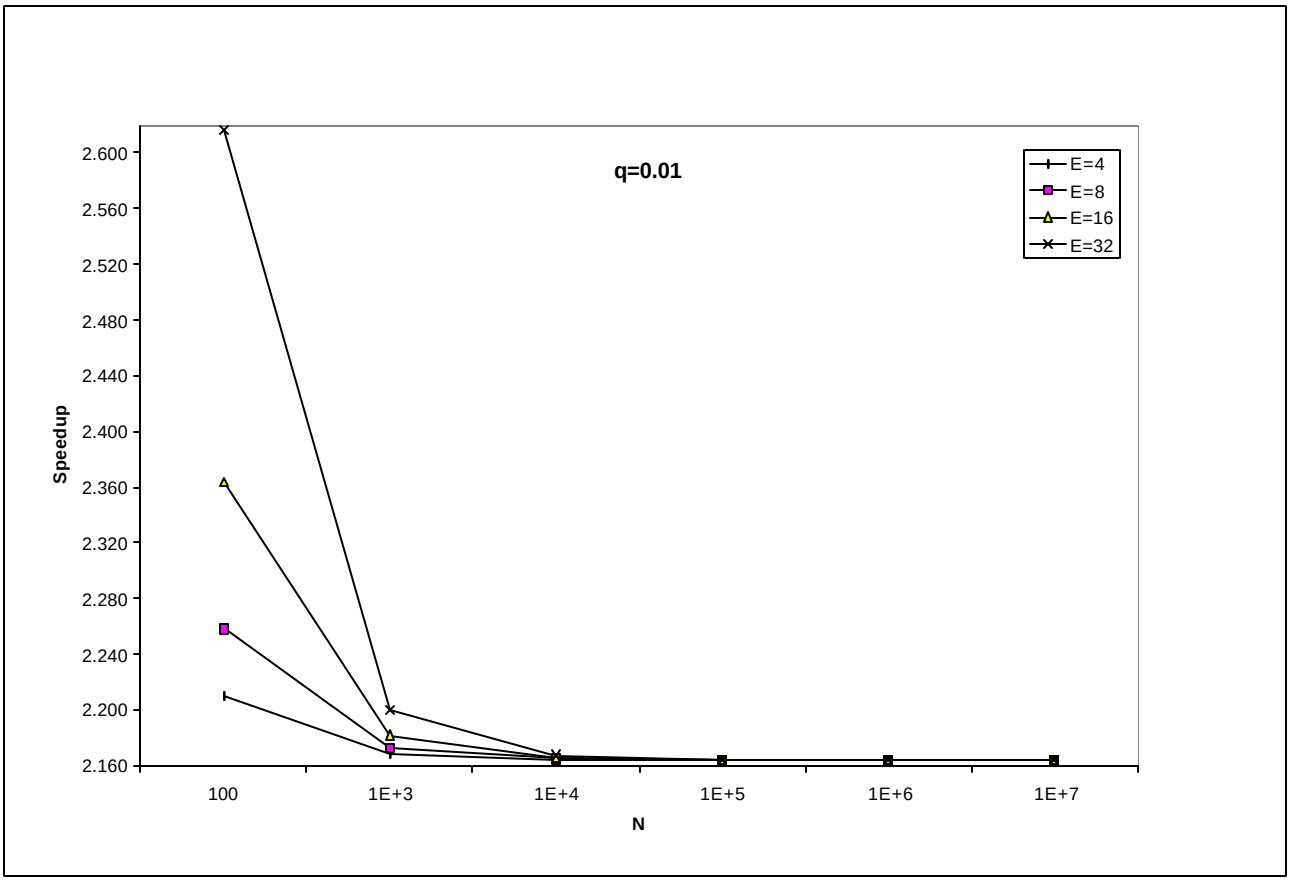

Figure 4. The speedup for *q=0.01.*

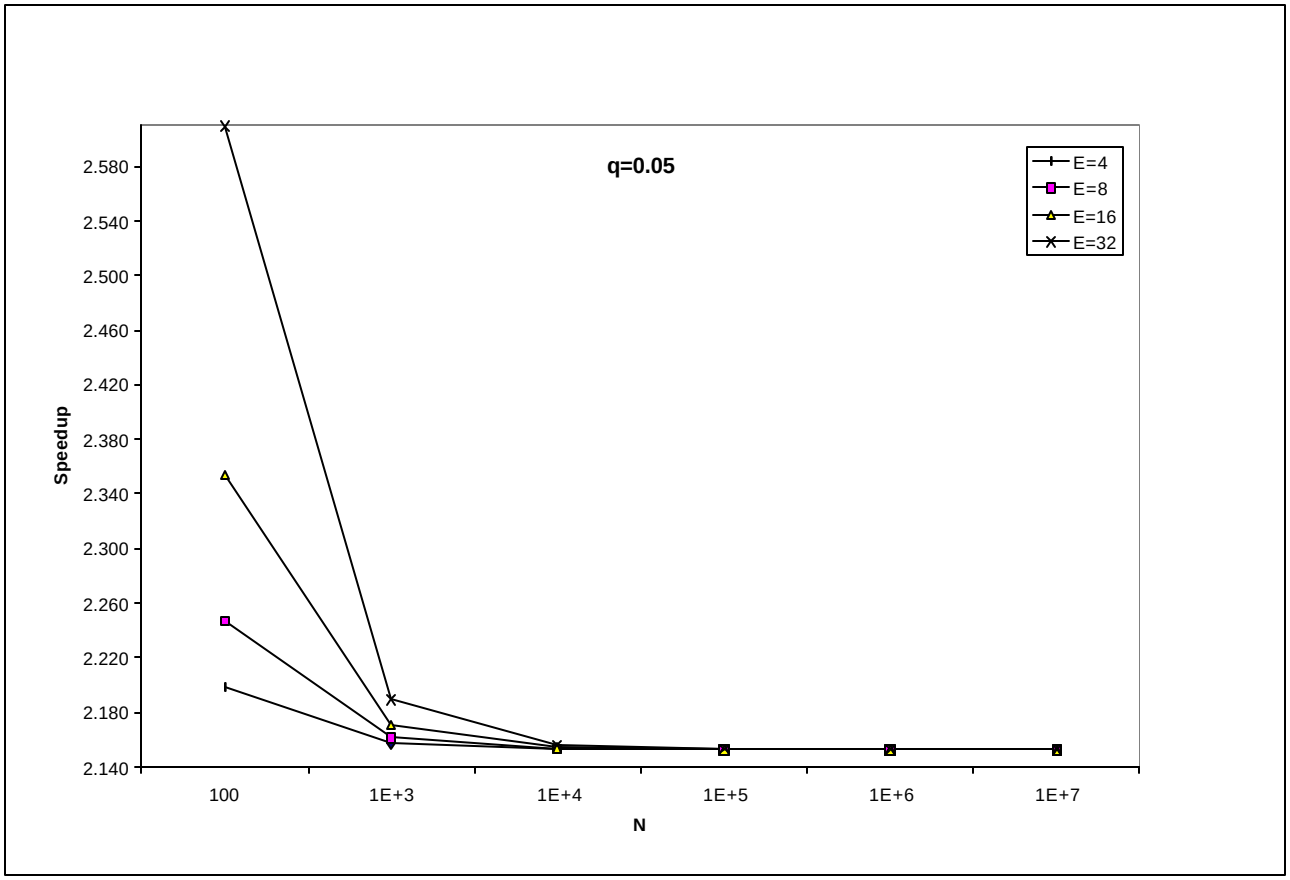

Figure 5. The speedup for *q=0.05*.

5. VHDL IMPLEMENTATION

5.1. PIM and Memory

We present here an implementation in VHDL of the PIM module that is used in the D^2-CPU. The PIM is made up of many modules that act as an interface between the ERU and the program memory (DRAM and DSRAM), or control some applications. The PIM has its own internal memory that it uses as cache and hash tables. A block diagram of the PIM is shown in Figure 6. Instructions are received through an external bus called the PIM I/O bus. The messages (PIM instructions) are stored inside a FIFO until retrieved via the PU-PIM. The counter is used as a clock divide-down timer. Timed operations are useful for programming memory cleanup and deallocation. These actions will be explained later. The PIM has a couple of internal memories that it uses for managing the main memory. The PIM must be able to maintain all user instructions currently in the system, that are to be executed by the dataflow processor. Every instruction is given a virtual ID so that it can be identified. The FIFO is a small memory queue of size 32 words X 32 bits. Messages stored inside of the FIFO are stored in different formats and sizes. The different sizes correspond to the possibility of a message being larger than one instruction segment of 32

bits. The only similarity among all of the messages is that the first 4 bits at the start of each message are used to identify the message. Normally, the FIFO output bus resides on the system data bus. Therefore, when it is inactive, its outputs are tri-stated. Even though the FIFO tri-states its outputs, it does not have to worry about any other module driving the bus. The performing of the read operation allows the FIFO to take its outputs out of tri-state into the position to output valid data from within the memory. Fortunately, this design has separated the bus from the FIFO to the PIM Processing Unit (PU) and the rest of the internal data bus. This tri-state condition is only used in case this same block must be reused in another location. All memories whose outputs are connected via a system bus must have the ability to take themselves off the bus by forcing their output into a tri-stated position.

The counter is a synchronous module with loading capability. The loading capability allows it to start counting from a value other than zero. This allows the processing unit to alter the amount of time it takes for the counter to reach its terminal count. The operation of the counter is quite simple. The reset and load lines are active low. The reset is asynchronous. In other words, regardless of the state of the clock, if the reset line is asserted, the output lines will reset. If the load line is asserted and a rising edge of a clock is detected, an internal count sets its starting count value to the load value. If the load or reset lines are not asserted, and a rising edge of a clock is detected, the output value will increment by one.

.

The size of the DRAM was kept small due to the need for multiple PIM/memory modules in the whole processor. The size of the memory is also restricted because we do not want to use up all of the processor logic within an FPGA or ASIC. Memory, unfortunately, takes up a lot of space when synthesized for a programmable device. Therefore, extreme care must be taken. Since the processor is a 16-bit device, the memory used is 16-bits wide. The depth of the memory was limited to 1Kbytes. The DRAM was designed to be a dual-port memory. The size of the DSRAM associated with each PIM is 256 X 16. There is no need for this device to be as large as the DRAM. The DSRAM modules are used more like caches. Each memory block contained within this module contains instructions that are waiting for the results of instructions that are currently in the ERU. In VHDL, the design of the DRAM and DSRAM memories is very similar. The difference would have to be when porting this to an ASIC or FPGA dye. The DSRAMs would have to be run at a quicker speed than the DRAMs in order to act more like caches. The DSRAM contains memory blocks just like the DRAM does. Each block contains 5 16-bit words in an effort to keep the structures uniform. However, since 256 is not divisible by 5, there will be invalid words contained within these memory devices as well. In this case, 5-word blocks will give us a total of 51 DSRAM blocks, with one word left over. This word is the NULL word (address 0x5000 – first location in DSRAM). The DSRAM is capable of holding two types of blocks, instruction blocks and pointer blocks. An instruction block contains an actual instruction that was copied from the DRAM. The main difference is that at least one of its operands has been filled with dependent information from another instruction that was executed in the ERU. Therefore, the instruction will be ready to execute after it receives a result from an instruction currently residing in the ERU. The other block is a pointer block. This is a pointer from DSRAM into DRAM. This is used when an

instruction in the DRAM is currently awaiting an instruction to be executed in the ERU but is still dependent upon another instruction to be executed in the future.

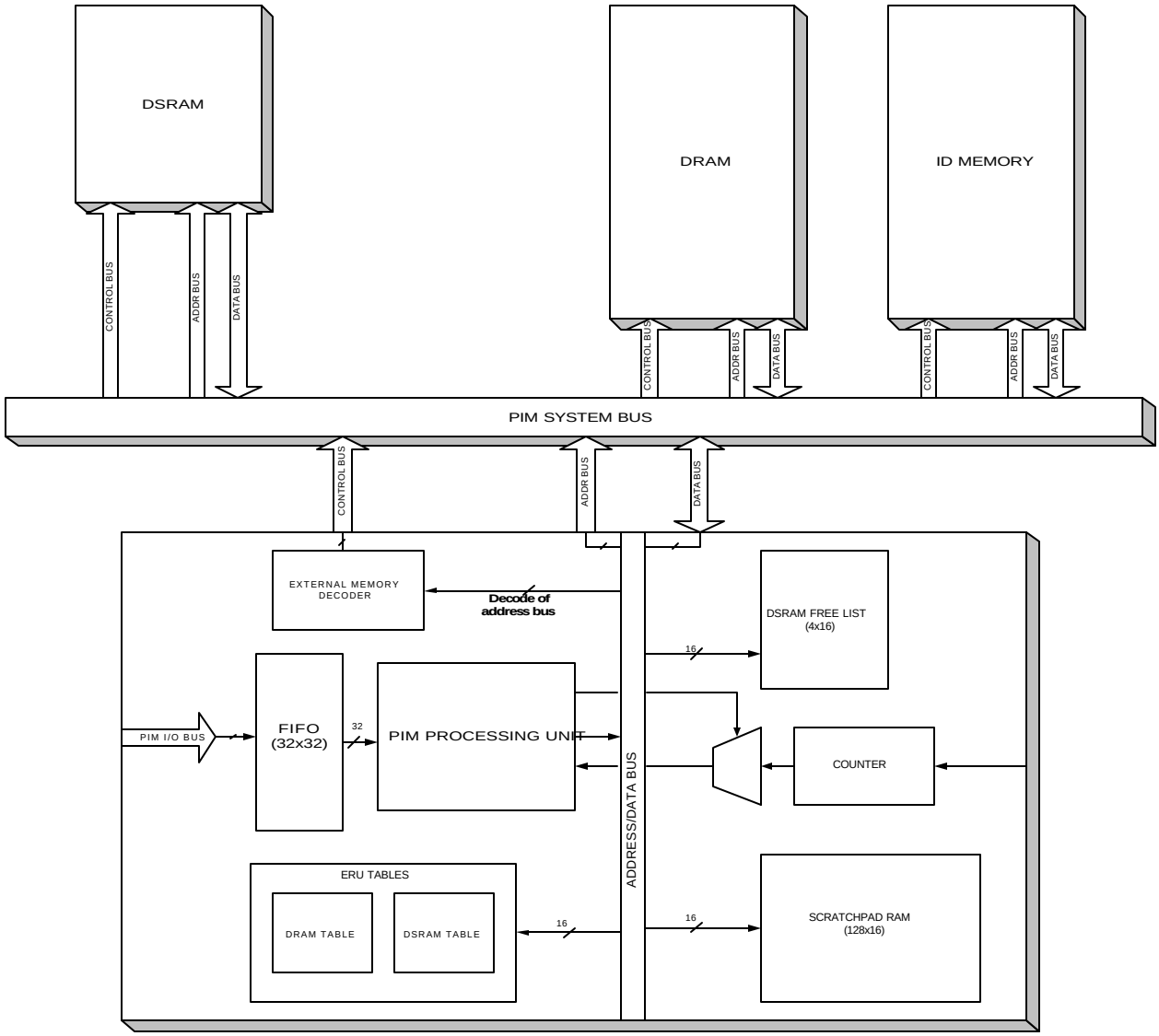

Figure 6. Block diagram of the PIM

The Scratchpad RAM is used as a temporary buffer for internal processing within the PIM and is also used in garbage collection. Every time an instruction is no longer needed (due to a loop ending or no more execution in the existing program), the instruction must be set to be deleted. Since the PIM performs scheduled garbage collection, it must save each instruction that must be deleted onto a list and then delete every instruction on the list at the given time (dictated by the counter). The Scratchpad RAM exists at the address range 0x0000 – 0x001F (64 bytes). The first location always contains the number of instructions that are on the "To-be-deleted" list. If a loop within the

main program ends, the ERU notifies the PIM that it is no longer needed. At a later time, the garbage timer will signal the PIM to empty its list and it will then go out to memory, reset the corresponding locations, and mark the memory deallocated.

5.2. PIM Instructions

The PIM has its own instruction set; this set is separate from the instruction set that is used for programs for the dataflow processor. Each instruction is generated externally and arrives at the PIM on the PIM I/O Bus. All instructions are stored in the FIFO to be read and translated at a later time. There are six instructions in our implementation. Table 2 gives an overview of these instructions. The following subsections describe the instructions.

Table 2. PIM Instruction Set.

Mnemonic	Instruction	Opcode	# 32-bit words	Memory Access Req (Y/N)
DTOK	Dissolve Token	0x0	1	Yes
IERU	Instr. To ERU	0x1	TBD	Yes
IRES	Instr. Result	0x2	2	Yes
GARB	Garbage Collect	0x3	1	No
OFLW	Overflow Instr.	0x4	1	No
LOAD	Load Instr	0x5	≥ 2	Yes

5.2.1 Dissolve Token (DTOK)

This instruction specifies to the PIM module that a loop has ended and the specified instructions are no longer needed. Therefore, the corresponding instructions are to be deleted. The PIM will receive a virtual ID (IAD) of an instruction that needs to be deleted. However, the PIM will not delete any traces of this instruction until the counter alerts the PIM. For instance, the PIM may be in the middle of executing another instruction or performing some memory access. Also, one loop will most likely have more than one instruction in it. So it would make more sense to have all of the instructions of a loop sent to the PIM around the same time (different bursts of the DTOK instruction). Therefore, when the time to do garbage collection arrives, then the PIM can traverse through the list and delete each instruction in succession. All that would be needed for reminding the PIM to do garbage collection

on a particular instruction would be an address ID and context ID. The ERU need not know where each instruction actually is. That is the PIM's job. So, we set aside part of our generic memory (the Scratchpad RAM) and use some of the locations for a *To-be-deleted* list. The main memory will be changed to delete the instruction and the ERU tables will have to be changed as well.

Thus, the actual operation of this instruction is rather simple. Upon detection and translation of this instruction, the PIM looks at the first location in the Scratchpad RAM (location 0x0000). This location holds the amount of instructions in the list. The next location (location 0x0001) holds the head of the list. This instruction adds an instruction identified by address ID and context ID to the list. It will add the 16-bit value to address *length* (retrieved by address 0x0000) +1 and then increment the length. The entire list is deleted during garbage collection. Since this is all that is needed for this instruction, the address ID and context ID will suffice for proper execution. Figure 7 shows the format of this instruction.

OPCODE = 0x0	N/A		
Address ID		Context ID	N/A

Figure 7. DTOK instruction format.

5.2.2. Instruction To ERU (IERU)

This instruction specifies to the PIM module that an instruction has gone to the ERU for execution. Before this happens, the PIM must know which instruction it is, so it can move all result-recipient instructions to the correct location. Instructions that are ready to be executed should reside somewhere that is easily and quickly accessible for faster execution speeds. Therefore, the instructions that are in a position for execution are stored in the fastest of memories that do not reside in the ERU, namely the DSRAM. In order to accomplish this, the PIM must manage all instructions and prepare them for movement to the ERU by keeping track of whether they are in a position to be moved for execution. Many instructions are created without any dependencies and can therefore be initially moved to the DSRAM. However, many instructions depend on other instruction's results. Those dependent instructions must wait in the DRAM for other instructions to execute. An instruction can exist in four states of dependency. The first state is for instructions that do *not* depend on any other instructions. These are the ones that can move to the DSRAM immediately. The second and third states are for instructions that still require operand #1 or operand #2 of the instruction. The fourth state is for instructions that still need both operands. These states can be examined in the DRAM by looking at the operand flags. See Table 3 for a further explanation of these states.

Table 3. Dependency state table.

State #	OPFL #1	OPFL #2	Explanation
1	0	0	The instruction has no dependencies and should be moved to the DSRAM immediately.
2	1	0	Has a dependency to fill operand #1 and should move to the DSRAM once its former instruction is in the ERU.
3	0	1	Has a dependency to fill operand #2 and should move to the DSRAM once its former instruction is in the ERU
4	1	1	Has a dependency to fill both operands. Once the 1st dependency is about to be filled, a pointer should be created in the DSRAM. When the 2nd operand is to be filled, the entire instruction should be copied over.

These states are important in determining what kind of information to put into the DRAM. If an instruction is waiting to fill both operands, then moving the instruction to the DSRAM immediately might not make much sense since it still needs another operand. Therefore, in this case the PIM will create a pointer block. A pointer to the DRAM location will be placed in the DSRAM in lieu of the entire instruction. When the next instruction that produces a needed result is moved to the ERU for execution, then the pointer block can be transformed into an instruction block. The design should update the flags in the DRAM when performing the IERU task. For example, if an instruction in the DRAM is waiting for two operands and then receives a result, the pointer block must be generated in the DSRAM and the operand flags must be changed accordingly to reflect the new dependency state of the instruction. The instruction uses two address IDs. The first ID is the virtual address ID of the instruction that has gone to the ERU for execution. At the present time, this ID is not used for anything inside the PIM. It is used as merely a reference, or for simulation testing. The other address ID is of the instruction that will have to be moved to the DSRAM (or pointer generated to it). See Figure 8 for the format of this instruction.

<table>
<tr><td>OPCODE = 0x1</td><td>N/A</td><td colspan="3">Address ID of instruction going to ERU</td></tr>
<tr><td colspan="3">Address ID of dependent instruction</td><td>Context ID</td><td>N/A</td></tr>
</table>

Figure 8. IERU instruction format.

5.2.3. Instruction Result (IRES)

This instruction informs the PIM of the result of an instruction. It will allow the PIM to fill the field of an instruction that is waiting for a particular result. The instruction only gives a result to an instruction waiting in the DRAM or DSRAM for a valid result. The format of this instruction is simple. However, since we need to include the actual 16-bit result, more than one FIFO read is necessary. Figure 9 shows the format of this instruction.

OPCODE = 0x2	N/A		
Address ID		Context ID	N/A
Result			
N/A			

Figure 9. IRES instruction format.

5.2.4. Garbage Collection (GARB)

This instruction will change the time duration needed for timed garbage collection. Periodically, a garbage collection pulse arrives via the garbage counter to alert the PIM processing unit of a garbage collection event. The mode section of the instruction will change the period of which this pulse arrives. The format of this instruction is shown in Figure 10.

OPCODE = 0x3	MODE	N/A
N/A		

Figure 10. GARB instruction format.

The number of clock cycles needed grows linearly with the number of instructions to be deallocated. Every execution of this instruction must do an instruction decode (1 clock cycle), read address 0x0000 which is the length (1 clock cycle), go to address location equal to the length (1 clock cycle), and delete the entry in the ERU table (2 clock cycles). See Figure 11 for a plot of this analysis.

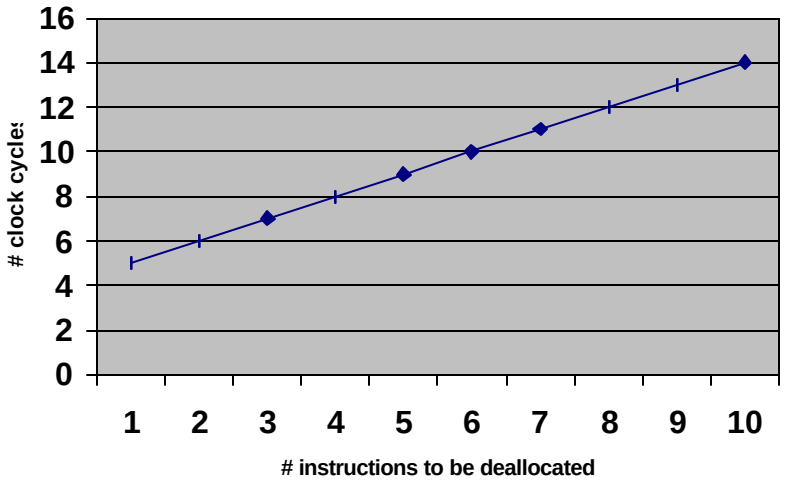

Figure 11. Number of clock cycles for GARB instruction execution .

5.2.5. Overflow Instruction (OVLW)

Once an instruction has left the PIM for the ERU, it can exist in one of three different memories, SRAM*, ERU-SRAM, or External Cache. A condition can exist where one of the memories can be filled. The instructions cannot be thrown away. If such a condition happens, they must be stored in the only other place that manages the memory, the PIM. This instruction sends the actual instruction back to the PIM for storage in the overflow list. The overflow list is separated into three separate sections. Each section is used to represent each respective memory in the ERU. The OVLW instruction contains origin bits that specify where the instruction is supposed to go in the ERU. See Table 4 for the definitions of these bits. The format of this instruction shows the origin bits and the instruction data. See the format of this instruction in Figure 12.

5.2.6. Load Program (LOAD)

This is the most complicated instruction. It requires multiple FIFO burst reads. That is the only way all of the necessary information can fit in a needed PIM instruction. The complicated execution comes up when the processor uses a multithreaded programming approach. Due to the complexity of the software design in this day and age, many programs (not all) require high throughput and efficiency, and benefit from executing on a processor using multiple threads. When this instruction is received and decoded, the PIM must attempt to load the instruction into the DRAM. This is where all the program memory is stored. However, since we want to avoid multiple copies of instructions that share the same opcode, operands and address ID, we must use a hashing technique to save thread IDs in accordance with its respective address ID. After reading the first 64-bits of data (2 FIFO reads), the PIM processing unit saves all needed instruction fields in local variables. An attempt must then be made to load the instruction into the memory. The rule used is to use the address ID as the DRAM index to store the instruction.

Since the DRAM location starts at address 0x6000, an address ID of 4 would be stored at location 0x6004. A problem arises when an attempt to load an instruction is made but there is already an instruction there. If there is an instruction there, it usually means that another thread is using the address ID. Therefore, the context ID must be used to store the new instruction of the same address ID in a table.

Table 4: Overflow origin bits definition.

Origin Bit #1 (MSB)	Origin Bit #2 (LSB)	Memory Origin
0	0	External Cache
0	1	ERU-SRAM
1	0	SRAM*
1	1	N/A

OPCODE = 0x4	ORIG	N/A
DATA		

Figure 12. OVLW instruction format.

If an instruction already exists at the location 0x6000 + address ID, then the instruction must be saved elsewhere. The same index into the DRAM in the ID Memory holds the address of a DRAM location that is the start of a hash table. This table for instructions that exist under the address ID in question but contain a different context ID. Finding this instruction may take a little longer but can be done. When the context IDs of the instructions don't match (when expected), the PIM must look to the ID Memory for the hash table location. If the address ID = 4, then the PIM would look in 0x6004 in the DRAM for the instruction. When it looks in the ID Memory, it will look at 0x7004. This location holds a DRAM address. This address is the starting address of a block that holds address locations of the actual instructions using the same address ID. The hashing key for each instruction in these blocks will be the context ID. The format of the instruction is shown in Figure 13.

OPCODE = 0x5	Program Instruct. OPCODE	Number of dependencies	
Address ID		Context ID	OPFL
Operand #1			
Operand #2			

Figure 13. LOAD instruction format.

To conclude, the VHDL implementation of the PIM indicates the viability of our approach.

6. CONCLUSIONS

A new architecture for CPU design was proposed based on the observation that current PC-driven architectures do not match well with the structure and requirements of application algorithms. In their effort to improve performance and hide the aforementioned mismatch, PC-driven CPUs apply significant amounts of redundant operations, introduce hardware resources of low effective utilization, and increase power consumption. In contrast, our data-driven uniprocessor design matches well with the structure of application algorithms, yields outstanding performance, minimizes the number of redundant operations, is distributed in nature for maximum parallelism, and results in lower power consumption. Our results prove the superiority of the resulting D^2-CPU uniprocessor design.

Further issues, among others, to be studied for our new design are related to: (a) the most effective implementation of the PU-PIM components, (b) appropriate configurations for the PU-PIM, EXT-CACHE, and hardware manager ensembles, (c) dynamic adaptation of the sizes for the ERU-SRAM and SRAM* units in the ERU for the best performance, (d) load balancing among the different DRAMs by appropriately assigning instructions, and (e) interconnection schemes for parallel computers utilizing our D^2-CPU.

Acknowledgment: The author would like to thank Mr. S.N. Kopec for the VHDL implementation of the PIM used in the proposed design.

REFERENCES

[1] J.A. Jacob and P. Chow, "Memory Interfacing and Instruction Specification for Reconfigurable Processors," *ACM Conf. FPGAs*, 1999, pp. 145-154.

[2] S.G. Ziavras, ``RH: A Versatile Family of Reduced Hypercube Interconnection Networks,'' *IEEE Trans. Paral. Distr. Syst*., Vol. 5, No. 11, Nov. 1994, pp. 1210-1220.

[3] D. Lopez, J. Llosa, M. Valero, and E. Ayguade, "Widening Resources: A Cost-Effective Technique for Aggressive ILP Architectures," *MICRO '98*, pp. 237-246.

[4] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *Intern. Symp. Comput. Arch.*, 1975, pp. 125-131.

[5] G.M. Papadopoulos and D.E. Culler, "Monsoon: An Explicit Token-Store Architecture," *Intern. Symp. Comput. Arch.*, 1990, pp. 82-91.

[6] W.-M. W. Hwu and Y.N. Patt, "HPSm, a High Performance Restricted Data Flow Architecture Having Minimal Functionality," *Intern. Symp. Comput. Arch.*, 1986.

[7] S.G. Ziavras et al., ``A Low-Complexity Parallel System for Gracious, Scalable Performance. Case Study for Near PetaFLOPS Computing," *6th Symp. Front. Mass. Paral. Comput., Spec. Sess. NSF/DARPA/NASA-funded New Millennium Computing Point Designs*, Annapolis, Maryland, Oct. 27-31, 1996, pp. 363-370.

[8] T. Golota and S.G. Ziavras, "A Universal, Dynamically Adaptable and Programmable Network Router for Parallel Computers," *VLSI Design*, Vol. 12, No. 1, Jan. 2001, pp. 25-52.

[9] M. Gokhale, B. Holmes, and K. Iobst, "Processing In Memory: the Terasys Massively Parallel PIM Array," *Computer*, April 1995, pp. 23-31.

[10] M.S. Schlansker and B.R. Rau, "EPIC: Explicitly Parallel Instruction Computing," *Computer*, Febr. 2000, pp. 37-45.

[11] X. Tang and G.R. Gao, "Automatically Partitioning Threads for Multithreaded Architectures," *Journ. Paral. Distr. Comput.*, Vol. 58, 1999, pp. 159-189.

[12] G. Alverson, S. Kahan, and R. Korry, "Processor Management in the Tera MTA System," *Techn. Rep., Tera Comput.*, Seattle, WA, 1995.

[13] R. Korry, C. McCann, and B. Smith, "Memory Management in the Tera MTA System," *Techn. Rep., Tera Comput.*, Seattle, WA, 1995.

[14] R.D. Blumofe et al., "Cilk: An Efficient Multithreaded Runtime System," ," *Journ. Paral. Distr. Comput.*, Vol. 37, 1996, pp. 55-69.

[15] R.D. Blumore and C.E. Leiserson, "Space-Efficient Scheduling of Multithreaded Computations," *SIAM Journ. Comput.*, Vol. 27, No. 1, Febr. 1998, pp. 202-229.

[16] J. Hennessy, "The Future of Systems Research," *Computer*, Aug. 1999, pp. 27-33.

[17] H.H.J. Hum et al., "A Design Study of the Earth Multiprocessor," *Int'l. Conf. Paral. Arch. Compil. Techn.*, 1995, pp. 59-68.

[18] D. Burger, J. Goodman, and A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors," *23rd Int'l. Symp. Comput. Arch.*, May 1996.

[19] J.A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," *10th Symp. Comput. Arch.*, 1983, pp. 140-150.

[20] M.J. Flynn, *Computer Architecture: Pipelined and Parallel Processor Design*, Jones and Bartlett Publ., 1995.

[21] Arvind and R.S. Nikhil, "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *IEEE Trans. Comput.*, Vol. 39, No. 3, Mar. 1990, pp. 300-318.

[22] H. Terada et al., “Design Philosophy of a Data-Driven Processor: Q-p,” *Journ. Inform. Proc.*, Vol. 10, No. 4, Mar. 1988, pp. 245-251.

[23] M. Chatterjee, S. Banerjee, and D.K. Pradhan, “Buffer Assignment Algorithms on Data Driven ASICs,” *IEEE Trans. Comput.*, Vol. 49, No. 1, Jan. 2000, pp. 16-32.

[24] H.T. Kung and M. Lam, “Wafer Scale Integration and Two Level Pipelined Implementation of Systolic Arrays,” *Journ. Paral. Distr. Comput.*, Vol. 1, No. 1, Sept. 1984, pp. 32-63.

[25] S. Ingersoll and S.G. Ziavras, “Intelligent Memories for Dataflow Computation and Emulation on Field-Programmable Gate Arrays," *Microprocessors Microsystems*, Vol. 26, No. 6, Aug. 2002, pp. 263-280.