A Multi-processor System for Prolog Processing

H. C. Fu, C. Chen, C. P. Chung, J. J. J. Shann

Computer System Laboratory
Department of Computer Engineering
National Chiao Tung University
Hsinchu, Taiwan 30050
R O C

Abstract
In this paper, we propose a RISC-based multiprocessor system, called MIEP (Multiple Inference Engine for Prolog), to construct an intelligent workstation for AI applications. The MIEP contains general purpose processor modules (processor with local memory) and PIER modules interconnected by a bus. A PIER module contains a LISCP as the kernel Prolog processor, a floating point co-processor, FPC, and a cache memory. The PIER module has been simulated and evaluated with some benchmark programs. Primary results show that PIER can achieve peak performance more than 500 KILIPS. In addition, a parallel execution model for Prolog on MIEP has been studied and evaluated. Many design issues and features of the MIEP system are presented in this paper.

1. Introduction
Since the Japanese 5th-Generation Computer project began in 1981, Prolog language has been considered as one of the major programming languages for artificial intelligence and symbolic manipulation applications [1, 2, 3]. Unfortunately, conventional implementations for Prolog execution have been hindered by slow execution rate [4] due to:
(1) dynamic type checking,
(2) dynamic memory allocation and deallocation,
(3) frequently context switching for backtracking, and
(4) interpretation type of execution.

Until now, several high speed Prolog machines have been proposed, to name a few, PLM [5], HPM[6], WAM[7], etc. These machines are designed with complex instruction sets and microarchitectures. In this paper, we propose a reduced instruction set architecture [8,9], PIER: a Prolog Inference Engine on a RISC. PIER is designed for compiled Prolog execution at a significant reduced instruction set and structure. In addition, using multiple RISCs to enhance the performance was also proposed and developed by [10]. In this paper, we propose a RISC-based multiprocessor system, called MIEP (Multiple Inference Engine for Prolog), as a research vehicle for intelligent workstations.

We initiated the MIEP since November 1985. The goal of the first stage is to design and to construct a RISC-based multiprocessor system to host and to execute the Prolog programs with high performance. The whole project has been planned to be completed in five years. In the first year, a kernel processor, called LISCP (Limited Instruction Set Computer for Prolog), was designed and evaluated [11,12]. In the second year, a parallel execution model for Prolog and its system architecture were developed [13]. The VLSI design of LISCP with its local cache are also considered [14]. The prototype of MIEP system will be built and attached to a super micro host system. The performance of the system will be evaluated and verified for future improvement.

In the following sections, the system overview of the MIEP will be introduced first. The design of the kernel processor, called LISCP, is described in section 3. The parallel execution model of Prolog on multiple LISCPs is presented in section 4. Then, the design issues of cache systems and their coherence protocols are explained in section 5. In addition, the numerical co-processor for LISCP, called FPC, is presented in section 6. Finally, current status and future development of the MIEP system are given at the end of this paper.

2. System Overview
This MIEP is to design a multiprocessor system for high-performance numerical computations and symbolic manipulations in order to improve the operational capabilities in conventional and artificial intelligent applications. In Figure 2.1, several Processor Groups and their shared Global Memory and I/O devices are connected to the system Global Bus. All these components are communicated with the host computer through the Interface Unit via the system Bus. A distributed operating system (D.O.S.) of the multiprocessor system has been designed by another group in our school. The D.O.S. will be located at the main processor of each processor group. When the host computer sends a message through the Interface Unit, the job partition part of the D.O.S. determines which of the processor group should receive and execute the command.

As shown in Figure 2.1, the MIEP is a subsystem dedicated for Prolog program execution. The MIEP consists of a PIER Controller, a Prolog Memory, a Local Bus, and one or more PIERs (Prolog Inference Engine on RISC) [15]. The PIER Controller performs some functions of the D.O.S. mentioned above. In addition, acting as the central controller of the MIEP subsystem, the PIER controller may partition tasks for the parallel processing of Prolog program on PIERs. The Prolog Memory is used to store the related information for Prolog execution. It is connected to the Local Bus of MIEP for data accesses of the Prolog Controller and PIERs, and is connected to the Global Bus of the system for the convenience of global data exchanges. The design of LISCP is mainly concerned with the sequential execution of Prolog. The MIEP subsystem is planned mainly for parallel processing capabilities of Prolog language. Thus, the investigation of the parallelism of Prolog is an important issue of the project, and is described in Section 4.

A PIER consists of a LISCP, a co-processor FPC, and a cache which is used as the local memory of the LISCP. LISCP is a special-purpose processor designed for the execution of Prolog
### 3. Processor Architecture

The LISCP processor design evolved from traditional RISC type processor design [9] to support Prolog programming. LISCP has a streamlined instruction set and a large register file to speed up procedure calls. In the followings, the instruction set, the micro-architecture, the pipeline, and the Splitable Register Window are described.

#### 3.1. Instruction Set Architecture of LISCP

The LISCP is a tagged, register-oriented RISC-type processor. Each 32-bit data word contains 5-bit tag field (see Figure 3.1). The 5-tag bits define (1) a cdr-bit (C) to support a cdr-coded representation of compound data of list or structure, (2) a garbage collection bit (G) to be used by garbage collection mechanisms, (3) 3-bit of type field to distinguish 8 different data types (e.g. bounded and unbounded variables, integers, atoms, lists, and structures, etc).

```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>C</th>
<th>Data fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 3.1 LISCP Data format.

"TRANS-tag" is used to manipulate tag field of a data word. Both "BRANCH-on-tag" and CMP-on-branch instructions test the data tag and then branch to new location. Similar to Warren’s SWITCH-on-term, these two instructions implement “type indexing” operations for the Prolog executions.

The CALL instruction changes the PC with a designated address field, IMM20, and then creates a new register window for a new procedure. The LISCP instruction set contains no "return" instruction for reduced instruction purposes. However, the "TRANS" instruction can be used to move current window pointer to last window pointer, and "TRANS_lpc" changes PC to the last PC. Thus, the "return" instruction can be easily replaced by TRANS and TRANS_lpc pair.

#### 3.2 The LISCP microarchitecture

The microarchitecture of LISCP (see Figure 3.3) is designed to enhance Prolog program execution. The major components in the LISCP include:

1. OP Code Controller: The OP Code controller is implemented by a PLA with 7 input lines, 25 output lines. Five of the 7 inputs are used to encode 30 LISCP instructions, and the other two inputs are used to differentiate floating point instructions from LISCP instructions.

2. Register file: The register file is designed to support a revolutionary overlapping window scheme called Splitable Register Window, which handles the context switching
between procedure calls for Prolog executions. Its operational principles are discussed in section 3.4.

3. Tag manipulator: The tag manipulator compares the input data types concurrently with ALU operations. When the input data types are detected to be incompatible, the tag manipulator interrupts and disables the destination register (DST) such that the results from ALU are ignored.

4. IMM: IMM register is used to latch and to sign-extend immediate constants in an instruction. It also transfers the tag bits in the instruction to the Tag manipulator.

5. DST: DST latch temporary holds the data from the ALU. Data in DST can be stored into the register file, or can be directly forward to ALU for next instruction.

6. PCs: There are three program counters (next PC, PC, and last PC) which hold the addresses of the instructions in the pipeline.

3.3 The LISCP pipeline

The pipeline organization of LISCP is presented here. It forms the basis of the micro-architecture of the LISCP implementations. The design of pipeline CPU normally has to carefully consider execution and programming environments, such as memory access rate or various branch operation frequency, so that the designed CPU can achieve high execution rate for various computing needs. On the considering of the Prolog programming environments, we find that (1) memory access rate is more than 30% of total fetched instructions, and (2) conditional jump operations occur very frequently. Thus, a pipeline CPU which performs no delay on data memory access and requires single instruction cycle for "compare then jump" operation is very desirable and necessary for high performance Prolog systems.

The LISCP CPU uses a three stage pipeline and attempts to complete an instruction in every cycle (see Figure 3.4). The LISCP pipeline stage allows memory referencing instructions to make cache accesses without stalling the pipeline stage and also allows register to register instruction to finish register write within an instruction cycle (3 pipeline stages).

3.4 The LISCP register file and the Splitable Register Window

The design goal of the LISCP register file is to enhance the register to register operations in order to reduce the frequency of memory accesses. In the LISCP, we have designed a modified Overlapping Register Window [9] scheme, called Splitable Register Window to simplify and to speedup the context switching between two consecutive procedures in a Prolog program.

The data structure design in the LISCP is based on an abstract model for Prolog execution originally described by Warren [16] and modified by Dobry et. al. [33]. Under this model, the address space of the LISCP is divided into code space for instruction fetch and data space for data items and state information. The data space is further divided into three major areas: the Heap and the Trail stacks in the memory, and environment and choice points in register file. The Heap contains all of the compound data items (List and Structural data). The Trail stack contains pointers to the variables in the register file and the Heap stack which have been bound and must be unbound upon backtracking. In the Warren's model, there is a third control stack, which contains environment parameters and choice points. In the LISCP design, we have allocated the environment parameters and choice points on the register file. Our register usage is shown in Table 3.1.

From Table 3.1, one can see that the LISCP register
allocation scheme is similar to the traditional RISC Overlapping Register Window scheme. Each Register Window is partitioned into input, local and output register area. Each window keeps the environment parameters and permanent parameters in local and overlaps argument registers with proceeding and following windows. The global register keeps the state registers (Heap pointer, Continuation pointer, Heap backtracking pointer, etc.) which are needed to each process in the execution of a Prolog program. The current environment and permanent parameters are saved in the local registers when a new process is to be created. Thus, when a process is returned from a finished process, its environment and permanent parameters are automatically available in the local register area. Therefore, the contents of choice points and other pointer values can be obtained from internal processor registers rather than a stack frame in memory. Thus, we can reduce the processor-memory bandwidth.

Backtracking is one of the major features in Prolog for efficient AI programming. When a goal matching process fails, the execution flow may jump to the most recently available choice point and continue to try another path of invocation. We have found that using traditional RISC type Overlapping Register Window Scheme to handle backtracking problem is very difficult.

Table 3.1 Register allocation in LISCP register window.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>Reg. Use</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>0</td>
<td>Hardwired 0, GR0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>Temporary storage, TEMP</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Structure Base Pointer, SBP</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Continuation Pointer, CP</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Local Stack Pointer, LSP</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Trial pointer, TRP</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Heap Backtracking Pointer, HBP</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Heap Pointer, HP</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Choice Backtracking Pointer, CBP</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>Environment Pointer, EP</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>Current Window Pointer, CWP</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>Top window Pointer, TWP</td>
</tr>
</tbody>
</table>

| Inputs     | 12..15  | Input Arguments |
| Local      | 16..23  | Environment Parameter |
| Outputs    | 24..27  | Permanent Parameter |
|           | 28..33  | Output Arguments |

Let us illustrate this problem with a hypothetical Prolog program:

```
a :- b, e.
b :- f, g, h.
c :- j.
g1 :- g, a.
g2 :- g, b.
```

The subgoal g in clause "b", has two choices, namely g1 and g2. Figure 3.5 shows the execution flow of the example program. For convenience, we use "Call" & "Rtn" pair to represent the start and the finish of a process, as shown in Figure 3.5. In the first run, the executed processes includes g1, g2, b, e and j. Since the process j may fail, the execution flow backtracks to g and select second path, starting with "Call g2," and Call g2, ..., etc., (shown in the right column in Figure 3.5).

Figure 3.5 Process creation and deletion of the example program.

Figure 3.6 depicts the creation and deletion of process windows for the example program by using the Overlapping Register Window. Each box in Figure 3.6, represents a process window which contains the execution environment (input parameters, register and stack pointers etc.) of a subgoal as indicated. As we can see in Figure 3.6 IV when the process j fails, the execution flow can not backtrack to g, since its parent process window g does not exist in register file. We can see that, in Figure 3.6 II, the process window g has been deleted, when the execution flow returns to process b.

Figure 3.6 Overlapping register window for Prolog execution.
Therefore, we propose the Splitable Register Window scheme to handle the problems stated previously. Similar to the Overlapping Register Window scheme, the Splitable Register Window contains a large size of register file (138 registers in our design). In addition, the Splitable Register Window scheme requires (1) a choice point marker (CP), and two register pointers, CWP (Current Window pointer), and TWP (Top Window Pointer). A CP marker is used to indicate a process involving with choice pointers (process g in the example). CWP points to the current active window and TWP points to the next available window. In general, CWP and TWP are separated by one register window. As shown in Figure 3.7, the CWP and TWP at the left-side of each diagram shows the creation of each of new windows (pointers move down) and the CWP and TWP at the right side shows the termination of processes (pointers move up). In Figure 3.7 II, process window g contains a marker (CP) which indicates g has two choices $g_1$ and $g_2$. Figure 3.7 III shows that even though the current process has returned from process g to process b, the TWP still points to the window beneath window g. Thus, the process environment of g can be preserved for a possible backtracking to process g due to a failed process later on. Therefore, when a new process window is to be created, the window can be allocated beneath the marked (cp) process window or the TWP (Figure 3.7 IV, V and VI). Hence, as shown in Figure 3.7 VI, if the process j fails, the execution flow can jump to process g and select the second choice subgoal $g_2$ to continue. Since process g has only two choices, at this moment, no more choice available for g. Therefore the marker "cp" is deleted from the process window g (Figure 3.7 VII). Inconsequence, when processes g & b has been finished, the process windows of g and b may be deleted (Figure 3.7 VIII, IX).

![Figure 3.7 Splitable Register Window scheme for Prolog execution.](image)

### 3.5 Register Window Overflow Scheme

This section describes the Register Window Overflow problem and proposes a solution to these problems. From our Splitable Register Window simulation results, we found that when one or more choice points exist in the register file, the register window may overflow very often (more process windows to create than the physical available register window). A simple and quick solution to this problem is storing the oldest process window (most recently unused scheme) into memory, when overflow occurs. However, this simple scheme does not work efficiently for the Splitable Register Window.

The inefficiency comes from (1) the preserved register windows for choice points, and (2) the backtracking procedures. Both cases cause window creation and deletion more than one window (see Figure 3.7 IV and VI) each time, thus more process windows may have to be stored to or loaded from memory at a time when overflow occurs. Our solution to the Register Window overflow problems, includes allocating a 1K memory, called Window Buffer Matrix, in cache and adding a 3-bit tag to each register window. As shown in Figure 3.8, the register file contain 8 register windows in total. We also partition the 1K memory into 8x8 blocks, 16 memory words per block. Thus a total of 64 Register Windows can be saved in this buffer area.

The basic concept in our overflow scheme is "storing the overflowed register window to its corresponding block of the Window Buffer Matrix". For example, register window 0 (RWO) is mapped to the first row of window buffer matrix. The tags in each window are used to specify the particular column in a row of the Window Buffer Matrix. Initially, the tag bit are reset to zero, that means we select the first column of the Window Buffer Matrix as the initial overflow buffer area. When all the eight register windows are used up, then the overflow may occur. Hence, the allocation of the next register window will be wrapped over to window 0. Therefore, the original contents of window 0 are saved in block (0,0) of the Window Buffer Matrix before the new window can be created. When more register windows are to be created, the overflow buffering process continues.

In general, our window buffering scheme works just the same as the previous stated simple scheme. However, when there is a choice point in the register window, and if a backtracking occurs, our scheme allows quick and direct returning to the choice point window, instead of moving in all the windows between the "failure" process window to the choice point window. Let us illustrate the overflow scheme by another example.

\[
\begin{align*}
\text{f('')} & : = \text{g}_1('') \cdot \text{g}_2('') \\
\text{g}_1('') & : = \text{e}_3('') \cdot \text{e}_4('') \\
\text{g}_2('') & : = \text{f}_5('') \\
\text{g}_3('') & : = \text{g}_6('') \cdot \text{g}_7('') \cdot \text{g}_8('') \\
\text{g}_4('') & : = \text{r}_9('') \cdot \text{r}_10('') \cdot \text{r}_11('') \\
\text{g}_5('') & : = \text{r}_12('') \cdot \text{r}_13('') \cdot \text{r}_14('') \cdot \text{r}_15('') \cdot \text{r}_16('') \cdot \text{r}_17('') \\
\end{align*}
\]
A complete flow diagram for the Splitable Register Window scheme is shown in Figure 3.9. Readers are referred to reference [16] for detail. In Figure 3.8 (a), the current active process is g17, and the first 9 process windows are saved in the Window Buffer Matrix. Suppose process g17 fails and process is going to backtrack to choice point process g5. Figure 3.8 (b) shows only process g5 has to be restored from Window Buffer Matrix and process can be resumed.

![Window Buffer Matrix](image)

Figure 3.8 The Window Buffer Matrix for register window overflow scheme.

![Flowchart](image)

Figure 3.9 Handling flowchart of SORWT

4. Parallel Execution Model

4.1 Prolog Execution on MIEP

The MIEP is a bus-connected multiprocessor system for parallel execution of Prolog programs. Initially, the user edits the Prolog program on the host and then compiles it into LISCP object code. After the system receives the "RUN" command from the user, it loads the 'object code' into the Prolog memory, and then activates the MIEP system to execute the code in parallel.

According to the user's query, the MIEP system creates a query. The query process will fork into several processes, which will be put into a ready process list. An idling LISCP retrieves and executes these sub-query processes from the ready process list. Naturally, some new processes may be created and put into the ready process list. The evaluation processing continues, until the ready list is empty, i.e. the query process is completed. Therefore, the MIEP acknowledges the host that the Prolog job has been completed. The host system then retrieves the solutions from the Prolog memory, and reports them to user.
In order to obtain high performance in a multiprocessor system, it is very important to construct a good execution model. In the followings, a pipelining execution model is proposed and some benchmarks are selected to evaluate our model.

4.2 Parallel Execution Model

Until now, there are several methods have been proposed for the parallel inference processing. They are AND, OR parallelisms [18,19,20,21], search parallelism [22], stream parallelism [23] and pipelining parallelism [24] etc. The AND, OR parallelism encounter the difficulty of binding shared variables and handling 'cut' function. The search parallelism is useful for distributed database system at the cost of adding some communication overhead. The criticism of stream parallelism is that the grain size of an entity for synchronization between processes may be too small to be efficient.

We have found that the pipelining parallelism is very suitable for the parallel execution of Prolog on the MIEP. The basic principle is to presearch for alternative solutions of a subgoal in advance. For instance, in the evaluation of f(x)&g(x), when the first solution of f(x) is passed to g(x), f(x) will presearch for alternative solutions. If f(x) fails, it will retrieve a presearched solution from f(x), and evaluate g(x) again. Thus, the number of processes will not be too large, and the execution control mechanism can be easily implemented. Since the sequence of solutions obtained is not changed, the cut operator can be introduced to control the backtracking. In addition, the presearching of pipelining parallelism can support nondeterministic Prolog processing efficiently.

In our model, the parallel processing starts partitioning a Prolog program into several subprograms, or processes, according to the following steps: (i) Constructing an AND/OR tree for the Prolog program. (ii) On this AND/OR tree, the depth-first search technique is used first. If a current searched node is an "OR" node, then this node with his father node belong to the same process. Otherwise, the node located to the left of the nodes at the same level in a AND/OR tree is selected to join with its father node to form a process. The rest of nodes at the same level are partitioned to different processes. This partition technique is selected due to its simple mechanism.

When a Prolog program has been partitioned into many processes, then how to allocate and execute those processes by the available LISCPs is another important issue. We propose both static and dynamic allocation and execution techniques for this issue. By using static method, the created processes are scheduled according to the program execution sequence and then allocated to each available processors. This method is simple and straightforward. In order to evaluate the performance of this technique, two benchmark programs, "data base inference" and "map coloring", were tested on our multi-LISCP simulator [13]. Table 4.1 and 4.2 shows the results of this static execution model. We find that the speed up gained from this method is very limited and it saturates very fast (only about 3 processors). The poor results comes from mainly waiting and idle time over each processors due static and prescheduled allocation.

To improve the static method, a "dynamic allocation and execution model" has been proposed. Instead of assigning each process to available processor in advance, each available processor can dynamically search for ready processes to run with [13]. Therefore, a lot of idle and waiting time of each available processor can be eliminated dramatically. Table 4.3 and 4.4 are the simulation results of the same two benchmarks programs as before. From Table 4.1 and 4.2, we find that the execution times are reduced greatly as the number of processor being increased.

It is also shown that the saturation number of processors is about 'ten processors' which is greater than that of static pipelining method.

Based on our simulations and evaluations, we have found that the dynamic allocation and execution technique has high execution performance in our multi-LISCP system. This results encourage us to do further system analysis and design in the future.

5. Cache System for MIEP

5.1 Cache Architecture

A cache memory has been inserted in between the shared global memory and every LISCP register file. The purpose of such a memory hierarchy is two-fold:

(a) inserting the cache memory in between can relieve the information retrieval rate bound by the low-speed device, i.e. the shared global memory [25].
(b) in the MIEP construct, all the Piers and the shared global memory communicate with each other via a common bus. A memory buffer attached to each PIER capable of lookahead and lookbehind buffering is necessary to balance the traffic load on the common bus [26].

Figure 1.1 shows the cache connections with other components in the system.

The clock phase three in stage one and phase one in stage three of the instruction timing are the time slots for accessing instructions and data from the instruction and data caches respectively. If a data access miss takes places, the corresponding cache controller will send a request for the block containing that datum through the common bus, and the requested information will be provided by either the shared global memory or one of the other data caches. In case of an instruction fetch miss, the only possible source of that information is the shared global memory. The accessed cache returns to its corresponding LISCP with a "ready" signal whenever the requested information is available.

A cache is divided into two submemories, the instruction cache and the data cache. This is due to the following observations:

(a) such a scheme reduces the degree of interference to the LISCP caused by the data coherence problem, since the instructions are read only.
(b) a simplified cache control is adequate for the read-only instruction cache.
(c) different replacement algorithms can be applied to the
instruction and data caches to enhance the cache performance, since the instructions and data inhere different locality characteristics. An example is that the looping, a common instruction fetch pattern, is seldom seen in data fetches.

(d) According to [27], the hit ratio of a heterogeneous cache is in general lower than that of two separate instruction and data caches with the equal combined size, and with the same replacement and write policies. The minimal requirements are that the cache capacity is above a certain lower bound, and the partition is competent, say let the two caches have the same capacity.

The set-associative mapping is chosen to be the cache placement algorithm. The associative mapping calls for too much silicon area for the tag storage for a large cache, and the direct mapping is somewhat inflexible and may result in a worst case hit ratio of zero. The set-associative placement is preferred for both of its economy in silicon area and its flexibility in information storage.

5.2 Cache Consistence Protocol

For the private cache construct in a multiprocessor system, a cache is attached to each processor. Data coherence problem arises when a processor writes a cache, since multiple copies may exist in both the shared global memory and any number of the caches.

Two solutions to the data coherence problem have been proposed: the static and dynamic methods [28]. In the static method, data are tagged as cacheable or noncacheable by the compiler, and only the cacheable data are permitted in the caches. The static method was undesirable for the increased compiler complexity and the software dependence on cache design [26], [28]. In the dynamic method, data coherence was maintained during the run time. Examples in this category include the write-through, the centralized and distributed directory methods.

The coherence protocol used in the MIEP is the distributed directory method. Both the write-through and the centralized directory methods increase the traffic load on the local bus significantly, and the system performance was degraded rapidly as the number of processors increases. The added complexity in the control of the distributed directory method is well compensated for by the local bus performance.

5.3 Local Bus for MIEP

A single bus is used to interconnect the PIERs and the shared global memory in the MIEP, for (a) its simple control; (b) an ordinary main memory structure is sufficient in such a scheme; and (c) most importantly, since the PIERs employ both the windowed register files and the private caches to buffer information, flow on the local bus is significantly reduced, and a simple common bus connection is adequate in such an application.

The local bus in the MIEP transmits both the data and control signals for maintaining data coherence among the caches and the shared global memory. Other than being able to transmit data blocks of a fixed length, the bus must be capable of broadcasting the messages for the coherence protocol chosen.

6. Floating-Point Co-processor and Its Interface to LISCP

6.1 Concurrent execution concepts and design

Most of the off-the-shelf floating-point co-processors are designed with Master-Slave type. The slave co-processor functions as a hardware floating-point subroutine to be called by a master CPU. The major weakness in the Master-Slave system is that the CPU has to be in the wait state when a floating-point operation is processing. To overcome this weakness, we propose a co-processor architecture that can concurrently process floating-point computations while the master CPU can continue its normal operations.

The major features of the proposed floating-point co-processor include:
(a) Co-processor has the same pipeline architecture as the CPU (LISCP), for better execution coherence.
(b) Co-processor can directly access cache (local memory) directly to reduce unnecessary data passing between LISCP and FPC.
(c) An intelligent compiler for floating point computation reduces pipeline suspensions between LISCP and co-processor.

(ii) Floating-Point Co-processor Structure

The design objectives of the FPC are:
(a) Follow the IEEE standard (W54).
(b) Speed up the arithmetic operations.
(c) Share as much common circuits between Multiplier and Adder as possible to reduce silicon area.

The block diagram of FPC is shown in Figure 6.1. The fraction part includes Addition/Subtraction unit and Multiplication/Division unit. One of the bottlenecks in floating-
point computation is the exponential operations which has to determine the larger exponent before the fraction part can start. Two subtractors are included in exponential part to speed up the comparison between the two operands. The fraction adder is composed by a 66-bit Manchester carry chain adder (6 sections of carry selecter to speed up carry propagation). We apply "modified Booth algorithm" in the multiplier design and apply "higher-radix division using the estimate of divider and partial remainder algorithm" in the divider design in order to share most of the circuits. The interface I/O unit is used to fetch instruction, access cache, and convert different data format between the internal and external data, for FPC.

6.2 Co-processor Interface

This subsection describes the interface logic among FPC, LISCP and cache unit. The interface diagram is shown in Figure 6.2.

The major interface signals are described below:
(a) Phi (from LISCP to FPC): Phi synchronizes the pipeline stages between LISCP and FPC.
(b) fpc_busy (from FPC to LISCP): The FPC sends this busy signal to indicate that a longer (two or three pipeline stages) floating-point operation is currently executed. LISCP used this signal to suspend any possible floating-point instructions.
(c) fpc_except (from FPC to LISCP): FPC sends this signal to LISCP to inform an error state occurred in the FPC (for example, divided by zero), then LISCP calls an error handling routine to resolve this erroneous situation.

Figure 6.2 Interface between LISCP, FPC and cache

6.3 FPC Instruction Set

The instructions are designed to support interface operations as well as floating-point computations. There are totally 20 instructions for FPC. They includes:
(a) Arithmetic instructions (FADD, FSUB, FMUL, and FDIV etc.).
(b) FLOAD and FSTORE for floating-point data storage.
(c) Compare branch instruction, FCBR.

FLOAD and FSTORE instructions are designed for FPC to directly access cache, such that the data communication between LISCP and FPC could be done through the cache. Addressing calculation is performed in LISCP for better concurrent operations between LISCP and FPC and simplify the control mechanism in FPC.

The FCBR instruction generates control signal to set the branch bit in the FPC_PSW (Status word of FPC).

6.4 Current Status

Until now, the microarchitecture of FPC has been designed. The functional simulation of FPC has also been implemented on VAX-11/780 by using PMS and ISPS languages [29, 30]. The primary simulation results show the LISCP+FPC can evaluate floating-point data 3 to 4 times faster than VAX/11-780 [31]. The VLSI implementation of the FPC is currently in progress.

7. Concluding Remarks

In this paper, we propose a RISC-based multiprocessor system, MIEP, as an research vehicle, to construct an intelligent workstation for AI applications. Until now, we have not been able to run multiprocessing experiments yet. Nevertheless, we have some preliminary results.

ACKNOWLEDGEMENT

The authors will thank to many other students in MIEP project for their help to run the simulation and to evaluate the performance of the LISCP.

REFERENCES


