Implementation of a CC-NUMA on RPM

Jaeheon Jeong, Yong Ho Song, Adrian Moga
and Michel Dubois

CENG 97-27

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2562
(213)740-4475
Fax: (213) 740-7290
{jaeheonj, yongho, moga, dubois}@paris.usc.edu

December 1997

Implementation of a CC-NUMA on RPM

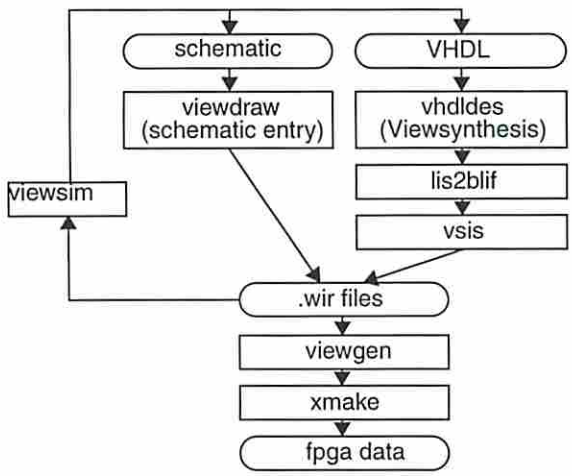
This document describes in details the implementation of the four controllers of RPM-2 for a CC-NUMA architecture: the memory/directory controller, the second-level cache controller, the network interface controller and the first-level cache controller. The overall description of the architecture of RPM-2 and of the CC-NUMA organization as well as the physical design of RPM-2 can be found in [4] and [5]. More deatails are available in [1, 2, 3]. Each controller of RPM is made of several FPGAs. During the course of the project two sets of tools have been used. First we introduce the tools, which will be referred to in the rest of the report.

1. RPM Programming Methodology

At the early stage of the RPM project, we had two tools to compile and map designs to Xilinx FPGAs. Viewsynthesis 2.2.1 provided by Viewlogic was our synthesis tool and the Xact 5.0 tool from Xilinx was then used to map the synthesized designs to FPGAs. Since Viewsynthesis did not support an interface to the Xilinx FPGA design library, we could not express the architecture dependent part of the design such as I/O pads, tri-state buffers and FPGA dedicated modules in VHDL. Therefore the architecture dependent part of the design was captured by schematic entry using the Viewlogic schematic entry tool and the architecture independent part of the design was written in VHDL. Both parts were merged in a top-level schematic. VHDL codes based on Viewsynthesis are not compatible with other tools since Viewsynthesis uses a variant of VHDL. We were unable to perform precise timing simulation in an efficient manner since the tool could not generate exact timing data for target devices.

As the designs were getting more complex, Viewsynthesis produced poorer designs. The next state logic was too deep and, as a result, the mapped designs failed to run at the target clock speed. To improve the quality of the synthesized designs, we decided to use the Berkeley SIS tool and developed an interface tool to convert the intermediate file produced by Viewsynthesis into bliff format accepted by the SIS tool. We refer to SIS+its interface as VSIS. With the VSIS tool, performance improved significantly. For example, the maximum delay of the control part of the SLC was improved from 148 ns to 130 ns. Figure 1 illustrates the design process using the original set of tools.

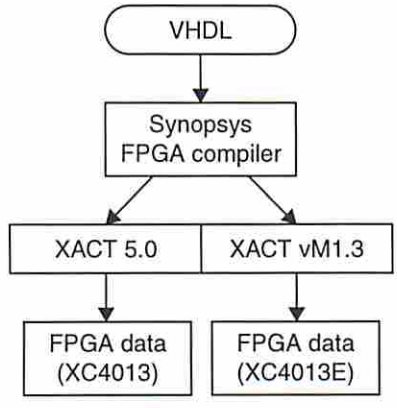
Figure 1. FPGA Design Process using Viewsynthesis and VSIS



After we completed the FPGA designs on v1994.2 PCBs [4], we obtained a set of new synthesis tools from Synopsys to replace Viewsynthesis and VSIS. Because the Synopsys tools come with XSI (Xilinx Synopsys Interface) which allows us to implement Xilinx FPGA designs with the Synopsys FPGA Compiler, designs are no longer divided into schematic entries and VHDL codes. The Network Interface Chip (NIC) in v1997 PCB has been implemented using the Synopsys tool. As indicated earlier, the old designs cannot be compiled directly with the Synopsys compiler. Therefore, we had to convert the schematic entries into VHDL codes and translate the codes of submodules written in Viewsynthesis VHDL into Synopsys VHDL codes.

Based on the FPGA performance statistics of our initial mapping, our target clock speed and additional features required in the future, we selected two different speed grades of Xilinx XC4013 FPGAs, XC4013-5 for the control units of FLC and SLC, and XC4013-6 for others. In the v1997 PCBs, we have added a NIC and purchased 20 additional XC4013E-3 which is an improved and downward compatible version of XC4013. The same bitstream can be used for both XC4013 and XC4013E. XC4013 consists of 576 CLBs (Combinational logic block) and 192 IO blocks, is equivalent to 13,000 gates and has 1,536 usable flip-flops. Currently each board is equipped two XC4013E-3, two XC4013-5 and four XC4013-6. For XC4013E, we optionally use the XACT vM1.3 package which provides better timing analysis and enhanced place and route. Figure 2 shows the new FPGA design process.

Figure 2. FPGA design process with the Synopsys Tools



The rest of the report is structured in four parts. The first part specifies the cache protocol of the CC-NUMA and the implementation of the memory controller. Then the descriptions of the second-level cache controller (SLCC), the NIC controller, and the first-level cache controller (FLCC) follow. We give statistics for every FPGA.

2. Cache protocol and Memory/Directory controller specification

2.1. Introduction

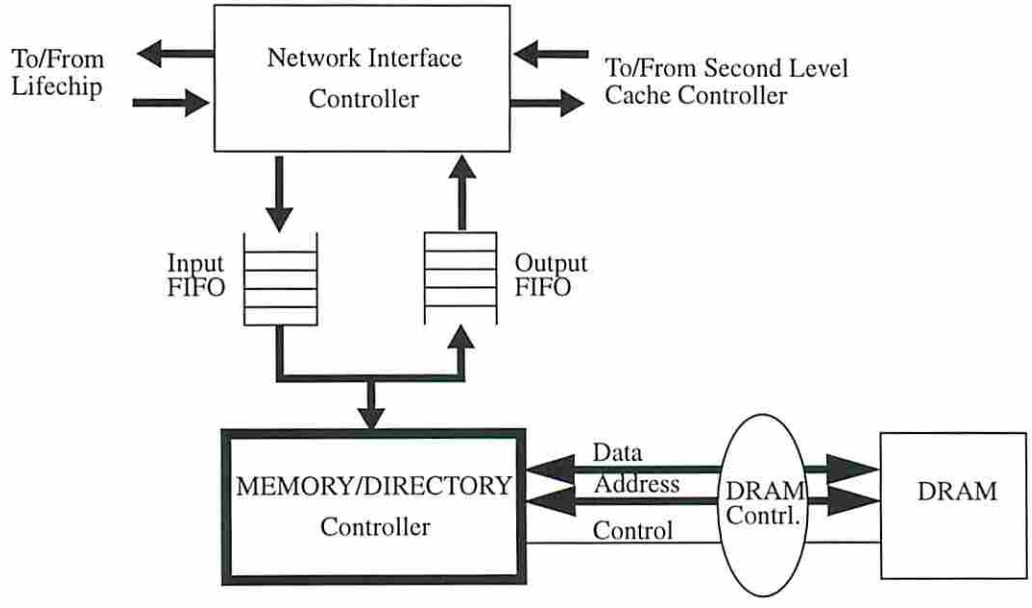
This section documents the specifications for the CC-NUMA memory/directory controller. It is a revision for the second-generation boards of RPM-2. It contains the high-level specification of the memory/directory controller FPGA to support a write-invalidate directory-based cache coherence protocol. It also specifies all test mode transactions, the hardware scheme to simulate multiple interleaved memory banks and the support for performance measurements.

2.2. The Memory/Directory controller architecture

The memory/directory controller is implemented with two FPGAs, and it is involved in all system actions requiring an access to a memory block or location. It communicates with the rest of the board and the system through the Network Interface Controller (NIC) through two pairs of FIFOs. It is also connected to a DRAM bank through a DRAM/ECC controller chip. A simplified diagram of the controller is shown in Figure 3. The DRAM controller relieves the FPGA from all functions necessary to manage a DRAM bank, including generation of row and column addresses, RAS and CAS timings, ECC checking and generation, and refresh signals.

The memory/directory can receive two types of requests: *coherence* (or emulated) requests or *test mode* requests. Since we emulate a directory-based cache coherence protocol, a coherence request starts by fetching the directory entry of the block and may trigger some coherence actions before the block is returned. A test mode request accesses the memory directly, bypassing the directory and the cache coherence protocol; it is used for testing, debugging, and initial downloading of code and data. Both directory and emulated main memory reside in the same DRAM bank at different offsets.

Figure 3. Memory/Directory Controller



2.3. Specification of the write-invalidate cache coherence protocol

We now describe the memory/directory controller actions needed to implement a full-map directory based protocol. In the present scheme a coherence transaction is first routed to the home node (the memory/directory controller of the board to which a given memory block maps to) which examines the contents of a directory entry, takes any appropriate actions to enforce a consistent view of the shared memory space, and replies to the sender. In some cases it is necessary for the directory/memory controller to send *secondary requests* to other caches before responding to the initial requester. The memory/directory controller is free to accept new requests while waiting for responses to secondary requests, provided that they are not to a block for which a coherence transaction is currently pending, in which case the new request is rejected and a negative acknowledgment is sent. When a coherence transaction is pending in the memory/directory controller we say that the directory entry for that block is *locked*.

Each directory entry contains the following information fields:

presence bit vector	10 bits
dirty bit	1 bit
locked bit	1 bit
lock type	2 bits
requester id	4 bits

In the description of the memory/directory controller actions for the write-invalidate cache protocol the following notation is adopted:

pbit(proc)	: value of the presence bit for processor "proc"
dbit	: value of the dirty bit
dirty	: 4-bit id of the node with the dirty copy of the block
req	: 4-bit id of the node that has sent a message to the memory/directory controller

In addition to that, the following shorthand notation is used to specify boolean conditions:

other_shared	: there is at least one 'x' | x != req & pbit(x) = 1
one_left	: there is only one 'x' | pbit(x) = 1

For the write-invalidate cache coherence protocol, a block frame in memory can be in one of the following states:

STATE	Situation
UNCACHED	for all 'x', pbit(x)=0
SHARED	there is at least one 'x' | pbit(x)=1 & dbit=0
DIRTY	there is only one 'x' | pbit(x)=1 & dbit=1
Sh_Dty_own	locked bit = 1, lock type = 00
Sh_Dty_miss	locked bit = 1, lock type = 01
Dty_Sh	locked bit = 1, lock type = 10
Dty_Dty	locked bit = 1, lock type = 11

Below is a listing of all the messages that are the memory/directory controller can receive or send:

Input messages:

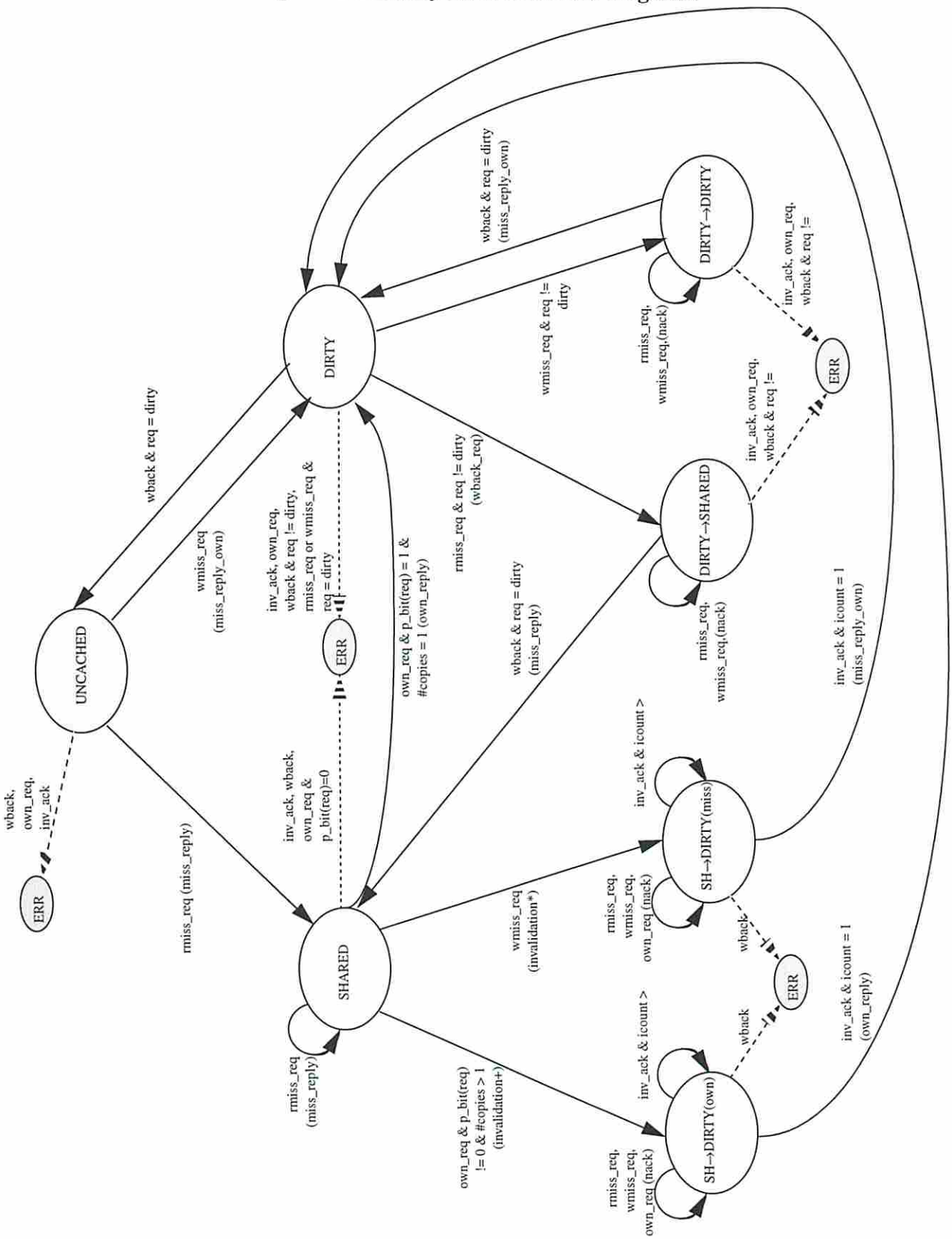
rmiss_req	: read miss request
wmiss_req	: write miss request (ownership)
own_req	: write on a clean block, i.e., request for ownership
inv_ack	: acknowledgment of an invalidation
wback	: write back message from the dirty processor
wword_req	: request to write a word (test mode)
rword_req	: request to read a word (test mode)
rblock_req	: request to read a block (test mode)
wblock_req	: request to write a block (test mode)

Output messages:

miss_reply	: reply for a read miss request
miss_reply_own	: reply for a write miss request
own_reply	: reply for an ownership request
invalidation	: secondary invalidation for all SHARED copies
wback_req	: secondary request for valid copy (dirty goes to RO)
wback_req_own	: secondary request for exclusive copy (dirty goes to INV)
nack	: negative acknowledgment; try later
rword_reply	: reply to a rword_req (test mode)
rblock_reply	: reply to a rblock_req (test mode)

Figure 4 shows the behavior of the memory/directory controller for the write-invalidate protocol as a state diagram. Tables 1 to 11 also specify the memory/directory controller behavior both as state transition tables and inverted tables.

Figure 4. Memory state transition diagram.



State Transition Tables

Table 1: State=UNCACHED

input	conditions	next state	actions	output
rmiss_req		SHARED	pbit(req)=1	miss_reply
wmiss_req		DIRTY	pbit(req)=1 & dbit=1	miss_reply_own
own_req		ERR		
inv_ack		ERR		
wback		ERR		

Table 2: State=SHARED

input	conditions	next state	actions	output
rmiss_req		SHARED	pbit(req)=1	miss_reply
wmiss_req	~(other_shared) other_shared	DIRTY Sh_Dty_miss	pbit(req)=1 & dbit=1 pbit(req)=0 & req_id=req	miss_reply_own forall x|pbit(x)=1 invalidate(x)
own_req	pbit(req)=0 pbit(req)=1 & ~(other_shared) pbit(req)=1 & other_shared	ERR DIRTY Sh_Dty_own	 pbit(req)=1 & dbit=1 pbit(req)=0 & req_id=req	 own_reply forall x|pbit(x)=1 invalidate(x)
inv_ack		ERR		
wback		ERR		

Table 3: State=DIRTY

input	conditions	next state	actions	output
rmiss_req	req=dirty req != dirty	ERR Dty_Sh	req_id = req	wback_req
wmiss_req	req=dirty req != dirty	ERR Dty_Dty	req_id = req	wback_req_own
own_req		ERR		
inv_ack		ERR		
wback	req = dirty req != dirty	UNCACHED ERR	pbit(req)=0 & dbit=0	

Table 4: State=Sh_Dty_own

input	conditions	next state	actions	output
rmiss_req		Sh_Dty_own		nack
wmiss_req		Sh_Dty_own		nack
own_req		Sh_Dty_own		nack
inv_ack	one_left ~(one_left)	DIRTY Sh_Dty_own	pbit(req_id)=1 & dbit=1 pbit(req) = 0	own_reply
wback		ERR		

Table 5: State=Sh_Dty_miss

input	conditions	next state	actions	output
rmiss_req		Sh_Dty_miss		nack
wmiss_req		Sh_Dty_miss		nack
own_req		Sh_Dty_miss		nack
inv_ack	one_left ~(one_left)	DIRTY Sh_Dty_miss	pbit(req_id)=1 & dbit=1 pbit(req) = 0	miss_reply_own
wback		ERR		

Table 6: State=Dty_Sh

input	conditions	next state	actions	output
rmiss_req	req != dirty req = dirty	Dty_Sh ERR		nack
wmiss_req	req != dirty req = dirty	Dty_Sh ERR		nack
own_req		ERR		
inv_ack		ERR		
wback	req = dirty req != dirty	SHARED ERR	dbit=0 & pbit(req_id) =1	miss_reply

Table 7: State=Dty_Dty

input	conditions	next state	actions	output
rmiss_req	req != dirty req = dirty	Dty_Dty ERR		nack
wmiss_req	req != dirty req = dirty	Dty_Dty ERR		nack
own_req		ERR		
inv_ack		ERR		
wback	req = dirty req != dirty	DIRTY ERR	pbit(dirty) = 0 & pbit(req_id) =1	miss_reply_own

Inverted Tables

Table 8: input message = rmiss_req

state	conditions	next state	actions	output
UNCACHED		SHARED	pbit(req) = 1	miss_reply
SHARED		SHARED	pbit(req) = 1	miss_reply
DIRTY	req=dirty req != dirty	ERR Dty_Sh	req_id = req	wback_req
Sh_Dty_own		Sh_Dty_own		nack
Sh_Dty_miss		Sh_Dty_miss		nack
Dty_Sh	req != dirty req = dirty	Dty_Sh ERR		nack
Dty_Dty	req != dirty req = dirty	Dty_Dty ERR		nack

Table 9: input message = wmiss_req

state	conditions	next state	actions	output
UNCACHED		DIRTY	pbit(req) = 1 & dbit = 1	miss_reply_own
SHARED	~(other_shared) other_shared	DIRTY Sh_Dty_miss	pbit(req)=1 & dbit=1 pbit(req)=0 & req_id=req	miss_reply_own forall x|pbit(x)=1 invalidate(x)
DIRTY	req=dirty req != dirty	ERR Dty_Dty	req_id = req	wback_req_own
Sh_Dty_own		Sh_Dty_own		nack
Sh_Dty_miss		Sh_Dty_miss		nack
Dty_Sh	req != dirty req = dirty	Dty_Sh ERR		nack
Dty_Dty	req != dirty req = dirty	Dty_Dty ERR		nack

Table 10: input message = own_req

state	conditions	next state	actions	output
UNCACHED		ERR		
SHARED	pbit(req)=0 pbit(req)=1 & ~(other_shared) pbit(req)=1 & other_shared	ERR DIRTY Sh_Dty_own	pbit(req)=1 & dbit=1 pbit(req)=0 & req_id=req	own_reply forall x|pbit(x)=1 invalidate(x)
DIRTY		ERR		
Sh_Dty_own		Sh_Dty_own		nack
Sh_Dty_miss		Sh_Dty_miss		nack
Dty_Sh		ERR		
Dty_Dty		ERR		

Table 11: input message = wback

state	conditions	next state	actions	output
UNCACHED		ERR		
SHARED		ERR		
DIRTY	req = dirty req != dirty	UNCACHED ERR	pbit(req)=0 & dbit=0	
Sh_Dty_own		ERR		
Sh_Dty_miss		ERR		
Dty_Sh	req = dirty req != dirty	SHARED ERR	dbit=0 & pbit(req_id) =1	miss_reply
Dty_Dty	req = dirty req != dirty	DIRTY ERR	pbit(dirty) = 0 & pbit(req_id) =1	miss_reply_own

2.4. Memory/Directory controller flowcharts and memory interleaving scheme

Here we specify the behavior of the memory/directory controller for each type of input message in terms of control flowcharts. We also describe the mechanism to emulate multiple interleaved DRAM banks. We first explain the main control loop and the hardware resources needed to implement the memory interleaving mechanism. After that we show the control flowcharts for each type of input message.

2.4.1. The main control loop

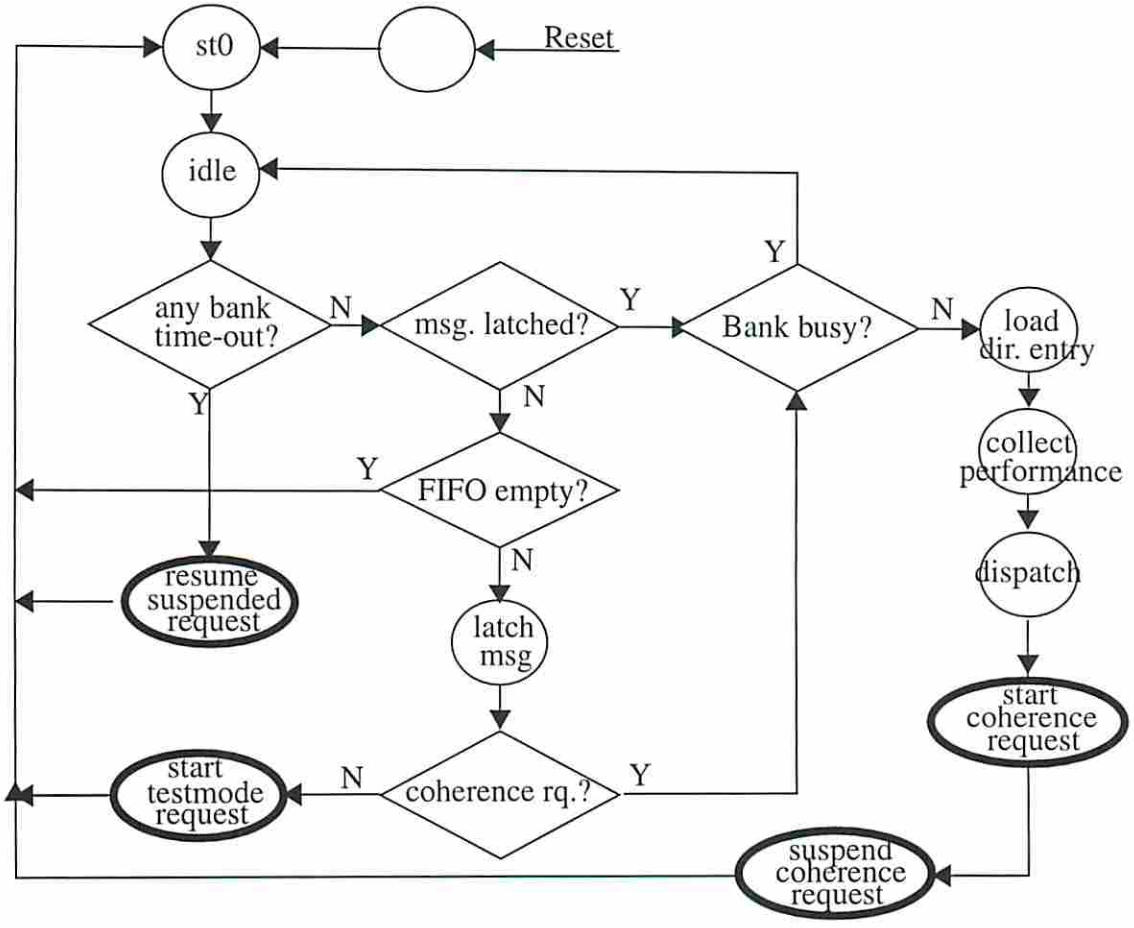
Figure 5 shows the main loop of the memory/directory controller. Requests are suspended as explained in the next section. Priority is given to requests that were suspended and are keeping a given emulated memory bank busy. At the beginning of the loop the controller first checks if the time-out value (time during which a bank is busy) for any bank has expired. The controller only accepts new requests if there are no suspended requests ready to proceed. To accept new requests it first checks whether there is a request already latched (only the header) in the input buffer and tries to execute it. If there are none it tries to read the next request from the input queue. After the header of a message is latched in the input queue the controller starts executing the request if the corresponding memory bank is free, otherwise the new request is blocked in the input buffer.

The four elipses in Figure 5 represents other subdiagrams. The flowcharts for starting and suspending all types of requests are shown later in this section. The flowchart for "resume suspended request" is shown in Figure 6.

As shown in Figure 6, there are three classes of suspended requests that have to be treated differently when resumed. In the case of miss_reply and miss_reply_own the block itself is fetched from memory, using the address contained in the suspended header, and appended to the header to form the full message. For own_reply, wback_req and wback_req_own, only the header is sent, and no further actions are necessary. For an invalidation request, the directory entry is fetched from memory so that (potentially) multiple invalidation messages can be sent. Finally, some requests do not send any messages, as is the case for a wback_req caused by replacement of a dirty block or when an invalidation_ack is received.

We assume here that requests that are nack'ed are not suspended. Furthermore, all test mode messages bypass the interleaving mechanism.

Figure 5. Memory controller: main flowchart

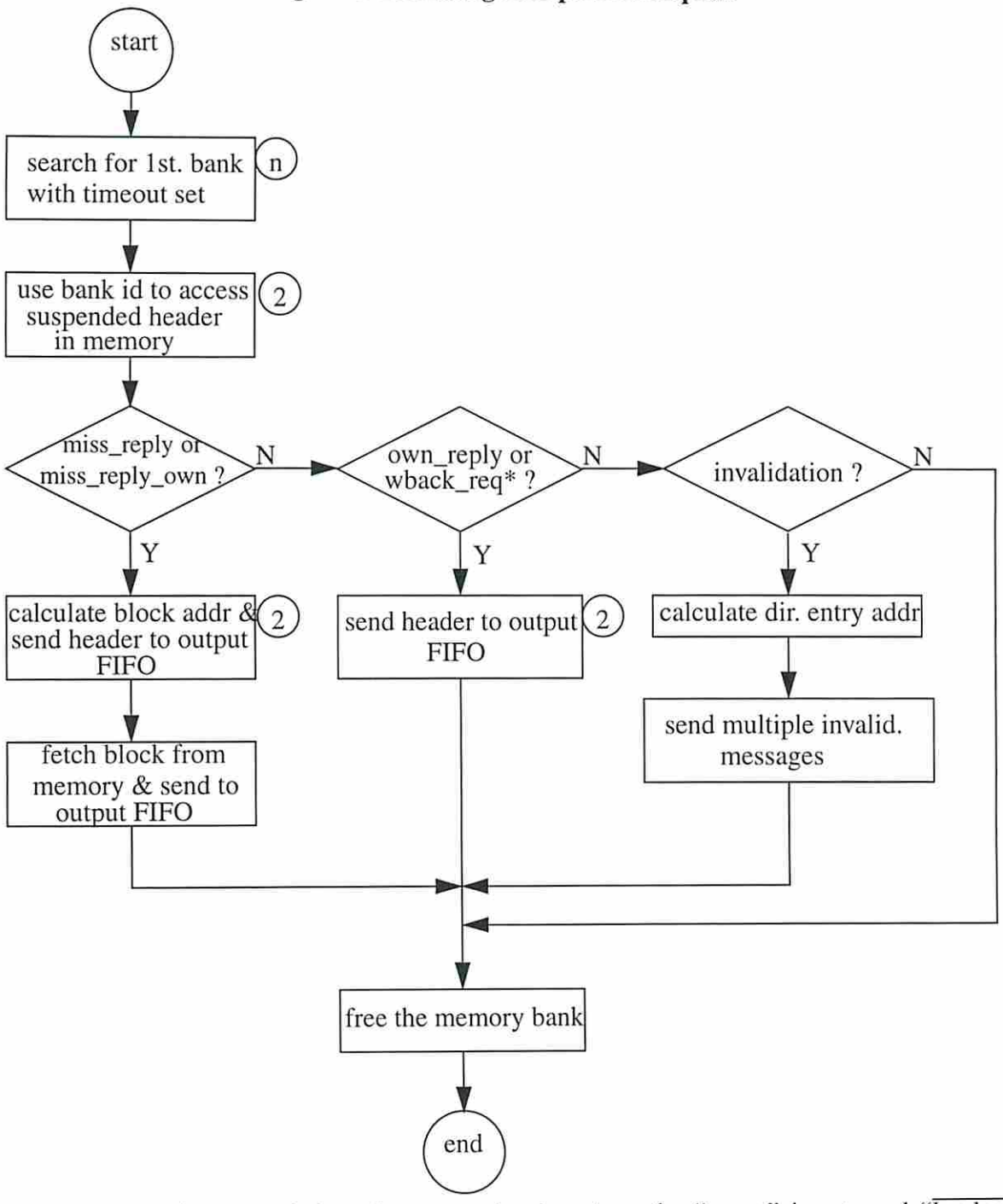


2.4.2. Hardware resources to simulate memory interleaving

The idea is to suspend every request that requires a non-negligible service time from the directory/ memory controller on the target machine for as much time (in pclocks) as they would take on the target machine. Multiple requests may be suspended at the same time provided that they map to different memory banks. When an incoming request maps to a bank that is currently busy it is buffered and blocked in the controller until the corresponding bank is freed. The request is suspended right before it is about to complete, which in most cases mean right before a reply message or a secondary request has to be sent. Therefore, the only information that needs to be kept to resume a suspended request consists of 2 32-bit words for a message header. The present scheme assumes that there is a single input buffer for all interleaved memory banks, therefore if the message in the input buffer is directed to a busy bank all other messages in the incoming FIFO are backed up.

For each of the emulated interleaved memory banks it is necessary to include a count-down register to keep track of the elapsed time that the bank should be busy. An extra flip-flop is included to indicate whether the bank is busy or not. Figure 7 shows the hardware scheme in detail for one bank.

Figure 6. Resuming a suspended request

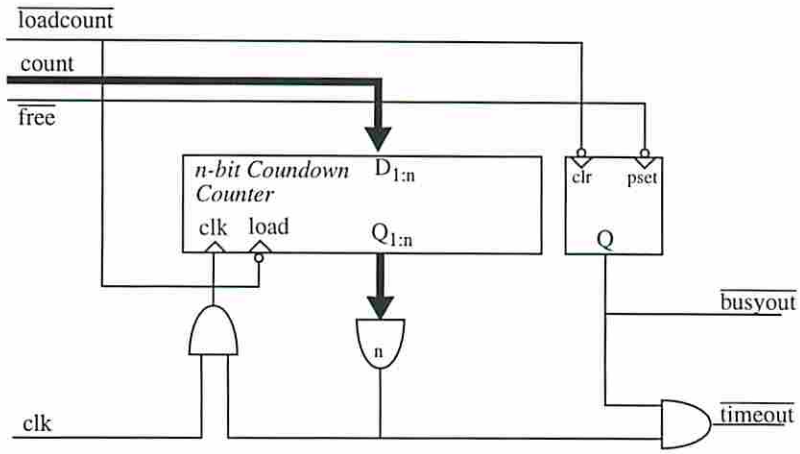


When a request is suspended, a time-out value is put on the "count" inputs and "$\overline{\text{loadcount}}$" is asserted. A non-zero "count" input releases the countdown clock. "$\overline{\text{loadcount}}$" also resets the flip-flop, which causes "$\overline{\text{busyout}}$" to be activated, indicating that this memory bank is busy. When the count reaches zero the clock is disabled and "$\overline{\text{time-out}}$" is activated, indicating that the request should now proceed. When the control circuit is done with the request it asserts "$\overline{\text{free}}$" which presets the flip-flop and causes "$\overline{\text{busyout}}$" to be deactivated, indicating that the bank is now free.

Since an incoming request for a bank that is busy has to be blocked, it will require two 32-bit

latches in the Receive Unit to latch its header until that particular bank is freed. It will also be necessary to identify whether there is a request header already latched in the receive unit.

Figure 7. Time-out circuit for one memory bank



There are six possible values for the counters for each target system configuration. The different values reflect the types of actions that are necessary in the target system to implement each request. Table 12 shows the six possible scenarios for the write-invalidate protocol.

Table 12: Counter values for the write-invalidate protocol

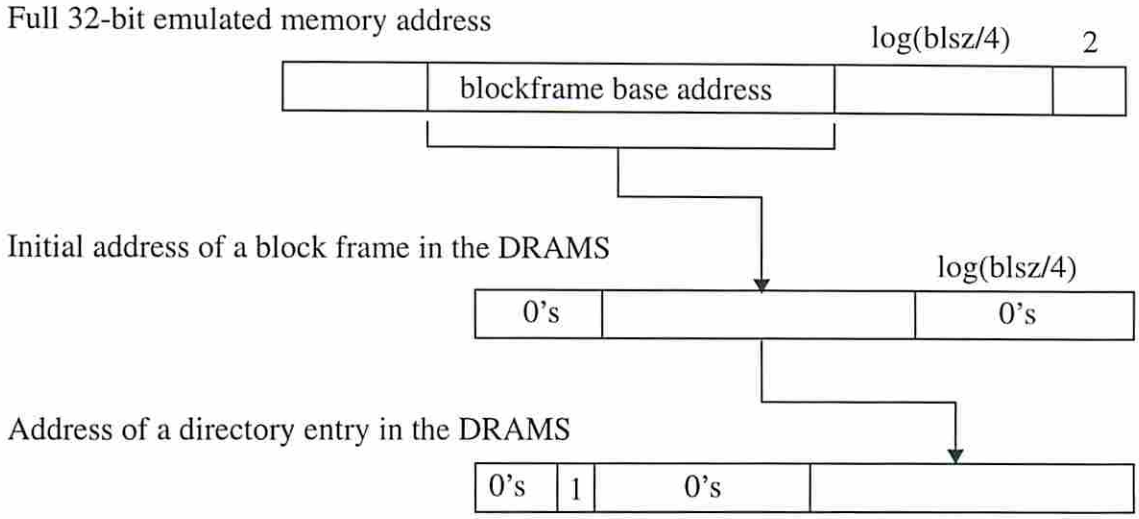
Count value	receive	transmit	situation
A	probe	block	read miss & block is clean write miss & block is uncached last inval. ack & state = Sh_Dty_miss
B	probe	probe	read miss & block is dirty write miss and block is cached elsewhere own. req. & no other_shared inval. ack & state = Sh_Dty_own
C	probe	probe*	write miss & other_shared own. req & other_shared
D	probe	nothing	inval. ack but not the last one
E	block	nothing	write back & state = DIRTY
F	block	block	write back & state = Dty_Dty or Dty_Shr

The actual values of the counters will depend on the parameters of the particular system being "simulated", and on the cache block size.

2.5. Flowcharts for all possible incoming requests

In the following figures we detail the controller actions that are taken to process each type of incoming request. Figure 9 to Figure 15 can be seen as an expansion of the ellipses labeled "start request" of Figure 5. Figure 8 below shows how the DRAM address of a data block and of a directory entry are computed from the address field in an incoming message.

Figure 8. Computing block and directory entry addresses



Prior to processing a request, performance metrics should be collected. The mechanism is simple and fixed regardless of the type of request being processed. Basically, a location in the DRAM event counting space is addressed using as address bits the following fields:

ID of the processor that sent the request	: 3 bits
request type identifier	: 5 bits
state of the memory block	: 3 bits (dirty bit, locked bit, locked type)
presence bits prior to request	: 8 bits

The counter is fetched, incremented by one, and stored back to the same position. It is important that this information be recorded before the directory entry for a block is modified in any way.

Figure 9. Read miss request

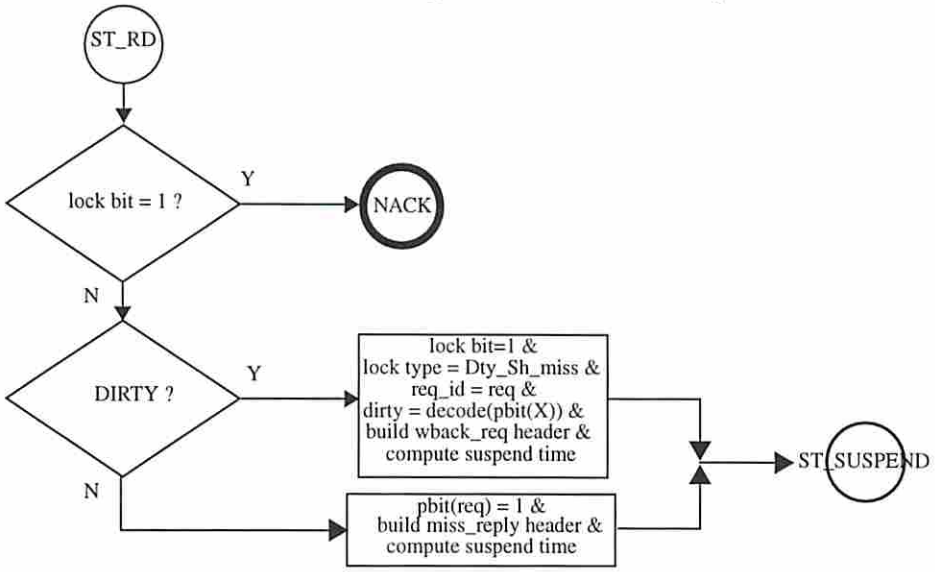


Figure 10. Write miss request

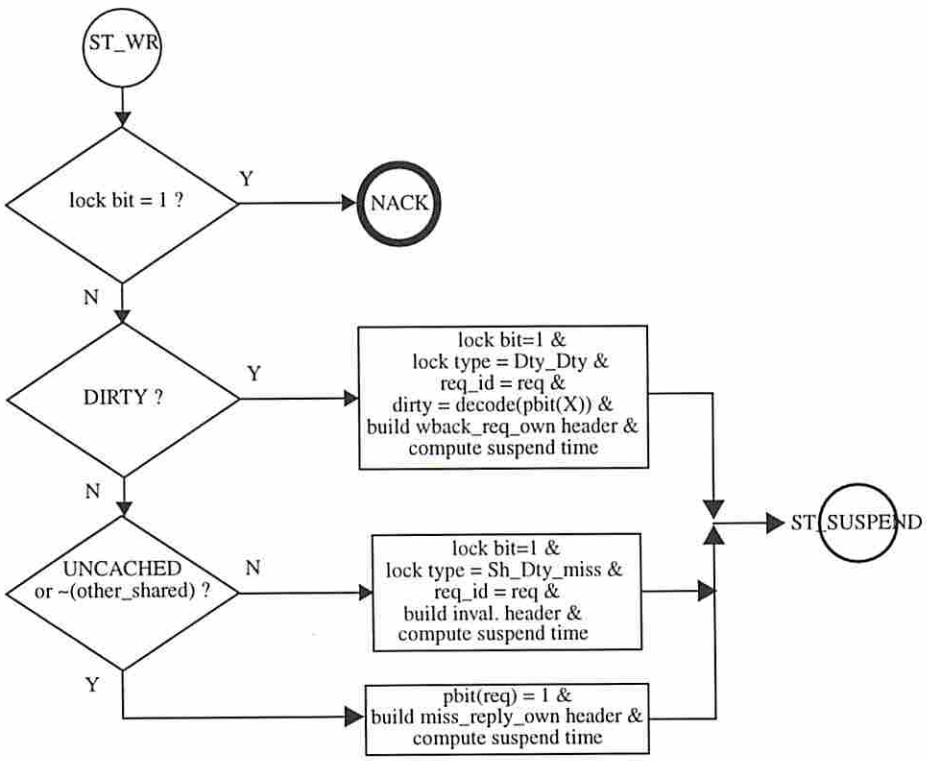


Figure 11. Ownership request

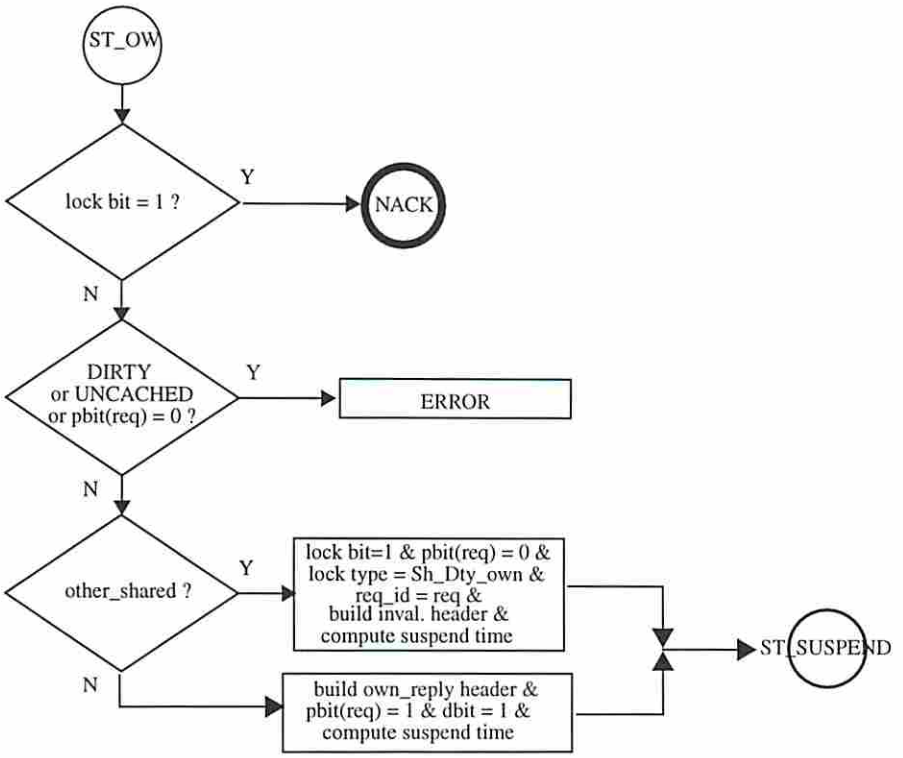


Figure 12. Invalidation acknowledgment

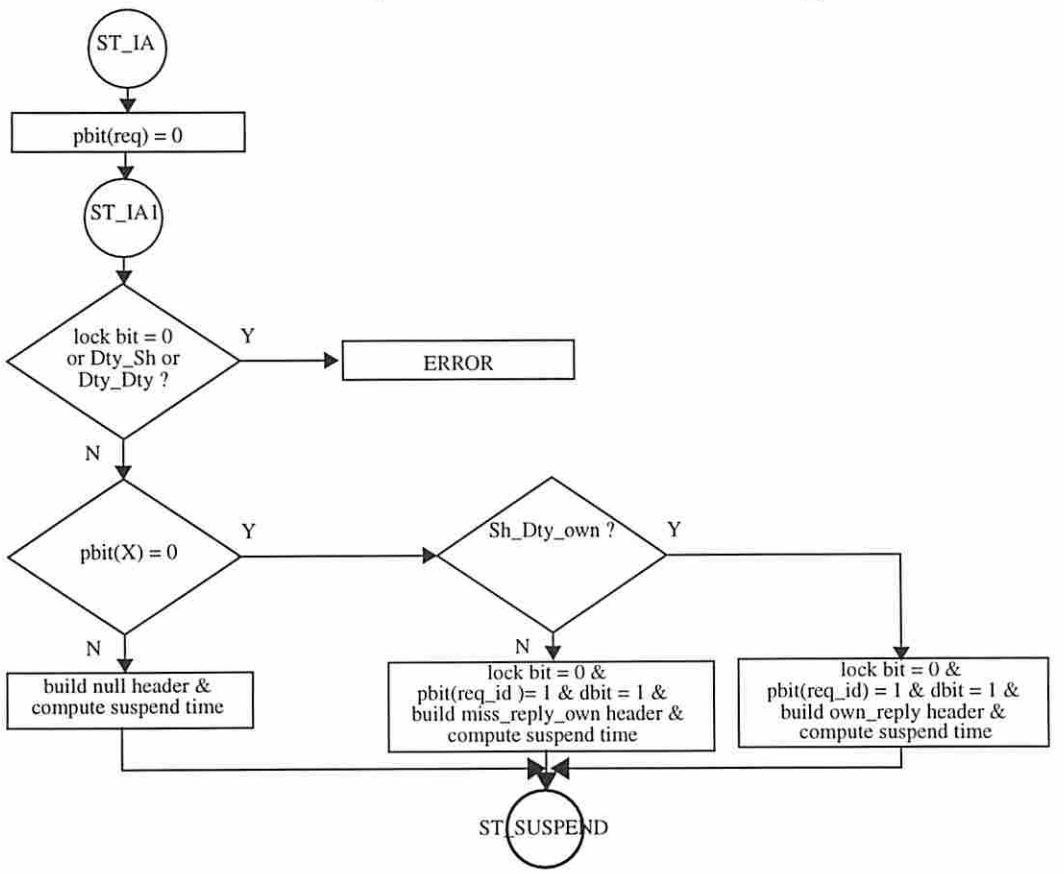


Figure 13. Write back

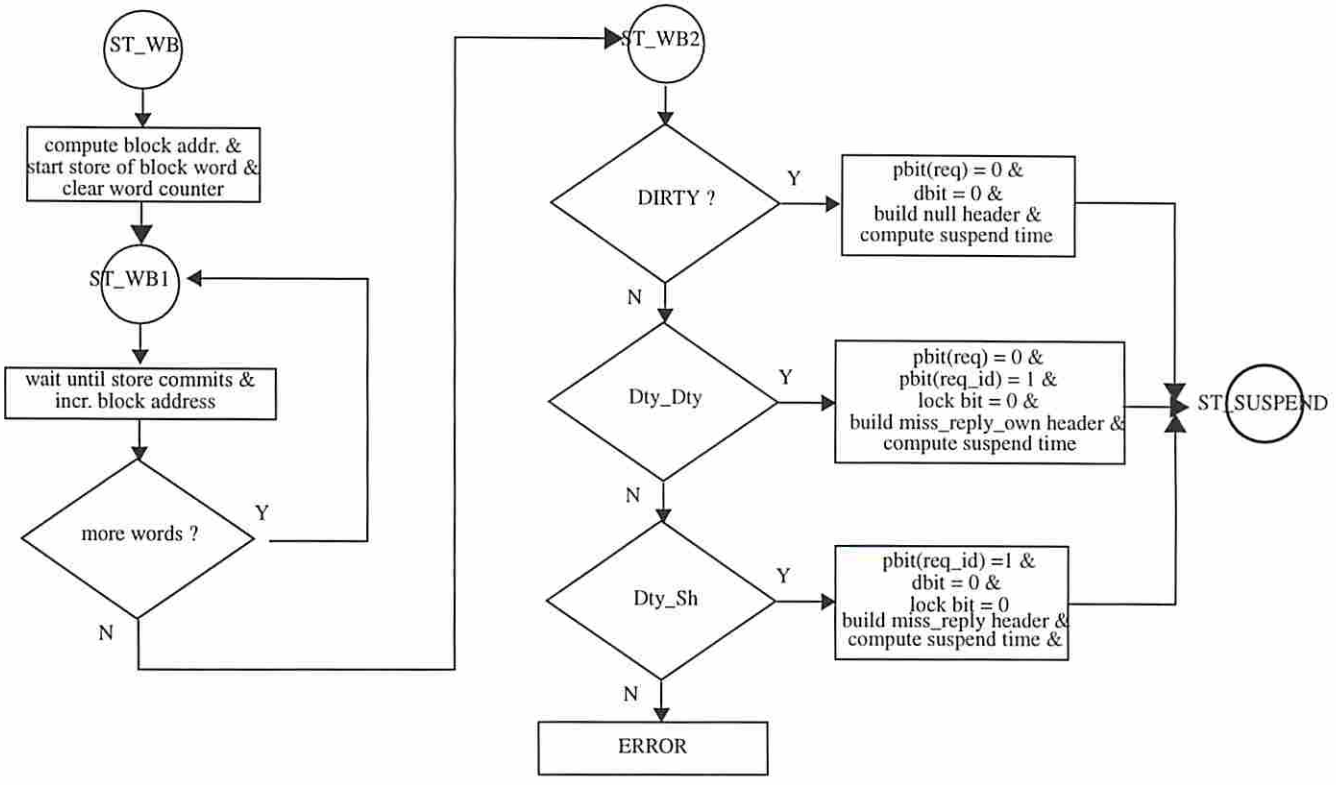


Figure 14. Request suspension

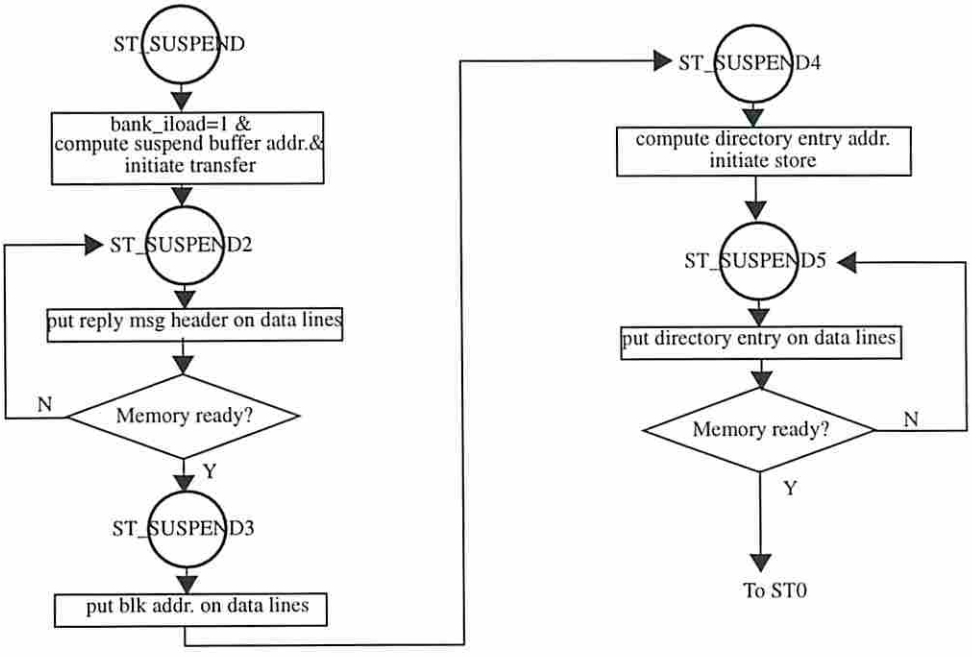
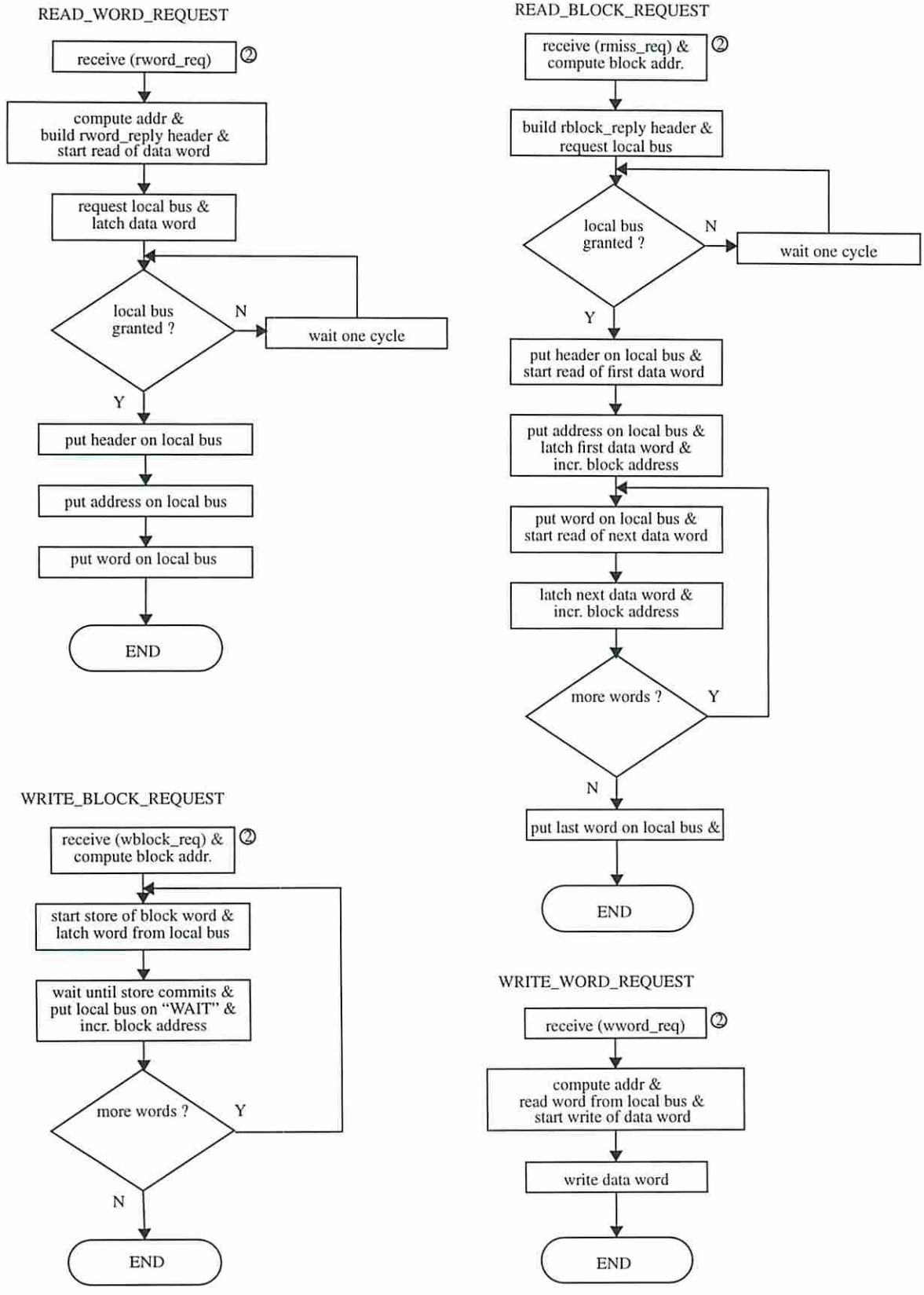


Figure 15. Test mode accesses (bypass the interleaving mechanism)



2.6. Implementation notes

The memory controller is implemented using two FPGAs, named MC1 and MC2. Both FPGAs have access to the same data, address, and control lines for a maximum of flexibility in the partitioning of the operations of the memory controller and for accomodating future developments. Both MC1 and MC2 are able to tri-state their shared outputs and drive them only during active cycles. The alternation of active cycles must be well coordinated by using the signals CHIP_ACTIVE1 and CHIP_ACTIVE2.

Currently, MC2 contains the states dealing with resuming suspended messages (see Figure 6). It also contains the adder that increments performance counts and its interface to the data bus. With its current functionality, MC2 does not need to receive packets from the FIFOs, it only needs to send out replies. It reads the DRAM (to fetch the suspended reply header and possibly a block of data or a directory entry when invalidations must be sent out). However, the DRAM is written (under controls from MC1) when performance counts are stored back after incrementation.

MC1 contains the bulk of the memory controller: the states from the main flowchart (Figure 5), the states for starting and suspending coherence requests (Figure 7-14), and for handling testmode requests (Figure 15). It also contains the states that control the fetching of a performance count, load it into MC2, control its incrementation, and write it back.

2.7. Conversion to SYNOPSYS

There were two major steps involved in the translation of old designs to the Synopsys format. First, several vendor-specific VHDL constructs had to be eliminated. These included predefined types (such as *vlbit*), conversion operators (such as *extendum* and *vld2int*) and operators (such as *addum*). It also required that some of the functions be rewritten using constructs with similar semantic.

Secondly, the mix of VHDL and Viewlogic schematics in the old designs was replaced with a unitary VHDL description using a minimum of source files.

In terms of performance, The state machine in MC1 was reduced from 96 states to 47 states. The simplification of the design along with the use of automatic state assignment feature in Synopsys allowed the design to clock up to 12MHz. MC2 is by far simpler than MC1 and can currently be clocked at 16 MHz.

In terms of resource utilization, MC1 is more compact:

70% utilization of I/O pins. (134 of 192)
41% utilization of CLB function generators. (471 of 1152)
10% utilization of CLB flip-flops. (111 of 1152)

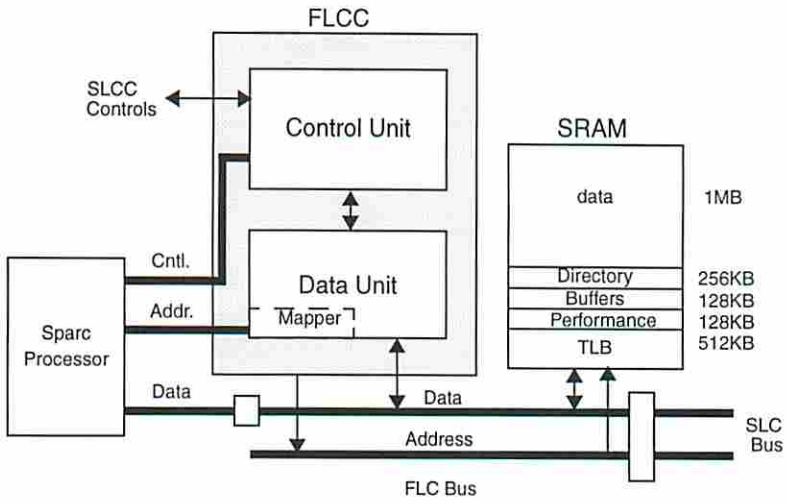
MC2 uses even less resources. It is certain that it is now possible to implement the entire MC in just one FPGA, but the impact on the maximum clock speed is not known.

3. First-level cache (FLCC)

3.1. Introduction

In RPM-2 the First-Level Cache Controller (FLCC) is implemented outside the processor. Figure 16 shows the organization of the FLC. Two FPGAs are used to implement the logic of FLCC: one is the control unit, called First-Level Cache Control Unit (FLCCU) and the other is the data path unit, First-Level Cache Data Unit (FLCDU).

Figure 16. Block Diagram of First-Level Cache Controller



FLCCU generates signals to control the correct operation of FLCC and interface logic to other modules such as the Sparc processor and the second-level cache. When it receives a memory requests from the processor, it checks if the requested data is present in the FLC memory by comparing the processor address with the tags stored in in the Directory region of the FLC SRAM shown in Figure 16. If the data is found, it is returned to the processor and the FLCC completes the memory access cycle. If the data is not present, the FLCC propagates the requests to SLCC.

In the current emulation each processor clock (pclock) is emulated by eight clocks. Therefore one access to FLC in the target system is implemented in eight steps accessing the FLC SRAM. In each pclock, the processor runs for one clock and "sleeps" for seven clocks.

The largest part of the FLCCU description is a state machine which replies differently according to the current operation mode and input signals from the processor and other modules. The details of the operation modes will be discussed in Section 3.4. The signals generated by FLCCU can be classified into three groups based on their function. First, a group of signals control the activation of the processor. If the current processor instruction accesses the memory hierarchy, FLCCU blocks the processor by asserting MHOLD in the processor; when the access is completed, it de-asserted MHOLD to wake up the processor. Second, if the data is not found in FLC, FLCC sends the decoded information for the access to SLCC. Third, the data that travels between different modules is temporarily stored in buffers, which are controlled by signals from FLCC.

The First-Level Cache Data Unit (FLCDU) controls the timing of FLC memory accesses and helps the operations of FLCCU by latching the signals from the Sparc processor and interpreting some of them. In particular, the MAPPER included in FLCDU decodes address-related information from the processor to classify memory accesses and converts them into signals that can be used to select the next state in the FLCCU state machine.

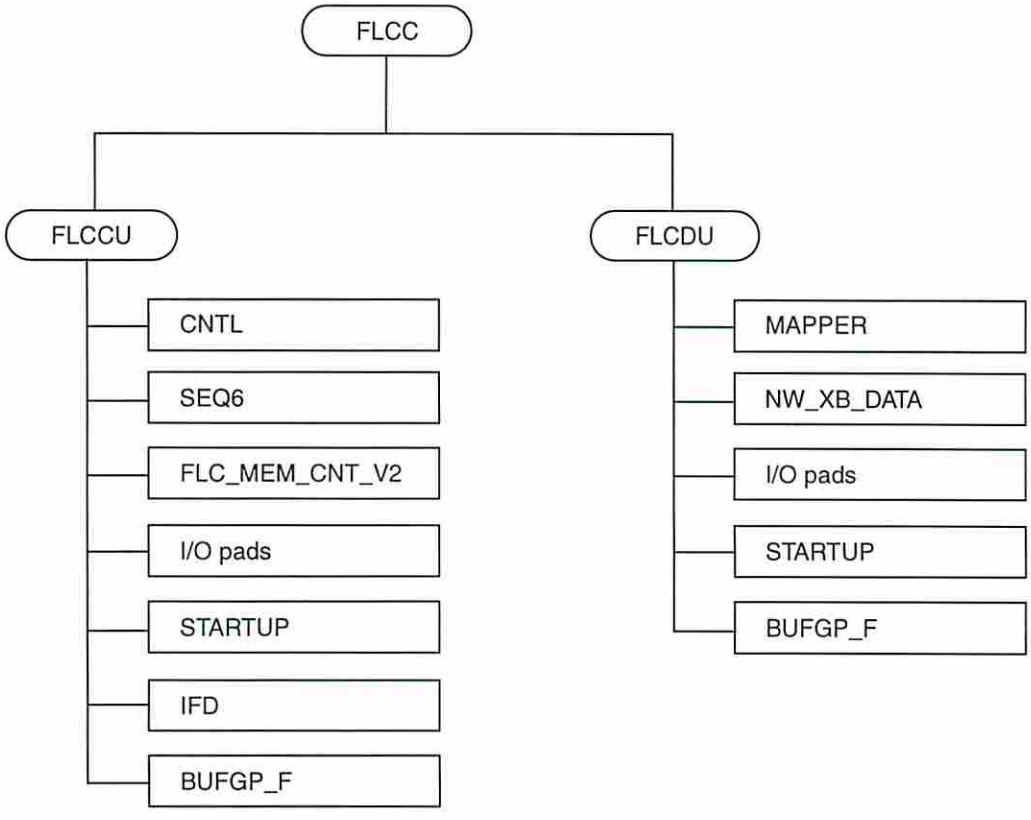
The memory needed by cache data, cache directory, TLB, data buffers and performance counters is all implemented in the same SRAM modules which are partitioned into different exclusive regions based on their address. Therefore accesses to these areas avoid conflicts in address and data ports.

3.2. Design Organization

The two FPGAs for FLCCU and FLCDU are implemented using multiple VHDL modules, shown in Figure 17.

The design framework of FLCCU is presented in `top_flcc.vhd` file, which is composed of other design components: cntl, seq6, flc_mem_cnt_v2, I/O pads(ipad1, ipad3, ipad13, ipad16), startup, IFD, and BUFGP_F.

Figure 17. FLC design organization



Cntl is the largest module of FLCCU, and is the central control engine of the FLC controller. The actual implementation of cntl is a single huge state machine, which is a collection of state sub-machines for each operation class. FLCCU contains different state sub-machines for read, write, atomic (test-and-set), prefetch and invalidation. This implementation uses more logic space on FPGA and has slower speed of operation, but it is a good model for interface synchronization between the state machines.

Seq6 generates the clock phase signal synchronized with the pclock. It uses a 8-bit shift register with leftmost bit initially set. In every clock the register shifts right in a circular fashion so that one complete rotation corresponds to one pclock.

Flc_mem_cnt_v2 takes address and read_in signals as inputs from the processor and the cntl module, and then generates control signals needed to control accesses to the SRAM modules including chip select and output enable signals.

I/O pad components specify the Xilinx-specific I/O terminals of the FPGAs. Synopsys provides a simple way to attach I/O pads to input/output pins, called `insert_pads`. However I/O pad components

in the FLCCU are provided for obsolete signals which had I/O pins assigned in FPGA and I/O nets in the PCB.

Startup is for Xilinx-specific FPGA initialization. Whenever FPGAs are reset, this module initializes the internal configuration.

IFD is a single input D flip-flop supplied in the Xilinx Libraries and is contained in an input/output block(IOB). The input of the flip-flop is connected to an IPAD or an IOPAD.

BUFGP_F, a primary global buffer, distributes high fan-out clock signals throughout FPGA device.

Table 13: Input Signals to FLC

Sources	Signal Names	Meanings
Sparc Processor	LOCK	Bus Lock
	INTACK	Interrupt Acknowledge
	ERROR_BAR	Error State
	DXFER	Indication of Data Transfer Cycle
	SIZE[1..0]	Bus Transaction Size
	WRT	Advanced Write
	RD	Read
	LDSTO	Atomic Load/Store Operation
	ASI[7..0]	Address Space Identifier
	A[31..0]	Address Bus
	INULL	NULL cycle indicator for integer operations
	FNULL	NULL cycle indicator for float-point operations
SLC	INVALIDATE	Cache Line Invalidation Request
	READY	Ready for accepting request
	BUSY	Data on Queue
	PRF_REQ	Request for Prefetch
IOCTL	FLCC_ACCESS	Read FLC
	NORMAL_MODE	Indication of Normal Mode
INT_GEN	IRL[3..0]	Interrupt Request Level
NIC	NIC_MSG[2..0]	Message path from NIC
ETC	CLK	System-wide Global Clock
	RESET	System-wide Global Reset

The design framework of FLCDU is contained in the `top_flcd.vhd` description file. As in FLCCU, FLCDU contains other design components: mapper, nw_xb_data, I/O pads(ibuf, iopad32), startup, and BUFGP_F.

Mapper receives address-related information from the processor and generates signals which select the next state in *cntl* of FLCCU. One of such signals is `testmode` set according to the current address space, ASI[7..0]. Even in emulation mode, RPM-2 uses test mode to access status registers, control registers, and performance counters. In particular, mapper maps the emulated address into a new address which points to the corresponding performance counter for the given event.

In **nw_xb_data,** multiplexors for address and data signals are connected to the SRAM modules. Each multiplexor choose one among multiple sources based on the access type and location. For example, the address for cache data is selected to read out the data stored in the cache while the address for the tag

accesses the tag information. .

Table 14: Output signals from FLC

Destination	Signal Names	Meanings
Sparc Processor	MEXC MDS MHOLDA	Memory Exception Memory Data Strobe Freeze the clock to the Processor
SLC	RELEASE_ACCESS P_MISS_REQ REQ_TYPE[4..0] F_READY RB_NOT_EMPTY F_SHARED F_WRT COLLECT_STATS	Release Operation Normal Read Miss Operation Type to SLC Completion of SLC Request Read Buffer Not Empty Shared Region Write Cycle Collection of Statistics
Addr Buffer	A_C[31..0] A_OESF_BAR A_CPFS A_SSF	Address for SLC Output Enable (74F652) Clock Pulse (74F652) Latch Enable (74F652)
Clearable Chip	A_C[31..0] CL_CS_BAR CL_OE_BAR CL_W_BAR CL_RESET_BAR	Address for Clearable Chips Chip Select (SRAM) Output Enable (SRAM) Write Enable (SRAM) Reset (SRAM)
Data Buffer (to SLC)	L_D[31..0] D_OESF_BAR D_CPFS D_SSF	Data for SLC Output Enable (74F652) Clock Pulse (74F652) Latch Enable (74F652)
Data Buffer (to Proc)	L_D[31..0] PD_DIR PD_CP_FP PD_CP_PF PD_S_FP PD_S_PF PD_OE_BAR	Data for Processor Direction Select (74F646) Clock Pulse for Traffic from FLC to Processor Clock Pulse for Traffic from Processor to FLC Latch Enable for Traffic from FLC to Processor Latch Enable for Traffic from Processor to FLC Output Enable
FLC Memory	L_D[31..0] A_C[19..2] E_BAR1[4..1] E_BAR2[4..1] READ_OUT[2..1]	Data Address Enable Enable Read Signal

In practice the cache data and tag information are read at the same time to reduce the access latency. However, in RPM because these information share the same SRAM module and because the processor clock is emulated using multiple system clock, data cache and tag information are accessed serially.

I/O pads, **startup**, **BUFGP_F** all have the same purpose as in FLCCU.

3.3. Interfaces

Signals which are connected to/from FLC are summarized in Table 13 and 14.

3.4. Operation Modes

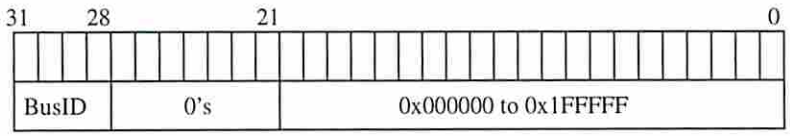
FLC can be in one of two different operation mode: *test mode* and *emulation mode*.

In test mode, all the physical memory regions are accessible by using alternate space instructions. The values ASI[7..0] identifying the specific alternate space are given below:

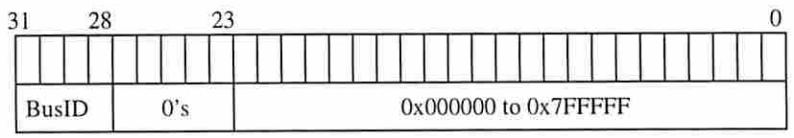
Table 15: Alternate space ASI mappings (Test Mode)

ASI[7..0]	Alternate Space
0x01	Main Memory
0x02	SLC Memory
0x03	FLC Memory + Registers decoded by FLCC
0x05	I/O (SCSI, Life, etc.)
0x00	Registers decoded outside the FLCC

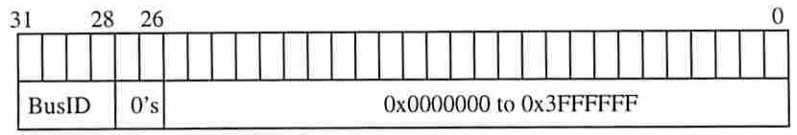
In test mode, every RAM location of the main memory and both caches is mapped to a linear address space so that it is accessible. This mode is principally used for testing, debugging and initial downloading of code and data. If the `normal_mode` signal which comes from IOCTL Mach chip is asserted, RPM runs in emulation mode. To access any physical location in the RAM of the first-level cache the following address should be used in conjunction with an ASI = 0x03:



To access any physical location in the RAM of the second-level cache the following address should be used in conjunction with an ASI = 0x02:

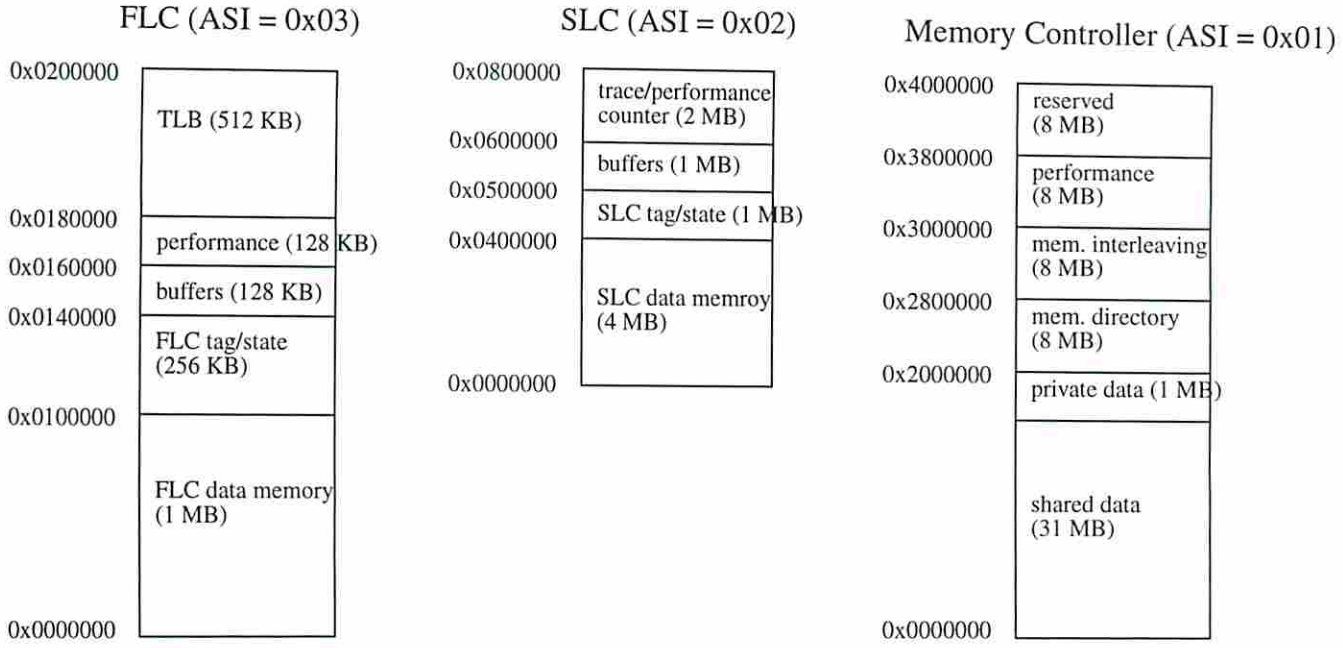


To access any physical location in the RAM of the main memory the following address should be used in conjunction with an ASI = 0x01:



The memory map for FLC, SLC, and main memory is shown in Figure 18:

Figure 18. Full Memory Space in Test Mode



I/O space has its own alternate space with ASI = 0x05. The address range assigned to individual devices are given in Table 16. :

Table 16: Memory Mapped I/O

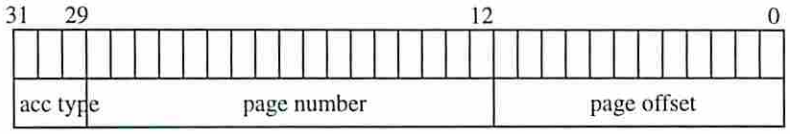
ASI[7..0]	Address Range	Device
0x05	0xXX0XXXXX	Control Register (pclock control, boot ROM address mapping, start control)
0x05	0xXX1XXXXX	FPGA programming (FPGA selector register)
0x05	0xXX2XXXXX	FPGA programming (FPGA data)
0x05	0xXX3XXXXX	FPGA status register
0x05	0xXX4XXXXX	7-segment display data register
0x05	0xXX5XXXXX	SCSI controller
0x05	0xXX6XXXXX	Serial I/O
0x05	0xXX7XXXXX	Real time clocker
0x05	0xXX8XXXXX	DRAM controller (data register)
0x05	0xXX9XXXXX	DRAM controller (address register)
0x05	0xXXAXXXXX	Delay unit (word register)
0x05	0xXXBXXXXX	LIFE chip registers
0x05	0xXXCXXXXX	Delay unit (initialization register)

In emulation mode, shared data and private data memory are mapped by their ASI and address

Table 17: Alternate space ASI mappings (Emulation Mode)

ASI[7..0]	Alternate Space
0x08	User Instruction
0x09	Supervisor Instruction
0x0A	User Data
0x0B	Supervisor Data

value given by processor. The address that comes out of the processor in emulation mode has the following format:



The 29 bits of address space allows 512MB of shared memory space in emulated mode. The three most significant address bits (A[31..29]) indicate to the first level cache the type of access. All emulation mode memory accesses have access type 00X. Nonblocking prefetch instructions to emulated memory the have access type 01X and can be a load or a store. Stores with access type 10X release a lock on a memory location. (loads are not defined for this access type). Memory accesses with type 11X provides controls over spin lock and statistics gathering mechanisms. The two bits of addresses (A[3..2]) select one of four control operation: spinlock on/off and statistics gathering on/off. In this case, only the address lines are decoded and the type of instruction is irrelevant.

Table 18: Decoding of instructions to simulated memory

A[31..29]	Instruction	Type of access
000	load/store/test-and-set/swap	normal access to simulated private memory
001	load/store/test-and-set/swap	normal access to simulated shared memory
010	load/store	prefetch access to simulated private memory
011	load/store	prefetch access to simulated shared memory
10X	store	lock release operation to an address
11X	load/store/test-and-set/swap	spin lock on/off, statistics on/off

3.5. Experience with Synopsys tools and performance analysis

In the Viewsynthesis environment, the FLCC design was input with Viewlogic schematics at the top level and VHDL descriptions for sub-level components.

The conversion from Viewsynthesis to Synopsys is straightforward. The first step of the conversion consists in changing top-level schematics into VHDL descriptions. One of the major changes of the design description is of the input/output pads. Because the Synopsys synthesizer supports pads automatically with the `insert_pads` command through the script file we don't need to include input/output pads in the design file unlike in the Viewsynthesis environment. However the I/O pads for obsolete signals remains to maintain compatibility. Also, Synopsys libraries provides design macros such as startup and BUFGP which were described with dedicated symbols in the schematic. Therefore, the conversion of the top-level design can be done in an one-to-one mapping fashion.

The second step of conversion is to remove obsolete macros and libraries or to change them to the corresponding new ones. One of such modification is `IEEE.std_logic_arith.all` package. In Synopsys, this package should be replaced with `IEEE.std_logic_unsigned.all` to implement the addition or subtraction function correctly in VHDL. For example, `mapper.vhd` frequently uses the addition and subtraction specified with '+' and '-' operators. With `IEEE.std_logic_arith.all`, these operators produce wrong values. For `IEEE.std_logic_unsigned.all` package to be used, we need to implement EXT macro properly, which is now included in `pp_const_flc.vhd` file.

The Synopsys synthesizer generates report files and log files after finishing the synthesis process. The report file named like *`designname`*`.rpt` shows statistics, summaries and, if any, warnings and errors about the design and synthesis results. The detail timings for each path and device usages are given in `ppr.log` file.

From our experience with FLCC, we have come to realize that the Synopsys synthesizer utilizes more device resources than the combination of Viewsynthesis and XSIS does as shown in Table 19 and 22, because the device-dependent designs are well described with primitives provided by Xilinx. When synthesized with Synopsys, the maximum delay is better for both FLCCU and FLCDU.

Table 19: Synthesis Results Comparisons for FLCCU

	Viewsynthesis	Synopsys
Occupied CLBs	408	521
Packed CLBs	266	305
Bonded I/O Pins	128	128
CLB Flip Flops	44	79
Maximum Delays	112.6 ns	111.8 ns

FLCCU designedwith the Synopsys tool uses 521 CLBs and its delay is decreased by 0.7%. The size and complexity of the state machine in FLCCU limits the efficiency of the synthesis process. However, FLCDU with Synopsys tool shows much improvement in terms of the maximum delays (25.8%). Even though it utilizes more CLBs than Viewsynthesis and XSIS, XC4013 FPGA can accommodate the required CLBs without any problem.

Table 20: Synthesis Results Comparisons for FLCDU

	Viewsynthesis	Synopsys
Occupied CLBs	385	529
Packed CLBs	260	333
Bonded I/O Pins	180	178
CLB Flip Flops	230	234
Maximum Delays	96.4 ns	71.5 ns

3.6. Detailed flowcharts

The cycles of FLC are read, write, atomic operation, prefetch and invalidation. Modern processors have on-chip FLCs so that the latencies to the cache could be minimized. The usual latencies are one or two processor clocks depending on their organizations. To emulate this behavior, FLC in RPM freezes the processor by asserting the MHOLD signal as soon as the data request comes out of the processor and wakes it up by de-asserting the signal after completing the designated operations.

Figure 19. Main state machine loop of FLC controller

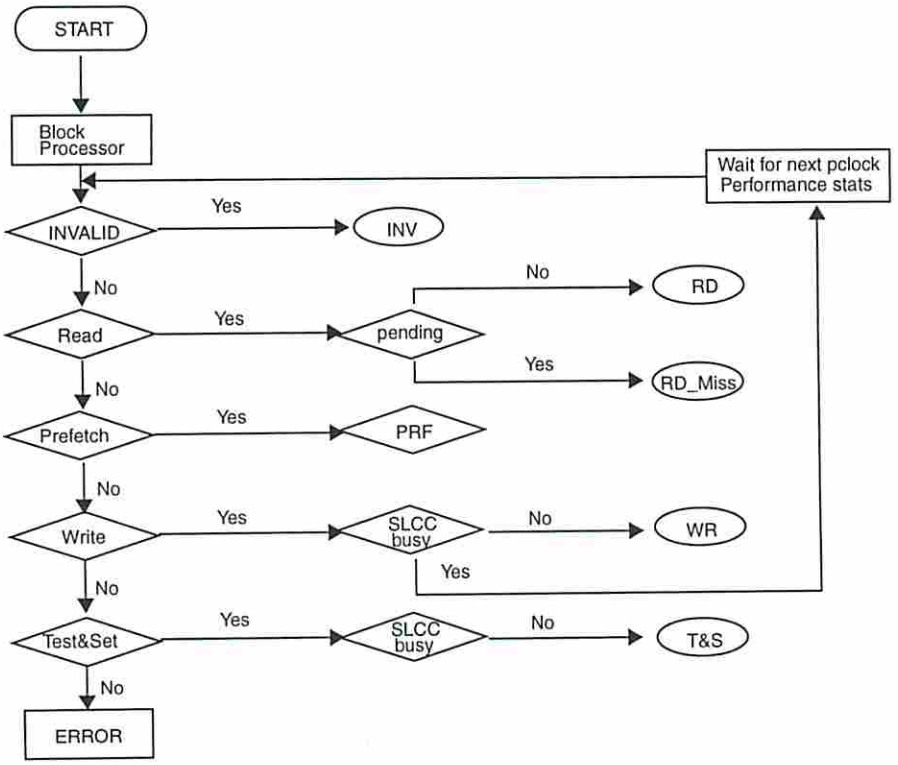


Figure 20. Invalidations and Prefetch

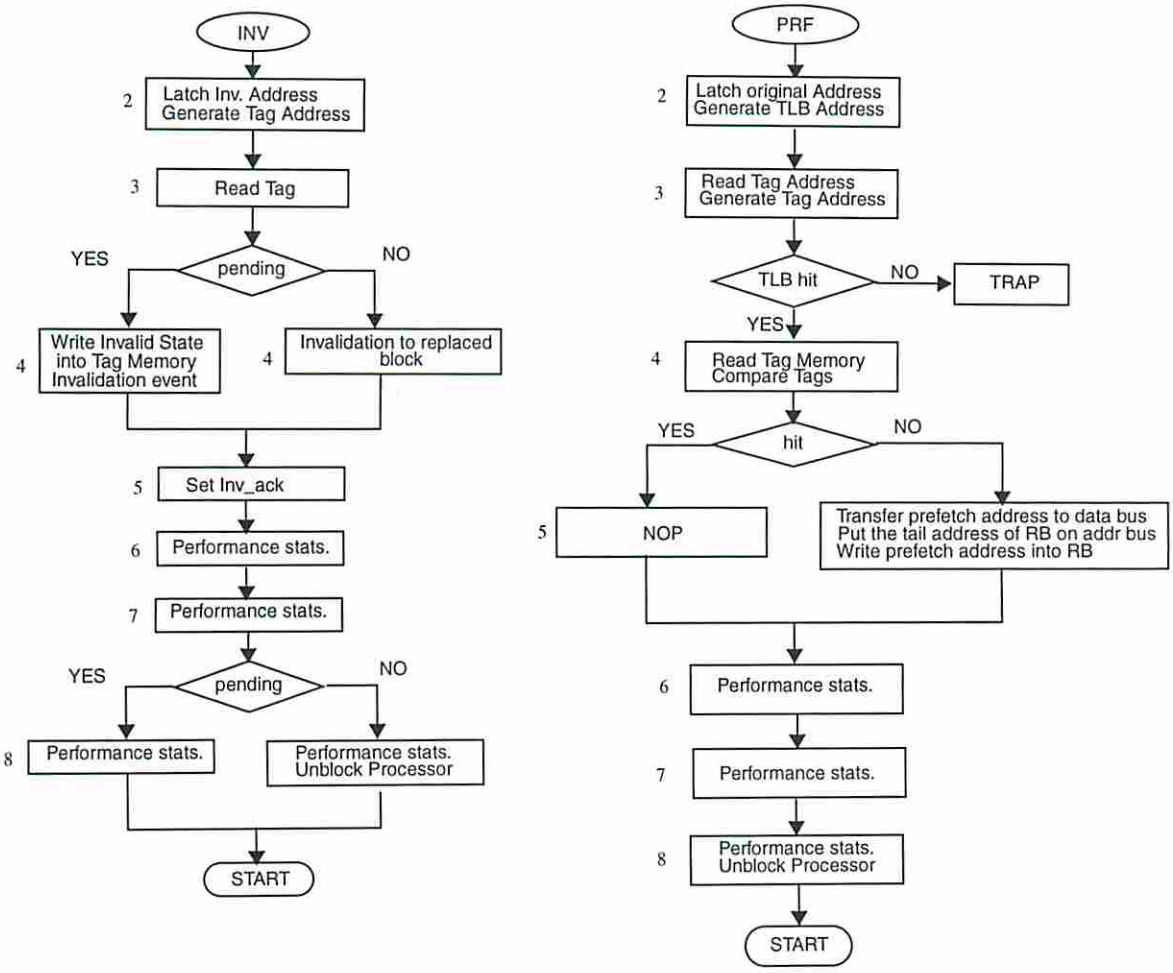


Figure 21. Read

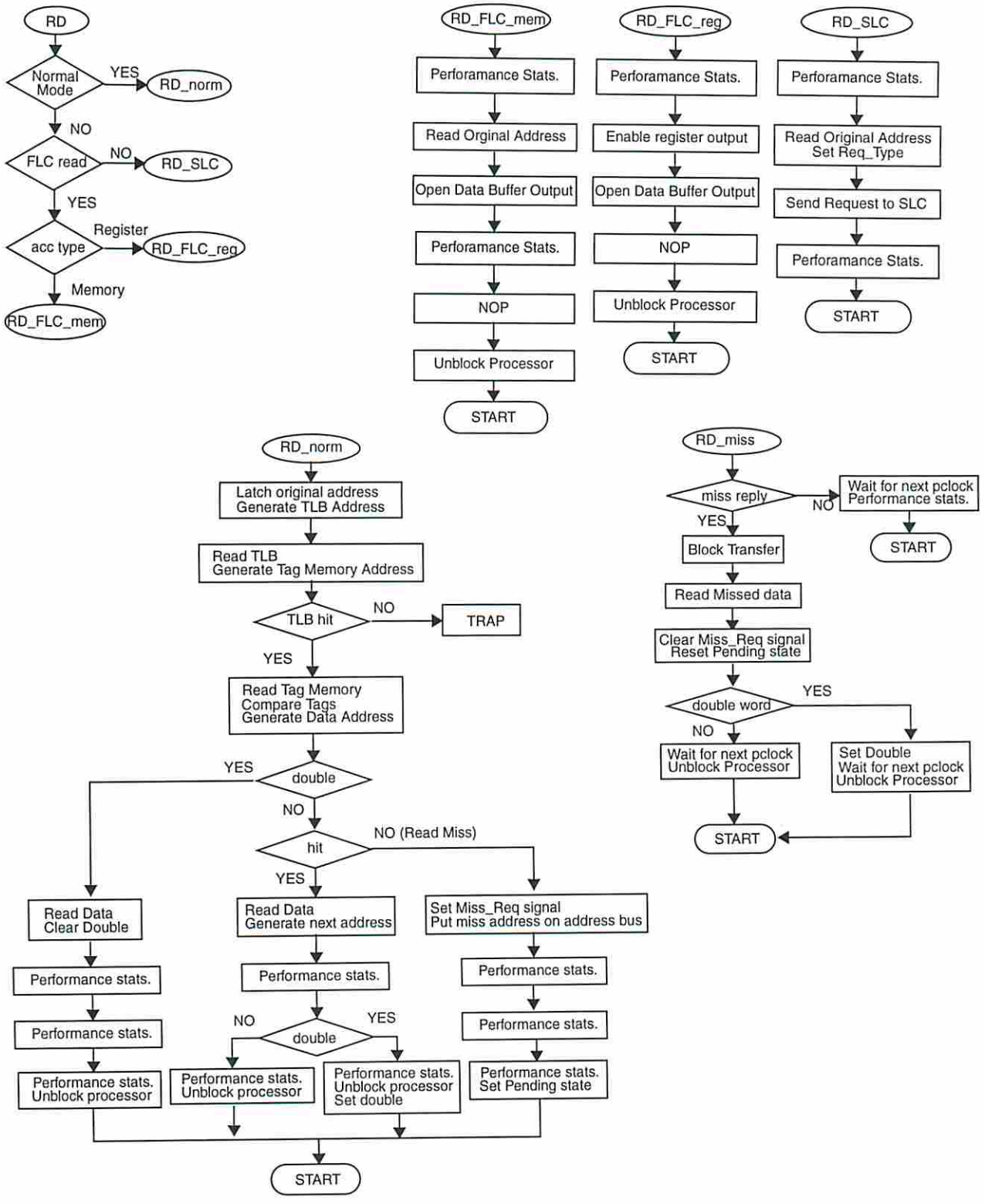


Figure 22. Write

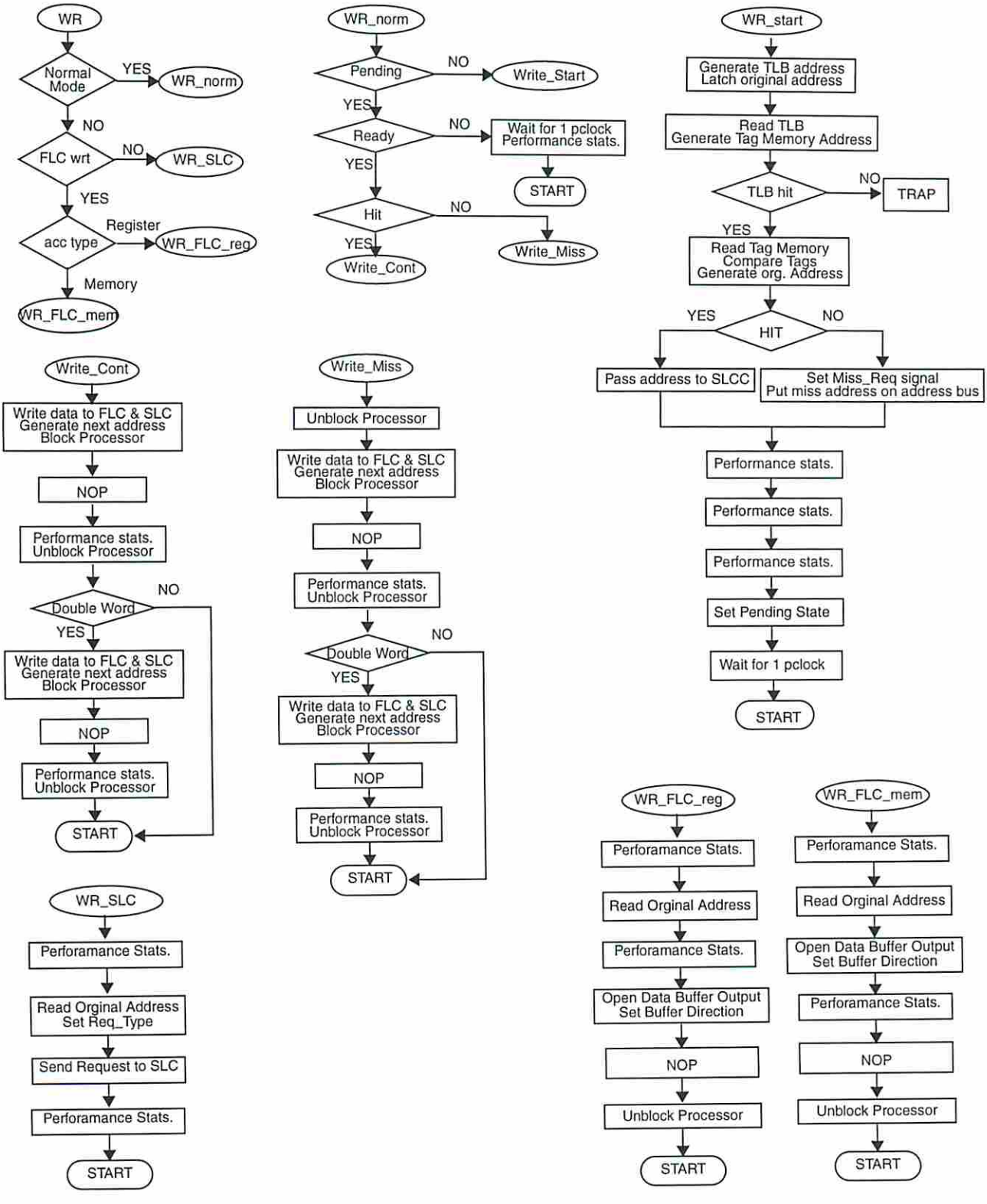
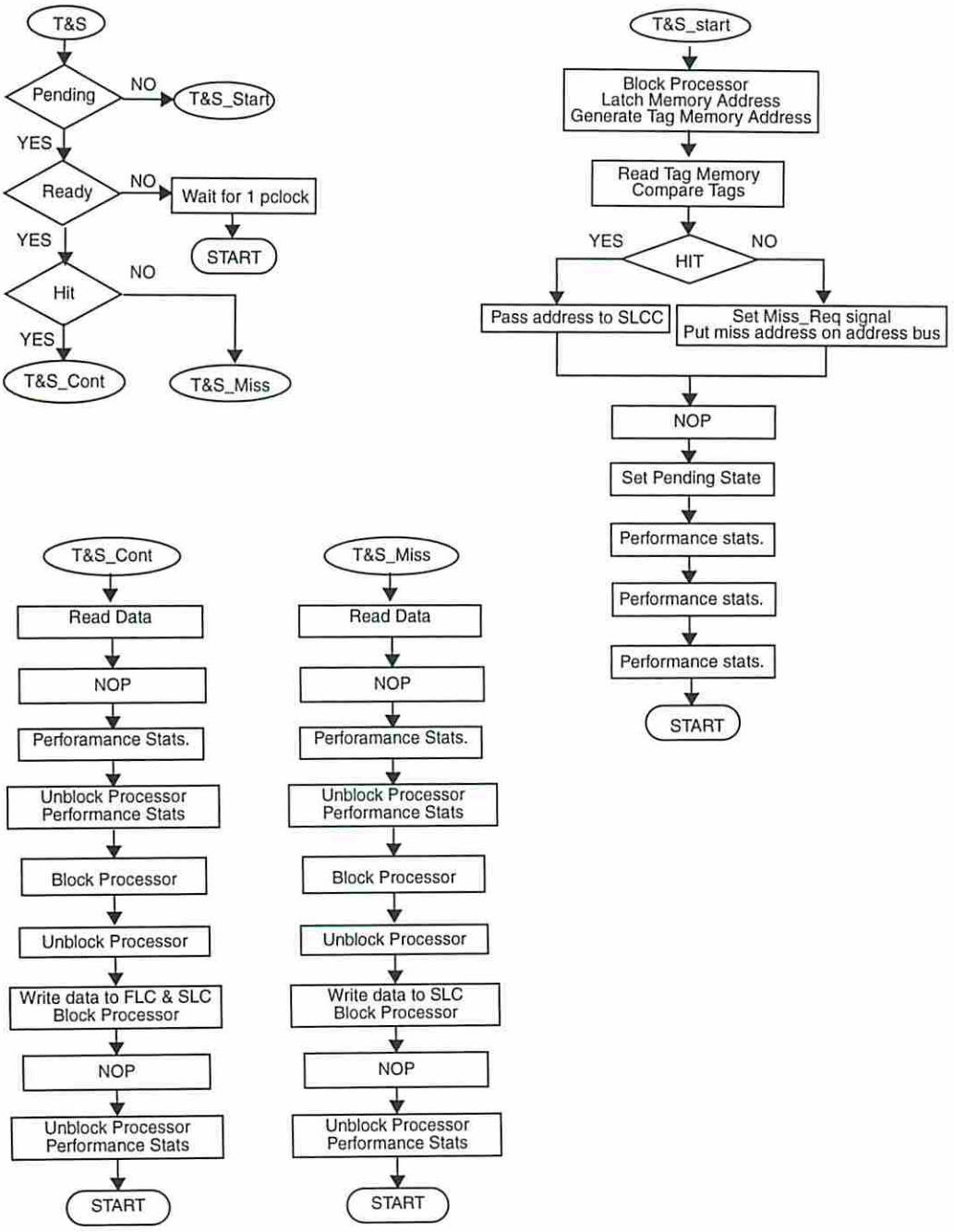


Figure 23. Test and Set

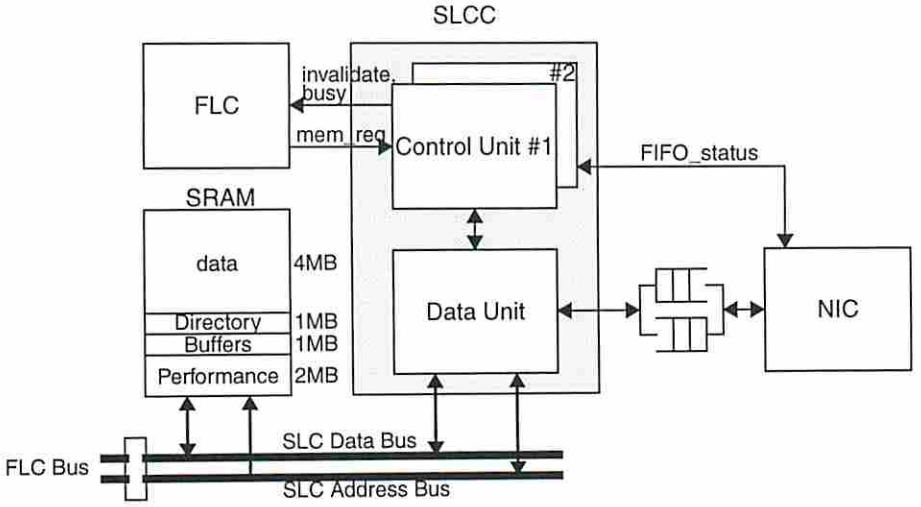


4. Second-level cache controller (SLCC)

4.1. Introduction

Figure 24 shows the block diagram of SLC. It consists of three FPGAs and 8 Mbytes of SRAM. The control unit consists of two identical FPGAs which have the same pinouts. Currently only the first control unit is used and the second one is reserved for implementing memory consistency mechanisms. They can communicate to each other using reserved connections between them. The control unit receives commands through either dedicated memory request signals from FLCC or through messages from incoming FIFO connected to NIC and it drives the data unit implemented with one FPGA. For each access, the directory of the cache stored in part of the SRAM is examined and corresponding data is forwarded to FLC or to outgoing FIFO connected to NIC. Currently it emulates two-way set associative or direct-mapped write-back coherent cache with variable block size and cache size. The functional description of SLCC in CC-NUMA emulation are provided in the flowcharts. SLCC also supports test mode in which the processor can access each level of memory hierarchy without forcing coherence of data.

Figure 24. Block Diagram of Second-Level Cache Controller



In this section, we give some details of the SLCC design which may not be obvious from the flowcharts and then we summarize our experience with the CAD tools and their performance.

4.2. SLCC Design Details

In the case of a read miss or a write in FLC, FLCC first blocks the processor and then transfers the appropriate request to SLCC. If it is a write, after getting the block in RW state, SLCC informs the FLCC (so that it can complete the write, if the block is valid in FLC), completes the write in SLC and unblocks the processor. In the case of a read miss, after getting the block in RO/RW state in SLC, SLCC transfers the block to FLC by interacting with FLCC. After that, FLCC unblocks the processor.

An access (write or a read miss from FLCC) may have to be kept pending in a register called Pending Access Register (PAR) at SLCC. This may happen either because SLCC has to access the home node to get write permission or a copy of the block, or because a miss for the block occurred in SLC but a blockframe can not be allocated (because all blockframes are pending in the set). In the former case, the pending access is serviced once the block is in the right state in SLC. In the latter case, once a blockframe gets out of the pending state, the pending access is retried.

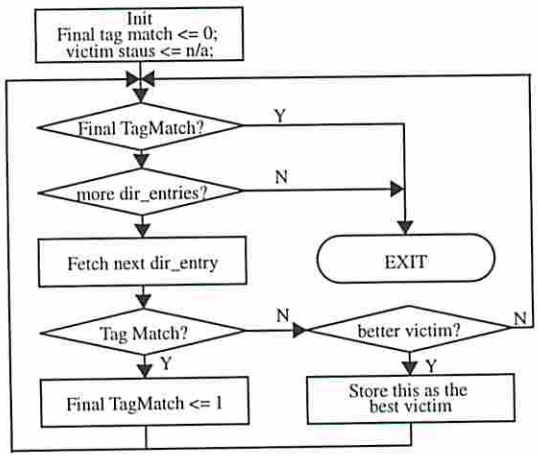
Double-word write is similar to ordinary write. When SLCC has the block in RW state (or after getting it in RW state) it unblocks the processor, completes the first word write at the following rising edge of the processor clock and second word write at the next rising edge of the processor clock. Between these two writes, SLCC cannot be interrupted.

The second level cache (SLC) is non-blocking, to implement non-blocking prefetch. There are two types of prefetches: shared and exclusive. Prefetches are first put in a request buffer (which could also be used as a write buffer in a relaxed memory model. Whenever SLCC is idle, it takes them out of the buffer one by one for processing. A prefetch is ignored if a blockframe cannot be allocated; this may happen when all blockframes in the set are in pending states. Also, an exclusive prefetch is dropped if a negative ack (nack) from home node is received in response to an earlier attempt to get a RW copy of the block. Since the request buffer is in the first level memory, two hand-shake signals called *req_buf_not_empty*, and *req_pf* are used to implement the buffer access.

Atomic Read-Modify-Write is almost like a double-word write. After getting the block in RW state, FLCC is informed (there may be a block transfer to FLC, which is the main difference with double-word write) so that it can execute the read part of RMW immediately followed by the write part at the following rising edge of the processor clock.

When more than one possible victim block of the same grade are available, the choice for replacement among them is random (two blockframes in the invalid state are of the same grade, whereas an invalid blockframe has a higher grade than a RO blockframe). However, if the choice is between a lower grade and higher grade blockframes, the one with higher grade is chosen for replacement. A flowchart for the implementation of the victim selection (and tag matching) is shown in Figure 25. Note that we need at most two registers to implement it, irrespective of the associativity. Furthermore, by changing the way we define the grades for potential victim blockframes, we can implement different replacement policies. For the current implementation we use the following grade ordering: PEND_* < RW < RO < INV.

Figure 25. Flowchart of Replacement



4.3. Experience with Synopsys tool and performance analysis

We have converted the design with old tools (Viewsynthesis+Vsis) into Synopsys version. The conversion was straight-forward since we can simply replace obsolete keywords or library functions into corresponding ones supported by Synopsys. The data unit is divided into two parts; one contains all the constants such as cache parameters to change its configuration easily and the other is the data path and its

control, which are mostly combinational. The control unit is the most complex design and implements a big finite state machine consists of 27 states and 32 substates for each state as described in the flowcharts. It also has two parts; one with all the constants and the other with the state machine.

Table 21 compares the performance results for the data and control units obtained with the old tools and with the Synopsys tool. The designs with Synopsys tool have two numbers for IO pins, CLB Function Generators (FG) and CLB Flip-Flops (FF) which indicate the utilization of FPGA. The first number shows the result after optimization by Xact tool whereas the second number shows the result after the synthesis process. The table also gives the delay after mapping the design on the FPGAs in four different measures. Pad to Setup (P2S) is the time delay from an IO pin to an input of an internal FF, which is usually the setup time for FFs in the design. Clock to Setup (C2S) is the delay between two internal FFs and Pad to Pad (P2P) is the sum of P2S, C2S and C2P. Finally the maximum delay is the estimated delay by Synopsys tool before mapping the design on FPGA.

The data unit designed with the Synopsys tool uses 516 FGs (64% of old tool) and its delay is decreased about 23%. This indicates that the Synopsys tool are much better than the old tools in terms of speed and resource utilization. The number of IO pins are different because we simply removed unused IO connections.

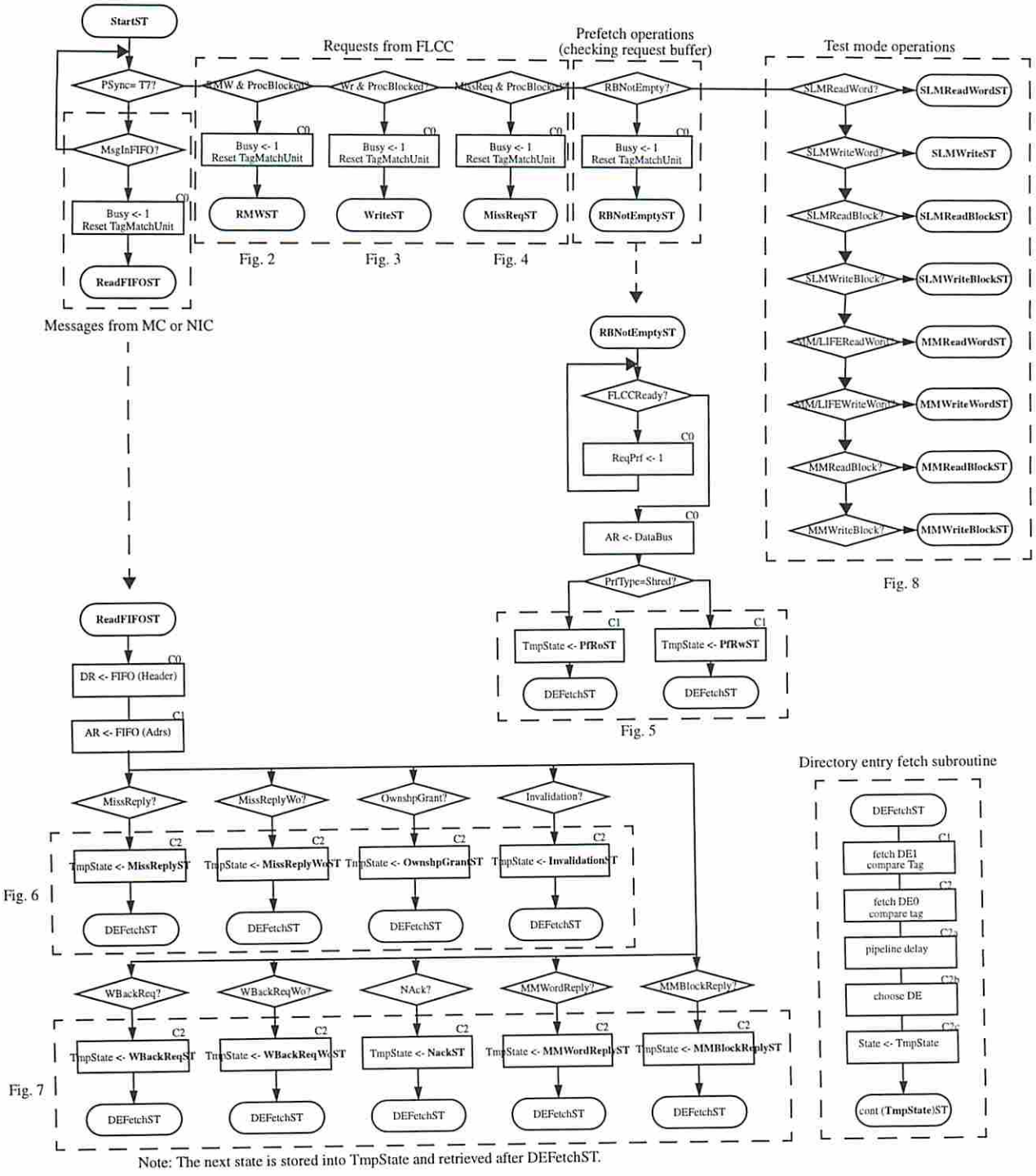
Table 21: Performance results of SLCC designs

designs	IO pins (192)	CLB FGs (1152)	CLB FFs (1152)	max P2S	max C2S	max C2P	max P2P	max delay (Synopsys)
data unit (old tool)	181	804	206	78.8 ns	77.0 ns	47.4 ns	N/A	N/A
data unit (Synopsys)	178 (178)	516 (557)	206 (206)	57.2 ns	60.4 ns	39.5 ns	40.0 ns	29.7 ns
control unit (old tool)	126	786	44	95.6 ns	118.0 ns	96.9 ns	N/A	N/A
control unit (Synopsys)	123 (123)	738 (1056)	119 (183)	105.0 ns	146.7 ns	111.2 ns	N/A	119.01 ns

For control unit, the result is different than for the data unit. First, the synthesized design obtained by directly converting VHDL codes by Synopsys tool could not be mapped on the FPGA and over 100 nets remained unrouted whereas the design with the old tool was mapped successfully by Xact 5.0. To map the design on FPGA, we took three approaches. First we played with state encoding and minimization tools provided by Synopsys. We tried one-hot encoding, binary encoding and automatic encoding by the FSM tool for both states and substates optionally using Xact vM1.3. This reduced the number of unrouted nets but the design still failed to map. Second, we removed many common flows in the design and the design mapped finally with Synopsys tool. Third, we could map the original design by relaxing timing constraints greatly and the results are shown in the table. It shows that the old tool works better than the Synopsys tool in the case when the design is highly sequential. We may need more experience to evaluate the usefulness of the Synopsys FSM tool and to further improve the performance of the control unit.

4.4. SLCC Flowcharts

Figure 26. Main Loop



Note: The next state is stored into TmpState and retrieved after DEFetchST.

Figure 27. Read-Modify-Write request from FLCC (RMWST)

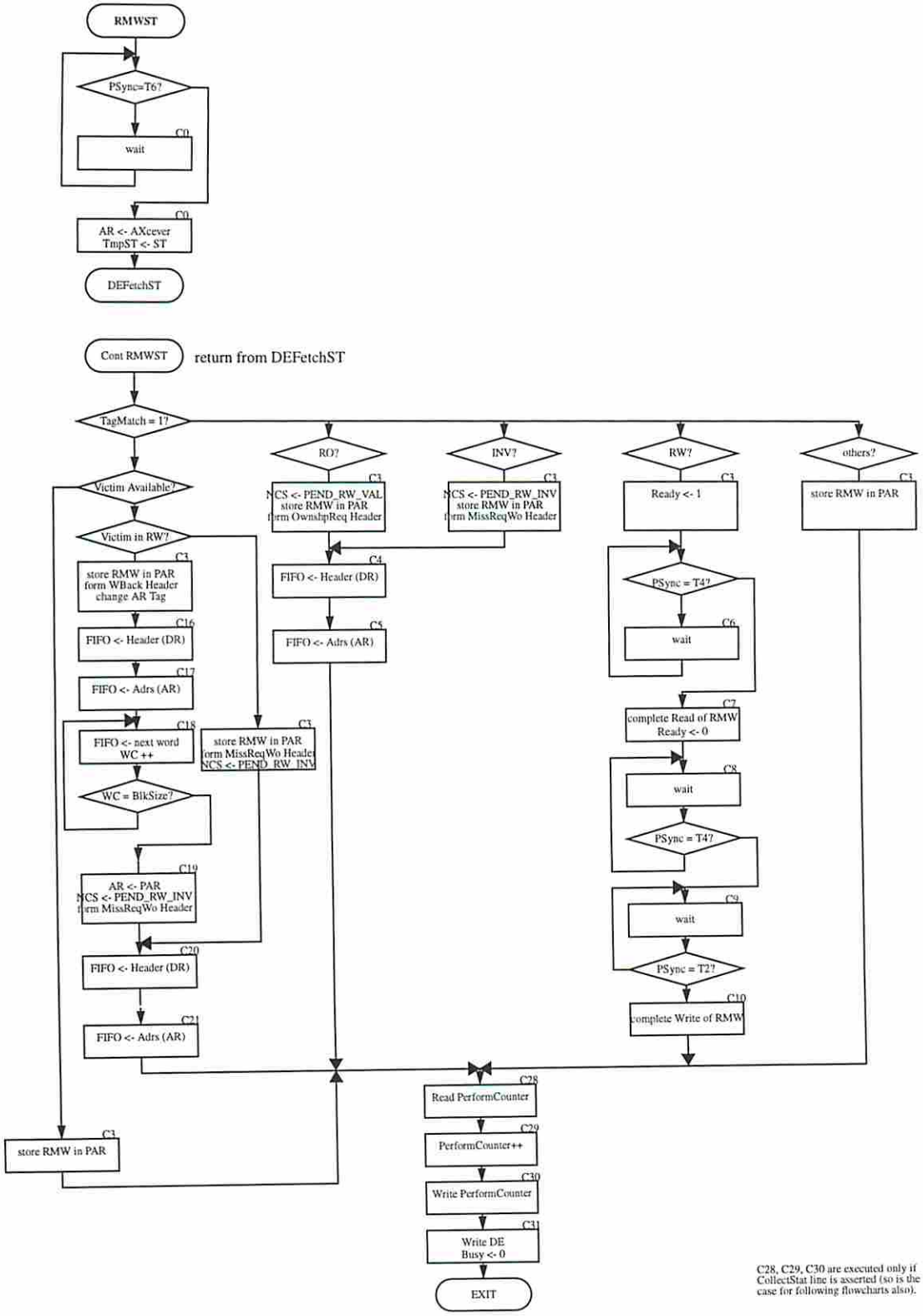


Figure 28. Write request from FLCC (WriteST)

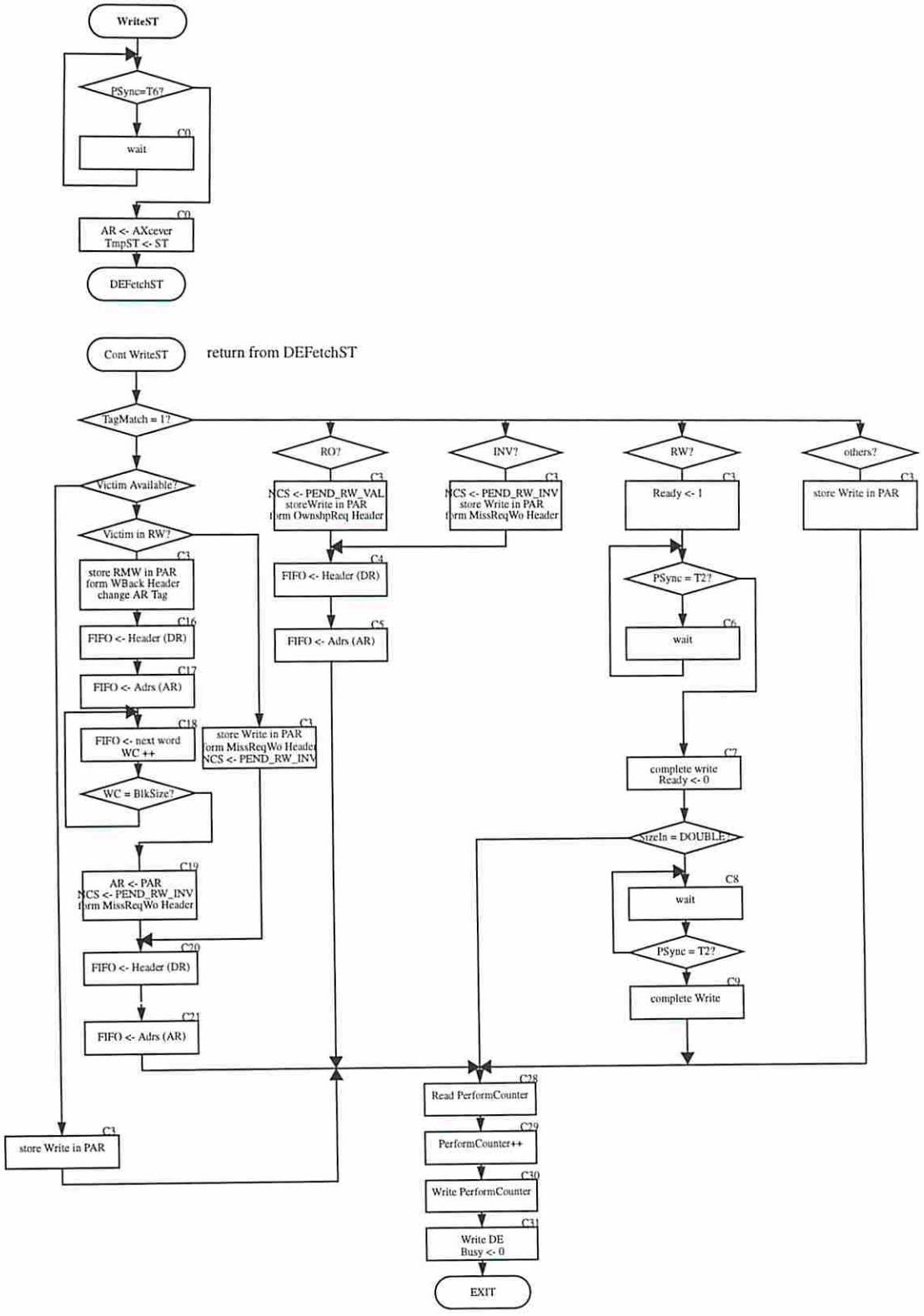


Figure 29. Miss requests from FLCC (MissReqST)

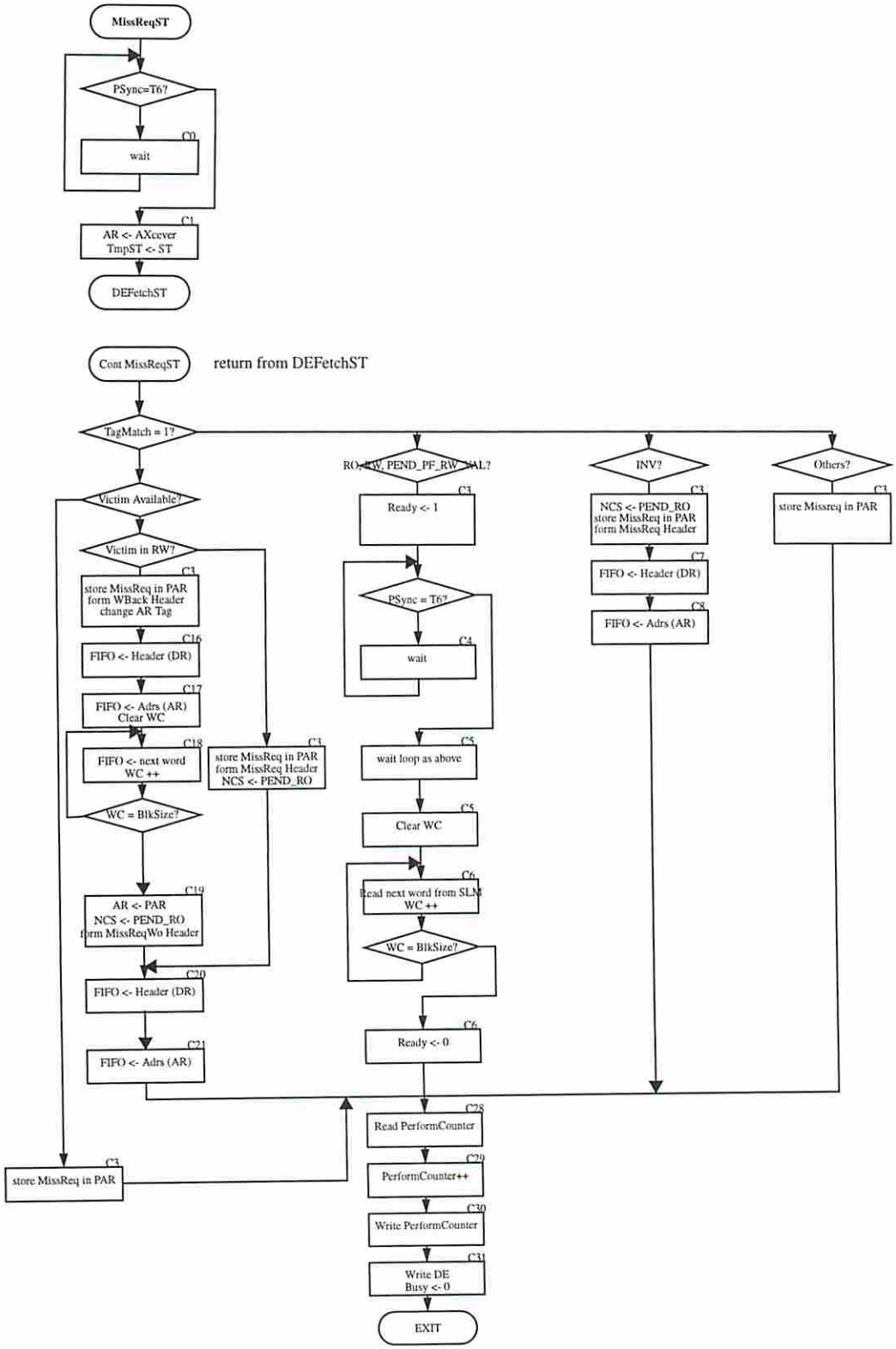


Figure 30. Prefetch operations (PfRoST, PfRwST)

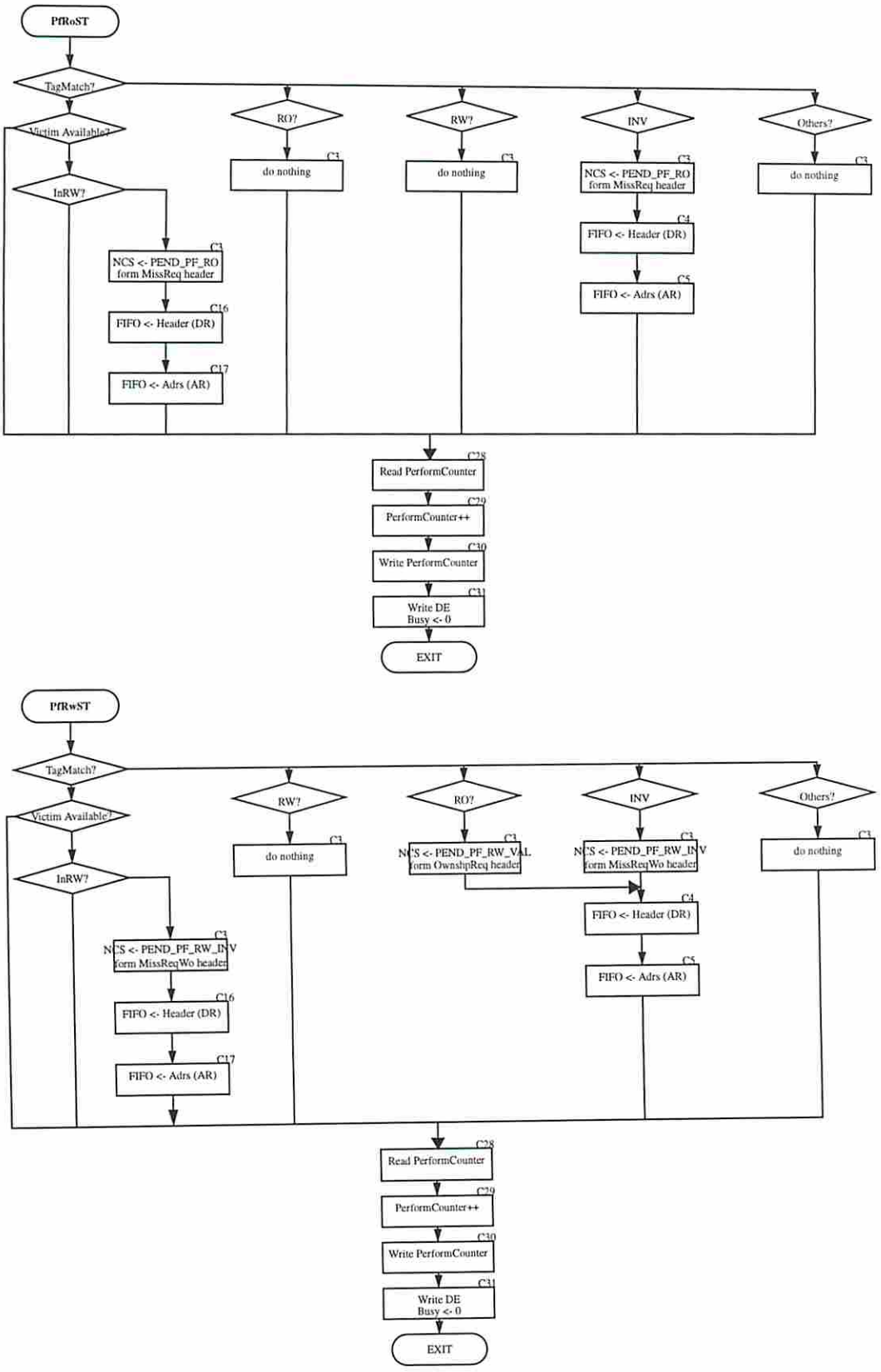


Figure 31. Requests from MC or NIC (part 1)

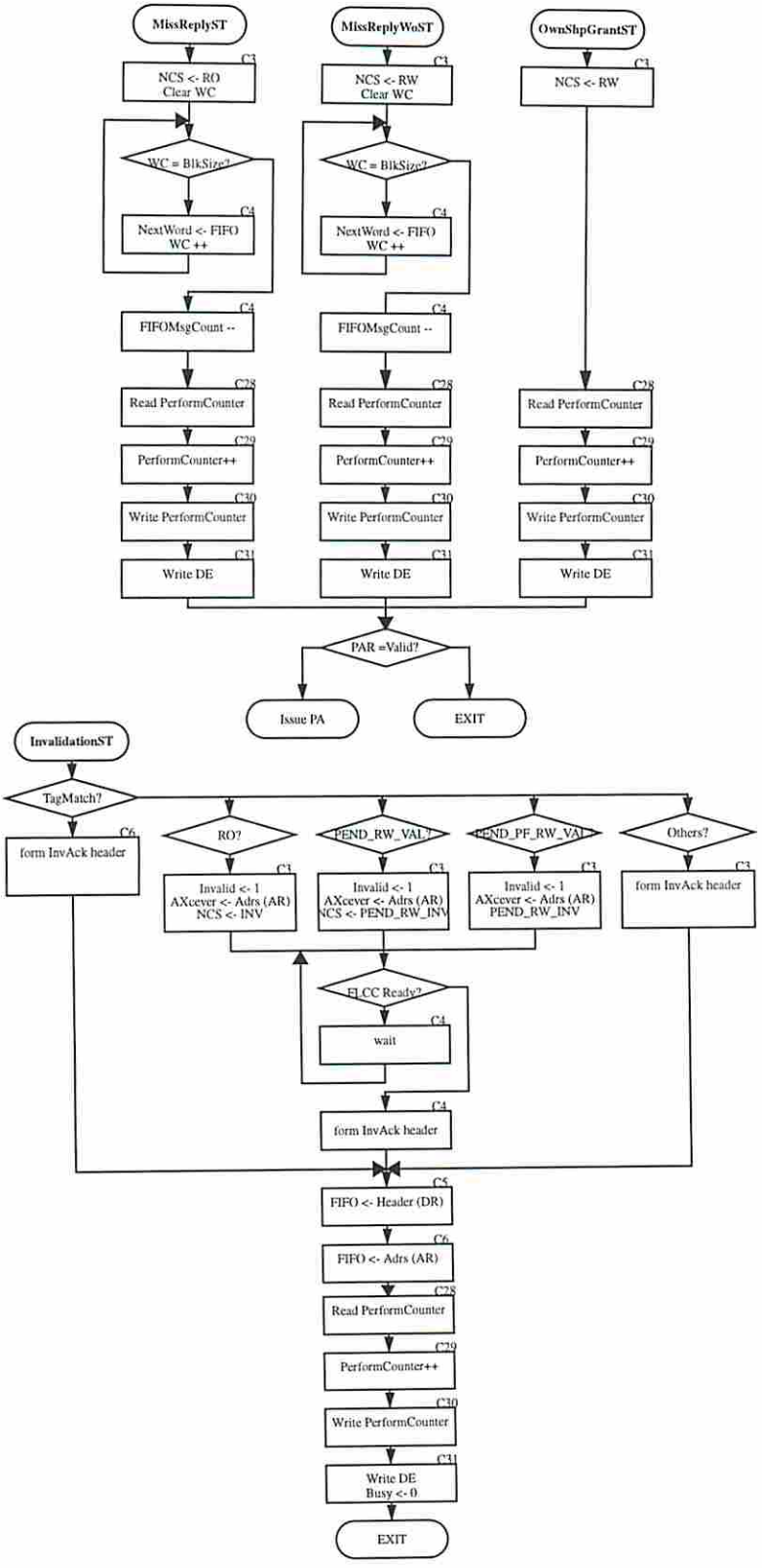


Figure 32. Requests from MC or NIC (part 2)

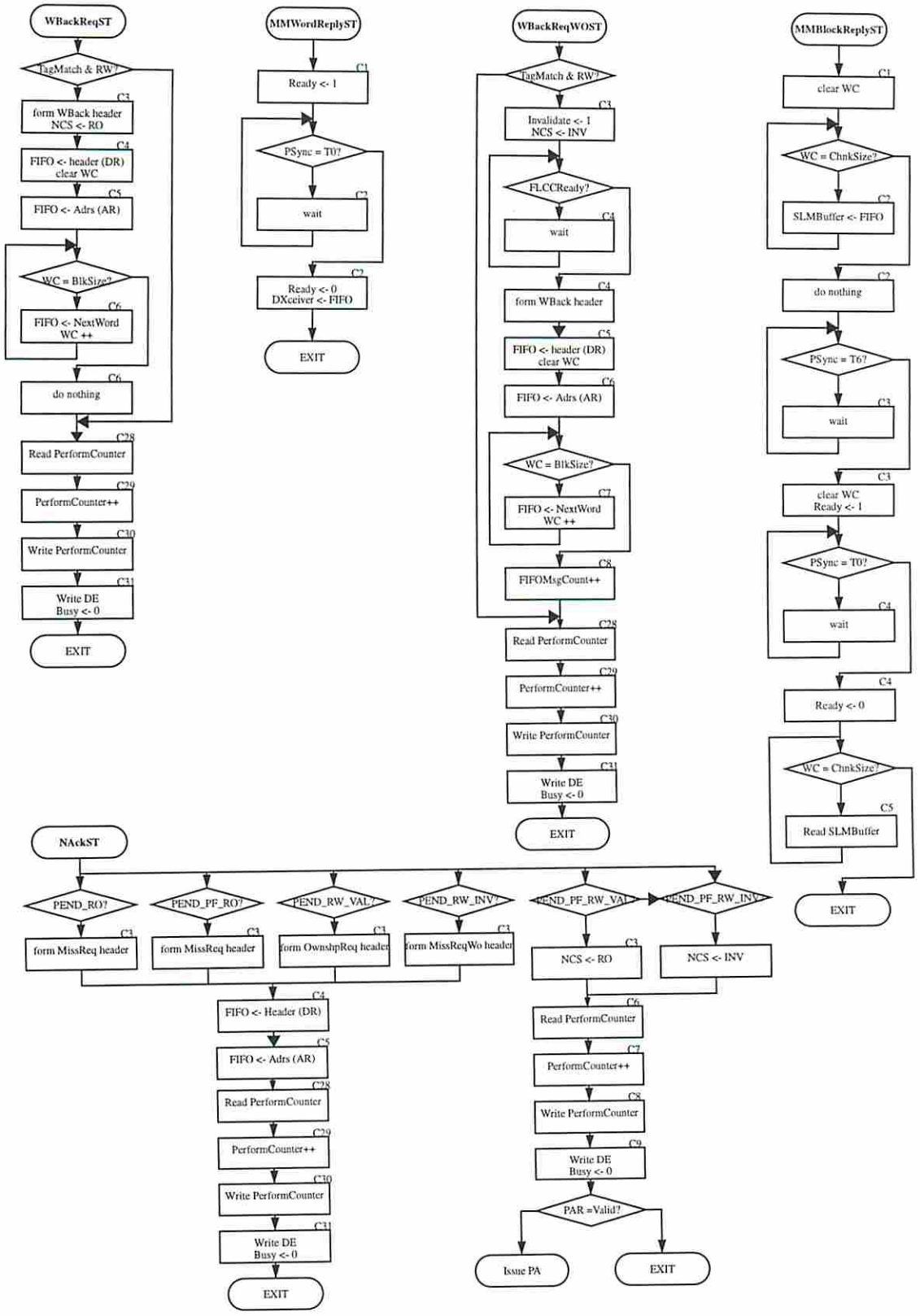
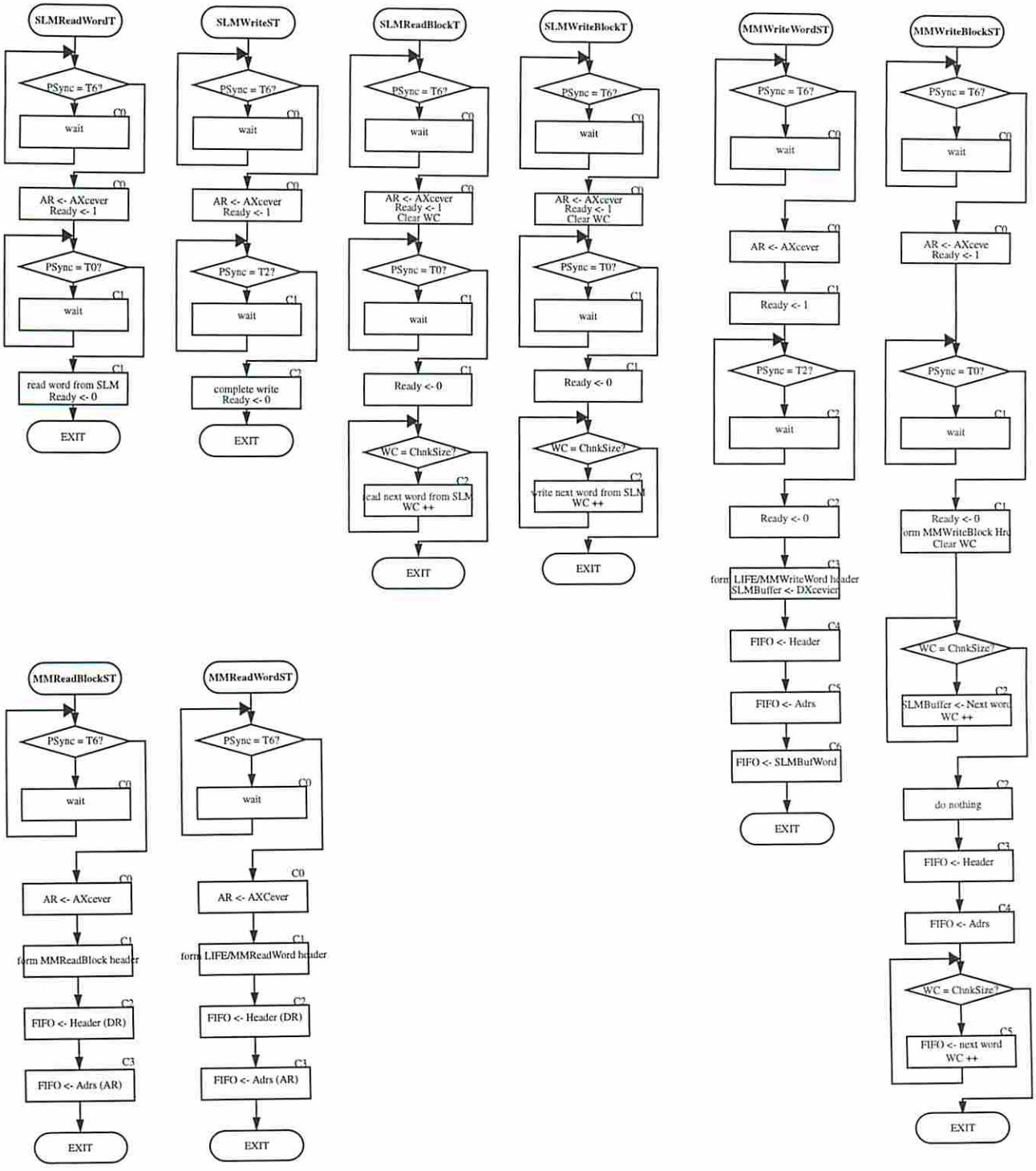


Figure 33. Test mode operations

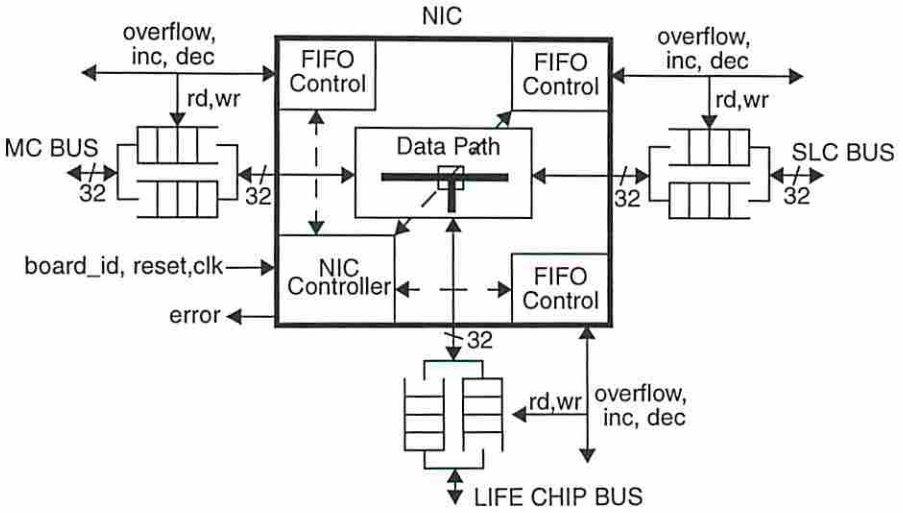


5. Network Interface Chip (NIC) design

The Network Interface Chip (NIC) in RPM-2 replaces the internal bus in the initial design of RPM [1]. We now describe it in details and show the design statistics.

Figure 34 shows the block diagram of NIC. NIC consists of three identical FIFO controls, data path and NIC controller. The FIFO control logic counts the exact number of messages in three sets of bidirectional FIFOs connected to the Second-Level Cache controller (SLCC), the Memory Controller (MC) and the LIFE chip and it transfers messages from NIC to outside buses. The data path is made of three sets of 2 to 1 multiplexors and the NIC controller routes messages with given priority when the data path is free.

Figure 34. Block diagram of NIC



Each FIFO control has two up/down message counters; one for incoming messages and one for outgoing messages. Each counter is increased or decreased by one whenever a message arrives or is sent, respectively. Based on the counter value, the FIFO control detects whether each FIFO contains one or several messages or is about to overflow. Currently the size of each FIFO is 4,096 words and the maximum number of messages in a FIFO is limited to 120 block messages .

The NIC controller is the main control unit of NIC. It consists of a small finite state machine and message header decoder. When the data path is free, it first checks whether there are incoming message(s) from the Futurebus+ via the LIFE chip, from SLCC or from MC by polling each message ready signal generated by each FIFO control. Messages from the LIFE chip have higher priority than messages from the SLCC and messages from MC have the lowest priority. If a messages is ready, then it fetches the message header and determines the destination and the length of the message. Once the header is decoded, it simply copies the incoming message to the outgoing FIFO connected to the destination through the data path. It also checks some error conditions. For example if a message from MC is for MC itself, an interrupt is generated to the processor.

NIC allows us to modify the message format easily. In the previous design, the format of message header was fixed and the design of the internal bus was hardwired to support the header format. For example, one bit in a fixed location of the message header gave the destination of the message. The message type field was also fixed and this prevented us from adding new message types. NIC also allows us to emulate more easily message passing systems. In this case, NIC can act as a communication assist such that it can issue DMA requests to the memory controller without processor intervention or generate interrupts to the processor upon receiving message from the network. The message is deposited either in a

specified region of main memory or in the internal registers of NIC depending on the type of message.

NIC is well structured and written in Synopsys VHDL format from its initial design. Table 22 shows the utilization and the delay of the Xilinx FPGA implementing the NIC controller after mapping by Synopsys tool and Xact 5.0.

Table 22: Design statistics for NIC

design	IO pins (192)	CLB FGs (1152)	CLB FFs (1152)	max P2S	max C2S	max C2P	max P2P	max delay (Synopsys)
NIC (Synopsys)	121 (121)	440 (466)	157 (157)	50.0 ns	74.4 ns	39.5 ns	74.0 ns	64.63 ns

Acknowledgments. Funding for this work has been provided by the National Science Foundation under Grants MIP-9223812 and MIP-9633542. Besides the authors, several individuals have contributed to the project. In particular, we want to thank Per Stenström from Lund University (Sweden) and Massoud Pedram from EE-Systems, U.S.C. Luiz Barroso, Jacqueline Chame, Koray Oner, Sasan Iman, and Krishnan Ramamurthy participated in the design of the RPM emulator. Through their University Program several companies helped reduce the cost of the hardware and software needed for the project. These companies are Advanced Micro Devices, Synopsys, Viewlogic, Axil Workstations and Xilinx. Finally, John Granacki from ISI offered the services of EZFAB, which is part of the ARPA-sponsored Systems Assembly Project.

6. References

[1] Barroso, L.A., Iman, S., Jeong, J., Öner, K., Ramamurthy, K., and Dubois, M., "RPM: A Rapid Prototyping Engine for Multiprocessor Systems," *IEEE Computer*, pp. 26-34, February 1995.

[2] Dubois, M., Gefflaut, A., Jeong, J., Moga, A., and Oner, K. Rapid Hardware Prototyping on RPM-2: Methodology and Experience. Technical Report No. CENG97-27, Dept of EE-Systems, University of Southern California, December 1997.

[3] Dubois, M., Jeong, J., Song, Y., and Moga, A., "Rapid Hardware Prototyping on RPM-2," to appear in *IEEE Design and Test*, September 1998.

[4] Jeong, J., Song, Y., and Dubois, M. The Physical Design of RPM. T. R. No. CENG97-26, Dept of EE-Systems, University of Southern California, December 1997

[5] Öner, K., Barroso, L.A., Iman, S., Jeong, J., Ramamurthy, K., and Dubois, M., "The Design of RPM: An FPGA-based Multiprocessor Emulator," *Proceedings of the 3rd ACM International Symposium on Field-Programmable Gate Arrays*, June 1995.