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
{jaehonj, 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 details 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](image)

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:

<table>
<thead>
<tr>
<th>STATE</th>
<th>Situation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNCACHED</td>
<td>for all ‘x’, pbit(x)=0</td>
</tr>
<tr>
<td>SHARED</td>
<td>there is at least one ‘x’</td>
</tr>
<tr>
<td>DIRTY</td>
<td>there is only one ‘x’</td>
</tr>
<tr>
<td>Sh_Dty_own</td>
<td>locked bit = 1, lock type = 00</td>
</tr>
<tr>
<td>Sh_Dty_miss</td>
<td>locked bit = 1, lock type = 01</td>
</tr>
<tr>
<td>Dty_Sh</td>
<td>locked bit = 1, lock type = 10</td>
</tr>
<tr>
<td>Dty_Dty</td>
<td>locked bit = 1, lock type = 11</td>
</tr>
</tbody>
</table>

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

**Input messages:**
- rmmiss_req: read miss request
- wrmmiss_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**

<table>
<thead>
<tr>
<th>input</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmiss_req</td>
<td>SHARED</td>
<td></td>
<td>pbit(req)=1</td>
<td>miss_reply</td>
</tr>
<tr>
<td>wmiss_req</td>
<td>DIRTY</td>
<td></td>
<td>pbit(req)=1 &amp; dbit=1</td>
<td>miss_reply_own</td>
</tr>
<tr>
<td>own_req</td>
<td>ERR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inv_ack</td>
<td>ERR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wback</td>
<td>ERR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2: State=SHARED**

<table>
<thead>
<tr>
<th>input</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmiss_req</td>
<td>SHARED</td>
<td></td>
<td>pbit(req)=1</td>
<td>miss_reply</td>
</tr>
<tr>
<td>wmiss_req</td>
<td>~(other_shared) &amp; other_shared</td>
<td>DIRTY</td>
<td>Sh_Dty_miss</td>
<td>pbit(req)=1 &amp; dbit=1</td>
</tr>
<tr>
<td></td>
<td>pbit(req)=0 &amp; ~(other_shared) &amp; pbit(req)=1 &amp; other_shared</td>
<td>DIRTY</td>
<td>Sh_Dty_own</td>
<td>pbit(req)=1 &amp; dbit=1</td>
</tr>
<tr>
<td>own_req</td>
<td>ERR</td>
<td></td>
<td>pbit(req)=0 &amp; req_id=req</td>
<td>own_reply for all xlpbit(x)=1 invalidate(x)</td>
</tr>
<tr>
<td>inv_ack</td>
<td>ERR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wback</td>
<td>ERR</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3: State=DIRTY**

<table>
<thead>
<tr>
<th>input</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmiss_req</td>
<td>req=dirty</td>
<td>ERR</td>
<td>req_id = req</td>
<td>wback_req</td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>Dty_Sh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wmiss_req</td>
<td>req=dirty</td>
<td>ERR</td>
<td>req_id = req</td>
<td>wback_req_own</td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>Dty_Dty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>own_req</td>
<td>ERR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inv_ack</td>
<td>ERR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wback</td>
<td>req = dirty</td>
<td>UNCACHED</td>
<td>pbit(req)=0 &amp; dbit=0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 4: State=Sh_Dty_own

<table>
<thead>
<tr>
<th>input</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>rm iss req</td>
<td></td>
<td>Sh_Dty_own</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>wm iss req</td>
<td></td>
<td>Sh_Dty_own</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>own req</td>
<td></td>
<td>Sh_Dty_own</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>inv_ack</td>
<td>one_left</td>
<td>DIRTY</td>
<td>pbit(req_id)=1 &amp; dbit=1</td>
<td>own_reply</td>
</tr>
<tr>
<td></td>
<td>~(one_left)</td>
<td>Sh_Dty_own</td>
<td>pbit(req) = 0</td>
<td></td>
</tr>
<tr>
<td>wb ack</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 5: State=Sh_Dty_miss

<table>
<thead>
<tr>
<th>input</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>rm iss req</td>
<td></td>
<td>Sh_Dty_miss</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>wm iss req</td>
<td></td>
<td>Sh_Dty_miss</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>own req</td>
<td></td>
<td>Sh_Dty_miss</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>inv_ack</td>
<td>one_left</td>
<td>DIRTY</td>
<td>pbit(req_id)=1 &amp; dbit=1</td>
<td>miss_reply_own</td>
</tr>
<tr>
<td></td>
<td>~(one_left)</td>
<td>Sh_Dty_miss</td>
<td>pbit(req) = 0</td>
<td></td>
</tr>
<tr>
<td>wb ack</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 6: State=Dty_Sh

<table>
<thead>
<tr>
<th>input</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>rm iss req</td>
<td>req != dirty</td>
<td>Dty_Sh</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td></td>
<td>req = dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wn iss req</td>
<td>req != dirty</td>
<td>Dty_Sh</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td></td>
<td>req = dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>own req</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inv_ack</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wb ack</td>
<td></td>
<td>ERR</td>
<td></td>
<td>miss_reply</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>req = dirty</td>
<td>SHARED</td>
<td>dbit=0 &amp; pbit(req_id) =1</td>
<td>miss_reply</td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 7: State=Dty_Dty

<table>
<thead>
<tr>
<th>input</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>rmis_req</td>
<td>req != dirty</td>
<td>Dty_Dty</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td></td>
<td>req = dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wmiss_req</td>
<td>req != dirty</td>
<td>Dty_Dty</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td></td>
<td>req = dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>own_req</td>
<td></td>
<td></td>
<td></td>
<td>ERR</td>
</tr>
<tr>
<td>inv_ack</td>
<td></td>
<td></td>
<td></td>
<td>ERR</td>
</tr>
<tr>
<td>wbback</td>
<td>req = dirty</td>
<td>DIRTY</td>
<td>pbit(dirty) = 0 &amp;</td>
<td>miss_reply_own</td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>ERR</td>
<td>pbit(req_id) = 1</td>
<td></td>
</tr>
</tbody>
</table>
### Inverted Tables

**Table 8: input message = rmiss_req**

<table>
<thead>
<tr>
<th>state</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNUNCACHED</td>
<td></td>
<td>SHARED</td>
<td>pbit(req) = 1</td>
<td>miss_reply</td>
</tr>
<tr>
<td>SHARED</td>
<td></td>
<td>SHARED</td>
<td>pbit(req) = 1</td>
<td>miss_reply</td>
</tr>
<tr>
<td>DIRTY</td>
<td>req=dirty</td>
<td>ERR</td>
<td>req_id = req</td>
<td>wback_req</td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>Dty_Sh</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sh_Dty_own</td>
<td></td>
<td>Sh_Dty_own</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>Sh_Dty_miss</td>
<td></td>
<td>Sh_Dty_miss</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>Dty_Sh</td>
<td>req != dirty</td>
<td>Dty_Sh</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td></td>
<td>req = dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dty_Dty</td>
<td>req != dirty</td>
<td>Dty_Dty</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td></td>
<td>req = dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 9: input message = wmiss_req**

<table>
<thead>
<tr>
<th>state</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNUNCACHED</td>
<td></td>
<td>DIRTY</td>
<td>pbit(req) = 1 &amp; dbit = 1</td>
<td>miss_reply_own</td>
</tr>
<tr>
<td>SHARED</td>
<td>~(other_shared)</td>
<td>DIRTY</td>
<td>pbit(req)=1 &amp; dbit=1</td>
<td>miss_reply_own</td>
</tr>
<tr>
<td></td>
<td>other_shared</td>
<td>Sh_Dty_miss</td>
<td>pbit(req)=0 &amp; req_id=req</td>
<td>forall x[pbit(x)=1 invalidate(x)]</td>
</tr>
<tr>
<td>DIRTY</td>
<td>req=dirty</td>
<td>ERR</td>
<td>req_id = req</td>
<td>wback_req_own</td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>Dty_Dty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sh_Dty_own</td>
<td></td>
<td>Sh_Dty_own</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>Sh_Dty_miss</td>
<td></td>
<td>Sh_Dty_miss</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>Dty_Sh</td>
<td>req != dirty</td>
<td>Dty_Sh</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td></td>
<td>req = dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dty_Dty</td>
<td>req != dirty</td>
<td>Dty_Dty</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td></td>
<td>req = dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 10: input message = own_req**

<table>
<thead>
<tr>
<th>state</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNUNCACHED</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARED</td>
<td>pbit(req)=0</td>
<td>ERR</td>
<td>pbit(req)=1 &amp; dbit=1</td>
<td>own_reply</td>
</tr>
<tr>
<td></td>
<td>pbit(req)=1 &amp; ~(other_shared)</td>
<td>ERR</td>
<td>pbit(req)=0 &amp; req_id=req</td>
<td>forall x[pbit(x)=1 invalidate(x)]</td>
</tr>
<tr>
<td>DIRTY</td>
<td></td>
<td>DIRTY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sh_Dty_own</td>
<td></td>
<td>Sh_Dty_own</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>Sh_Dty_miss</td>
<td></td>
<td>Sh_Dty_miss</td>
<td></td>
<td>nack</td>
</tr>
<tr>
<td>Dty_Sh</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dty_Dty</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 11: input message = wback

<table>
<thead>
<tr>
<th>state</th>
<th>conditions</th>
<th>next state</th>
<th>actions</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNCAUGHTED</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHARED</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIRTY</td>
<td>req = dirty</td>
<td>UNCAUGHTED</td>
<td>pbit(req)==0 &amp; dbit=0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sh_Dty_own</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sh_Dty_miss</td>
<td></td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dty_Sh</td>
<td>req = dirty</td>
<td>SHARED</td>
<td>dbit=0 &amp; pbit(req_id) =1</td>
<td>miss_reply</td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dty_Dty</td>
<td>req = dirty</td>
<td>DIRTY</td>
<td>pbit(dirty) = 0 &amp; pbit(req_id) =1</td>
<td>miss_reply</td>
</tr>
<tr>
<td></td>
<td>req != dirty</td>
<td>ERR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Count value</th>
<th>receive</th>
<th>transmit</th>
<th>situation</th>
</tr>
</thead>
</table>
| 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**

Full 32-bit emulated memory address

| blockframe base address | log(blsz/4) | 2 |

Initial address of a block frame in the DRAMS

| 0's | log(blsz/4) | 0's |

Address of a directory entry in the DRAMS

| 0's | 1 | 0's |

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

- **ST_RD**
  - lock bit = 1 ?
    - Y: NACK
    - N: DIRT Y ?
      - Y: lock bit=1 & lock type = Dty_Sh_miss & req_id = req & dirty = decode(pbit(X)) & build whack_req header & compute suspend time
      - N: ST_SUSPEND
      - pbit(req) = 1 & build miss_reply header & compute suspend time

Figure 10. Write miss request

- **ST_WR**
  - lock bit = 1 ?
    - Y: NACK
    - N: DIRT Y ?
      - Y: lock bit=1 & lock type = Dty_Dty & req_id = req & dirty = decode(pbit(X)) & build whack_req own header & compute suspend time
      - N: ST_SUSPEND
      - UNCA NCHED or ~ (other_shared) ?
        - Y: lock bit=1 & lock type = Sh_Dty_miss & req_id = req & build inval. header & compute suspend time
        - N: pbit(req) = 1 & build miss_reply own header & compute suspend time
Figure 11. Ownership request

ST_OW

lock bit = 1 ?

Y -> NACK

N

DIRTY or UNCAVED or pbit(req) = 0 ?

Y -> ERROR

N

other_shared ?

Y

lock bit = 1 & pbit(req) = 0 &
lock type = Sh_Dry_own &
req_id = req &
build inval. header &
compute suspend time

N

build own_reply header &
pbit(req) = 1 & dbit = 1 &
compute suspend time

ST_SUSPEND

Figure 12. Invalidation acknowledgment

ST_IA

pbit(req) = 0

ST_IA

lock bit = 0 or Dty_Sh or Dty_Dry ?

Y -> ERROR

N

pbit(X) = 0

Y

Sh_Dry_own ?

Y

lock bit = 0 & pbit(req_id) = 1 & dbit = 1 &
build miss_reply_own header &
compute suspend time

N

N

lock bit = 0 & pbit(req_id) = 1 & dbit = 1 &
buid own_reply header &
compute suspend time

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

**READ_WORD_REQUEST**
- receive (rword_req)
- compute addr & build rword_reply header & start read of data word
- request local bus & latch data word
- local bus granted?
  - Y: put header on local bus
  - N: wait one cycle
- put header on local bus
- put address on local bus
- put word on local bus
- END

**READ_BLOCK_REQUEST**
- receive (rmiss_req) & compute block addr.
- build rblock_reply header & request local bus
- local bus granted?
  - Y: put header on local bus & start read of first data word
  - N: wait one cycle
- put address on local bus & latch first data word & incr. block address
- put word on local bus & start read of next data word
- latch next data word & incr. block address
- more words?
  - Y: ...
  - N: ...
- put last word on local bus & END

**WRITE_BLOCK_REQUEST**
- receive (wblock_req) & compute block addr.
- start store of block word & latch word from local bus
- wait until store commits & put local bus on "WAIT" & incr. block address
- more words?
  - Y: ...
  - N: ...
- put last word on local bus & END

**WRITE WORD REQUEST**
- receive (sword_req)
- compute addr & read word from local bus & start write of data word
- write data word
- END
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 accommodating 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 SYNOPSIS

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).
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 M HOLD in the processor; when the access is completed, it de-asserted M HOLD 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 pelock. 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 pelock.

Fle_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**

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

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
Table 14: Output signals from FLC

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

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

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:

```
31  28  21  0
BusID  0's  0x000000 to 0x1FFFFF
```

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:

```
31  28  23  0
BusID  0's  0x000000 to 0x7FFFFF
```

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

```
31  28  26  0
BusID  0's  0x000000 to 0x3FFFFFFF
```

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

<table>
<thead>
<tr>
<th>ASI[7..0]</th>
<th>Address Range</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x05</td>
<td>0xXX0XXXXXX</td>
<td>Control Register (pclk control, boot ROM address mapping, start control)</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXX1XXXXXX</td>
<td>FPGA programming (FPGA selector register)</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXX2XXXXXX</td>
<td>FPGA programming (FPGA data)</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXX3XXXXXX</td>
<td>FPGA status register</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXX4XXXXXX</td>
<td>7-segment display data register</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXX5XXXXXX</td>
<td>SCSI controller</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXX6XXXXXX</td>
<td>Serial I/O</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXX7XXXXXX</td>
<td>Real time clocker</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXX8XXXXXX</td>
<td>DRAM controller (data register)</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXX9XXXXXX</td>
<td>DRAM controller (address register)</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXXAXXXXXX</td>
<td>Delay unit (word register)</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXXBXXXXXX</td>
<td>LIFE chip registers</td>
</tr>
<tr>
<td>0x05</td>
<td>0xXXCXXXXXX</td>
<td>Delay unit (initialization register)</td>
</tr>
</tbody>
</table>
In emulation mode, shared data and private data memory are mapped by their ASI and address

<table>
<thead>
<tr>
<th>ASI[7..0]</th>
<th>Alternate Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x08</td>
<td>User Instruction</td>
</tr>
<tr>
<td>0x09</td>
<td>Supervisor Instruction</td>
</tr>
<tr>
<td>0x0A</td>
<td>User Data</td>
</tr>
<tr>
<td>0x0B</td>
<td>Supervisor Data</td>
</tr>
</tbody>
</table>

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

```
<table>
<thead>
<tr>
<th>31</th>
<th>29</th>
<th>12</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc type</td>
<td>page number</td>
<td>page offset</td>
<td></td>
</tr>
</tbody>
</table>
```

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 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>
<thead>
<tr>
<th>A[31..29]</th>
<th>Instruction</th>
<th>Type of access</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>load/store/test-and-set/swap</td>
<td>normal access to simulated private memory</td>
</tr>
<tr>
<td>001</td>
<td>load/store/test-and-set/swap</td>
<td>normal access to simulated shared memory</td>
</tr>
<tr>
<td>010</td>
<td>load/store</td>
<td>prefetch access to simulated private memory</td>
</tr>
<tr>
<td>011</td>
<td>load/store</td>
<td>prefetch access to simulated shared memory</td>
</tr>
<tr>
<td>10X</td>
<td>store</td>
<td>lock release operation to an address</td>
</tr>
<tr>
<td>11X</td>
<td>load/store/test-and-set/swap</td>
<td>spin lock on/off, statistics on/off</td>
</tr>
</tbody>
</table>
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, `mapperr.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 `designdata.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>
<thead>
<tr>
<th>Table 19: Synthesis Results Comparisons for FLCCU</th>
</tr>
</thead>
<tbody>
<tr>
<td>!<img src="image" alt="" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Viewsynthesis</th>
<th>Synopsys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occupied CLBs</td>
<td>408</td>
<td>521</td>
</tr>
<tr>
<td>Packed CLBs</td>
<td>266</td>
<td>305</td>
</tr>
<tr>
<td>Bonded I/O Pins</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>CLB Flip Flops</td>
<td>44</td>
<td>79</td>
</tr>
<tr>
<td>Maximum Delays</td>
<td>112.6 ns</td>
<td>111.8 ns</td>
</tr>
</tbody>
</table>

FLCCU designed with 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>
<thead>
<tr>
<th></th>
<th>Viewsynthesis</th>
<th>Synopsys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Occupied CLBs</td>
<td>385</td>
<td>529</td>
</tr>
<tr>
<td>Packed CLBs</td>
<td>260</td>
<td>333</td>
</tr>
<tr>
<td>Bonded I/O Pins</td>
<td>180</td>
<td>178</td>
</tr>
<tr>
<td>CLB Flip Flops</td>
<td>230</td>
<td>234</td>
</tr>
<tr>
<td>Maximum Delays</td>
<td>96.4 ns</td>
<td>71.5 ns</td>
</tr>
</tbody>
</table>

### 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

INV

2  Latch Inv. Address
    Generate Tag Address

3  Read Tag

4  Write Invalid State into Tag Memory
    Invalidation event

5  Set Inv_ack

6  Performance stats.

7  Performance stats.

8  Performance stats.
    Performance stats.
    Unblock Processor

START

PRF

2  Latch original Address
    Generate TLB Address

3  Read Tag Address
    Generate Tag Address

4  TLB hit

5  Read Tag Memory
    Compare tags

6  NOP

7  Performance stats.

8  Performance stats.
    Unblock Processor

START
Figure 21. Read

```
RD
Normal Mode YES -> RD_norm
NO -> RD_SCL
FLC read NO -> RD_SCL
YES Register RD_FLC_reg
RD_SCL
RD_FLC_reg
RD_FLC_mem

RD_FLC_mem
Performance Stats.
Read Origial Address
Enable register output
Open Data Buffer Output
Performance Stats.
NOP
Unblock Processor
START

RD_FLC_reg
Performance Stats.
Read Original Address
Set Req_Type
Open Data Buffer Output
NOP
Unblock Processor
START

RD_SCL
Performance Stats.
Send Request to SLC
Performance Stats.

RD_norm
Latch original address
Generate TLB Address
Read TLB Tag Memory Address
TLB hit NO TRAP
YES
Read Tag Memory
Compare Tags
Generate Data Address
double NO
hit YES

Read Data Clear Double
Performance stats.
Performance stats.
Performance stats.
Unblock processor
START

double NO

READ

RD_miss
miss reply NO Wait for next clock
Performance stats.
YES Block Transfer
Read Missed data
Clear Miss_Req signal
Reset Pending state
double word
wait for next clock
Unblock Processor
START

set double
Wait for next clock
Unblock Processor

set pending state
START
```
Figure 22. Write

- **WR**
  - Normal Mode: **YES** → **WR_norm**
  - FLC wrt: **NO** → **WR_SLC**
    - **YES** → Register acc type **WR_FLC_reg**
      - Memory → **WR_FLC_mem**
        - **Write_Cont**
          - Write data to FLC & SLC
          - Generate next address Block Processor → **NOP**
            - Performance stats. Unblock Processor → **Double Word**
              - **YES** → Write data to FLC & SLC
                - Generate next address Block Processor → **NOP**
                  - Performance stats. Unblock Processor → **START**
            - **NO** → Double Word
              - **NO** → Write data to FLC & SLC
                - Generate next address Block Processor → **NOP**
                  - Performance stats. Unblock Processor → **START**
  - **WR_SLC**
    - Performance Stats.
    - Read Orginal Address Set Req_Type
    - Send Request to SLC
    - Performance Stats.
    - **START**
  - **WR_start**
    - Generate TLB address
      - Latch original address
      - Read TLB Generate Tag Memory Address
      - TLB hit
        - **NO** → **TRAP**
        - **YES** → **Write_Miss**
          - Read Tag Memory Compare Tags Generate org. Address
          - **YES** → Hit
            - **NO** → Pass address to SLCC
              - Set Miss_Req signal
                - Put miss address on address bus
      - **NO** → **START**
      - **NO** → **START**
    - **Performance stats.**
    - **Performance stats.**
    - **Performance stats.**
    - Set Pending State
    - Wait for 1 pclock
      - **START**
  - **WR_FLC_reg**
    - Performance Stats.
    - Read Orginal Address
      - Performance Stats.
      - Open Data Buffer Output Set Buffer Direction
        - **NO** → **NOP**
          - Unblock Processor
            - **START**
  - **WR_FLC_mem**
    - Performance Stats.
    - Read Orginal Address
      - Performance Stats.
      - Open Data Buffer Output Set Buffer Direction
        - **NOP** → **NOP**
          - Unblock Processor
            - **START**
      - **START**
Figure 23. Test and Set

```
T&S
  Pending NO T&S_Start
     YES
  Ready NO Wait for 1 p clock
     YES
  Hit NO START
      YES
T&S_Cont
      T&S_Miss

T&S_start
  Block Processor
  Latch Memory Address
  Generate Tag Memory Address
  Read Tag Memory
  Compare Tags
  YES
  HIT
  NO
  Pass address to SLCC
  Set Miss_Req signal
  Put miss address on address bus
  NOP
  Set Pending State
  Performance stats.
  Performance stats.
  Performance stats.
START

T&S_Cont
  Read Data
  NOP
  Performance Stats.
  Unblock Processor
  Performance Stats
  Block Processor
  Unblock Processor
  Write data to FLC & SLC
  Block Processor
  NOP
  Unblock Processor
  Performance Stats
  START

T&S_Miss
  Read Data
  NOP
  Performance Stats.
  Unblock Processor
  Performance Stats
  Block Processor
  Unblock Processor
  Write data to SLC
  Block Processor
  NOP
  Unblock Processor
  Performance Stats
  START
```
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](image)

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**

```plaintext
Init
Final tag match <= 0;
victim state <= n/a:

Final TagMatch?
Y

more dir_entries?
N

Fetch next dir_entry

EXIT

Tag Match?
N
better victim?
N
Y

Final TagMatch <= 1
Store this as the best victim
```

### 4.3. Experience with Synopsys tool and performance analysis

We have converted the design with old tools (Vesysynthesis+Vsis) into Synopsys version. The conversion was straightforward 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>
<thead>
<tr>
<th>designs</th>
<th>IO pins (192)</th>
<th>CLB FGs (1152)</th>
<th>CLB FFs (1152)</th>
<th>max P2S</th>
<th>max C2S</th>
<th>max C2P</th>
<th>max P2P</th>
<th>max delay (Synopsys)</th>
</tr>
</thead>
<tbody>
<tr>
<td>data unit (old tool)</td>
<td>181</td>
<td>804</td>
<td>206</td>
<td>78.8 ns</td>
<td>77.0 ns</td>
<td>47.4 ns</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>data unit (Synopsys)</td>
<td>178 (178)</td>
<td>516 (557)</td>
<td>206 (206)</td>
<td>57.2 ns</td>
<td>60.4 ns</td>
<td>39.5 ns</td>
<td>40.0 ns</td>
<td>29.7 ns</td>
</tr>
<tr>
<td>control unit (old tool)</td>
<td>126</td>
<td>786</td>
<td>44</td>
<td>95.6 ns</td>
<td>118.0 ns</td>
<td>96.9 ns</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>control unit (Synopsys)</td>
<td>123 (123)</td>
<td>738 (1056)</td>
<td>119 (183)</td>
<td>105.0 ns</td>
<td>146.7 ns</td>
<td>111.2 ns</td>
<td>N/A</td>
<td>119.01 ns</td>
</tr>
</tbody>
</table>

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

Messages from MC or NIC

StartST

- Fetch ST
- Msg from FLCC

Requests from FLCC

- Busy c 1
- Busy c 1
- Busy c 1
- Busy c 1

Read/FIFO

Prefetch operations
(checking request buffer)

- RBBNotEmptyST
- WorkST
- MissReplST
- RBBNotEmptyST

Test mode operations

- MMReadWordST
- MMWriteWordST
- MMReadBlockST
- MMWriteBlockST
- MMReadWordST
- MMWriteWordST
- MMReadBlockST
- MMWriteBlockST

- ReqPt c 1
- ReqPt c 1

AR c Datasrc

- Dir c FIFO (dirsrc)
- AR c FIFO (dirsrc)

Directory entry fetch subroutine

Fig. 2
Fig. 3
Fig. 4
Fig. 5
Fig. 6
Fig. 7

- TmpState c MissReplST
- TmpState c MissReplWST
- TmpState c MissReplWST
- TmpState c MissReplWST
- TmpState c MissReplWST
- TmpState c MissReplWST
- TmpState c MissReplWST
- TmpState c MissReplWST

- DEFxeST
- DEFxeST
- DEFxeST
- DEFxeST
- DEFxeST
- DEFxeST
- DEFxeST

Note: The next state is stored into TmpState and retrieved after DEFxeST.
Figure 27. Read-Modify-Write request from FLCC (RMWST)

RMWST

FetchReq?

wait

AR ← Access
TSST ← ST

DEFetchST

RMWST

return from DEFetchST

TagMatch ← 1

Vector Available?

Vector in RW?

store RMW in PAR from Wishack Reader
change AR Tag

FIFO ← Header (DR)

FIFO ← Addr (AR)

FIFO ← next word
RW ←

WST ← RWST

AR ← PAR

FIFO ← FEND_RW_INV
from MissReqNo-Header

FIFO ← Header (DR)

FIFO ← Addr (AR)

store RMW in PAR

Ready ← 1

FIFO ← Tag

wait

store RMW in PAR

complete Read of RMW
Ready ← 0

wait

store RMW in PAR

complete Write of RMW

C71, C72, C70 are executed only if
Conflict logic is specified to use the
case for following branch militia.
Figure 28. Write request from FLCC (WriteST)

WriteST

PSqStk = T

wait

AR = ALCore
TagST = ST

DEFetchST

Con WriteST

TagMatch = 1

Victim AVAILABLE

Victim in RW

store RW in PPH with Write in PPAK with OverrideFlag
Header

FIFO = Header (DR)

FIFO = Ads (AR)

FIFO = text word

WC = Hash

AR = PPAK

NCS = PEND_RW_INV

store Write in PPAK with MissReqWoHeader

NCS = PEND_RW_INV

store Write in PPAK

FIFO = Header (DR)

FIFO = Ads (AR)

store Write in PPAK

Read PerfCounter

PerfCounter*

Write PerfCounter

Write DIB

Busy = 0

EXIT
Figure 29. Miss requests from FLCC (MissReqST)

- MissReqST
  - Psys = True?
    - wait
  - AR ← AX?Vec
    - TagST ← ST
    - DEFetchST

- Cont.MissReqST
  - TagMatch = ST
    - MissAvailable
      - Miss in PAM
        - store MissReq in PAM from MissReq Header
          - change AR Tag
        - FIFO ← Header
        - FIFO ← Adv (AR) Clear WC
      - FIFO ← next word
        - WC ← BackErr
      - AR ← PAM
        - NCS ← PEND_REQ form MissReq No Header
        - FIFO ← Header
        - FIFO ← Adv (AR)
      - FIFO ← PEND_REQ form MissReq Header

- Ready ← 1
  - FIFO = True
    - FIFO ← WC
      - wait
  - FIFO = False
    - FIFO ← Header (DR)
  - FIFO ← Adv (AR)

- NCS ← PEND_REQ form MissReq in DMA
  - FIFO ← Header
  - FIFO ← Adv (AR)

- Others?
  - store Missreq in PAM

- BOUSH: PEND_REQ AL?
  - Ready ← 0
    - FIFO ← WC
      - wait loop as above
    - FIFO ← WC
      - WC ← BackErr
      - ready ← 0

- Read PerforCount
  - PerformCount++
  - Write PerformCount
  - Write DE
    - Busy ← 0

EXIT
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

![Block Diagram of NIC](image)

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 message 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

<table>
<thead>
<tr>
<th>design</th>
<th>IO pins (192)</th>
<th>CLB FGs (1152)</th>
<th>CLB FF's (1152)</th>
<th>max P2S</th>
<th>max C2S</th>
<th>max C2P</th>
<th>max P2P</th>
<th>max delay (Synopsys)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIC (Synopsys)</td>
<td>121 (121)</td>
<td>440 (466)</td>
<td>157 (157)</td>
<td>50.0 ns</td>
<td>74.4 ns</td>
<td>39.5 ns</td>
<td>74.0 ns</td>
<td>64.63 ns</td>
</tr>
</tbody>
</table>

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


