# Design, Development and Validation Testing of a Versatile PLD Implementable Single-Chip Heterogeneous, Hybrid and Reconfigurable Multiprocessor Architecture

**J. Robert Heath**, Sridhar Hegde, Kanchan Bhide, Paul Maxwell, Xiaohui Zhao, and Venugopal Duvvuri**
Department of Electrical and Computer Engineering
University of Kentucky
Lexington, KY 40506
heath@engr.uky.edu
**Corresponding Author

## Abstract

There appear to be an increasing number of real-time and non-real-time computer applications where the application may be described by process and/or data-flow graphs (from here on we use the term "process flow graphs"). Such applications include radar/sonar signal processing; collection, processing and storage of data from multiple sensors/instruments; various system simulation environments; communications signal processing, routing; image processing, etc. For such applications, a first goal is the availability of a computer system/architecture platform which will allow an application described by a process flow graph of any topology to be mapped to and executed on the computer system/architecture. The application process flow graph could be single or multiple input/output and cyclic or acyclic. Processes are represented by nodes of the graphs. Further, it would be desirable for the computer system/architecture to be able to continue execution of the application with minimum interruption if the application process flow graph topology were to dynamically change during application execution. This goal is referred to as *application level reconfigurability*. A second goal for the same computer system/architecture would be that it have the ability to dynamically on-the-fly configure, move, or assign processors or other physical resources to application processes (and/or vice versa, the assignment of additional copies of a process to additional processors) that may need them at any time. This goal is referred to as *node level reconfigurability*. A third goal for the same computer system/architecture would be that it be a single-chip heterogeneous multiprocessor system and that it would have the capability to dynamically on-the-fly configure and reconfigure, if and when needed, single processor architectures within the overall multiprocessor architecture. We refer to this goal as *processor architecture level reconfigurability*. With proper Operating System (OS) and other system software support, a computer system/architecture platform which can meet these three goals should be able to execute a wide range of non-real and real-time applications described by process flow graphs of any topology in a fault tolerant manner. The contributions of this paper are in that it describes the research and development and current status of the development, testing and evaluation of such a computer system architecture. HDL "virtual prototype" functional and performance simulation testing results are shown for the architecture executing simple hypothetical applications. Future research, development and testing of the architecture is addressed. The described architecture paradigm and platform is known as a single-chip Hybrid Data/Command Driven Architecture (HDCA) system. A reconfigurable/dynamic production HDCA system would be implemented to Programmable Logic Devices (PLDs).

**Key Words:** Dynamically reconfigurable architecture, single-chip heterogeneous multiprocessor architecture, hybrid architecture, data or command driven architecture, hardware load balancing, Programmable Logic Device (PLD) implementation, virtual prototyping.

## 1 Introduction and Background

The concept, paradigm, and the first high-level functional and architectural view of a Hybrid Data/Command Driven Architecture (HDCA) appeared in [1]. Shortly thereafter, we presented the results of high-level behavioral/functional simulations of the architecture which further and more extensively validated the concept and paradigm of the architecture and functionality of individual initially envisioned functional units of the architecture [2]. Also, via the simulations, we were able to validate the functionality and operational characteristics of a hardware based load-balancing strategy for the architecture [2].

The architecture as originally conceived was to be reconfigurable at two levels, the *'application'* and *'node'* levels and it was envisioned to be used for non-real-time or real-time applications. It is somewhat of a 'hybrid' architecture in that it can be used in either a normal non-real-time command mode or in a real-time or non-real-time data driven mode. When operating in the data driven mode it operates in that mode at the *'process' level*, not at the single-instruction data driven level.

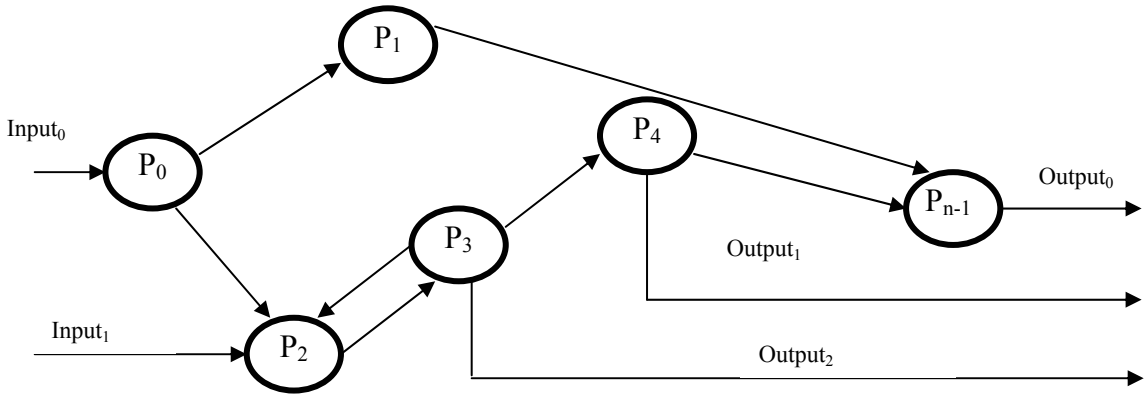



**Figure 1-1. Application Data/Process Flow Graph (Cyclic Topology with 2 Inputs and 3 Outputs)**

The architecture is structured and was conceived to facilitate natural and efficient mapping of applications to the architecture. The architecture is structured and offers highest performance when executing applications that can be modeled as directed graphs and more specifically, acyclic or cyclic process flow graphs. An example cyclic process flow graph topology describing an application which may be mapped to the HDCA is shown in Fig. 1.1. Process flow graphs can be used to model a wide range of applications; radar signal processing, digital system simulation and communications system packet processing are applicable example application areas amenable to being modeled as process flow graphs and mapped to an HDCA system. The architecture is *'reconfigurable' at the application level* in that we feel it has the capability to execute most applications which may be described by a cyclic or acyclic process flow graph of any topology.

Typically, in a process level data driven environment as depicted in Fig. 1-1, data or data-packets/state-vectors (a data-packet or state-vector may typically contain anywhere from a few to 1K or more bytes) enter the HDCA system and as they enter the system, the data packets initiate processes (initially processes $P_0$ and $P_2$ in Fig. 1-1). The initial processes $P_0$ and $P_2$ process these input data-packets and finally produce output data-packets which are forwarded to successor processes ($P_1$ and $P_3$ of Fig. 1-1). The forwarded data-packets initiate processes $P_1$ and $P_3$ which in turn produce output data-packets which are forwarded to successor nodes (processes) $P_{n-1}$, $P_2$, and $P_4$, etc. Data being forwarded to these three nodes initiates the respective processes within these three successor nodes and so on. Since input data-packets may continually enter the system at nodes representing the processes $P_0$ and $P_2$, there may be times when all processes Pi of Fig. 1-1 are active on different processors of the parallel processing HDCA system. Even though this may be viewed as a data flow/driven environment, it was our intent from the beginning to minimize the data flow within the architecture in order to simplify the bussing and control structure of the architecture. In this architecture much shorter control tokens flow through the architecture rather than the more voluminous data-packets/state-vectors. Our goal was to keep all data-packets/state-vectors in a commonly shared memory of the multiprocessor system hence minimizing the flow of data packets through the architecture other than data-packet flow between the set of processors and the commonly shared memory.

As stated above, a goal is the development of a computer system/architecture with *application level reconfigurability* capabilities such that it can execute applications described by process flow graphs of any topology. The nodes of the graphs can represent medium or coarse grained processes. The directed arcs of the graphs represent data flow between the processes as discussed above, but in reality and more importantly, they also represent precedence and control flow between the processes of the nodes.

As a way of illustrating the second goal of *node level reconfigurability* for the HDCA which can execute applications described by process flow graphs, consider Fig. 1-2. In many cases, for some application/architecture combinations, a static resource allocation algorithm is first executed to statically assign specific application processes to specific processors within a multiprocessor architecture prior to execution of the application. Some less frequently requested processes may be assigned to only one processor where as other more heavily requested processes may be assigned to several processors in order to meet real-time requirements when operating in a real-time mode. In general, each processor of such a system would not contain a copy of every process of an application. In Fig. 1-2, the lower of three rectangles with solid line input/output represents initial static resource allocation assignment of a single copy of $Process_j$ (Pj) to a single processor Computing $Element_1$ (CE1). In many cases, especially in real-time applications such as radar signal processing, unexpected events may occur (such as the number of objects being tracked unexpectedly exceeding limits used during static resource allocation) which require additional

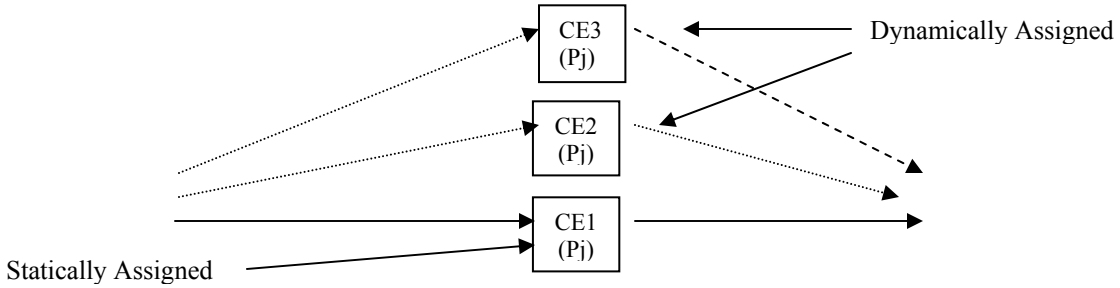



**Figure 1-2. Dynamic Assignment of a Process $P_j$ to Two Additional CEs (CE2 and CE3) – (Dynamic Node Level Reconfigurability).**

copies of processes to be quickly and dynamically assigned and initiated on-the-fly on existing or newly configured CE processors within the multiprocessor architecture (*node level reconfigurability*). In Fig. 1-2, the two upper rectangles with dotted line input/output represent a situation where it has been dynamically, in real-time, determined that two additional copies of process Pj are required to meet application real-time computing requirements and thus must be loaded to and execution begun on two additional processor CEs, in this case CE2 and CE3.

In the hybrid HDCA system [1,2], it was decided that the directed arcs (all except system input/output arcs) of an application data/process flow graph (Fig. 1-1) would represent short "control tokens" or "commands" which activate processes rather than sometimes voluminous data.

Figure 1-3 shows a high functional level architectural view of the initially envisioned parallel and scalable HDCA [1]. Any application data/process flow graph single process nodal structure ($P_i$) of Fig. 1-1, differentiated by number of node input/output arcs, can be implemented by a single CE (processor) and Queue (Q) hardware structure (slice) of Fig. 1-3. The hardware implementation consists of a CE (processor) fronted by a multifunctional Queue (Q). There is not a one-to-one relationship between the number of application processes, of a application data/process flow graph, and the number of CE-Q pairs of an HDCA system which may run the application [1,2].

The HDCA of Fig. 1-3, in addition to being a parallel multiprocessor system that is heterogeneous, scalable, token controlled, pipelined and reconfigurable/dynamic at the application and node levels as proposed and simulated in [1] and [2], is now being further enhanced and is now proposed to be *reconfigurable/dynamic at the CE processor architecture level* and the desire is to implement the multiprocessor system within a single chip except for possibly the Large File Memory of the architecture. Being reconfigurable/dynamic at the CE processor architecture level means that the architecture of each individual CE processor of the multiprocessor HDCA may be dynamically configured when needed and reconfigured to best-fit the algorithmic and data structure of the process currently executing on each of the individual processors when needed for performance enhancement. Such capability may be desired for some compute intense real-time applications or compute intense non-real-time applications. Each CE of Fig. 1-3 contains one processor. The architectures of processors within an HDCA system may vary widely. The goal of implementing the reconfigurable architecture within a single chip is now becoming more feasible with existing and future anticipated Programmable Logic Device (PLD) and other Integrated Circuit (IC) technologies.

In the HDCA of Fig. 1-3, system input data in the form of data-packets/state-vectors enters the system via the Input FIFOs and an input process $P_i$ resident within a CE reads the input data from there and after doing initial processing on it, writes the data in the form of a data-packet/state-vector to a commonly shared RAM Data Memory Block. For large data-packets (such as images or portions of images), the data-packets are written to the Large File Memory (may be off-chip) of Fig. 1-3. In general, as each process running on the processor of a CE completes execution, resultant (output) data in the form of data-packets, forwarded to following successor processes, is either written to the Data Memory Blocks or the Large File Memory of Fig. 1-3. One of the last functions a completing process running on a processor of a CE does is generate a "true" control token followed by a number of, to be explained, "dummy" tokens that are sent to the Control Token Mapper (CTM) unit of the architecture (Fig. 1-3). The control tokens are then dynamically mapped by the CTM to the multifunctional Q of the "most available" CE(s) containing a copy of the requested successor process(es) which processes the resultant data written to the data or file memory by the predecessor processes. A "most available" CE among several containing a copy of a requested process is the one where the true wait time for initiation of processing of the requested process is minimum. A single process Pj resident in more than one CE is referred to as a "multi-copy process". The control token generated by the process of a fork node (one with multiple output paths or successor processes) would be mapped by the CTM to more than one CE. Once the Q of a CE receives a control token and it's trailing dummy token(s) that were mapped to it by the CTM, that particular CE would first have configured to it, if not previously configured, a best-fit processor architecture for the process requested by the incoming control token once that control token has worked its

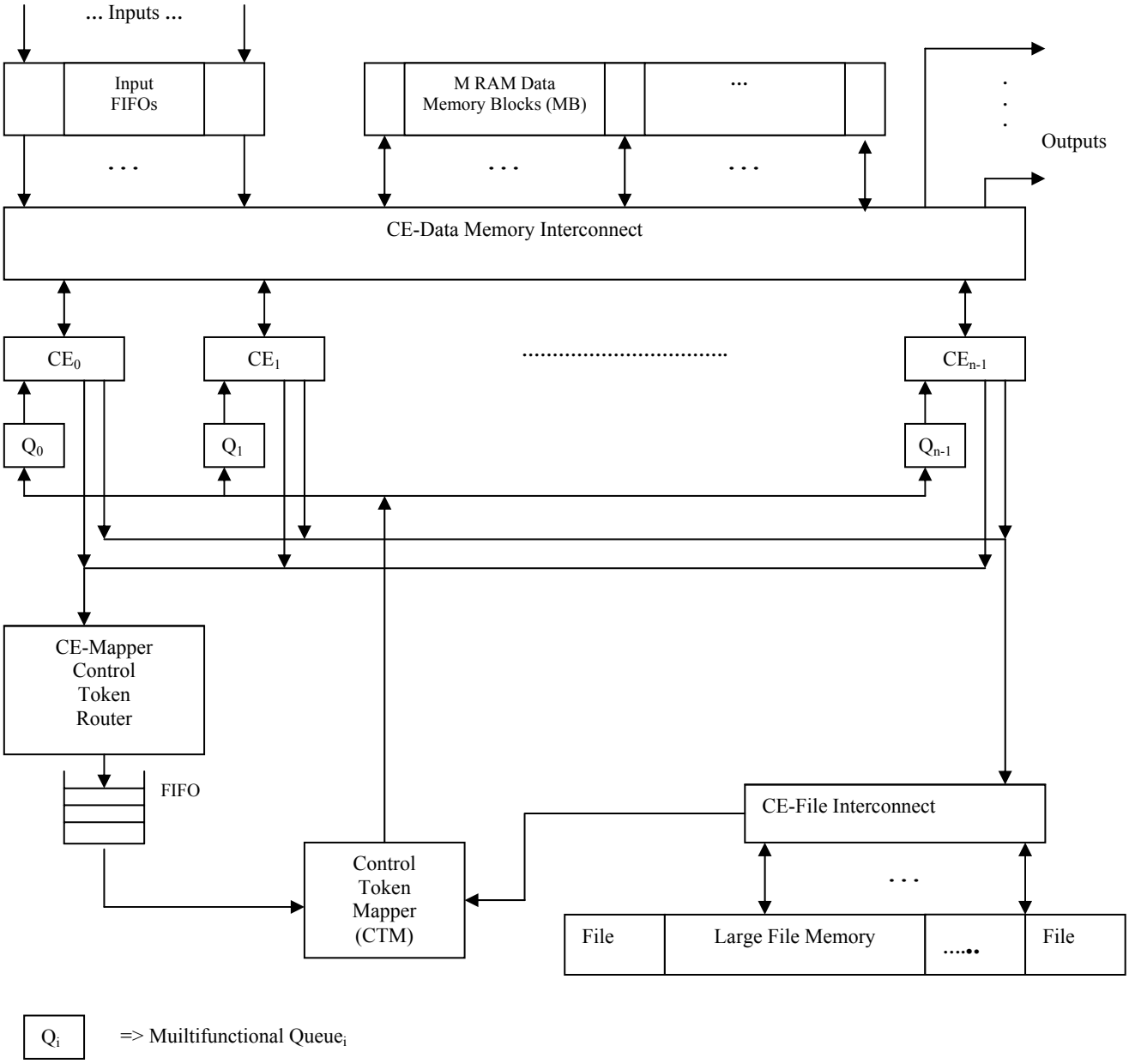



**Figure 1-3. Single-Chip Reconfigurable HDCA System (*Large File Memory* May Be Off-Chip)**

way to the front of the Q and then the process requested by the incoming control token would be initiated and run on the configured best-fit processor architecture. Data-packets/state-vectors needed by an initiated process are read by the process from either the data or file memories of the HDCA of Fig. 1-3. The Q(s) of Fig. 1-3 perform much more functionality than the normal First-In-First-Out (FIFO) functionality as will be addressed in a later section of the paper. The additional functionality is required for proper control of the HDCA system as it operates including control of the CE-Data Memory Interconnect (see Fig. 1-3).

As a process executing in a CE processor of Fig. 1-3 completes, it generates a number of "dummy" control tokens to be packaged with and sent immediately following the one "true" control token generated by the process to be sent to the Control Token Mapper (CTM) of Fig. 1-3 via the shown path from each CE to the CE-Mapper Control Token Router to the CTM. The CTM (see Fig. 1-4), using information contained within a incoming "true" control token, finds the CE processor, from among several for multicopy processes, containing a copy of the requested successor process where "wait time" to execute the requested process is minimum. The number of "dummy" tokens generated by a completing process is directly proportional to the execution time of the process to be requested by the "true" token preceding the dummy tokens. Using this mechanism, filled "queue depth" for the Qs of the

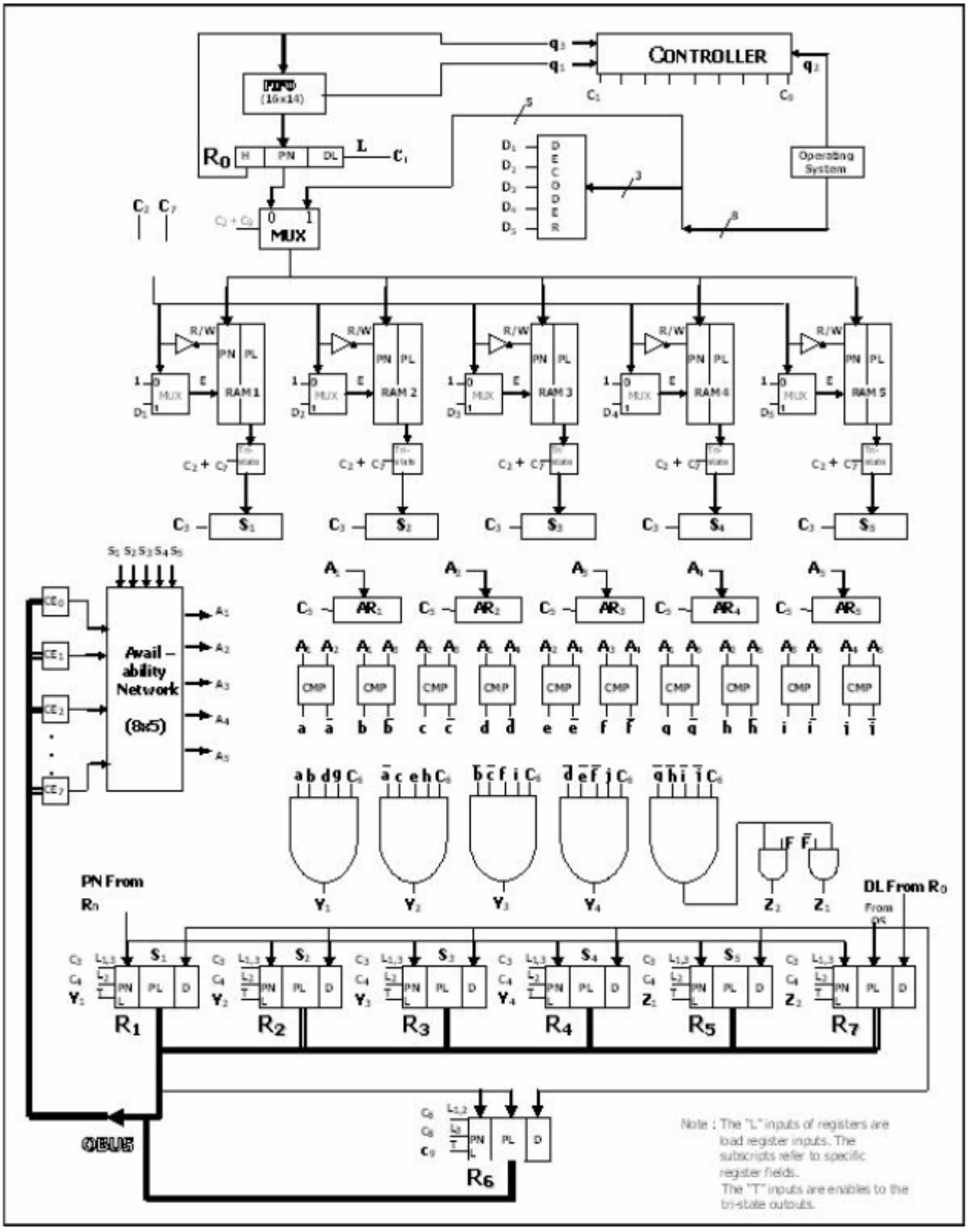



**Figure 1-4. Functional Organization of the Control Token Mapper (CTM) of a HDCA system.**

architecture can be used as a measure of true wait time for the initiation of a requested process in a CE processor. Keeping a proper processing (load) balance between the processors in a parallel processing environment is important in terms of increasing system performance. In short, the hardware load balancing strategy and mechanism for the HDCA when operating in either a real-time or non-real-time mode and in either a data or command driven environment, is based around the CTM being able to map control tokens requesting a successor process, to the CE processor of possibly several containing the requested process, which has the shortest Q depth. Routing of the process requesting true control token and its trailing dummy tokens to the CE containing the process with the shortest Q depth results in the requested process being initiated with minimum wait time.

CE processor Q depth will also be used by the CE-Data Memory Interconnect to resolve memory access conflicts resulting when multiple CE processors of Fig. 1-3 try to access data within the same memory block of the shared RAM Data Memory Blocks (MBs) of Fig. 1-3. Each multifunctional queue (Q) of the HDCA of Fig. 1-3 provides Q depth and additionally provides the input token rate over a programmable time period, the change in input token rate over a programmable time period, can swap the position of tokens in the Q etc. This information is required and utilized by an operating HDCA system for load balancing purposes and in making control and dynamic resource allocation decisions. In addition to the CTM of Fig. 1-4, we will now address the most current functional, architectural, operational, and performance requirements and characteristics of all other major functional units of the HDCA system of Fig. 1-3. We will then take a current and more detailed view of the HDCA as it is now structured, which is quiet different from the view of Fig. 1-3.

# 2 HDCA Operation, Functional Units and Current Architectural Structure

The HDCA architecture operates on the principle that applications can be modeled using process flow graphs which are implemented on a HDCA system. The fundamentals of process flow graphs start with their three basic structures as shown in Fig. 1-1. A process flow graph can basically consist of linear pipelines, forks, and joins. In a linear pipeline Control Tokens simply move from one process to the next in a uniform manner. Each process has a single-input and a single-output. Data needed by the processes are resident in a shared memory (the MBs of Fig. 1-3). Within a fork ($P_0$ of Fig. 1-1), Control Tokens are distributed to multiple follow-on or successor processes. Forking may be to two or more processes and may be selective or non-selective. To potentially increase the amount of parallelism in an application, a scheme for multiple forking has been introduced to the HDCA. The join is the complimentary function to the fork where Control Tokens from two or more sources are selectively or non-selectively combined for execution in one process ($P_{n-1}$ in Fig. 1-1). The *non-selective fork* represents a total broadcast of data along all output arcs, whereas the *selective fork* represents a broadcast of data along a single output arc or a subset of output arcs. Similarily, during the execution of a selective join, only a selected subset of input arcs to a process is active. A non-selective join is triggered when all the inputs to a process are active. When these basic structures are combined, any application composed of multiple processes can be modeled.

Processes on individual Computing Elements (CE's) do not start execution until an initializing token has arrived. Once a token is received, indicating the location and availability of data needed by the process, the CE parses it in order to determine the proper process to execute. This is due to the fact that each CE can hold several processes in its Instruction memory or only one process. The CE then executes the appropriate process and upon completion issues the follow-on token(s) for the successor process (es). These tokens are the sole communication between CE's. . An example Control Token format is shown in Fig. 2-1.

In this token the Hold Field is used to indicate a requested process that is a member of a join operation. It is also used by the system in a manner such that the processor token queue depth represents true wait time for the initiation of a requested process. The Physical address denotes the destination CE or functional unit for the token. For example the five different CEs used in the HDCA system presented later have addresses of two, three, four, five and six The Process Number indicates which process to execute and the Data Location provides the address of the data in shared memory which is accessed through the Crossbar interconnect switch as described in [7,14,15].

## 2.1 Control Token Mapper (CTM)

An important function of the CTM (see Fig. 1-4 for this functional unit in the DPCA) is to maintain the dynamic system workload balance. In order to achieve this goal, it constantly monitors the input control token Queue (Q) lengths/depths of each CE in order to determine the most available CE. Control tokens are sent first to the CTM where it is cross-referenced in a RAM table to determine which CE's are able to run the desired process. Not all CEs can run all the processes. The workloads of the eligible CEs are then compared, resulting in a control token being issued to the least loaded CE i.e the one with the lowest amount of work to be done. In order to determine which CE has the least amount of work, the concept of shortest wait time is used. The CE that has the shortest wait time indication in its input control token queue is the most available since it will service the token before its corresponding CE. Once the eligible CE's are known, it compares the workloads of those CE's to determine which is the least utilized. A new control token is then created using the physical address of the selected CE and the location of the associated data. The newly formed token is then output on the

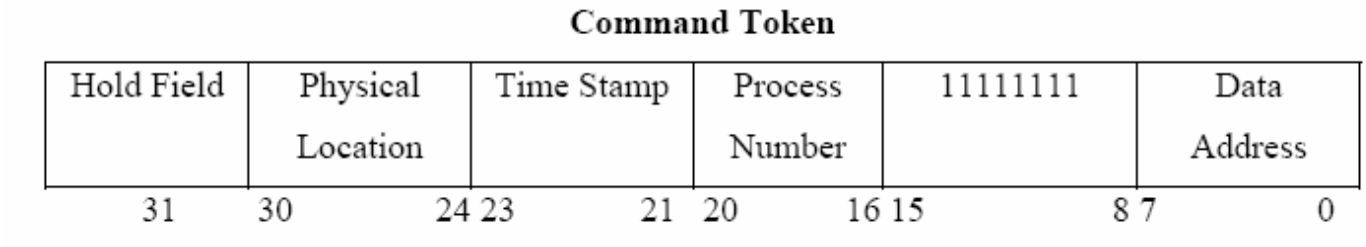



**Figure 2-1. Control (Command) Token Format.**

Token Bus via the OBUS (Fig. 1-4.) to the appropriate CE. This new control token (Fig. 2-1) contains the Process Number to be executed, the physical location of the destination CE, and the address of the required data in the shared data memory in addition to a "Time Stamp" to be later addressed.

In addition to the load balancing function of the CTM, the state of the system is continuously monitored in order to detect faults and system failures. If a CE node fails, the system has the ability to shift the work of a failed node to another location. Additionally, the system is designed with the intent to allow it to reconfigure its processing elements in the event of a failure or to create additional copies of a resource that is heavily used. This happens when the tokens have been queued sufficient enough, that the queue depth reaches a pre-defined "threshold" determined by the user/operating system. At this stage, an additional processor is dynamically initiated and configured, on the fly, to "help-out" this overloaded CE and help it reduce the queue depth by executing some of the follow on processes. This allows the system to dynamically maintain the desired application system input to output rate and functionality of the system even if elements fail or workloads are higher than initially and statically predicted from the application process flow graph.

## 2.2 CE-Data Memory Interconnect Network and Requirements

The shared RAM Data Memory Blocks (MBs) of the HDCA, as shown in Fig. 1-3, is organized as a number of individual memory blocks. This type memory organization is required by the HDCA. If two or more processors want to communicate with memory locations within the same memory block, lower priority processors have to wait until the highest priority processor gets its transaction done. Only the highest priority processor will receive a request grant and the requests from other lower priority processors must be queued and these requests are processed only after the completion of the first highest priority transaction. The interconnect network developed for the HDCA, the CE-Data Memory Interconnect of Fig. 1-3, is able to connect requesting processors on one side of the interconnect network to the memory blocks on the other side of the interconnect network. The efficiency of the interconnect network increases as the possible number of parallel connections between the processors and the memory blocks increases.

When there is contention for access to data in the same MB of the shared memory of an HDCA system by CE processors, the memory contention resolution policy is that the processor with the highest magnitude Q depth count is given highest priority in accessing the contested MB. As earlier addressed, Q depth count is a count of the number of real and dummy process request control tokens (see Fig. 2-1) in the Q of each CE processor at anytime. The magnitude of Q depth count is directly proportional to true wait time for initiation of the process requested by a control token which may be entered into the control token Q of a CE processor. The general philosophy behind the policy is that those CE processors with the largest magnitude Q depths are the "most behind" in their processing since Q depth indicates wait time and therefore they should be allowed to access the data they need so that they may catch up. This policy is applicable to most non-real-time and real-time applications and applications that are either data or command driven.

The policy described above to be utilized by the HDCA to resolve memory contention conflicts within a MB is a variable priority protocol. The priorities assigned to CE processors based on the Q depth of the processor will dynamically vary over time. Thus, the CE-Data Memory Interconnect of the HDCA must be able to accommodate and implement a variable priority protocol in terms of resolving memory contention conflicts.

Interconnect networks can be classified as static or dynamic [8,9]. In the case of a static interconnection network, all connections are fixed, i.e. the processors are wired directly, whereas in the latter case there are routing switches in between. We desire to be able to run a wide range of applications on the HDCA, therefore, we need a dynamic interconnect network.

For the HDCA system, the desired interconnect should be able to establish non-blocking, high speed connections from the requesting CEs to the MBs. The interconnect should be able to sense if there are conflicts such as when two or more CE processors requesting connection to the same MB and give only a highest priority processor the ability to connect.

A factor in selecting the processor-to-shared-memory interconnect in a multiprocessor system such as the HDCA is the anticipated maximum number of processors to be implemented into the system. Since the HDCA is a single-chip multiprocessor system, we currently envision a maximum number of approximately 64 processors within an HDCA system. A majority of anticipated applications can be run with

32 or fewer processors. Therefore, we will select a double sided interconnect assuming a majority of applications can be handled with a 32 X 32 or smaller double sided interconnect.

### 2.2.1 CE-Data Memory Interconnect Network Selection

Considering the above CE-Data Memory Interconnect Network requirements relative to structure and connection capability (double sided with N inputs and N outputs and nonblocking) including the desire for it to be dynamic, high-speed, able to handle a variable priority protocol and have competitive complexity for $N \leq 32$, we considered three (3) candidate interconnect networks. They were the Clos, Benes, and Crossbar networks. All of these networks can be implemented as double sided (N X N) nonblocking interconnect networks. We based our decision on the complexity of the networks for the size we needed, speed (inversely proportional to the number of stages through the network), and the ability to implement the variable priority protocol in the network related to resolving memory block contention from CE processors.

Interconnect networks have been extensively studied and evaluated in the past. In terms of comparing the complexity of the Clos, Benes, and Crossbar networks, one may refer to [7] and [9] for a more detailed view of their structure/interconnect capability and development of the complexity metric for each.

The HDCA system normally requires an interconnect with $N \leq 32$. The crossbar implemented interconnect best suits the system as it has minimum or competitive complexity for the sizes of interconnect needed by the HDCA system, it is highly non-blocking as no processor has to share any bus with any other processor and it is a very high speed implementation as it has only one intermediate stage between processors and memory blocks. Also, it is anticipated the variable priority shared memory contention protocol can be most easily and efficiently (minimum logic complexity) implemented within a Crossbar network. For these reasons, a Crossbar interconnect topology was chosen for the CE-Data Memory Interconnect network of Fig. 1-3.

### 2.2.2 Organization, Architecture, and Design of New Crossbar Type CE-Data Memory Interconnect

We developed a new double sided crossbar processor-to-shared-memory interconnect for the less restrictive general case of N inputs and M outputs, ie, an (N X M) network [7,14,15]. M may equal N if desired. The interconnect network serves as the dynamic interconnect between N CE processors and M memory blocks of the HDCA. We let ($M=2^c$). We assumed each MB has a capacity of $2^b$ words and that each word is 'a' bits wide. The total capacity of the shared M RAM Data Memory Blocks of Fig. 1-3 will therefore be $2^{b+c}$ a-bit words. Figure 2-2 is a functional and architectural model view of the (N X M) crossbar interconnect network. The N CE processors (P[0], P[1], - - P[N-1])are seen to the left and the ($M=2^c$) MBs (MB[0], MB[1], - - MB[M-1]) are seen to the right of the figure. Internal to the interconnect network one sees N Decode Logic (DEC) functional units (DEC[0], DEC[1], - - DEC[N-1]) and M Priority Logic (PRL) functional units (PRL[0], PRL[1], - - PRL[M-1]).

The input bussing structure of the interconnect consists of the PI[i] and MB_ADDR[i] busses for each CE processor P[i]. The internal bussing structure of the interconnect consists of the DM[i][j] busses shown in Fig. 2-2 and the output bussing structure of the interconnect consists of the IM[j] busses appearing in Fig. 2-2. Manuscript page limitations prevent us from explaining details of how the below interconnect implements all required functionality. All interconnect functionality and performance requirements for the below interconnect were validated via FPGA based experimental hardware prototype testing as addressed in [7,12,15].

## 2.3 Multifunctional Queue (Q)

When the original DPCA was conceived [1,2], as represented in Fig. 1-3, it was known that the CE's would each require a FIFO queue to hold control tokens that were yet to be parsed and executed. CEs can be busy executing a process when control token(s) may come in. The incoming tokens have to wait for their turn in the queue. If there was no queue provided, these tokens would be lost and hence the system would not behave as expected. As the HDCA was further developed, it was determined that this queue needed additional functionality. The additional Queue functionality allows the HDCA to operate in both a real time and non-real time environment and the new functionality supports dynamic node-level re-configurability of the HDCA. The functionality of the FIFO queue was expanded to implement six different functions [4]. It can read and write simultaneously, maintain a count of elements in the queue, and signal when a programmable queue depth threshold is met. It can also switch the order of any two tokens in the queue and report the net rate at which tokens are entering or leaving the queue over a programmable time period. A high-level block diagram of the Multi-Function Queue is found in Fig. 2-3. Figures 2-4 and 2-5 show a functional level diagram of the FIFO and Rate blocks respectively.

The Queue's ability to switch the order of tokens can allow the system to give priority to a given token. If the system sees that a process is waiting for an input token that is stuck in an unusually long queue, it can re-organize the queue such that the token of interest is swapped with the token at the top of the queue which is about to be serviced. This helps to reduce execution time by allowing processes to be executed faster. The queue achieves this by placing the tokens in a temporary buffer and then swapping them. The swapping is implemented by an address interchange between the two tokens using the RAM1 and RAM2 of Fig. 2-4.

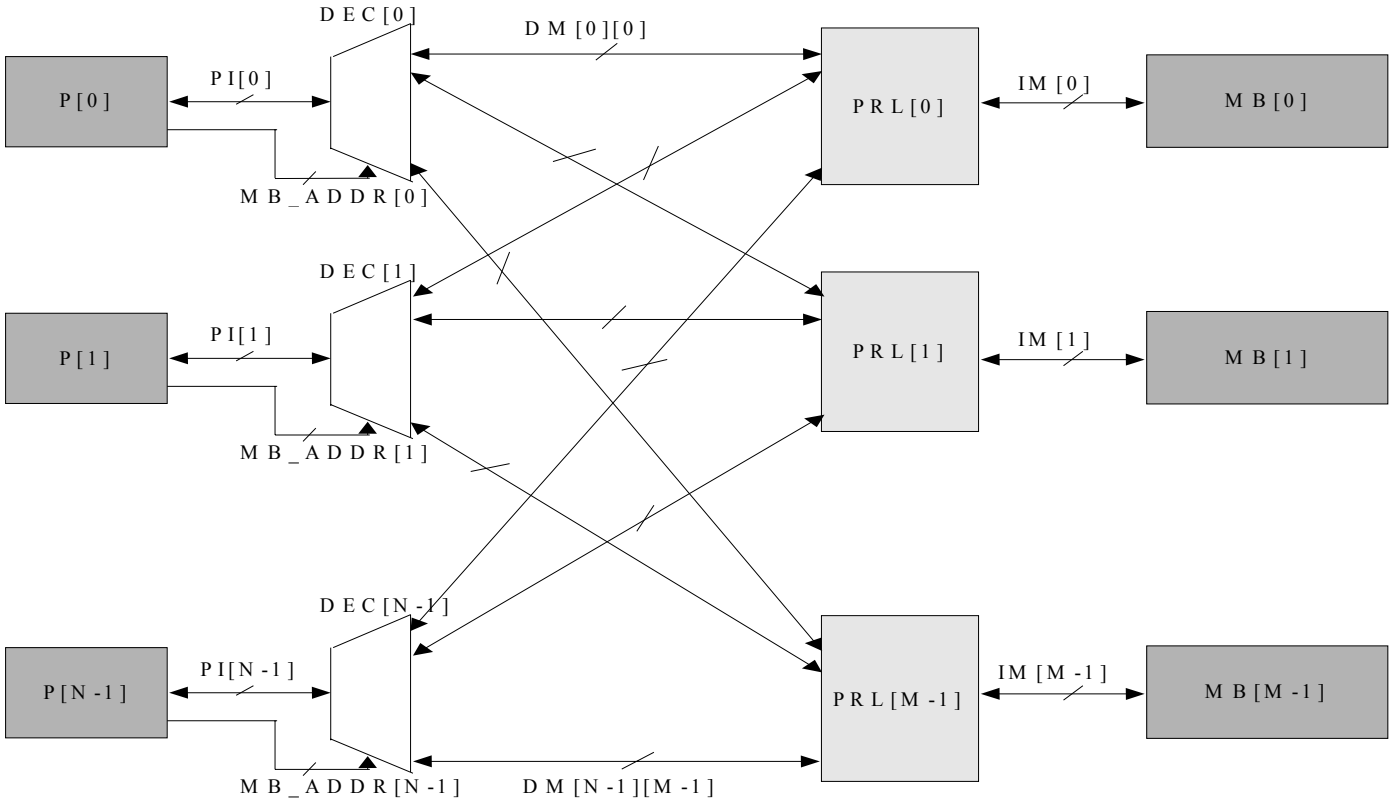



**Figure 2-2: Block Diagram of the New CE-Data Memory Crossbar Interconnect Network**

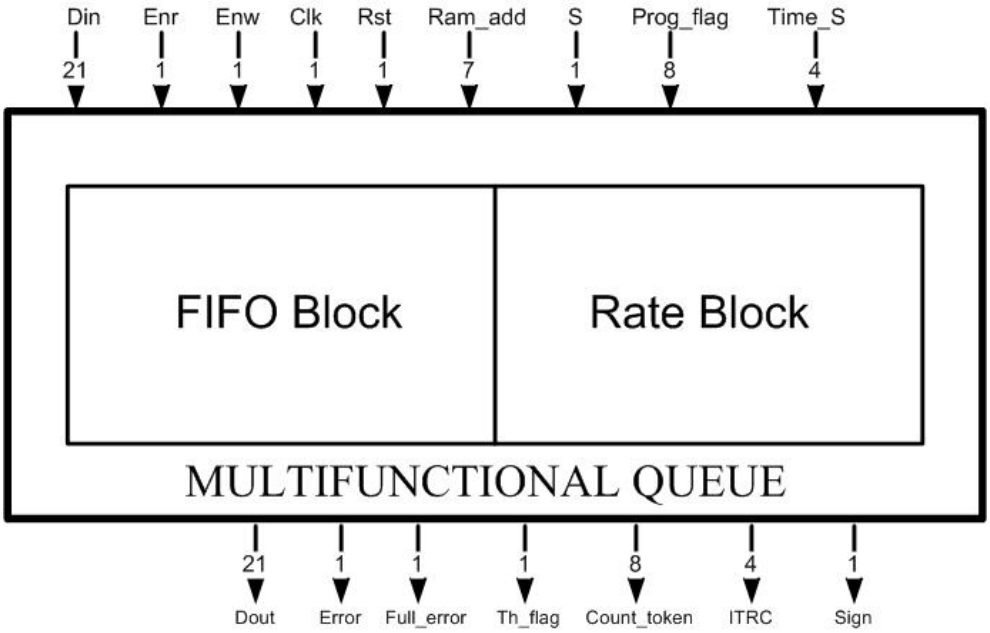



**Figure 2-3: Multifunctional Queue (Q) of HDCA.**

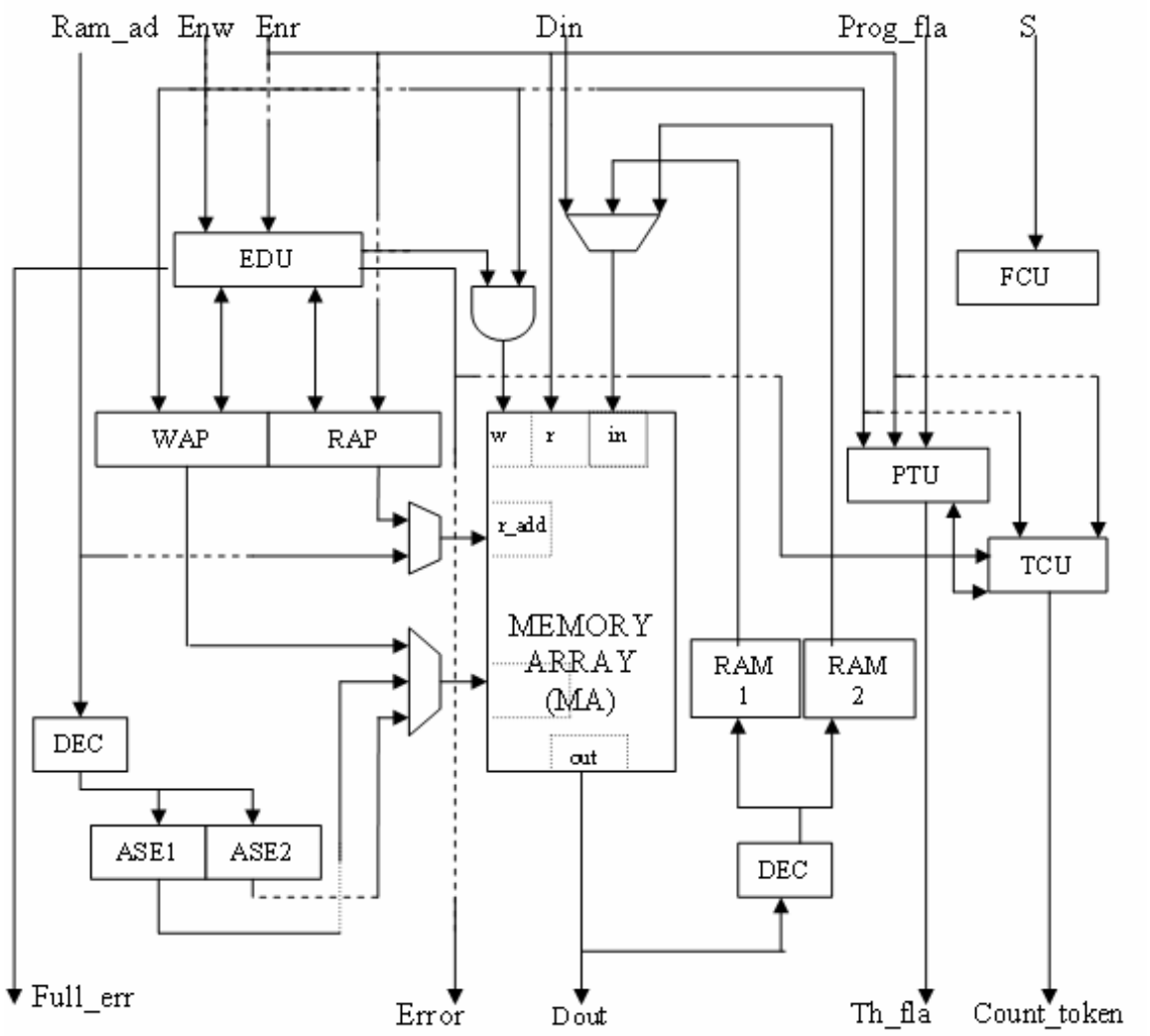



**Figure 2-4: FIFO Block of Multifunctional Queue (Q).**

The Rate Block of Fig. 2-5. measures the Input Token Rate Change (ITRC) over a programmable time interval (Time_S). This time period indicates the time period over which to base the calculations. The Queue then determines whether there was a net increase or decrease in the number of tokens passing through the Queue over the given time period. The outputs of this function are a sign bit (Sign) and a magnitude (ITRC). Thus the Operating System can determine the workload of a CE by the number of tokens arriving or departing a given queue.

## 2.4 The Computing Elements (CEs)

A first phase prototype of the HDCA consisted of 3 CEs [5,6]. The first phase prototype could only execute applications described by acyclic process flow graphs and it did not have node level nor processor architecture level dynamic capability. We have now developed and tested, as is being reported in this paper, a second phase virtual prototype of the HDCA [14,16]. The second phase prototype contains 5 CEs, it can execute applications modeled by both cyclic and acyclic process flow graphs and it is dynamic at both the application and node levels. To demonstrate the heterogeneous nature of the 5 CE prototype of the HDCA, 3 of the CEs are a Memory/Register Architecture programmable processor developed by the authors which operates on 16-bit words, a fourth CE is a high performance special purpose Multiply unit and the fifth CE is a special purpose Divide unit. In this section we present a brief architectural view of each of the CE architectures. In a later section of the paper, we will present an architectural view of the entire current 5 CE (CE0, CE1, CE2, CE3 and CE4) second phase prototype.

Two of the CEs, CE0 and CE1, are 16-bit unpipelined memory/register computer architectures as shown in Fig. 2-6. The assembly language instruction set for CE0 and CE1 is shown in [16]. Both processors have full functionality: a register set in the data path available to the assembly language programmer, a Hardware Vectored Priority Interrupt System (HVPIS) in addition to other functional units such as Arithmetic and Logical Unit (ALU), a Program Counter (PC) and simple Input/Output (I/O) structure. The instruction set of CE0 and CE1 is sufficient to write interesting test programs

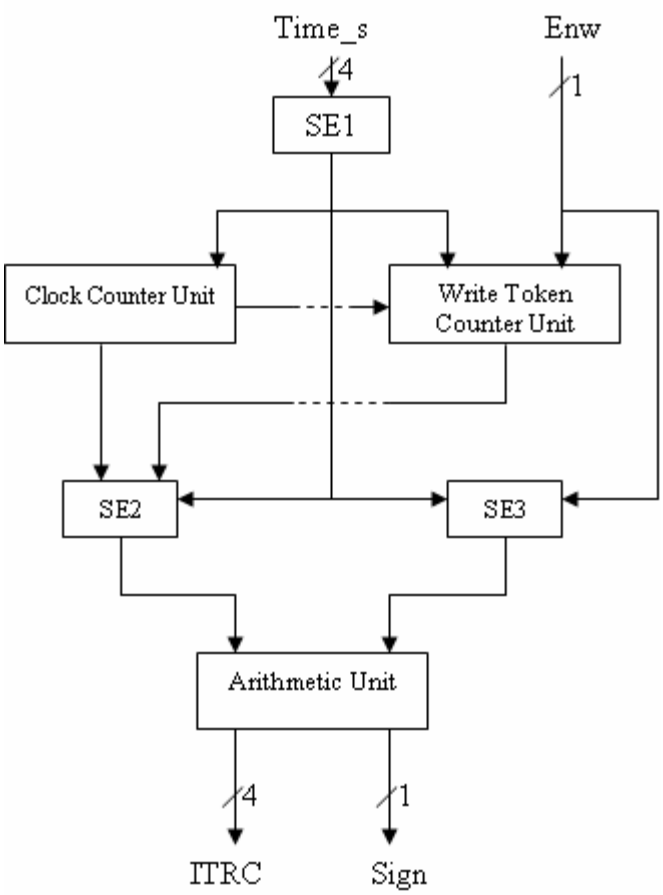



**Figure 2-5: Rate Block of Multifunctional Queue (Q) of HDCA.**

sufficient to test the functionality of the second phase model of the HDCA.

The processor used for CE2 is a pipelined Divider circuit as shown in Fig. 2-7. This divider can be considered as a special purpose circuit for a system that needs additional computational power and it allows the single-chip multiprocessor prototype system to be heterogeneous. Each CE, as shown in the 5 CE HDCA prototype of Fig. 2-11, has a Controller, which includes the multifunctional queue, a Lookup Table (LUT) and an Interface Controller (see Fig. 2-9 for the CE Controller). Additionally, as part of work done to build the second phase prototype, CE3 and CE4 were added to the HDCA system in order to execute a wider variety of applications and allow the implementation and testing of node level dynamic functionality to the HDCA. The need for a special purpose Multiplier CE (CE3) was felt. Often, in DSP and Image Processing applications, multiplication is an important aspect of any operation and hence the new special purpose multiplier was added to the HDCA system. Its architectural view is seen in Fig. 2-8.

The fifth CE, CE4, is architecturally the same as the memory/register architecture CE0 and CE1 of Figure 2-6., but it is unique in the sense that it is not utilized under normal conditions. Under normal operating conditions, when the Queues of CE0 and CE1 have not built up to their threshold (indicating they are not overloaded or over subscribed), this CE acts as a stand-by CE monitoring the queue depth of either of the two CEs. Once the queue depth of both CE0 and CE1 of the operational CEs exceeds the pre-programmed threshold, this additional CE is dynamically configured, on the fly, to initiate and start accepting control tokens from that point on that would normally have been routed to CE0 and CE1 by the CTM, and executing them. Implementation of this concept results in node-level dynamic capability of the architecture. Once the queue depth is reduced below the pre-programmed threshold, the CE goes back to its sensing state where it silently monitors the queue depth of CE0 and CE1.

### 2.4.1 CE Controller

The HDCA has recently been revised and enhanced as addressed in this paper and now appears as shown in Fig. 2-11. This view of the architecture is somewhat different from the structure shown in Fig. 1-3. One difference is in that each CE has a Controller associated with the CE as shown in Fig. 2-9. It consists of an Interface Controller, the Multifunctional Queue (Q) and a Look up Table (LUT). The LUT contains all the information necessary to communicate with a CE. During system initialization, the LUT is loaded with information about all of the processes that a given CE can execute. It consists of

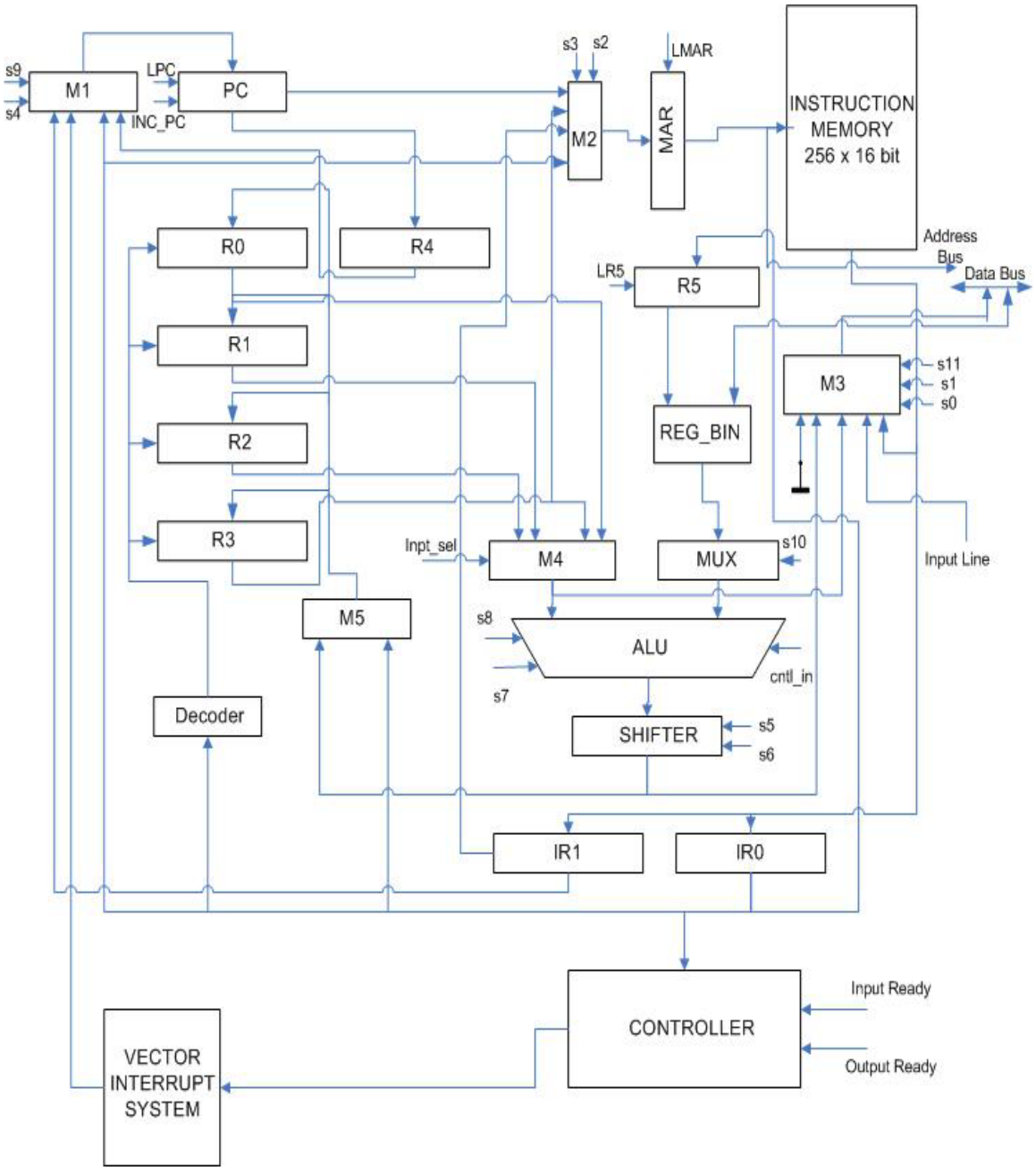



**Figure 2-6 : Memory/Register Computer Architecture of CE0, CE1 and CE4 of HDCA Prototype.**

the process number identifier (PN), the address of the Process Number's first instruction in memory (Instruction Location), follow-on process numbers (PN0, PN1), a hold bit (H) and a join bit (J). Since the only communication between CE's is tokens, any CE must know what the next processes are in order to issue the correct follow-on token. This is why there are follow on process numbers in the LUT. The functionality of Hold and Join bits come into picture when the process flow graph is non-linear, or in other words, has forks and joins as earlier explained. The Hold bit is set to logic one if the follow-on process to be executed is a member of a Join operation. The Join bit when set to logic one indicates that the Process to be run is a join process and thus will have more than one token associated with it in the Queue.

### 2.4.2 The Interface Controller

The Interface Controller of Fig. 2-10 provides the logic to integrate the LUT, the Queue, and the CE. One of the functions of the Interface Controller is to receive Tokens from

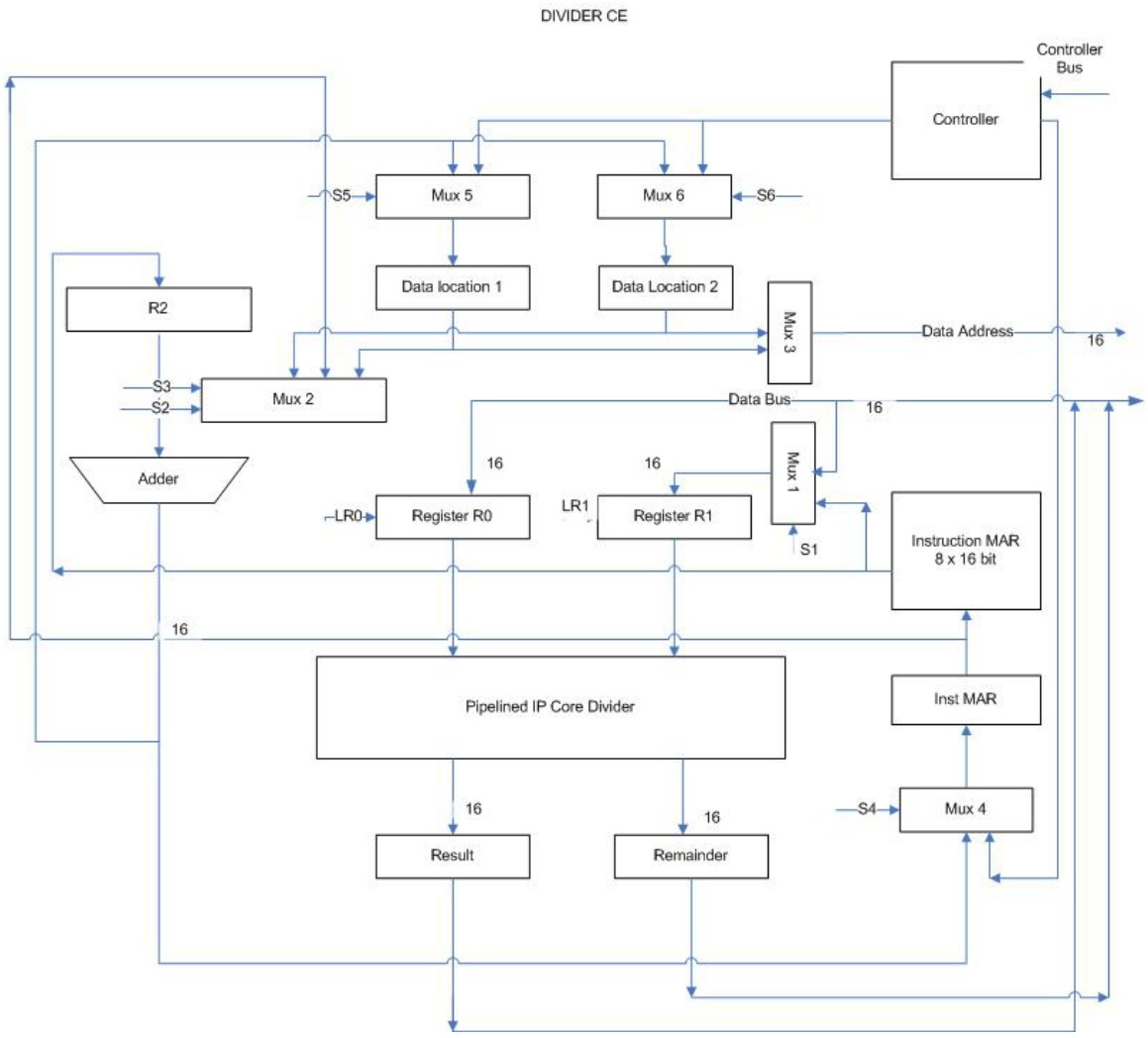



**Figure 2-7: Divider CE Architecture of HDCA Prototype.**

the Token Bus and transmit output Tokens. On the receive side, it has the previously described FIFO buffer to temporarily hold fifteen inbound tokens. Besides this simple task, the Interface Controller is a State Machine for the control of the LUT and Queue. The State Diagram for the Interface Controller is found in Figure 2-10.

The controller starts functioning, as soon as the reset signal goes active low. The first state, after "System Reset" is the "Load Table" state. It remains in this state until the Look up Table described above has all its entries populated. How many entries are needed to fill up the Look up Table depends on the topology of the process flow graph. Once the Look up Table is full, the controller moves to the "Get Token" state. Here the controller waits for properly addressed tokens to arrive from the Token Bus. The first thing the controller checks is if a process previously sent to the CE has completed executing. If it has, and another token is available in the instruction buffers for execution (Next_Loaded is true), then the state switches to Send PRT. If no token is ready for execution, then the state moves to the Dummy Read State. If the CE is still busy executing a process and a token is in the queue, the controller moves to the "De-Queue" state. If none of those conditions are met, then the inbound token is parsed to determine what type of command it contains. The state will then move to "Check Status", "Load Table", "PRAM", or it will loop back to Get Token. The Get Token State is the Default State when the system is waiting for a token arrival or Process completion.

The "De-Queue" state simply removes a token from the Queue and passes it to the LUT. The state then moves to "Get Token" if the CE is busy (not Finished) or to Issue if the CE is ready for another Process (Finished). In the Issue state, the LUT records the last Process executed, if any, and issues a new process to the CE. After issuing the Process, if another token is in the Queue, it will go to the "De-Queue" State to keep the LUTs instruction buffer full. Otherwise, it will go back to the Default State.

The "Dummy Read" state is only used in the case where a Process completes and there is no token available in the instruction buffer to send to the CE. The state allows the

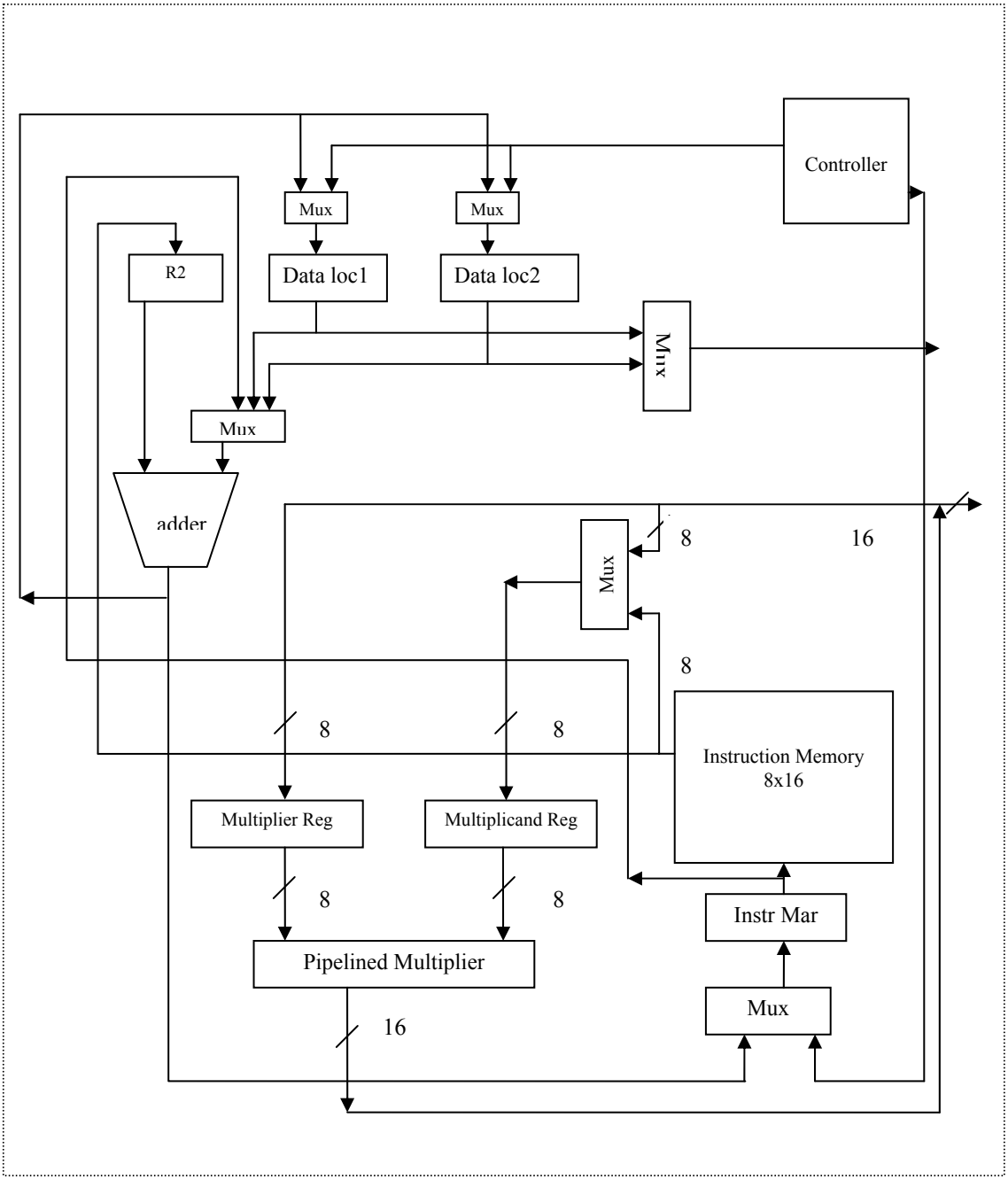



**Figure 2-8: Multiplier CE of 5-CE HDCA Prototype.**

LUT to record the finished Process' information without issuing another Process. This state always transitions to the Send PRT State. In the applications described here, the CEs are fairly efficient and hence the system never goes into this state.

The "Send PRT" state transmits the follow-on tokens of a completed process from the LUT to the Interface Controller. The Interface Controller then negotiates for the Token Bus and submits the tokens to the PRT mapper. Upon completion of the send, if another token is loaded in the LUT's instruction buffer, the state moves to Issue. If a token is not loaded the state returns to the "Get Token" state.

The "Check Status" and "PRAM" states are for the Multifunctional Queue. The PRAM is used to aid in the swap function. The instructions place the Queue in the swap mode and then provide the swap address locations from where tokens are to be swapped. Finally the Queue is removed from the swap mode when this is accomplished.

The HDCA can not start functioning until it has received all the information it needs to start system operation. This information is in essence a set of Tokens.

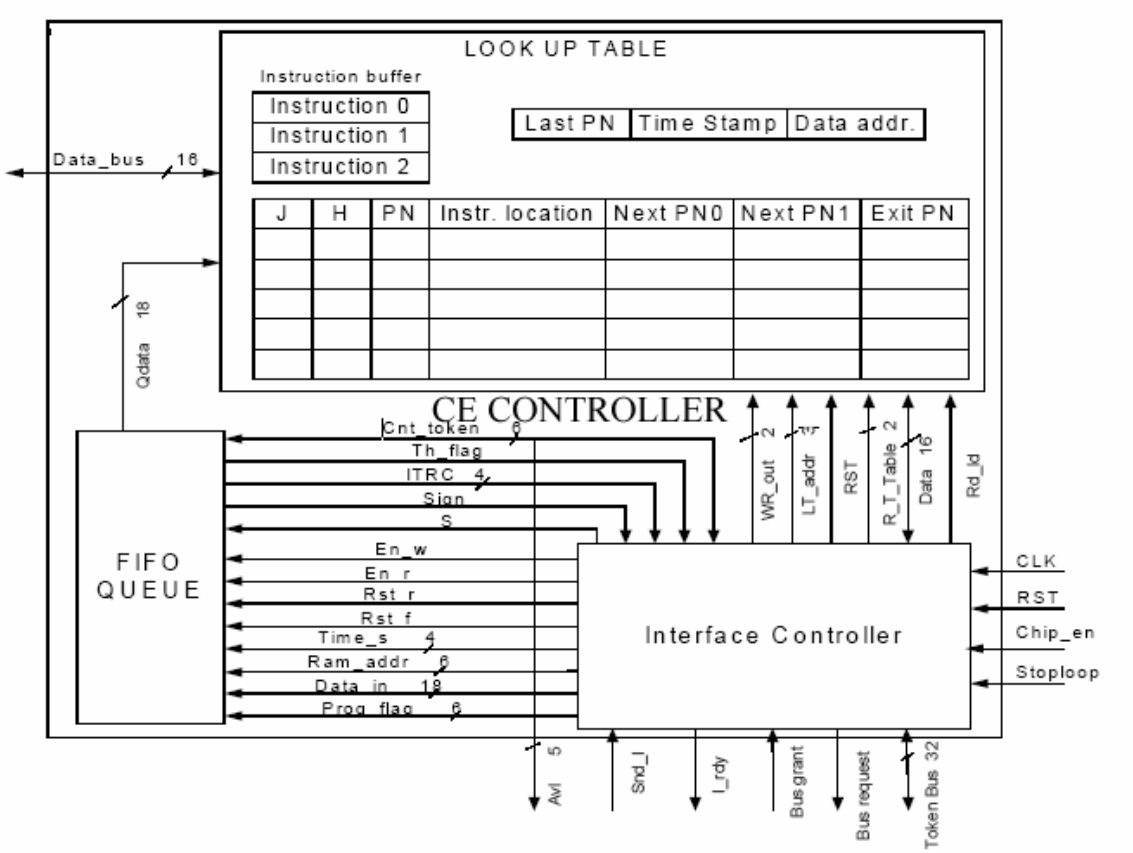



**Figure 2-9: CE Controller Within Each CE of HDCA**

There are different sets of Token formats for the HDCA, each performing a unique function. The token names were chosen sensibly to indicate the function of the token. Table 2.1 below represents the tokens that can be used in the HDCA system. Though not all Token formats are used in the work reported here, some of the Token formats are needed for special functionalities incorporated in the core components that were designed earlier.

Out of the possible token formats in Table 2.1, Token formats a, b, g and h were used for all applications. The "Load Threshold" token was used in the application for demonstrating dynamic node level reconfigurability. The tokens that are used to initialize the Look up Table are the "Table Load" and "Table Input" tokens. These tokens, in essence, contain information about the processes different CEs could possibly execute. They provide information on the current process number, the following process numbers for the successor nodes, the address of the process's first instruction in local memory, a Hold and a Join field. The remaining four tokens are used to access the advanced functionality of the multifunctional queue if required. The "Load Threshold Token" identifies the queue for a CE by the "physical location" of the CE and programs the threshold for the queue and the time period (Time_s) desired for sampling the input and output rate. The "Switch Tokens" token is utilized to swap tokens in the queue by address as previously mentioned in this section. The "Read Status Token" and "Send Status Token" are designed to obtain status information of a queue. The "Read Status Token" is sent by the operating system to a CE directing it to provide status information. The "Send Status Token" is like an "ack" containing the Input Token Rate Change (ITRC) over the specified time, its sign (positive and negative) and a flag to indicate whether or not the threshold has been crossed for the queue. "Load PRT" token is used to initialize the RAM in the PRT mapper upon system startup. It contains information about the physical location (address) of the CE, the process number that CE holds, and the RAM address within PRT to load this information. This token is primarily responsible for starting application execution.

Each CE has a unique address which distinguishes it from the other CEs. The below Table represents the physical addresses of the CEs as used in the current HDCA. These addresses are essential for proper functionality of the token bus with the set of tokens described below in Table 2.1.

| Element | Physical Location |
|---|---|
| PRT mapper | 0000001 |
| CE0(MR16) | 0000011 |
| CE1(MR16) | 0000010 |
| CE2(DIV) | 0000100 |
| CE3(MULT) | 0000101 |
| CE4(STANDBY) | 0000110 |

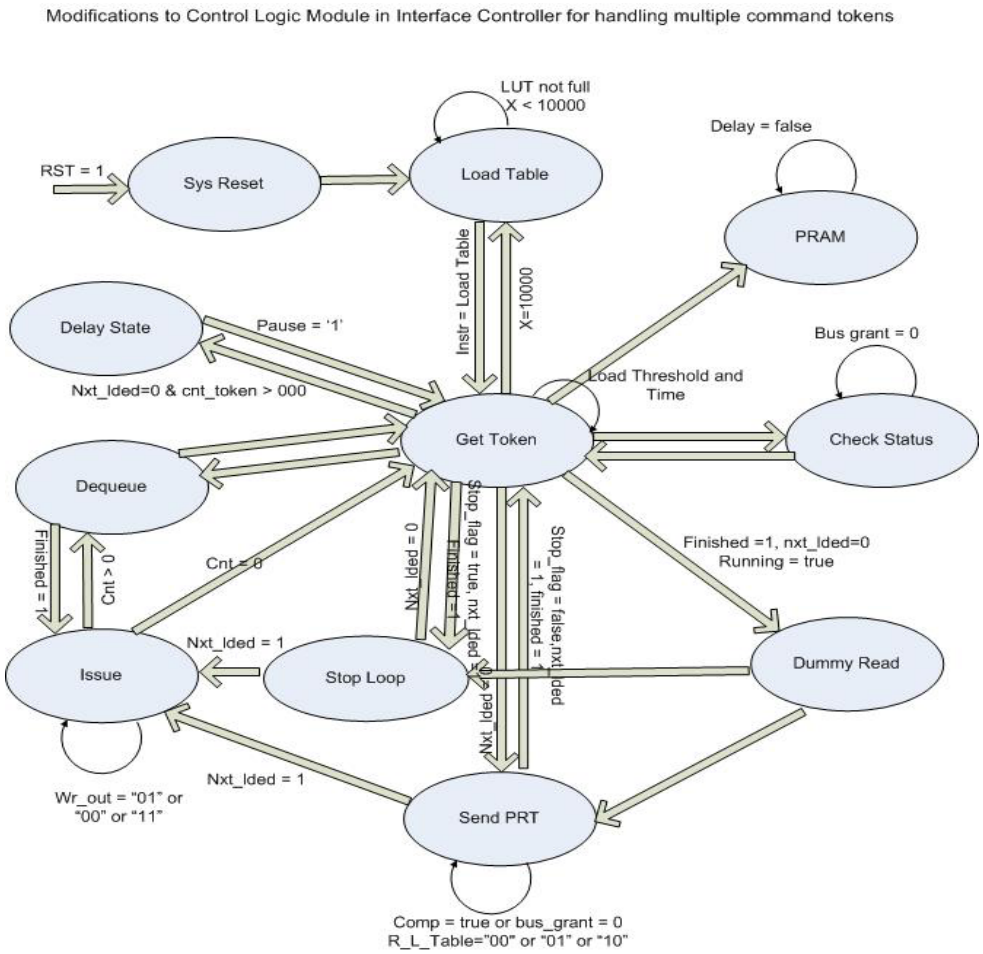



**Figure 2-10: "Interface Controller" of CE Controller (see Fig. 2-9) State Diagram.**

## Table 2.1, Token Formats Available for the HDCA System

### a. Table Load Token

| 1 | Physical Location | 11111 | XXXXXXXXX | Join Field | Hold Field | Instruction Address |
|---|---|---|---|---|---|---|
| 31 | 30 24 | 23 19 | 18 10 | 9 | 8 | 7 0 |

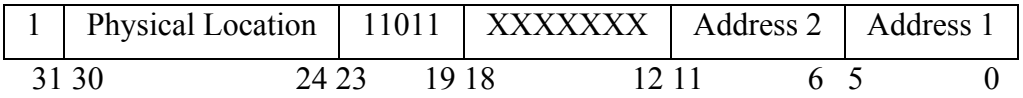
### b. Table Input Token

| 1 | Physical Location | 11110 | Process Number (PN) | Next PN0 | Next PN1 | XXXX |
|---|---|---|---|---|---|---|
| 31 | 30 24 | 23 19 | 18 14 | 13 9 | 8 4 | 3 0 |

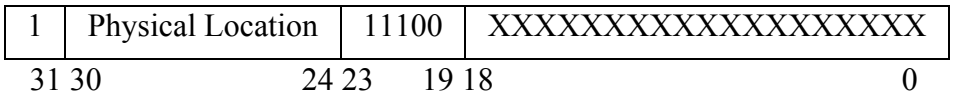
### c. Load Threshold Token

| 1 | Physical Location | 11101 | XXXXXXXXX | Time_S | Threshold |
|---|---|---|---|---|---|
| 31 | 30 24 | 23 19 | 18 10 | 9 6 | 5 0 |

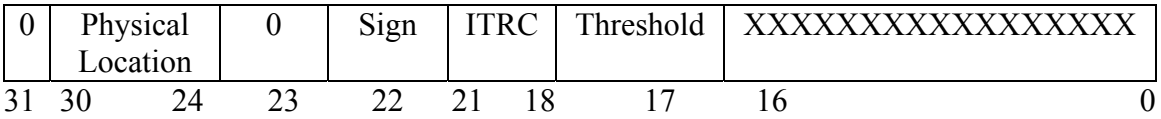
**d. Switch Tokens Token**



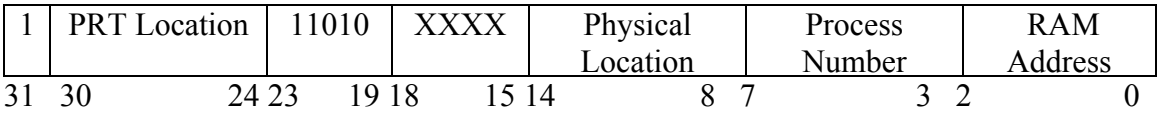

**e. Read Status Token**



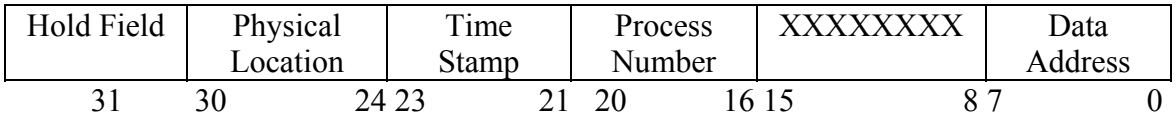

**f. Send Status Token**




**g. Load_PRT Mapper Token**




**h. Command Token**



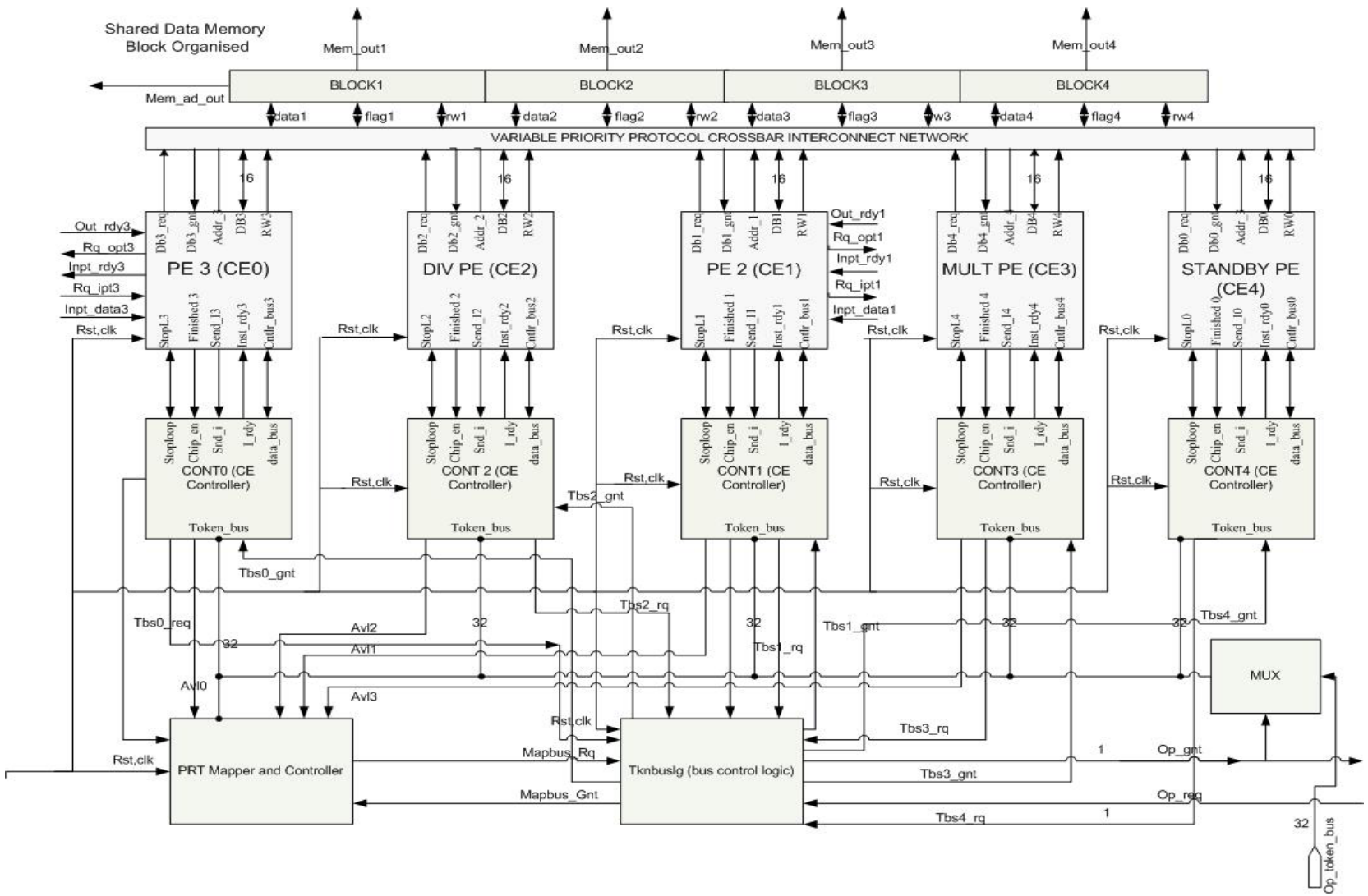



**Figure 2-11: Current 5-CE HDCA System FPGA Based Prototype (Process Request Token (PRT) Mapper is Same as previously described Control Token Mapper (CTM) shown in Fig. 1-4)**

## 3 Synthesis and Implementation of HDCA System Prototype

We have now addressed and shown all major functional units of an HDCA system, which when incorporated into and connected into a correct overall HDCA structure as shown in Fig. 2-11, should allow and implement expanded, new and improved operational and performance functionality into the HDCA system. We next synthesized and implemented a Field Programmable Gate Array (FPGA) based virtual prototype of the HDCA as shown in Fig. 2-11. We utilized Hardware Description Language (HDL) post-implementation functional and performance simulations to evaluate and validate operational, functional and performance characteristics of the newly expanded and functionally enhanced HDCA. We utilized the VHDL HDL to capture the design of the HDCA system of Fig. 2-11.

The HDCA system prototype was synthesized, implemented and simulation tested using Xilinx ISE 6.2.3 ISE CAD software [11] synthesis and implementation tools and the Mentor Graphics ModelSim PE 5.7g version (13) HDL simulator. The host PC was a high performance AMD Athlon processor running Windows XP, 32 bit edition at 2.16 GHz with 2GB of RAM. Input stimuli were added through HDL Bencher within the Xilinx ISE tool set. Timing constraints could also be specified. After synthesis, the HDCA system was implemented to a Xilinx Virtex 2 XC2V8000 FPGA chip. Post-Implementation simulation was then carried out using Modelsim with the test vector sets provided in the Appendices of [16,14] for different applications and after an Input ROM and the CE processor Instruction Memories had been initialized using the Memory Editor tool provided in Xilinx. HDL virtual prototype simulation results were then compared with known correct results in order to validate correct operation and architectural functionality of the HDCA system.

## 4 HDCA System Functionality and Performance Validation via HDL Post-Implementation Simulation (Virtual Prototyping)

Via post-implementation HDL simulation of the HDCA system of Fig. 2-11, we desired to validate the flexible heterogeneous nature and following new functionality of the HDCA by achieving the following goals:

- Structure and implement a heterogeneous HDCA system consisting of more than 3 CEs.
- Demonstrate that an HDCA system can successfully execute applications described by both acyclic and cyclic process flow graphs of any topology *(application-level dynamic/reconfigurability capability).*
- Demonstrate *node-level dynamic/reconfigurability capability.*
- Demonstrate an ability to *fork to two or more successor nodes (processes)* in a process flow graph describing an application.
- Demonstrate that multiple copies of an application can simultaneously execute in parallel on the HDCA, each operating on a different set of data.

HDL post-implementation simulation validation of architectural functionality and performance for a single-chip multiprocessor architecture system such as the HDCA requires sometimes tedious time-consuming evaluation of many complex signal waveform traces. This was the case for the HDCA. Via our post-implementation simulation efforts, we were able to *achieve all above goals* related to *demonstrating that the HDCA is a flexible hybrid single-chip heterogeneous multiprocessor architecture* that now has *node-level dynamic capability, it can execute application process flow graphs that fork to more than two nodes (any number)* and it can *correctly execute applications described by both acyclic and cyclic process flow graphs*. *It can also simultaneously execute multiple copies of the same application with each application executing on a different set of data.* Performance wise, we demonstrated the parallel processing HDCA executes applications much faster as compared to execution of the applications on a sequentially executing single processor version of the architecture (utilization of only one CE at the time to sequentially execute the processes of an application). Example applications included matrix multiplication and computation/evaluation of other recursive mathematical equations and algorithms. The applications required use of all CEs of the HDCA. Achievement of all goals, including showing and explanation of numerous supporting signal waveform traces, is thoroughly documented in [16,14]. We will briefly address and show here two simple applications we developed and then validated their correct execution on the HDCA system of Figure 2.11 via virtual prototyping. The first application is an acyclic pipelined integer manipulation algorithm and the second application is a cyclic non-deterministic value swap application.

### 4.1 Acyclic Pipelined Integer Manipulation Algorithm

Figure 4-1 below shows the acyclic process flow graph for the integer manipulation algorithm. The interest in showing this application is in that the flow graph is acyclic and the implementation of this algorithm utilizes all heterogeneous CEs of the HDCA system of Figure 2-11 (PE2, PE3, DIV PE and MULT PE). Additionally, virtual prototyping of the HDCA executing this application validates correct control, functional, operational and performance operation of the HDCA in executing both single and multiple copies of the application, ie, all processes of the application of Figure 4-1 are correctly executed in a correct order in a parallel manner

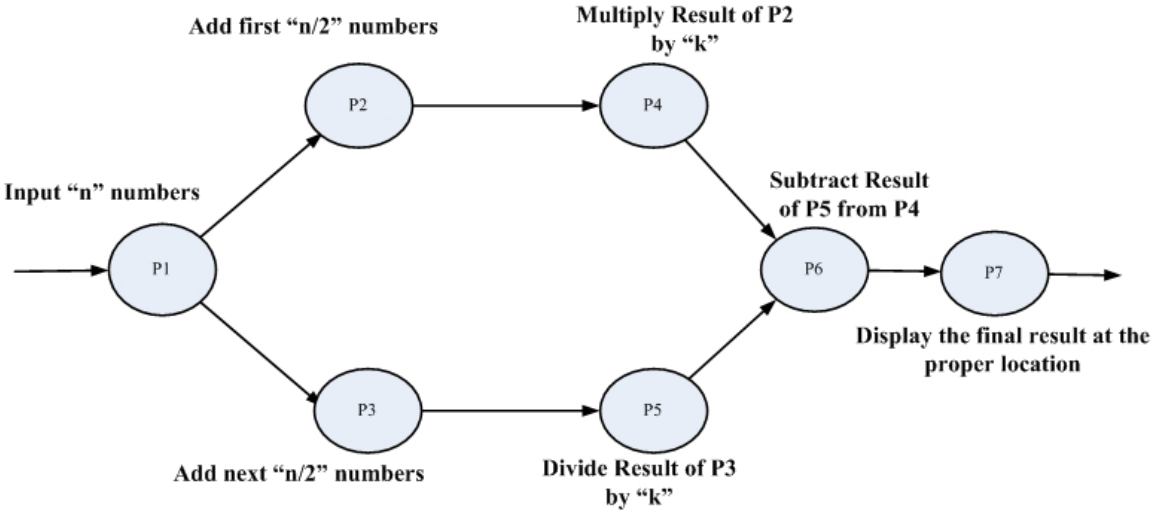



**Figure 4-1. Process Flow Graph for Acyclic Pipelined Integer Manipulation Algorithm.**

on a heterogeneous mix of processors in the single-chip multiprocessor architecture.

The process flow graph for this application is split into the following processes –

P1 – Input "n" numbers into the Shared Data Memory of the HDCA from an Input Data ROM on the PCB of the prototype system.

P2 - Add the first half of these numbers and store the result in the Shared Data Memory.

P3 – Add the remaining numbers in parallel and store the results in the shared data memory.

P4 – Multiply the result of P2 by value "k" stored in the instruction memory of the Multiplier CE.

P5 – Divide the result of P3 by value "k" stored in the instruction memory of the Divider CE.

P6 – Subtract the result of P5 from P4 and store the result in the data memory.

P7 – Display the address and value of the final result calculated in P6.

Via virtual prototyping, we validated correct execution of the above integer manipulation algorithm for all tested cases of n integer numbers by the HDCA of Figure 2-11. We show below for illustrative purposes, for a single copy of this application, two HDL post implementation simulation wave traces to illustrate the level of detail (system clock cycle) at which we validated correct HDCA system control, functionality, operation and performance. The first wave trace (Figure 4-2), shows process P1 running on CE0 of Figure 2-11, inputting the first 5 of 10 (n=10) input integer numbers. Of the several shown system waveforms in Figure 4-2, the one showing the 5 numbers is indicated. The 5 numbers, and eventually all 10 input numbers, are written by process P1 to the shared Data Memory of the HDCA. Many system clock cycles later, and after evaluation of many similar wave traces, one finally can see the result of process P7 appear as indicated on the wave trace of Figure 4-3 for the case of each input number having a value of 2 and (k=2). The indicated address within the shared Data Memory where the final result has been written by process P6 is correct and the value of the final result at that address is correct.

In a similar manner, let us briefly show and address the second application.

## 4.2 Cyclic Non-Deterministic Value Swap Application

Figure 4-4 below shows the process flow graph for the cyclic non-deterministic value swap application. The application is non-deterministic in that the number of times the feedback loops are traversed are not know apriori. This application was developed to show that the HDCA system can execute complex cyclic applications and those involving "while-do" or "if-then" loop structures. This application basically swaps two values over a period of time. The application was further extended to validate the dynamic node level reconfigurability capability of the HDCA by increasing the rate at which data was entering the system and causing the queues at CEs to build up to a *threshold* value making it necessary for an additional standby CE to dynamically configure to prevent system overload and failure.

For the flow graph shown below, a value of x"0A" was chosen for k and values of T2 and T1 were chosen to be x"4C" and x"64" respectively, to keep the application small and provide a working proof of concept. The processes of the flow graph do the following:

P1 – Input 2 values, T1 and T2 with T1>T2.

P2 – Add a value of unsigned '10' to T2 to get a new value of T2.

P3 - Check if T2 =T1 original. If yes, branch to P6 (Exit PN), display both T1 and T2

Else branch to P2 again (feedback loop)

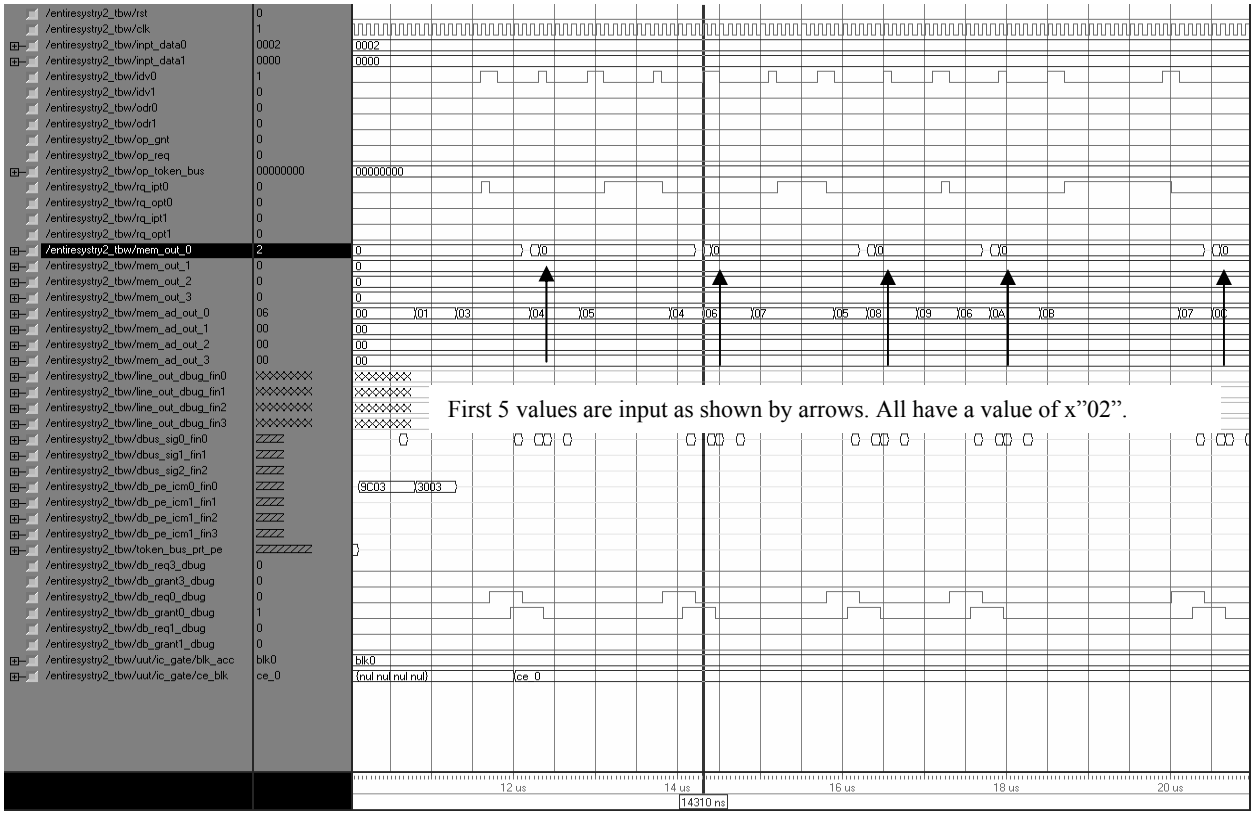



**Figure 4-2. First 5 of 10 Input Numbers Read From a ROM and Written to Shared Data Memory of HDCA by Process P1.**

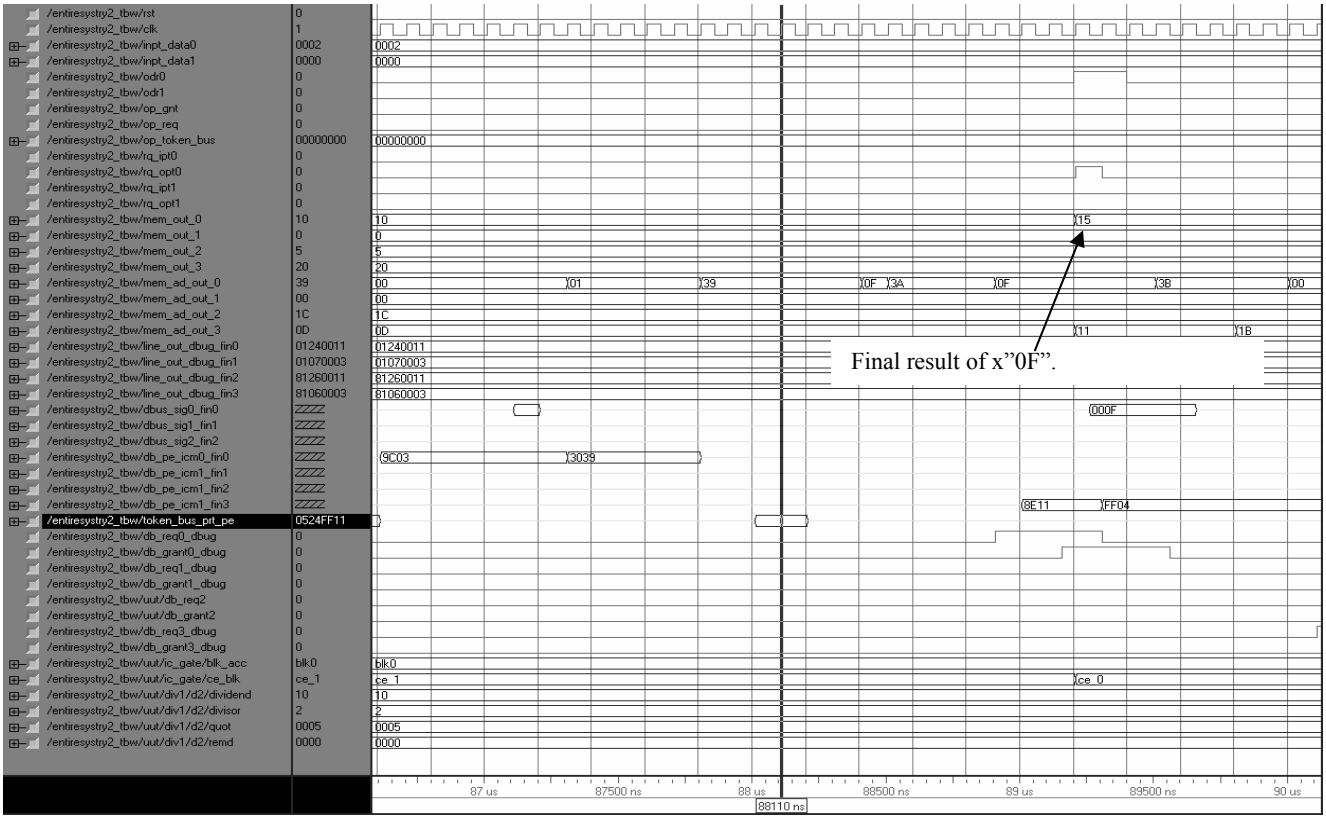



**Figure 4-3. Instruction for P7 and Final Results for Application Displayed**

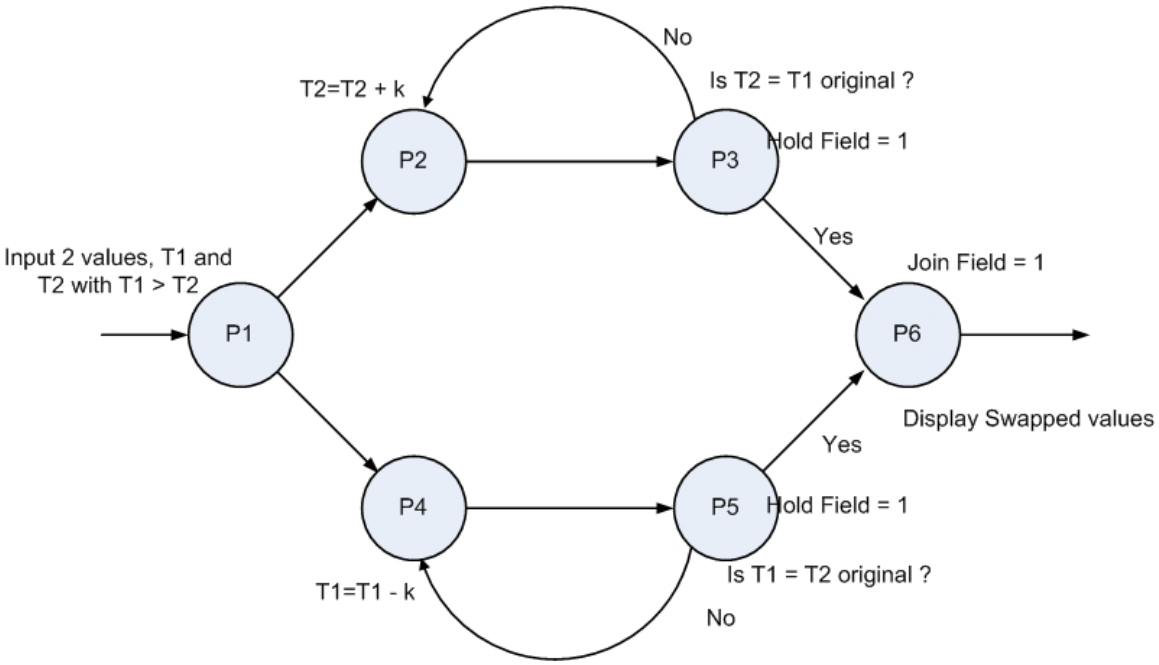



**Figure 4-4. Process Flow Graph for Cyclic Application of Swapping Two Sets of Values**

P4 – Subtract ”0A” from T1 and update T1 to its new value.
P5 – Check if T1=T2 original. If yes, branch to P6 (Exit PN), display both T1 and T2
Else branch to P4 again (feedback loop)
P6 – Display the values of T1 and T2 and then exit.

Post implementation VHDL simulation of the HDCA system executing the application described by Figure 4-4 validated correct execution of the cyclic application by the HDCA for all tested values of T1, T2 and k. Execution of this application with its cyclic topology versus the different and acyclic topology of the application described by Figure 4-1 illustrates the HDCA can correctly execute cyclic and acyclic applications of different topologies. While executing the application of Figure 4-4, the input data rate was increased to a point such that control tokens at various nodes queued to the *threshold* levels set for each node indicating node processor (CE processor executing the process) overload. When this would happen, a stand-by CE processor in the single-chip architecture system was activated to also execute the overloaded process(es), thus demonstrating the node level dynamic capability of the architecture.

Because of manuscript page constraints, we only show Figure 4-5, a post implementation simulation wave trace showing process P1 inputting values for T1 and T2 and Figure 4-6, which shows process P6 outputting T1 and T2 with their values swapped.

In short, we have now briefly illustrated via VHDL post implementation functional and performance simulation that the HDCA can, in a parallel manner, correctly execute applications described by acyclic and cyclic process flow graphs of different topologies and that it can correctly implement its node level dynamic capability resulting in other CE processor(s) being activated on-the-fly to help-out overloaded node CE processors. Post implementation simulation also validated that a HDCA system can simultaneously execute multiple copies of an application, each operating on a different set of data.

## 5 Conclusions and Future Research/Development

The focus of this paper has been the functional development and validation of the single-chip hybrid and heterogeneous multiprocessor HDCA to a point such that it can implement real-time and/or non-real-time applications (assuming proper Operating System support) described by cyclic and acyclic process flow graphs of any topology. Functionality was also added to the HDCA to give it node-level dynamic capability, that is, if a processor (CE) executing a process of an application begins to overload, the architecture has the ability to dynamically on-the-fly initiate execution of additional copie(s) of the process on idle stand-by processors within the single-chip multiprocessor architecture. Once the computational load of the overloaded processor ceases, the helping stand-by processors systematically terminate their help of the overloaded processor and return to an idle state.

In the future, research and development will focus on identification, mapping and testing of more complex and realistic non-real-time and real-time applications on the HDCA. Concurrent with this will be the identification and mapping of required operating system and system software support to the HDCA. An example potential communications

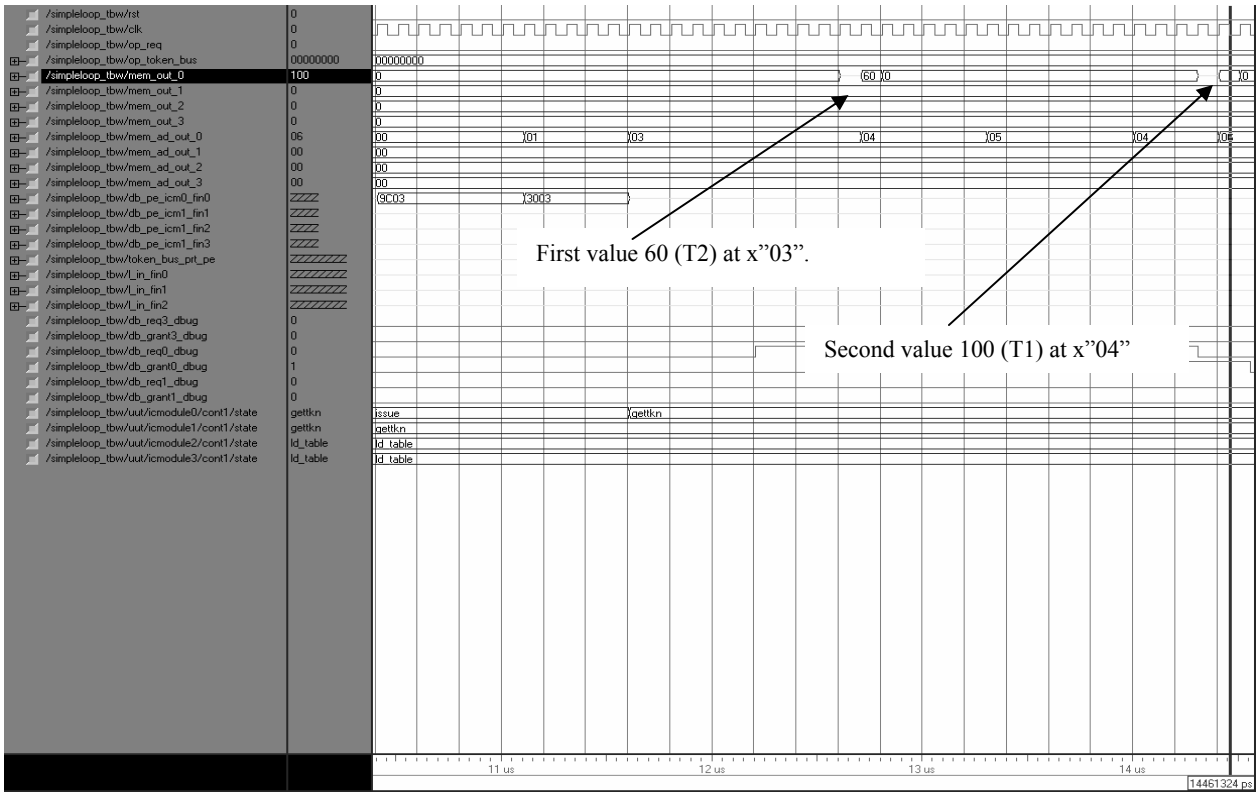



**Figure 4-5. First Values for T1 and T2 Input from Input ROM into the HDCA Shared Data Memory.**

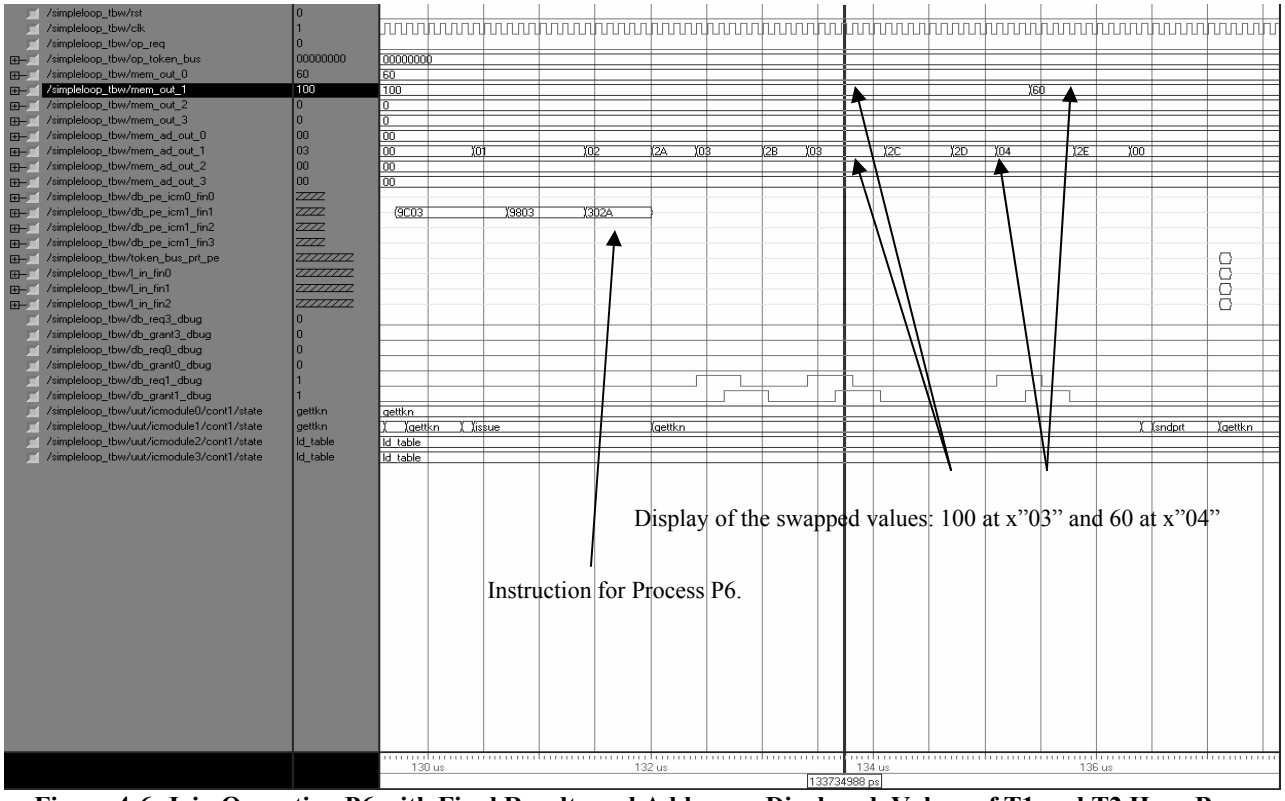



**Figure 4-6. Join Operation P6 with Final Results and Addresses Displayed. Values of T1 and T2 Have Been Swapped.**

oriented (packet driven) application is Ephemeral State Processing as addressed in [17]. Another area of investigation is implementation of *processor architecture dynamic capability* as earlier addressed in the paper.

## References


1. Heath, J.R, Ramamoorthy, S, Stroud, C.E, and Hurt, A.D, "Modeling, Design, and Performance Analysis of a Parallel Hybrid Data/Command Driven Architecture System and Its Scalable Dynamic Load Balancing Circuit", *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 44, no 1, pp. 22-40, Jan. 1997.
2. Heath, J.R, and Balasubramanian, S, "Development, Analysis, and Verification of a Parallel Dataflow Computer Architectural Framework and Associated Load-Balancing Strategies and Algorithms via Parallel Simulation", *SIMULATION*, vol. 69, no.1, pp. 7-25, July.1997.
3. Fernando, U. Chameera R., "Modeling, Design, Prototype Synthesis and Experimental Testing of a Dynamic Load Balancing Circuit for a Parallel Hybrid Data/Command Driven Architecture", *Masters Project Report,* Department of Electrical Engineering, University of Kentucky, Lexington, KY, December, 1999.
4. Heath, J.R. and Tan, A. "Modeling, Design, Virtual and Physical Prototyping, Testing, and Verification of a Multifunctional Processor Queue for a Single-Chip Multiprocessor Architecture", *Proceedings of the 2001 IEEE International Workshop on Rapid Systems Prototyping,* Monterey, California, 6 pps. June 25-27, 2001.
5. Maxwell, P., "Design Enhancement, Synthesis, and Field Programmable Gate Array Post-Implementation Simulation Verification of the Hybrid Data/Command Driven Architecture", *Master's Project Report*, Department of Electrical Engineering, University of Kentucky, Lexington, KY, May, 2001, (Available at: http://www.engr.uky.edu/~heath/Masters_Proj_Report_Paul_Maxwell.pdf).
6. Xiaohui Zhao, "Hardware Description Language Simulation and Experimental Hardware Prototype Validation of a First-Phase Prototype of a Hybrid Data/Command Driven Multiprocessor Architecture", *Master's Project Report,* Department of Electrical and Computer Engineering, University of Kentucky, Lexington, KY, May, 2002, (Available at: http://www.engr.uky.edu/~heath/Masters_Proj_Report_Xiaohui_Zhao.pdf).
7. Venugopal Duvvuri, "Design, Development, and Simulation/Experimental Validation of a Crossbar Interconnect Network for a Single-Chip Shared Memory Multiprocessor Architecture", *Master's Project Report,* Department of Electrical and Computer Engineering, University of Kentucky, Lexington, KY, June, 2002, (Available at: http://www.engr.uky.edu/~heath/Masters_Proj_Report_Venugopal_Duvvuri.pdf).
8. http://www.gup.uni-linz.ac.at/thesis/diploma/christian_schaubschlaeger/html/chapter02a5.html.
9. George Broomell and J. Robert Heath, "Classification Categories and Historical Development of Circuit Switching Topologies", *ACM Computing Surveys*, Vol.15, No.2, pp. 95-133, June 1983.
10. M. L. Bos, "Design of a Chip Set for a Parallel Computer based on the Crossbar Interconnection Principle", *Proceedings Circuits and Systems, 1995, Proceedings of the 38th Midwest Symposium,* Vol.2, pp. 752-756, 1996.
11. Xilinx ISE 6.2i CAD Tool Set, www.xilinx.com.
12. Datasheets for Virtex 2 Series Devices, www.xilinx.com, www.digilent.com.
13. Mentor Graphics ModelSim PE 5.7g HDL Simulation Tool Set.
14. Kanchan P. Bhide, "Design Enhancement and Integration of a Processor-Memory Interconnect Network into a Single-Chip Multiprocessor Architecture", *Masters Thesis,* Dept. of Electrical and Computer Engineering, Univ. of Kentucky, Lexington, KY, Dec. 2004. (See http://www.engr.uky.edu/~heath/m_thesis.htm)
15. Venugopal Duvvuri, J. Robert Heath, Kanchan Bhide and Sridhar Hegde, "A New Processor-to-Memory Crossbar Interconnect Network with a Variable Priority Memory Contention Resolution Protocol for Multiprocessor Architectures", *Proceedings of the 2005 International Conference on Information Systems: New Generations,* Las Vegas, NV, 6pps, April 4-6, 2005.
16. Sridhar Hegde, "Functional Enhancement and Applications Development for a Hybrid, Heterogeneous Single-Chip Multiprocessor Architecture", *Masters Thesis,* Dept. of Electrical and Computer Engineering, Univ. of Kentucky, Lexington, KY, Dec. 2004. (See http://www.engr.uky.edu/~heath/m_thesis.htm)
17. M. Muthulakshmi, J. Robert Heath, Kenneth L. Calvert, and James Griffioen, "ESP: A Flexible, High-Performance, PLD-Based Network Service", *Proceedings of the 2004 IEEE International Conference on Communications,* Paris, FRANCE, 5 pps, June 20-24, 2004.