ELSEVIER

Microprocessors and Microsystems 26 (2002) 263–280

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.com/locate/micpro

Dataflow computation with intelligent memories emulated on field-programmable gate arrays (FPGAs)

Segreen Ingersoll, Sotirios G. Ziavras*

Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102, USA

Received 24 September 2001; revised 10 February 2002; accepted 10 May 2002

Abstract

This paper presents a new design that implements the data-driven (i.e. dataflow) computation paradigm with intelligent memories. Also, a relevant prototype that employs FPGAs is presented for the support of intelligent memory structures. Instead of giving the CPU the privileged right to decide what instructions to fetch in each cycle (as is the case for control-flow CPUs), instructions in dataflow computers enter the execution unit on their own when they are ready to execute. This way, the application-knowledgeable algorithm, rather than the application-ignorant CPU, is in control. This approach could eventually result in outstanding performance and elimination of large numbers of redundant operations that plague current control-flow designs. Control-flow and dataflow machines are two extreme computation paradigms. In their pure form, the former machines follow an inherently sequential execution process while the latter are parallel in nature. The sequential nature of control-flow machines makes them relatively easy to implement compared to dataflow machines, which have to address a number of issues that are easily solved in the realm of the control-flow paradigm. Our dataflow design solves these issues at the intelligent memory level, separating the processor from dataflow maintenance tasks. It is shown that using intelligent memories with basic components similar to those of FPGAs produces a feasible approach. Expected improvements within the next few years in underlying intelligent memory and FPGA technologies will have the potential to make the effect of our approach even more dramatic.

Keywords: Dataflow computation; Field-programmable gate arrays; Prototyping; Intelligent memories

1. Introduction

Program counter (PC)-driven or control-flow CPUs are very widely used in all kinds of application environments. Current designs are the result of more than 25 years of evolution. Incremental performance improvements during this period have been primarily the result of ever higher transistor densities and larger chip sizes. Transistor densities and single-chip processor performance double approximately every 18 months (Moore's law). Impressed by these facts, CPU designers have rarely attempted to circumvent this evolutionary path. Reduced instruction set computers (RISC) CPUs have basically the same structure with complex instruction set computers (CISC) CPUs; they differ only in the complexity of individual components and instruction sets. Also, very large instruction word (VLIW) CPUs are similar to RISCs, while relying heavily on the compiler and their wide data buses for good performance. All basic ideas built into CPUs have been around for more than 20 years [7]. 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. For example, the first phase of the instruction fetch operation is required only because of the chosen computation model (i.e. PC-driven); 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

* Corresponding author. Tel.: +1-973-596-5651; fax: +1-973-596-5680.
E-mail address: ziavras@njit.edu (S.G. Ziavras).

0141-933/02/$ - see front matter
PII: S0141-9331(02)00038-8

or the attached cache. However, these fetch cycles also should be avoided, if possible.

Current high-end microprocessors implement wide instruction issue, out-of-order instruction execution, aggressive speculation and in-order retirement of instructions [14]. They generally implement on the fly only a small, dynamically changing window of dataflow execution. Under the pure *data-driven* (*dataflow*) *model/paradigm of computation*, an instruction is executed as soon as all its operands become available [2]. Instructions are not ordered and also carry with them their operands. Three steps implement a dataflow instruction: (a) *instruction issuance/firing*: it is the departure of the instruction for the execution unit just after all its operands become available to it. (b) *token propagation*: as soon as an instruction completes execution, it makes copies of its result for all other instructions, if any, that needs it. Different tokens that contain the same result are then forwarded to different instructions. (c) *Instruction dissolvement*: it is the destruction of the instruction just after it produces all tokens for other receiving instructions. Loop instructions can be treated differently, as discussed later.

The main *advantages* of dataflow computation are: (a) instructions are executed according to the natural flow of data propagated in the program. (b) Most often, there is a high degree of embedded parallelism in programs and, therefore, very high performance is possible. (c) It is free of any side effects (because of the natural flow of data that guides the execution of instructions). (d) It reduces the effect of memory-access latency because all operands are attached to their instructions. (e) 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). (f) 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 dataflow computation are: (a) increased communication (or memory-access) overhead because of explicit token passing. (b) Instructions are forced to use their data (in incoming tokens) when they arrive, even if they are not needed at that time. (c) 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. (d) The hardware for matching recipient instruction addresses in the memory with tokens may be complex and expensive.

We do not yet know how to implement the dataflow paradigm efficiently with current hardware design practices and silicon technologies. Ref. [9] showed that the instruction firing rule can be implemented through state-dependent instruction completion. They treated each 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 stored in the memory and allocated to an instruction just before it starts executing; it facilitates the storage of *tokens*. 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 dataflow computation is characterized by very significant computation and storage overheads.

Past implementations of dataflow have been primarily carried out on parallel computers [4], where dataflow is basically applied among instructions running on different processors; the latter processors are PC-driven. The majority of the designs have made many compromises because they are constrained into developing systems with commercial off-the-shelf (COTS) processors [1,9]. In contrast, a dataflow processor was introduced in Ref. [13] that utilizes a self-timed pipeline scheme to achieve distributed control. This design is based on the observation that dataflow 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 design also suffers from several constraints imposed by current design practices. Other related work appeared in Refs. [6,10,11]. Several dataflow architectures have been introduced for the design of high performance ASIC devices [3]. Also, several techniques have been developed for the implementation of ASICs in VLSI when the dataflow graphs of algorithms are given. However, these techniques employ straightforward one-to-one mapping of nodes from the graph onto distinct functional units in the chip.

Multithreading is now common in CPU designs; each program is partitioned into a collection of instructions, called threads. Instructions in a thread are issued according to the von Neumann (i.e. PC-driven) model of computation (i.e. they are run sequentially). Similarly to dataflow, instructions between threads are run based on data availability [6,12]. 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.

Efficient architecture for running threads (EARTH) is a multiprocessor that contains multithreaded nodes [8]. Each

node contains a COTS RISC processor (called execution unit (EU)) 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 EU chooses a thread to run from this queue. EU executes the thread to completion and then chooses the next one from the ready queue. EU's interface with the network and SU is implemented with an event queue that stores messages. SU manages the latter queue. The local memory shared by EU and SU in the local node is part of the global address space. A thread is activated when all its input data become available. 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 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 EU, not SU. The implementation of EU with a COTS processor implies that its communication with SU is made via loads and stores to special addresses. However, multithreading does not implement dataflow at the instruction level and for the entire program. Also, multithreading and prefetching significantly increase the memory bandwidth requirements.

The conceptual simplicity of dataflow computation strucks a chord in many computer designers, but the lack of methods to efficiently implement dataflow has been a great challenge. As discussed earlier, the problem of implementing dataflow has been tackled numerous times at the processor or multiprocessor levels. Dataflow computers in the past have been implemented by using a modified processor, often called the *processing element* (PE), which was composed of a processing unit along with memory to store partially active instructions and tags. The PE would also contain a matching unit for incoming tokens/tags and was assigned the task of sending and receiving tags, from/to other PEs. Most of the inactive instructions would reside in some memory outside the PE.

Dataflow implementation at the intelligent memory level has not yet been attempted. Each dataflow instruction in this memory should have its own logic unit that determines when the instruction is ready to execute. This approach should incur lesser overhead than previous methods. Our objective is to design and implement a proof-of-concept dataflow computer based on intelligent memories that can be prototyped with current FPGAs. Our prototype should demonstrate the viability of our approach while future FPGAs will have the potential to better match the requirements of our solution.

2. More dataflow computing challenges

It is difficult to implement on dataflow machines simple programming constructs that are taken for granted in von Neumann architectures, like conditionals, loops and modules (procedures and functions). Accommodating loops and conditionals requires nodes that implement controlled branching. For efficient implementation of loops, each iteration can be executed as a separate instance/copy of the reentrant subgraph (representing the loop code). This *code-copying* method requires facilities to create a new instance of a subgraph and to direct tokens to the appropriate instance. A potentially more efficient way to implement code-copying is to share the node descriptions between the different instances of a graph without confusing tokens that belong to separate instances. This is accomplished by attaching a *tag* to each token that identifies the instance of the node that it is directed to. These *tagged-token* architectures enable a node (instruction) if all its input arcs contain tokens with *identical tags*. This method increases concurrency but its implementation is not easy and involves considerable overhead. Problems with procedure calls are similar to those with re-entrancy. The methods described above can still be applied. In code-copying architectures, a copy of the called procedure is made. In tagged-token architectures, a new tag area is allocated for each procedure call so that each invocation executes in its own context. Nested procedure calls, recursion and co-routines can, therefore, be implemented without any additional problems. However, it is required to direct the output tokens of the procedure to the proper calling location.

Machines that handle re-entrancy by the lock or acknowledge method are called *static*. Those involving code-copying or tagged tokens are called *dynamic*. Static machines are much simpler to implement than dynamic machines but for most algorithms their effective concurrency is lower. The earliest design at MIT had a two-stage structure with heterogeneous functional units, where each enabling unit was dedicated to one node [2]. It was later extended into a series of machines differing in the way they handled re-entrancy and data structures. They ranged from the elementary Form I processor, which was static and could only handle elementary data, to the full-fledged Form IV processor, which had extensive structure facilities and

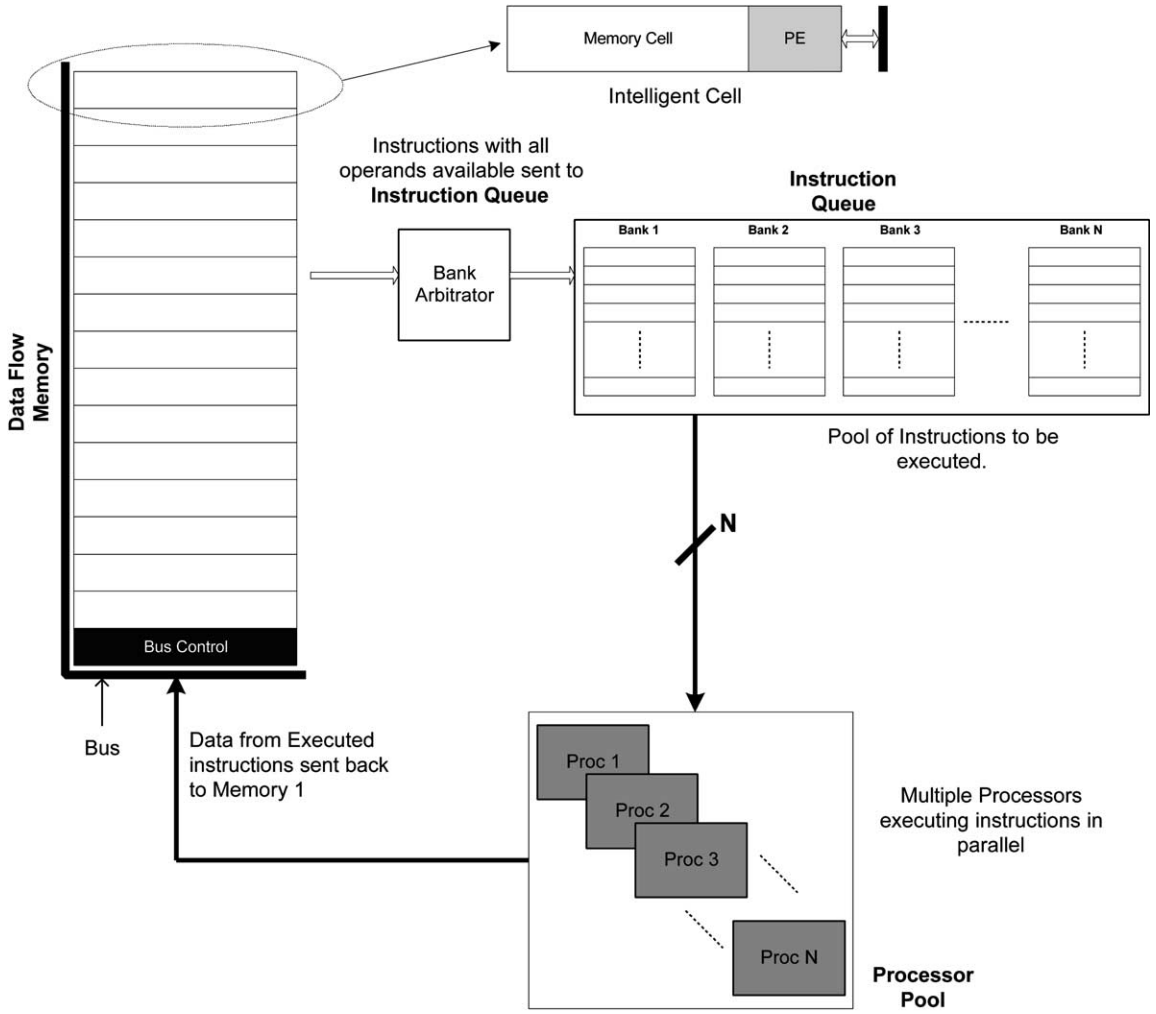

Fig. 1. The basic structure of our dataflow computer.

could copy subgraphs on demand. Sandia National Laboratories designed and implemented around 1990, the ϵpsilon static dataflow computer [5]. It was designed as a scalable multiprocessor architecture, consisting of ϵpsilon processors and structure memory units connected with a packet switched network. The whole design was implemented on a single board using COTS components.

3. Overview of our dataflow computer design

The basic structure of our dataflow computer is shown in Fig. 1. It has three major components: the instruction memory *data flow memory* (DFM), the intermediate buffer *instruction queue* (IQ) and the execution unit *processor pool* (PP). It also has minor components, such as the *bank arbitrator*, *bus* and *bus controller* in DFM.

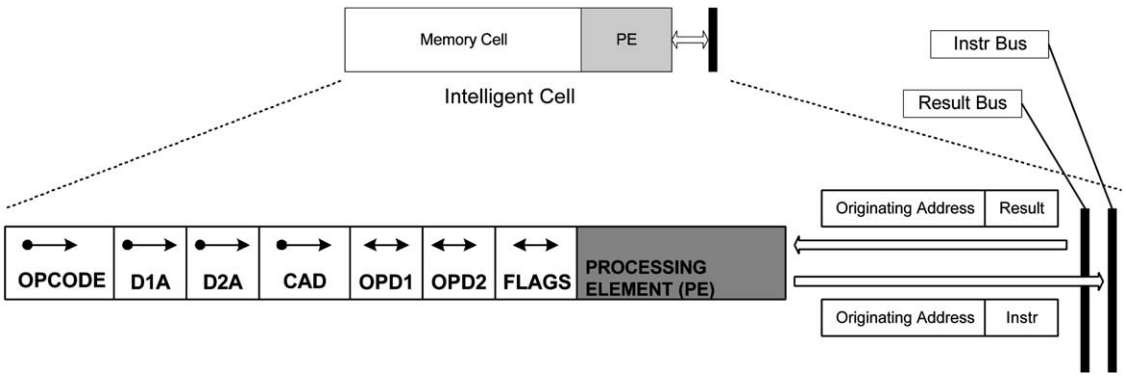

Fig. 2. Internal structure of an intelligent cell in DFM.

3.1. Data flow memory

DFM is the heart of our design. It is a special kind of memory, where each memory location is an *intelligent cell* consisting of the *instruction* (*memory cell*) and *processing element* (PE) parts. The instruction part is in turn divided into seven components as shown in Fig. 2, where dyadic operations are assumed: instruction opcode (OPCODE), source of the first operand (D1A), source of the second operand (D2A), source of the clause operand (CAD), operand to be obtained from the first source (OPD1), operand to be obtained from the second source (OPD2) and flags that control the behavior of the instruction (FLAGS). The little arrows within each cell in Fig. 2 denote the direction of information flow between the cell and the PE. The instruction bus and the result bus are also shown along with the format in which the PE communicates with them.

The result of an instruction that finishes executing is immediately broadcasted to all cells in DFM. Each *result packet* contains the result along with the address of the instruction in DFM that produced it. The PE in each cell is responsible for picking up broadcasted result packets sent by the PP. If the address in a broadcasted result packet is either equal to D1A or D2A for an instruction in DFM, then the result is written into OPD1 or OPD2, respectively, and appropriate flags are set. When both OPD1 and OPD2 are available (determined by examining the appropriate flags) to the dyadic instruction, the PE sends an *executable packet* (composed of the instruction opcode, operands and cell address of the sending PE) to the bank arbitrator. It is important to point out here that the bus on which the result packets are broadcasted is independent of the bus on which executable packets are relayed to the bank arbitrator. This is done to avoid congestion that would occur if only one bus was used. The CAD field is used to store the address of an instruction that sends the *clause*. A clause is a boolean value stored as a flag in the flags field; it acts as a permission for the instruction that needs it, i.e. an instruction will execute only if its clause bit is 1. The clause is useful in the construction and execution of conditional or looping program constructs, as discussed later. The *bus controller* (BC) in DFM controls the arbitration of the result packets sent by the PP to DFM on the *result bus*.

3.2. Instruction queue

IQ is an intermediate buffer between DFM and PP. It is made up of the *bank arbitrator* (BA) and multiple banks of memory. The number of banks in IQ is equal to the number of processors in PP. Each processor is assigned one bank exclusively. This assures that all processors can access the memory at the same time; it also keeps the initial design simple. Ideally, a crossbar switch should be implemented so that a processor could access any bank. No processor is busy or idles all the time during the execution; this is because BA makes sure that the number of executable instructions allotted to each memory bank is the same, using a round robin scheme.

3.3. Processor pool

PP is a pool of execution units. Each processor sequentially executes ready instructions (executables) from its assigned memory bank. The instructions are executed in no particular order because all instructions in the memory bank are waiting to be executed. When a processor receives an instruction, it also gets the *originating address* (OA) of that instruction. After execution, the produced result along with OA is sent to the *bus controller* of DFM in the form of a result packet; this packet is then broadcasted to all cells via the result bus. This design is suitable for small-scale dataflow computers, i.e. machines having up to about 32 processors in the PP. This restriction is foreseen due to two reasons. First, the number of memory banks in IQ will increase linearly with the number of processors, which for a large number of processors may be unrealistic. Secondly, having a large number of processors may increase the number of broadcasted messages causing congestion on the bus.

4. Field-programmable gate arrays

Fastest performance is achieved when a design is implemented directly on silicon, with dedicated/specialized logic for all units. Since the major objective of our prototype was to prove the viability of our intelligent memory based implementation of dataflow, the performance of the prototype was to be of secondary importance. Hence, a device was needed that could easily be reconfigured, if design flaws were detected. In addition, this device is required to have sufficient logic to be able to accommodate the different components and also implement intelligent memory. The ideal solution for prototyping the dataflow computer was to use field-programmable gate arrays (FPGAs) that have recently achieved tremendous technological advances.

FPGAs are user-programmable devices, which are now widely accepted as an excellent technology for implementing and prototyping moderately large digital circuits. They offer a cost-effective solution for prototyping because they have a fast turn around time (i.e. short design and development cycles). Since FPGAs can be reprogrammed an unlimited number of times, they can be used in innovative designs where hardware is changed dynamically and must be adapted to different user applications. Though dynamically changeable hardware is not a consideration in this project, this feature is particularly useful when prototyping, where the design is constantly being changed and updated. In terms of speed, most FPGAs are slower than complex programmable logic devices. However, rapid advances in FPGA technology are quickly closing the gap

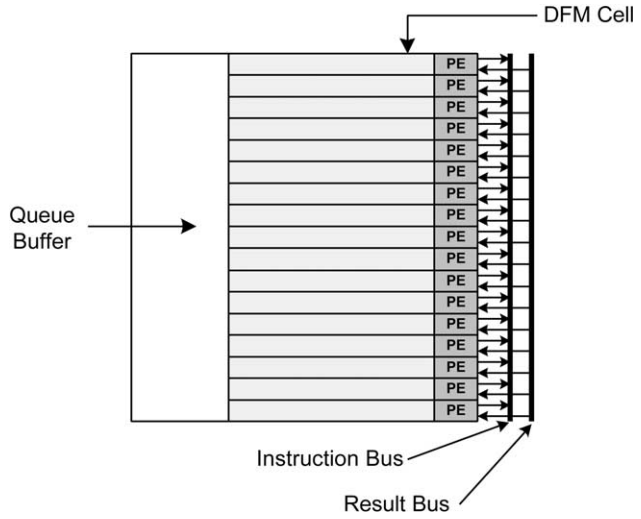

Fig. 3. Dataflow memory (DFM) structure.

on speed and denseness. One prominent disadvantage of FPGA technology is that circuit propagation delays are dependent on the performance of the design implementation tools used, which is not a major handicap for this project.

The internal architecture of an FPGA consists of several uncommitted logic blocks in which the design is to be encoded. These logic blocks consist of several universal gates that can be programmed to operate like multiplexers, decoders, registers, random access memory (RAM) and as many other digital logic devices. The logic blocks are connected through a set of programmable interconnects that implement buses and direct connections. FPGAs have elaborate clocking schemes and optimization methods (programs) can be used to use faster hardware with fewer logic blocks. Since logic blocks are independent, a single FPGA can be used to implement multiple units, all of which can work independently and in parallel (a key factor in the design of a dataflow computer).

The FPGAs used for our prototype are those made by Altera Corp. Three Altera devices were chosen to implement the three major components of our dataflow computer: DFM was implemented on a FLEX10KE, IQ and the bank arbitrator were implemented on an ACEX1K and PP was implemented on a MAX9000. The FLEX10KE was a good candidate for implementing DFM because each device provides up to 98,304 RAM bits that can be configured as dual-port memory. Also, each device contains sufficient logic, up to 200,000 gates depending on the device chosen, to implement the PEs in DFM. All these are connected together by a fast interconnect network that has predictable interconnect delays. The ACEX1K is a less powerful relative of the FLEX10KE. It has the same features as the FLEX10KE, except there is less of everything. The largest ACEX1K has 49,152 RAM bits and about 100,000 gates. Since the logic and memory requirements for IQ are less than those for DFM, this was a good choice. The MAX9000 is the smallest of the three devices used. It contains no memory bits, but has sufficient logic gates and flip-flops, up to 12,000 and 772, respectively, to implement PP.

A major factor in deciding to use Altera's FPGAs to develop the prototype was the availability of a free development kit from Altera called MAX + PLUS II BASELINE version 10. In addition, Altera has a university support program through which it is possible to get free manuals on how to use the software, a programming language reference and tutorial on Altera hardware description language (AHDL), and sample boards for hardware development. In conclusion, Altera FPGAs provided a low cost, highly configurable solution along with a simple environment to develop the dataflow prototype.

5. DFM description

For our prototype, DFM is divided into the *queue buffer* (QB), *DFM cells* and *PEs*, as shown in Fig. 3. The internal structure of a DFM cell is shown in Fig. 4. Each DFM memory cell is broken up into four sections, *cell sections* CS1 through CS4. Each cell section holds a piece of the instruction to be executed. Controlling each CS is a *logic unit* (LU). The LUs, LU1 through LU4, manage CS1 through CS4, respectively, and constitute a PE. The bits in CSs are grouped, so that minimum communication is needed between different LUs controlling different CSs. By having an LU controlling only one CS, work on CSs is done

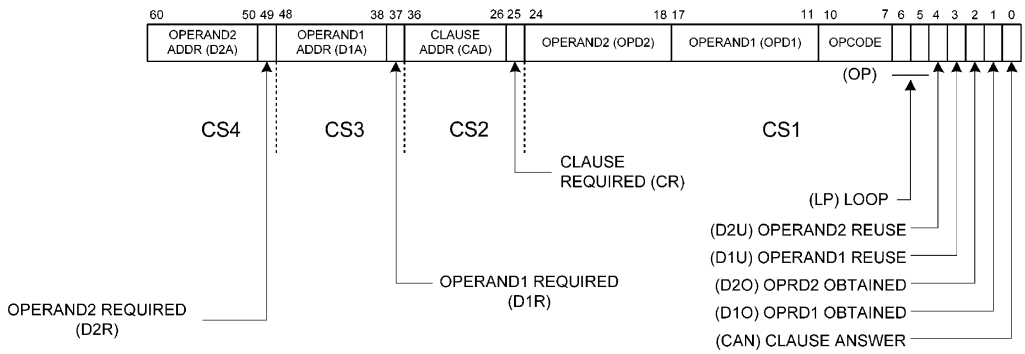

Fig. 4. DFM cell structure.

independently and in parallel, avoiding potential waits that would arise if an LU controlled more than one CS across the cell.

5.1. Cell section 1

Cell section 1 (CS1) is made up of nine fields. Each field is described below.

- *Operand1* (OPD1). It holds the first operand, which is received from the instruction whose address matches the value in D1A (source of operand1). OPD1 is copied into CS1 after LU3 picks up a result packet and makes a match between the *originating address* (OA) in the packet and D1A.
- *Operand2* (OPD2). It holds the second operand, which is received from the instruction whose address matches the value in D2A (source of operand2). OPD2 is copied into CS1 after LU4 picks up a result packet and makes a match between OA in the packet and D2A.
- *Opcode* (OP). The opcode of the instruction to be executed.
- *Operand1Obtained* (D1O). A one-bit flag set by LU1 when LU1 receives OPD1 from LU3. This bit may be set at compile or run time. If it is set at compile time, then the operand is already available (immediate value).
- *OPRD2 Obtained* (D2O). A one-bit flag set by LU1 when LU1 receives OPD2 from LU4. This bit may be set at compile or run time. If it is set at compile time, then the operand is already available (immediate value).
- *Clause Answer* (CAN). A one-bit flag which holds the boolean answer that LU2 picked up from the packet it received from the instruction whose OA matches the value in CAD. This bit is used during execution of conditional and loop constructs. It is set to 1 at compile if a clause is not required by the instruction.
- *Operand1 Reuse* (D1U) and *Operand2 Reuse* (D2U). Fields D1U, D2U and LP (described below) are used to implement loops. Often instructions inside a loop require values from outside the loop, but instructions from outside the loop execute and transmit their values only once (i.e. an instruction is fired only once). Hence, the instruction inside the loop will receive that value only once and will normally fire only once. To overcome this problem, the reuse bit is used. Setting this bit at compile time allows LU1 to realize that the received value has to be reused, and will not change the D1O/D2O bit of the firing instruction whose D1U/D2U bit(s) is (are) set.
- *Loop* (LP). Loops are difficult to manage in dataflow machines, but they also are the most commonly used constructs in programming. Usually the value of the loop-controlling variable is checked before a loop is entered (e.g. 'while' and 'for' loops). Thus, the first time the value of the variable is obtained from outside the loop and every subsequent time it is obtained from inside the loop. Hence, we need a primitive to obtain the same variable from two different sources. Also, dataflow machines are runaway machines; firing one instruction subsequently fires many instructions in different parts of the code. In particular, when a dataflow computer executes a loop it is highly probable that the machine may be simultaneously executing different iterations of the same loop. This would not be a problem if there were no dependencies between consecutive iterations, but would be a disaster if there were any. Some way to control the execution of the loop is needed. To accomplish this control, the 2-bit flag *LOOP* (LP) is used to implement loop constructs. It can only be set at compile time with a value from 0 to 3. Only instructions that enclose a loop have their LP values greater than 0. All other instructions inside and outside the loop have LP = 0. The LP values of 1 through 3 are not used to distinguish between different types of loop constructs but to control how a loop executes, as follows:
- LP = 1 is used to initiate a loop. The instruction which has its LP set to 1 is the *SPecial instruction SP*. It is not actually an instruction at all (it is never sent for execution). For a loop, there would be an SP instruction whose LP value would be 1; its D1A field would contain the OA from where the value of the loop-controlling variable is obtained the first time and D2A would contain the OA from where the value is obtained all other times. The 'firing' of SP does not require the instruction to proceed to PP.
- An SP instruction is always followed by a *conditional instruction* in the program graph. The latter fires only if SP has received a value. The result of this conditional instruction is sent out as a clause to all instructions inside the loop. This instruction has LP = 2, because if it is treated as a regular instruction (LP = 0), upon execution its CAN bit will be reset instantly and this instruction will not execute again (until its CAN bit is set). LP = 2 instructs LU1 to leave the CAN bit intact after instruction firing.
- An instruction with LP = 3 is used along with SP to avoid situations where the runaway effect will cause a problem due to data dependencies between consecutive loop iterations. With LP = 3, a process similar to when LP = 1 goes into action, except that instead of firing SP after the value of the variable is obtained from either D1A or D2A, SP will fire only when the value is obtained from both D1A and D2A. D1A is always the address of the SP instruction with LP = 1 (beginning of loop) and D2A is always the address of the instruction which must be executed before the next iteration starts.

5.2. Cell section 2

Cell section 2 (CS2) is made up of two fields.

- *Clause Address* (CAD). It stores the address of the instruction that sends a clause. This instruction will

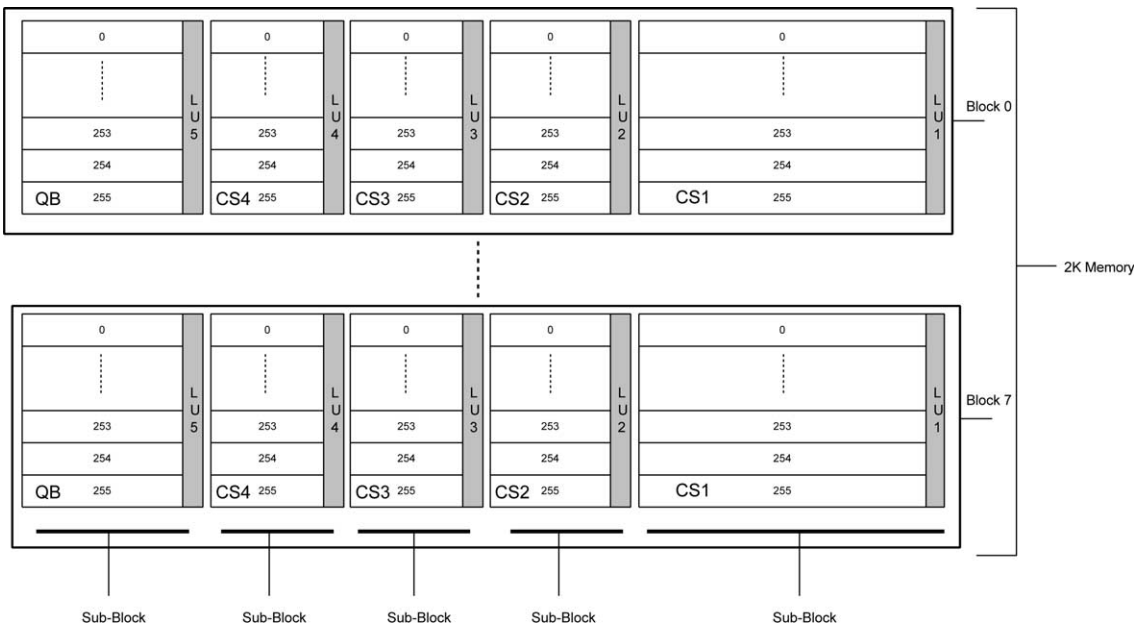

Fig. 5. Implementation of DFM cells.

execute only if its CAN field contains 1 (claused received). If the instruction does not need a clause, then its CAN field is set to 1 at compile time and the value in the CAD field is irrelevant.

- *Clause Required* (CR). A one-bit flag set to 1 at compile time if the instruction requires a clause.

5.3. Cell section 3

Cell section 3 (CS3) is made up of two fields.

- *Operand1 Address* (D1A). It holds the address of the instruction from which the value of the first operand is

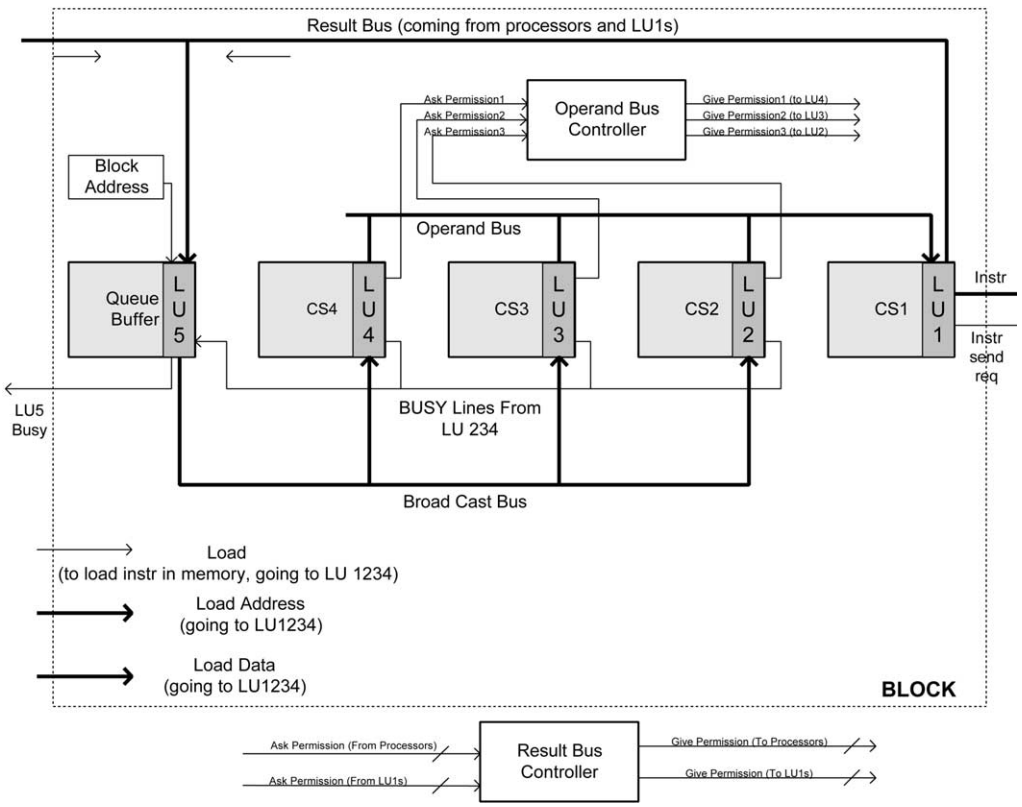

Fig. 6. Internal structure of a DFM block.

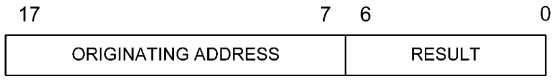

Fig. 7. Result arrival format.

expected. If the first operand is an immediate value, then the address in this field is irrelevant.

- *Operand1 Required* (D1R). A one-bit flag set at compile time to let LU3 know if this instruction needs to receive the first operand. When OPD1 is an immediate value, this bit is set to 0.

Cell Section 4 is similar to CS3. The fields are D2A (Operand2 Address) and D2R (Operand2 Required).

5.4. Queue buffer

It is a repository where result packets coming on the result bus are deposited if the results are coming faster than they can be absorbed. As LUs become free, QB sends queued result packets in the order that they were received.

6. Detailed implementation

This section covers the actual implementation of the different components in the dataflow computer, and the design decisions that were made that did not completely conform to the design presented earlier. The reasons for these decisions are also discussed. Very detailed diagrams are included in Appendix A.

6.1. DFM implementation

The structure of a single DFM cell was shown in Fig. 4. It is 61 bits long and all addresses are 11 bits long, thus giving a total addressable memory of 2Kwords. All operands are seven bits wide. This small operand size is not considered a limitation because our main purpose is to prove the viability of our design. While implementing DFM, some design decisions were made so that it would fit in a single Altera device. One in particular was the implementation of the PE, i.e. units LU1–LU4. Ideally, each intelligent cell should have its own set of LU1–LU4, but that required more logic than would fit in a single Altera device. Thus, each LU was assigned to a group of cells called a *sub-block*, as shown in Fig. 5. Each LU controls a sub-block of 256 cells; LU1 controls a sub-block of 256 CS1s, LU2 controls a sub-block of 256 CS2s, and so on. The group of five sub-blocks formed by LU1–LU5 is called a *block*. There are eight blocks in this design, thus giving a total space of 2K memory words. Notice that a new LU, namely LU5, was introduced. LU5 controls QB in a block and is not part of the PE, which is made up of LU1–LU4; it takes no part in the manipulation of instructions.

6.1.1. Block

Fig. 6 shows the internal structure of a block. Instructions are loaded into each block using the LOAD, LOAD ADDR and LOAD DATA lines. Each block waits for a result to arrive before an instruction is fired. The result comes on the result bus; it may arrive from any of the processors in PP, any of the LU1s or an external source (user input, interrupt, etc.). To put a result on the result bus, the sender should get permission from the result bus controller. This controller manages the information flow into LU5 of each block, which in turn determines if the result packet should be queued or sent straight through to LU2, LU3 and LU4. Once an LU5 gets a result off the result bus, it broadcasts this information to the LU2, LU3 and LU4 units (to be collectively referred to in the future as LU234) in its respective block, if LU234 are not busy. If LU234 are busy, then the incoming result is queued into QB to be dispatched later when LU234 are free.

Upon reception of the result by LU234, each LU checks if any of the instructions contained in its cell section requires the result. If an LU finds an instruction that needs the result, it dispatches the result to LU1. The Operand bus controller controls the dispatching of results by LU234 to LU1. On receiving a result, LU1 dispatches the *executable* for execution if all its operands are available. LU1 uses the Instr_Send_Req line to confirm if it is safe to send an instruction for execution. The three most significant bits of an 11-bit address determine the block number. Address 255 was used to initiate execution by sending a positive clause on the result bus. Finally, the block was broken up into two parts because it could not be fit in a single Altera device (Appendix A).

6.1.2. Logic units

An LU1 keeps tabs on which instructions have already been fired and those that need to refire (loop instructions). LU234 process anything that is sent by LU5 in the format shown in Fig. 7. LU1 is also in charge of dispatching instructions for firing when all operands/clauses become available. It sends them on the instruction bus in the format shown in Fig. 8. LU1 processes operands and clauses sent by LU234 in the format of Fig. 9.

6.2. Instruction queue

The IQ is composed of the bank arbitrator (BA) and memory banks. A memory bank within IQ is shown in Fig.

Fig. 8. Instruction firing dispatch format.

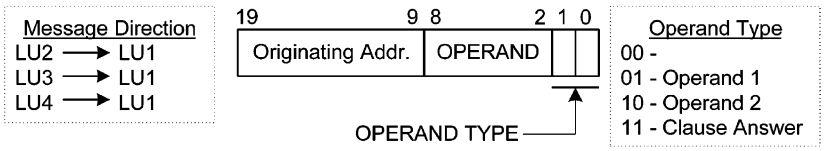

Fig. 9. Format of data sent from LU234 to LU1 (operand type 00 is not used).

10. Each bank has two pointers, the *new instruction load pointer* (NILP) and *next instruction execute pointer* (NIEP) to implement a circular queue. NILP holds the address of the next available location where the bank arbitrator can insert an executable coming from DFM. NIEP holds the address of the next instruction that is ready to execute. The processor associated with the bank uses this address to fetch the next instruction. For the sake of simplicity, we implemented two processors. *Memory controller* (MC) in each bank uses two signals to communicate with its processor. Instr_RD is used by a bank to indicate to its processor that a new instruction(s) is (are) available for execution. When an instruction has finished executing, the processor notifies its MC via Instr_done, upon which MC increments NIEP, if NIEP $\neq$ NILP. Each memory bank is implemented using a *dual port memory*, allowing BA to send instructions to it while its processor can read from it. BA puts instructions into each bank using a round robin scheme. Each bank has 16 words in its circular queue. The number of memory words in each bank is not an optimized number, but an arbitrary number was chosen to build the prototype.

6.3. Processor pool

Each processor is associated with only one memory bank in the memory pool, and vice versa. The processors are more like execution units than general-purpose processors. When a processor is ready for execution, it looks for the Instr_RD signal from its corresponding memory bank. When it sees the signal go high, it reads the instruction from its memory bank and executes it. An instruction is always executed with OPD1 on the left side and OPD2 on the right (e.g. OPD1–OPD2, OPD1/OPD2, check if OPD1 > OPD2, etc.). The processor then sends a request to the result bus controller in DFM, asking permission to send the result to DFM. Upon acknowledgement, the requesting processor sends the result over to DFM in the format shown in Fig. 7 and sends an Instr_done signal to its bank, which in turn readies a new instruction for execution. Since all dataflow maintenance work is relegated to DFM, either processor is blindly executing instructions and sending signals, independently of the other. The instructions that the processor can currently execute are set to the bare minimum, just enough to make this dataflow machine work. The instructions and their corresponding opcodes are shown in Table 1.

7. Programming the dataflow computer

All instructions are commonly used instructions, except for SP and LK. This computer has six compare instructions, which is unlike most machines that usually have four. The two compare instructions not usually found in other computers are CGE and CLE, because these two instructions can normally be made up by combining other compare instructions (e.g. CGT and CEQ for CGE). However, this is not suitable in our architecture, e.g. when a combination of instructions are used to build a conditional instruction to control a loop, each instruction within the loop should be capable of receiving two clauses and fire if either one is true. This ability to receive two clauses is not provided in this implementation because it would make the DFM design

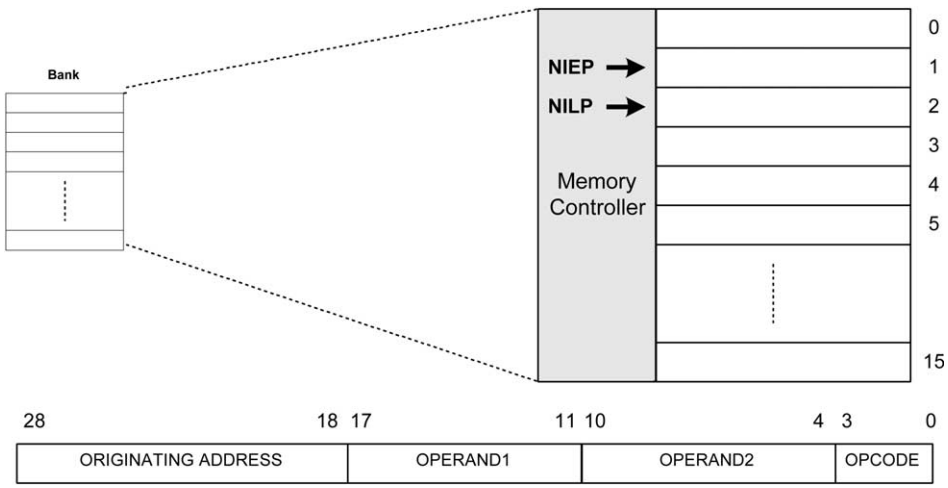

Fig. 10. (a) A memory bank in IQ; (b) contents of a cell.

Table 1
Instruction set

INSTR OPCODE	INSTR OPCODE
ADD 0001 → Addition	CGT 1010 → Compare if Greater than
SUB 0010 → Subtraction	CLT 1011 → Compare if Less than
MUL 0011 → Multiplication	CGE 1100 → Compare if Greater or Equal to
DIV 0100 → Division	CLE 1101 → Compare if Less or Equal
CEQ 1000 → Compare if Equal	SP 0000 → SPecial
CNE 1001 → Compare if Not Equal	LK 0000 → LocK

more complex. SP is not really an instruction because it is never sent to the PP for execution. Though its opcode is 0000, it is irrelevant. SP is a directive for LU1 as explained earlier, instructing it to forward all values it receives for SP to other cells in DFM. What constitutes an SP instruction is the value of LP in an instruction; if the value of LP is 1, LU1 realizes that this is a directive and treats it accordingly. Currently, SP is used exclusively as part of a loop construct. When LP = 1, LU1 sends the operands it receives from either LU3 or LU4 out onto the operand bus, if the CAN bit of SP is 1. SP always has its D1R and D2R bits set, but unlike other instructions where LU1 will wait for both operands to arrive before dispatching them, LU1 dispatches the arriving operands immediately.

LK also is a directive that has its LP set to 3 (opcode irrelevant). While setting LP to 1 makes SP transmit any operand it receives (from either LU3 or LU4) onto the Operand bus, setting LP to 3 causes LU1 to transmit OPD1 (the value it receives from LU3), but only after LK has received both operands and the CAN bit is set to one. LK is particularly useful in loop constructs where before incrementing the control variable of a loop it may be necessary to check if all instructions within the loop have been executed. This is to avoid race conditions, which would start a new iteration before the completion of an earlier one. Concurrently executing multiple iterations of a loop is often required in parallel computing, but it is disastrous if there are dependencies between iterations.

Before we present an example program, it is important to learn to interpret the nodes of dataflow graphs. Fig. 11 shows three primitives that are used to represent nodes. Fig. 11(d) is a variation of Fig. 11(a). Arrows going in and out of a node are divided into four categories based on *arc type*, *arrowhead*, *label type* and *tail*. Table 2 shows the classification of arrows. The *INSTR* node is used to implement every kind of instruction this dataflow computer offers, except for SP and LK which have their own

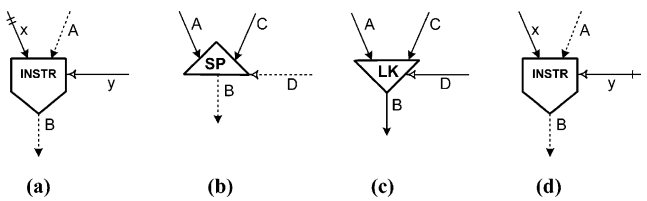

Fig. 11. Primitives of program dataflow graphs.

representations. The two solid headed arrows coming into the top of each node represent the two operands required by the node. The operand arrow on the left is OPD1 and the operand arrow on the right is OPD2. The hollow headed arrow on the side of a node is the clause needed by the node and the solid headed arrow at the bottom of each node is the result pushed out by the node.

This style of representing a dataflow program is new. Though the nodes bear some similarity to earlier representations, their interpretation is completely different. One may use only solid lines for the arcs to show less detail, but all other components are absolutely important to show reusability of an arc; the two different arrowheads to show the difference between values and clauses, and the single tail to indicate an LP value of 2. Using dashed and solid lines shows what state a program is in, when it is loaded into memory, giving a clue of the operands that are available to this program and those that it is waiting for. Labeling the arcs with memory addresses does have the drawback of not being able to draw the flow graph until the program has already been written. Eliminating all memory address labels overcomes this problem but reduces the amount of information the graph conveys. Some output arcs of conditionals get converted into clause arcs for some nodes; this is allowed because the result from the execution of a conditional is either 0 or 1, thus it can be a clause arc for a node.

Table 3 shows a program segment that contains conditional statements, reentrant code and reusable variables. The code is presented in two formats, in high-level pseudo-code and pseudo-assembly code. The equivalent program in dataflow machine language is presented in Table 4. A dataflow language and compiler were not developed for our machine, so the code in Table 4 is not how one would write the program for this computer to be compiled but is actually the code resident in DFM. The flow graph for this code is shown in Fig. 12.

The two loops execute concurrently and the values being used by them are not interdependent. In a von Neumann machine, it would be quite difficult to concurrently execute the two loops because the variable x is needed by both loops; while one (FOR) uses variable x, the other (WHILE) modifies x. The FOR loop must finish executing before the WHILE loop starts. However, in the dataflow machine each loop is sent its own copy of x, thus allowing the loops to

Table 2
Classification of arrows associated with dataflow graph nodes

Arrowhead	Solid Head	Indicates an incoming operand or an outgoing result
	Hollow Head	Indicates an incoming clause
Arc Type	Solid Arc	Value along that arc is not yet available
	Dotted Arc	Value along that arc is available
Label Type	Lower Case	Value associated with arc (immediate value)
	Upper Case	Address associated with arc (incoming or outgoing)
Tail	Single Tail	LP value of the instruction is 2
	Double Tail	Reuse value on the arc

execute concurrently. The FOR loop is bounded by an LK instruction because there are dependencies between consecutive iterations of the loop. Such dependencies do not exist between consecutive iterations of the WHILE loop, hence no LK instruction is needed to bind the loop. Writing a program for the dataflow machine is tricky and tedious because the programmer has to be aware of the data dependencies in the program. This issue can, however, be overcome using an intelligent compiler.

8. Conclusions and remarks

A new hardware design was presented for dataflow computers. The incorporation of intelligent memories is at the core of the proposed design, so that processing can be separated from all other dataflow related operations. Recent advances in FPGAs were taken advantage of to prototype our design. The logic capabilities of FPGAs allow the implementation of intelligent memories for a proof-of-concept approach. Our results prove that our approach is valid and further technological improvements in FPGAs and/or intelligent memories may make our design even more attractive in a few years' time.

Higher primitives such as code-copying and tagged tokens were not implemented here, hence procedure invocation and indirect memory addressing are not currently possible. However, it is possible to add these features into the current architecture by making some modifications in the design. There is one feature of this architecture that eliminates a problem faced by past dataflow designs, and that is the problem of data fan out. In past designs, the data fan out of an instruction was limited, usually to two or three. So, programs to be run on such machines had to be written

Table 3
Sample code

Calculation of functions $F1$ and $F2$ in pseudo-high level language		Equivalent pseudo-assembly language code	
•		Get (Y)	
Get (y)	← Get y from user	MOV X, Y	(X) ← (Y)
$x = y^*9 - 15$	← *Reuse variable* (x)	MUL X, #9	X ← X ∗ 9
If $x \neq 0$ then	← *Conditional*	SUB X, #15	
$F1 = 0$		CMP X, #0	
$F2 = 1$		BEQ DN2	
For $i = 1$ to x	← *Loop*	MOV R1,#0	
$F1 = F1 + x^*(i + 2)$		MOV R2,#1	
End For		MOV R3,#0	
While $x > 1$	← *Loop*	LP1: CMP R2, X	For loop: $R1$ is $F1$, $R2$ is i, $R3$ is temp
$F2 = F2^*x$		BGT DN1	
$x = x - 1$		ADD R3,R2	
Endwhile		ADD R3,#2	
Endif		MUL R3, X	
•		ADD R1,R3	
•		CLR R3	
•		ADD R2,#1	
		BRA LP1	
		DN1: MOV R2,#1	
		LP2: CMP X, #1	While loop: $R2$ is $F2$
		BEQ DN2	
		BLT DN2	
		MUL R2,X	
		SUB X,#1	
		BRA LP2	
		DN2: •	

Table 4
Equivalent dataflow machine code

ADDRESS	**CS4**		**CS3**		**CS2**		**CS1**									
	OPERAND2 ADDR. (D2A)	OPERAND2 REQD. (D2R)	OPERAND1 ADDR. (D1A)	OPERAND1 REQD. (D1R)	CLAUSE ADDR. (CAD)	CLAUSE REQUIRED (CR)	OPERAND2 (OPD2)	OPERAND1 (OPD1)	OPCODE (OP)	LOOP (LP)	OPERAND2 REUSE (D2U)	OPERAND1 REUSE (DIU)	OPRD2 OBTAINED (D2O)	OPRD1 OBTAINED (D1O)	CLAUSE ANSWER (CAN)	
•																
•																
A									GET (X)							
B	0	0	A	1	0	0	9	0	MUL	0	0	0	1	0	1	Calculate X
C	0	0	B	1	0	0	15	0	SUB	0	0	0	1	0	1	
D	0	0	C	1	0	0	0	0	CNE	0	0	0	1	0	1	If stmt.
E	0	0	0	0	D	1	1	0	ADD	0	0	0	1	1	0	Initialize *i*
F	L	1	E	1	D	1	0	0	SP	1	0	0	0	0	0	For Loop
G	C	1	F	1	D	1	0	0	CLE	2	1	0	0	0	0	Start
H	0	0	F	1	G	1	2	0	ADD	0	1	0	1	0	0	Calculate *F*1
I	C	1	H	1	G	1	0	0	MUL	0	1	0	0	0	0	
J	C	1	I	1	G	1	0	0	ADD	0	0	0	1	0	0	Lock instr
K	J	1	F	1	G	1	0	0	LK	3	0	0	0	0	0	
L	J	0	K	1	G	1	1	0	ADD	0	1	0	1	0	0	Increment *i*
M	0	1	C	1	D	1	0	0	SP	1	0	0	0	0	0	While loop
N	P	0	M	1	D	1	1	0	CGT	2	1	0	1	0	0	Start
O	0	1	M	1	N	1	1	0	MUL	0	0	0	1	0	0	Calculate *F*2
P	0	0	M	1	N	1	1	0	SUB	0	1	0	1	0	0	

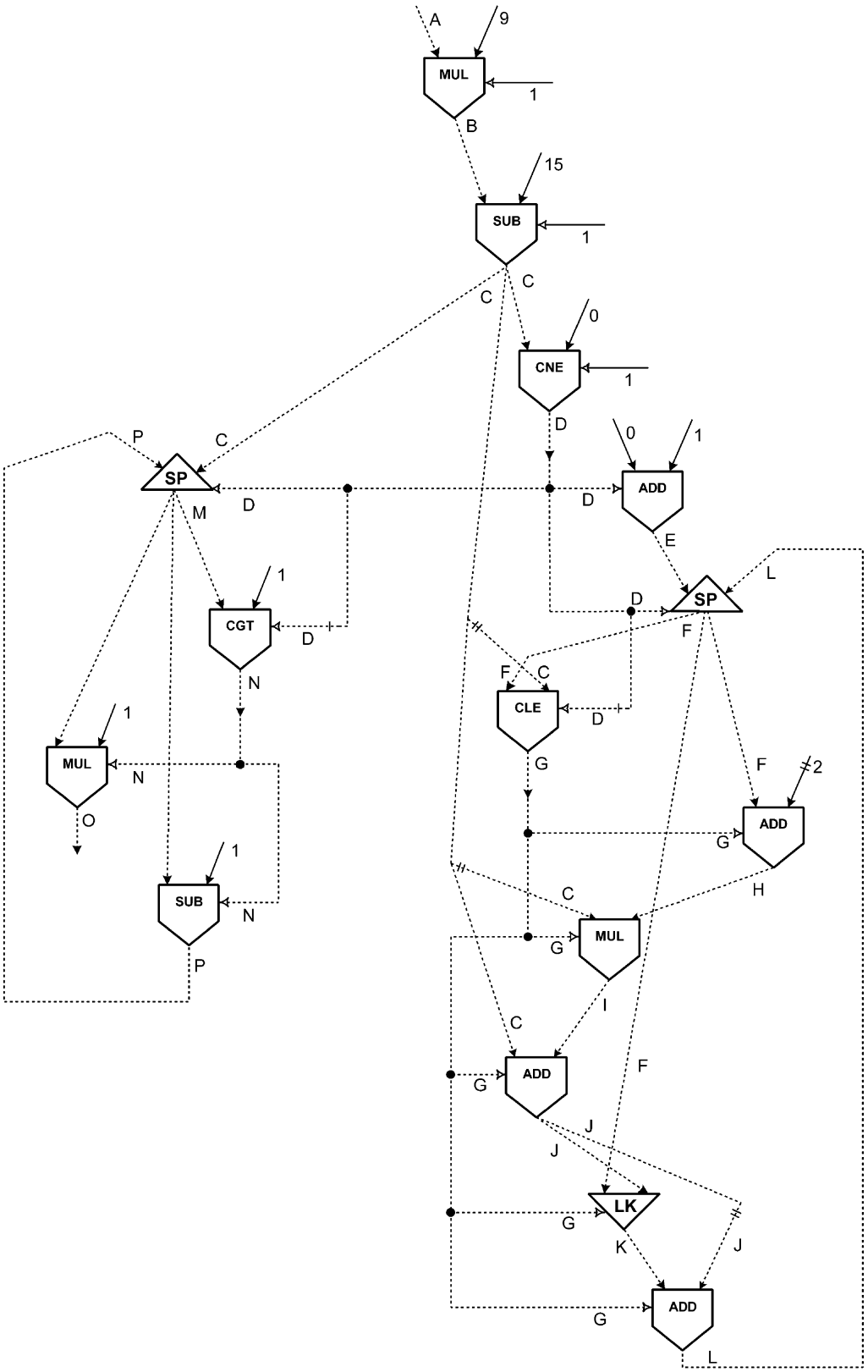

Fig. 12. Flow graph of the code presented in Table 3.

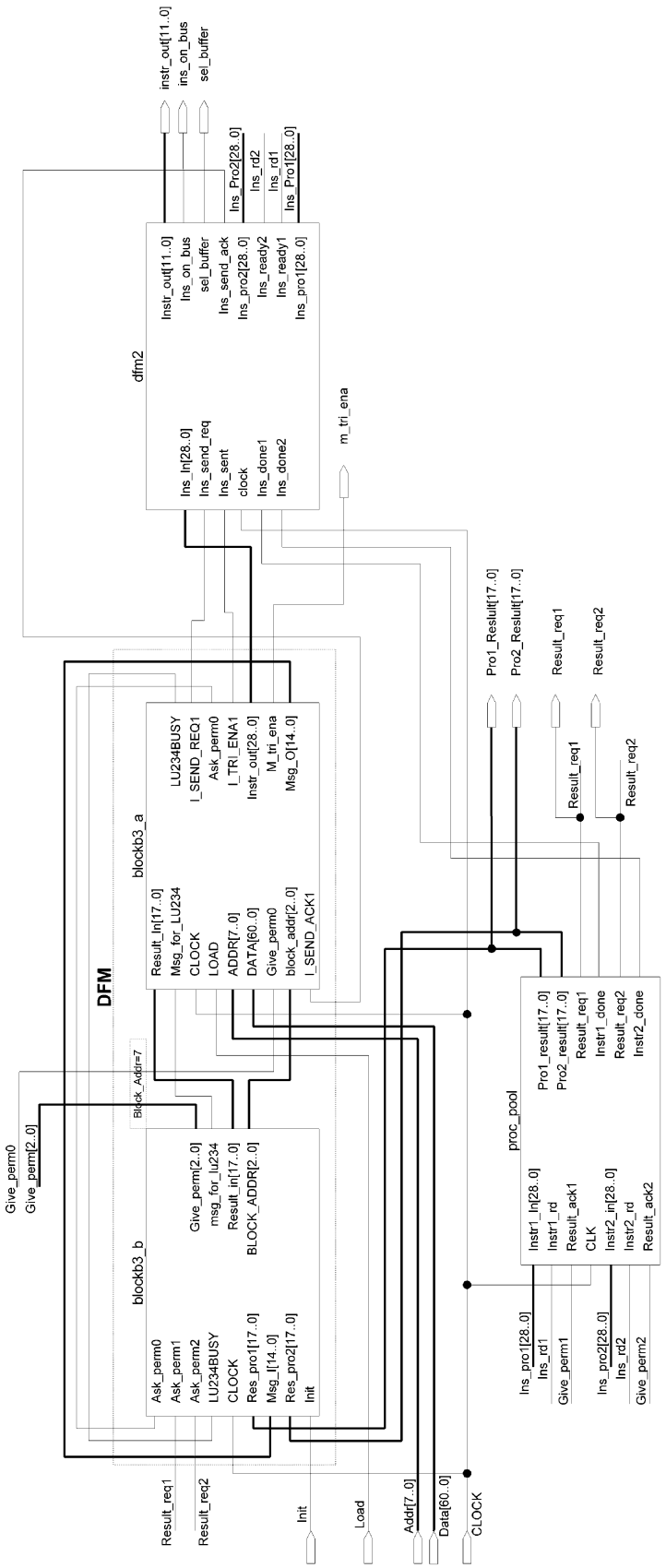

Fig. 13. Graphic design file of the dataflow computer.

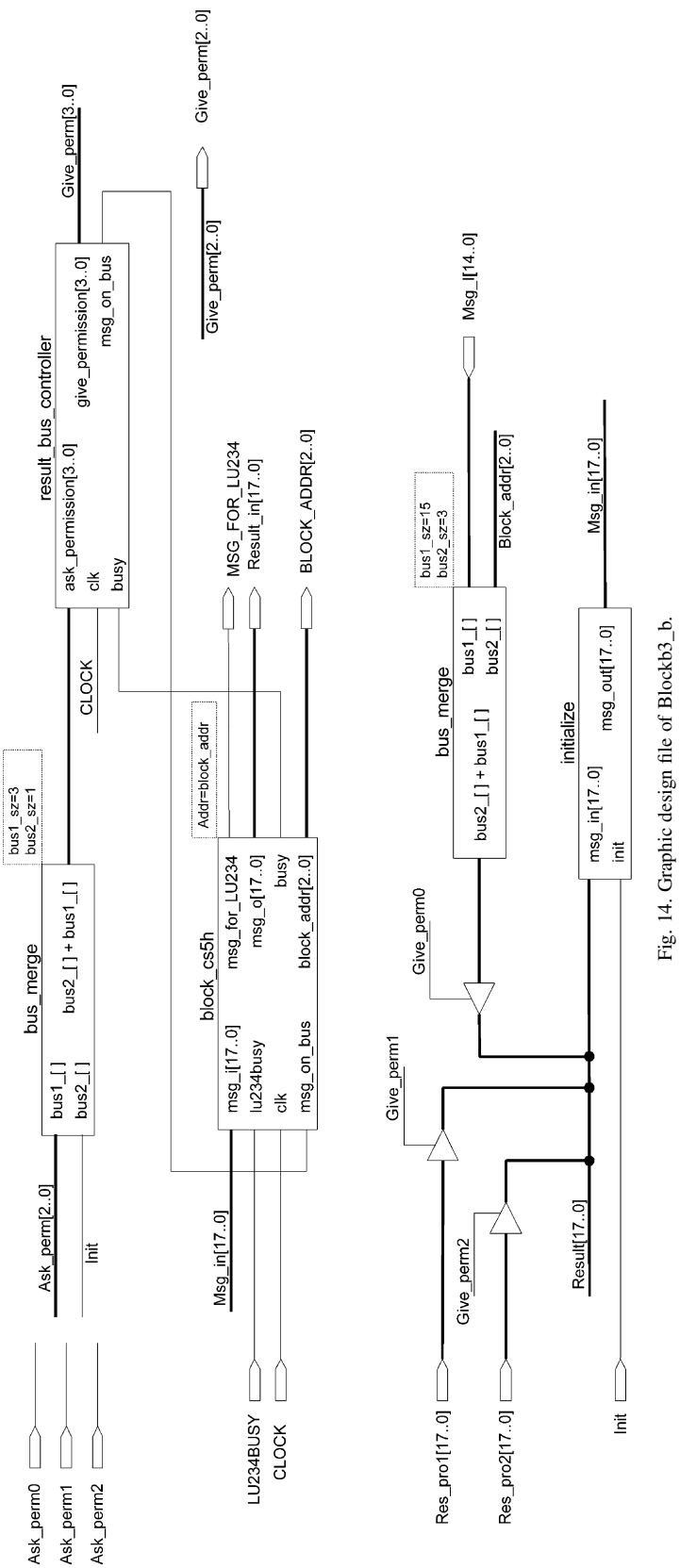

Fig. 14. Graphic design file of Blockb3_b.

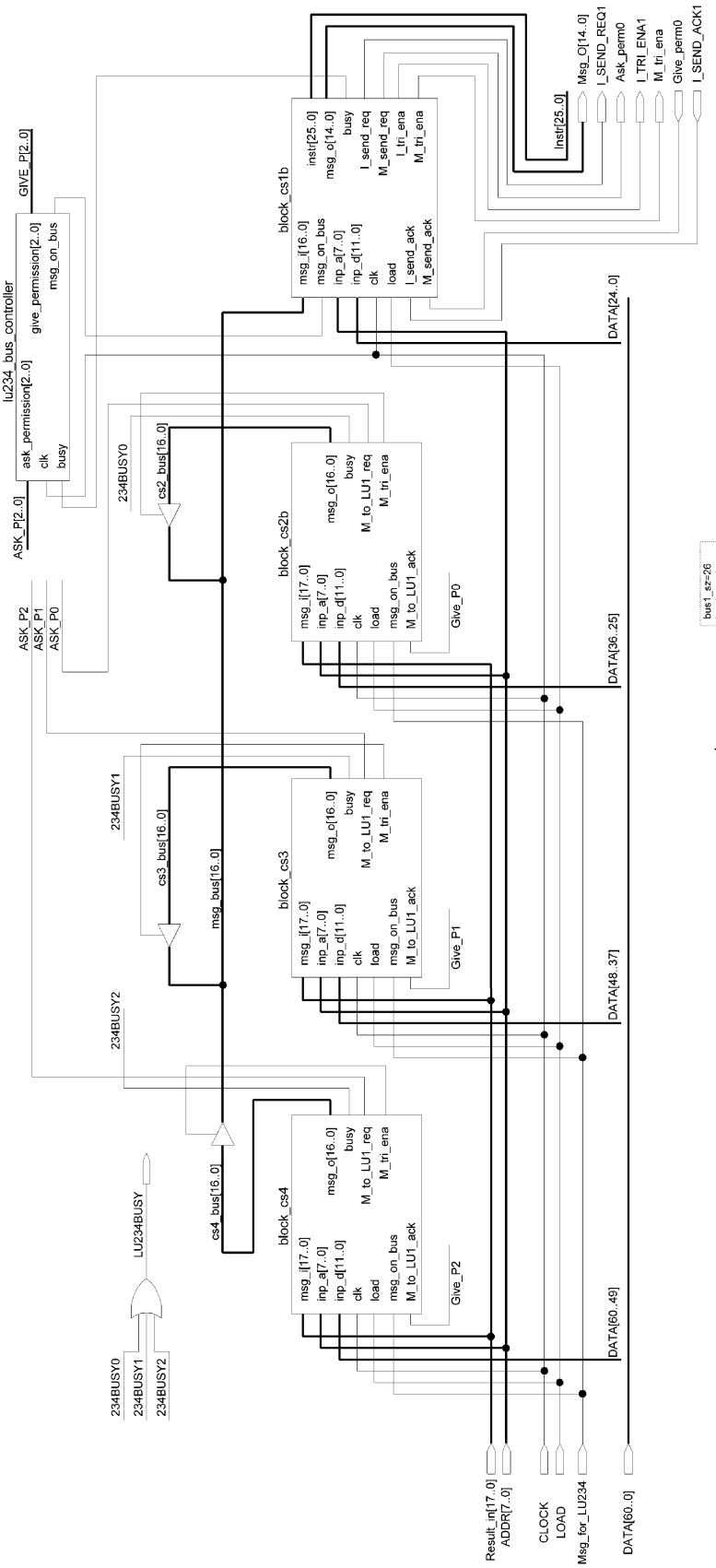

Fig. 14. Graphic design file of Blockb3_a.

adhering to this restriction. Of course, this was a major drawback and different schemes were developed to overcome this restriction. Two ways that were devised to nullify this restriction were through hardware, as implemented in the ϵpsilon dataflow processor [5] with a repeat hardware unit that circulates the result value using a tagging scheme. The second method uses specialized instructions, which hold addresses of additional instructions (beyond the allowed limit) that need the result. The address of the special instruction is on the list of addresses that the executing instruction needs to send the result to. When the special instruction receives the result, it forwards this result to its list of destination addresses. However, no such means need be employed in our architecture, since no instruction maintains a list of destination addresses to send the result to; instead, each instruction has the addresses of the sources for its two operands. When a result packet is sent out, all instructions pick up the packet and compare the source addresses they have with the originating address in the result packet. All the instructions that make a positive match absorb the result (this eliminates the data fan out problem).

Our removing the processor from dataflow memory maintenance tasks has the advantage of simpler processors and the ability to easily replace simpler execution units with powerful ones having a compatible interface. Besides, dataflow in DFM is performed using only the flag bits, thus offering opcode independence. Hence, a compatible language can be used to write the same program or new opcodes can be added to the existing language without affecting dataflow; of course, a compatible processor needs be used to execute different or new instructions.

Appendix A

See Figs. 13–15.

References

[1] Arvind, R.S. Nikhil, Executing a program on the MIT tagged-token dataflow architecture, IEEE Trans. Comput. 39 (3) (1990) 300–318.
[2] J.B. Dennis, D.P. Misunas, A preliminary architecture for a basic data-flow processor, Int. Symp. Comput. Arch. (1975) 125–131.
[3] M. Chatterjee, S. Banerjee, D.K. Pradhan, Buffer assignment algorithms on data driven ASICs, IEEE Trans. Comput. 49 (1) (2000) 16–32.
[4] T.I. Golota, S.G. Ziavras, A universal dynamically adaptable and programmable network router for parallel computers, VLSI Design 12 (1) (2001) 25–52.
[5] V. Grafe, G. Davidson, J. Hoch, V. Holmes, The Epsilon Dataflow Processor, 16th Annual International Symposium on Computer Architecture, 1989, pp. 36–45.
[6] J.R. Gurd, C.C. Kerkham, I. Watson, The manchester prototype dataflow computer, Commun. ACM 28 (1) (1985) 34–52.
[7] J. Hennessy, The future of systems research, Computer (1999) 27–33.
[8] H.H.J. Hum, O. Maquelin, K.B. Theobald, X. Tian, X. Tang, G.R. Gao, P. Cupryk, N. Elmasri, L.J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S.S. Nemawarkar, P. Panangaden, X. Xue, Y. Zhu, A design study of the earth multiprocessor, Int. Conf. Paral. Arch. Compil. Technol. (1995) 59–68.
[9] G.M. Papadopoulos, D.E. Culler, Monsoon: an explicit token-store architecture, Int. Symp. Comput. Arch. (1990) 82–91.
[10] S. Sakai, Y. Kodama, Y. Yamaguchi, Prototype implementation of a highly parallel dataflow machine EM4, Int. Paral. Proc. Symp. (1991) 278–286.
[11] T. Shimada, K. Hiraki, K. Nishida, An architecture of a data flow machine and its evaluation, Compcon (1984) 486–490.
[12] X. Tang, G.R. Gao, Automatically partitioning threads for multi-threaded architectures, J. Paral. Distr. Comput. 58 (1999) 159–189.
[13] H. Terada, H. Nishikawa, K. Asada, T. Okamoto, S. Miyata, H. Asano, T. Tokura, M. Shimizu, S. Hara, S. Komori, K. Shima, Design philosophy of a data-driven processor: Q-p, J. Inform. Proc. 10 (4) (1988) 245–251.
[14] W.-M.W. Hwu, Y.N. Patt, HPSm, a high performance restricted data flow architecture having minimal functionality, Int. Symp. Comput. Arch. (1986).

Segreen Ingersoll received the BS degree in Physics from Bombay University, in India (1992), BS degrees in Computer Engineering and Computer Science from New Jersey Institute of Technology in 1998, and the MS degree in Computer Engineering from the latter institution in 2001.

Dr Sotirios G. Ziavras received the Diploma in Electrical Engineering from the National Technical University of Athens, Greece, in 1984, the MSc in Computer Engineering from Ohio University in 1985, and the PhD degree in Computer Science from George Washington University (GWU) in 1990. He was a distinguished graduate teaching assistant at GWU. He was also with the Center for Automation Research at the University of Maryland, College Park, from 1988 to 1989. He was a visiting Assistant Professor at George Mason University in Spring 1990. He is currently a Professor of Electrical and Computer Engineering, and Computer Science, and the Associate Chair for Graduate Studies in ECE at New Jersey Institute of Technology. He is an associate editor of the pattern recognition journal. He is an author/co-author of about 100 research and technical papers. He is listed, among others, in Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, Who's Who in Engineering Education, and Who's Who in the East. His main research interests are conventional and unconventional processor designs, FPGAs, embedded computing systems, parallel computer architectures and algorithms, network router design, and computer architecture design. He is a member of the IEEE (Senior Member), Pattern Recognition Society, Greek Chamber of Engineers, and Eta Kappa Nu.