An Examination Of Designs For The Instruction Pipeline Of The G-machine

Bill Hostmann
B.S.E.E, University of Portland, 1984

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

October 1, 1987

The thesis "An Examination Of Designs For The Instruction Pipeline Of The G-machine" by Bill Hostmann has been examined and approved by the following Examination Committee:

Shreekant S. Thakkar, Thesis Research Advisor
Assistant Adjunct Professor,
Sequent Computer Systems, Inc.

Richard Hamlet
Professor,
Department of Computer Science and Engineering

Dan Hammerstrom
Associate Professor,
Department of Computer Science And Engineering

Richard Kieburtz
Professor and Chairman,
Department of Computer Science and Engineering

Acknowledgments

I wish to thank my Program Committee for being involved and for taking the time to read and comment on this work, especially Dr. Thakkar, the committe chairman, for providing motivation and encouragement. My thanks also goes to my employer, Hewlett-Packard and my manager Frank Hunt, for providing the environment and time for me to complete this work. Thanks are also due to Boris Agapiev for providing instruction traces for the simulator as well as timely and helpful information on the G-machine.

List Of Figures

List Of Tables

ABSTRACT

Functional programming languages have been advanced as a means of increasing programmer productivity, enhancing program clarity and simplifying the task of program verification. The slow execution of functional programs on conventional computer architectures has been their major drawback. Several architectures have been proposed for executing functional programming languages more efficiently. One such architecture, the G-machine, provides architectural support for the evaluation of functional programming languages by graph reduction.

In this study, designs for the instruction processing pipeline of the G-machine are examined and compared via simulation. A microcoded pipeline design, named Design 1, is proposed and it's performance is evaluated using a range of enhancements which are known to reduce delays associated with instruction memory access and branches. While the enhancements increase performance, they also increase the complexity to implement the pipeline. A RISC design, Design 2, is then proposed for the instruction pipeline. Design 1 and Design 2 are compared, via simulation, to determine whether Design 2 can provide the functionality and throughput of Design 1. Reccommendations for the design of the instructon pipeline of the G-machine are then made based on these simulations.

1. INTRODUCTION

The G-machine, a research project at the Oregon Graduate Center (OGC), provides architectural support for the evaluation of functional languages by programmed graph reduction. In a programmed reduction system, computations consist of transformations of an expression graph by application of reduction rules. Control of the computation is derived from a static analysis (compilation) of the original expression form. The result of the compilation is a sequence of instructions similar to that generated for a conventional von Neumann computer. The efficiency with which this instruction stream can be fed to the execution hardware has a significant effect on overall system performance.

The role of an instruction pipeline is to form an efficient communication interface between an instruction stream and the instruction execution hardware. Different instruction pipeline organizations produce different performance results for any given instruction stream. The design of an instruction pipeline should maximize the instruction stream throughput while minimizing implementation complexity. The intent of this thesis is to examine and characterize the relative performance and complexity of several possible design configurations for the instruction pipeline of the G-machine. A set of high level simulation tools provides an environment to explore the design configurations and their associated tradeoffs.

An overview of the function and organization of the G-machine is given in Section 2. A microprogrammed instruction pipeline for the G-machine is presented in Section 3. A design based on an instruction pipeline proposed for the G-machine by Kieburtz [Kie84] is presented in Section 3; this design will be referred to as Design 1.

In Section 4, a simulation environment for examining the performance of Design 1 is described. In Section 5, a series of simulations examines the relative performance of design enhancements when incorporated into the instruction pipeline of Design 1.

By adding the architectural support for performance enhancements, it is possible to achieve better performance; however, as more support is added the instruction pipeline becomes increasingly complex in terms of operation and implementation issues. "Is there a less complex design than that of Design 1 which can achieve similar performance?" is a question that needs to be asked. In answering this question, a second instruction pipeline based on RISC [1] design principles [Pat85] is presented in Section 6 and shall be referred to as Design 2. A simulation environment to compare the relative performance of Design 1 and Design 2 is described in Section 7; the results of the comparisons are given in Section 8.

Section 9 concludes the thesis with a review of the results of the simulations and suggestions are made for a design of the instruction pipeline of the G-machine.

1. Reduced Instruction Set Computer

2. THE G-MACHINE

The G-machine provides architectural support for the evaluation of functional programming languages by programmed graph reduction. The abstract model for the G-machine architecture is described in detail by Johnsson [Joh83]; Kieburtz [Kie84] describes the G-machine implementation under study at the Oregon Graduate Center (OGC). A brief description of graph reduction and the context and components of the G-machine is presented in this section.

2.1 Graph Reduction

Expressions in the G-machine are represented directly in memory as graphs. The evaluation of an expression consists of applying transformations (*reduction* rules) to the graph data structure. Unlike combinator reduction, in which control is inferred by inspecting the expression graph at each step in a computation, control of the evaluation in graph reduction is directed by an instruction stream obtained from static analysis (a compilation) of the original expression form. This method of reduction can be viewed as an active agent (the instruction stream) that transforms (reduces) the passive data (graph) into a normal form (its value). The G-machine is distinct from conventional architectures in that it provides hardware support for graph traversal and evaluation and a graph memory system specifically suited to graph reduction [Ran86].

2.2 Architecture of the G-machine

The G-machine is implemented as a loosely coupled coprocessor attached to a conventional host processor (Fig 2.1). The host processor treats the G-machine as an asynchronous I/O device. The host handles the memory management functions as

well as running the operating system and other utilities (editors, compilers, etc).

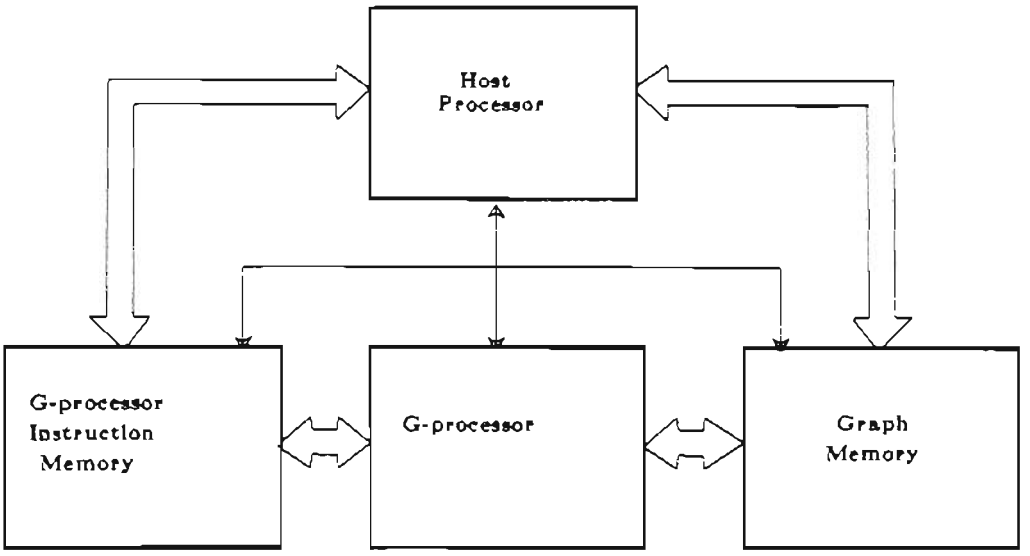

Figure 2.1 The G-machine and its host processor

Graph Memory consists of a dual-ported memory which provides support for efficient allocation of graph nodes and for concurrent garbage collection. Instruction Memory contains the instructions, obtained from compilation, for controlling the evaluation of an expression graph. The internal organization of the G-processor is described next.

2.2.1 The G-processor

The G-processor consists of:

— The instruction pipeline. The instruction pipeline carries out the fetching of instructions and generation of the necessary control signals to the different functional units for execution of the operation indicated by the fetched instruction.

— The functional units. The functional units make up the hardware execution environment. Control signals generated by the instruction pipeline are interpreted by the functional units as hardware operations. The functional units consist of the Pointer-stack (P-stack) and the ALU along with its associated Value-stack (the V-stack).

— The G-bus. An internal data bus which interconnects the operational units of the G-processor and Graph Memory.

— State Registers. Several registers assist in maintaining the state of the G-machine. The A-register holds a graph store address for a READ or WRITE operation. The T-register holds current values of the relevant data tags associated with the top of the P-stack.

Architectural support for graph reduction is provided by the P-stack. The P-stack provides operations for graph traversal associated with the evaluation and manipulation of an expression. Pointers (into Graph Memory) for the current expression-evaluation environment are held on the stack. Typical push and pop stack operations are possible as well as operations to copy, rotate and move elements within the stack.

The ALU along with its associated V-stack, implements the arithmetic and logical operations for the G-processor. The ALU also provides for shift operations, byte insertion, and constant zero. The top two elements of the V-stack are dual-ported so that the ALU can receive two operands simultaneously. Since the P-Stack and the ALU are independent functional units, an ALU operation can be carried out concurrently with a P-stack operation.

A design for the instruction pipeline is presented in the next section.

3. A DESIGN FOR THE INSTRUCTION PIPELINE OF THE G-MACHINE

In programmed graph reduction systems, the steps of computations are transformations of an expression form by application of reduction rules. Control is derived from the original expression form by static analysis -- compilation of a program, resulting in a stream of instructions as in a conventional von Neumann computer. The instruction pipeline of a computer forms the communication link between the instruction stream and the hardware. The efficiency with which this instruction stream can communicate with the execution hardware has a significant effect on overall system performance.

This section introduces the benefits of instruction pipelining. A design for the instruction pipeline of the G-machine is then presented.

3.1 Pipelining

Any instruction pipeline needs to perform the following tasks:

1) Instruction Fetch
2) Instruction Decode
3) Instruction Execution

In non-pipelined systems, each instruction sequentially follows the previous instruction and all phases of an instruction (fetch, decode, execute) must complete before any phase of the next instruction can start. By contrast, in a pipelined system, each phase proceeds independently, so the fetch phase of the second instruction can begin as soon as the fetch phase of the first instruction completes, without having to wait for the first instruction to finish its decode or execute phases. Figure 3.1 illustrates the performance benefits of a simple form of pipelining.

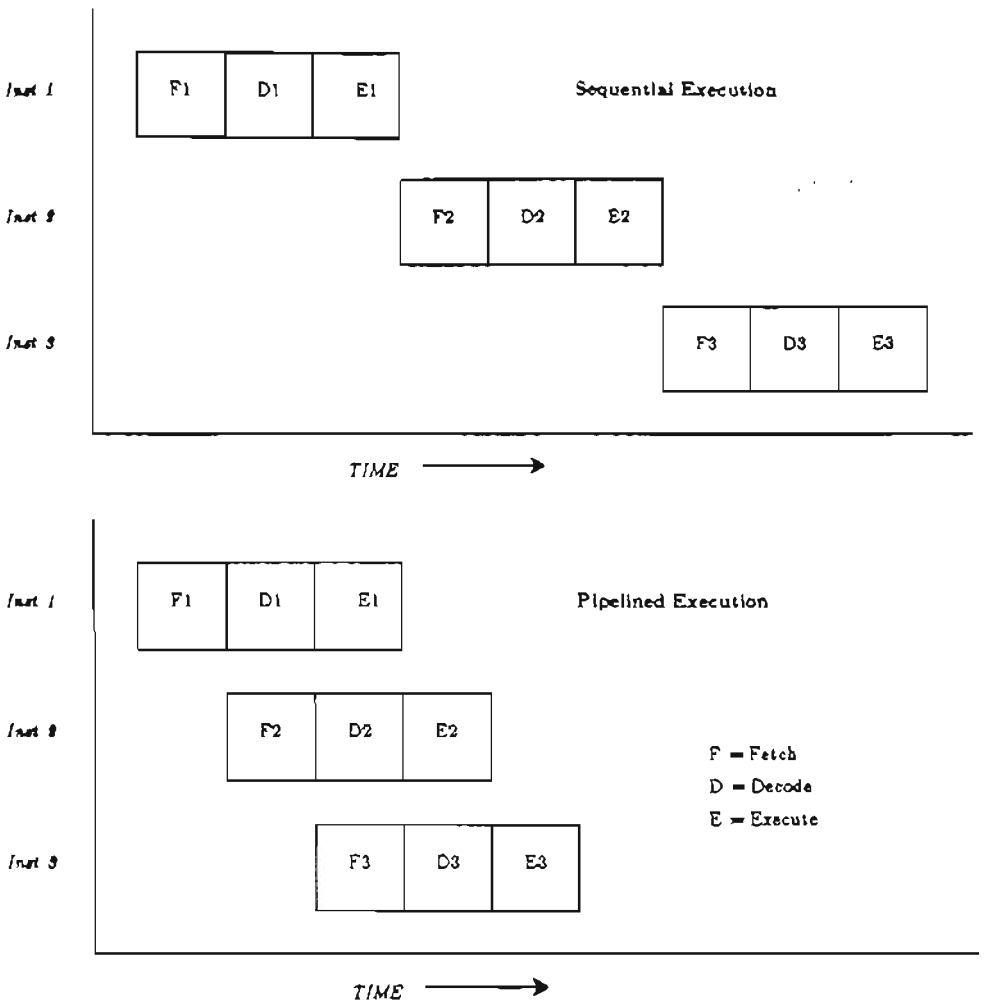

Figure 3.1 Pipelining

Current instruction pipeline designs can be generalized into two basic types: Microprogrammed Instruction Pipelines and Reduced Instruction Set Computer (RISC) Pipelines.

3.1.1 Microprogrammed Instruction Pipelines.

In a microprogrammed instruction pipeline, the fetched instruction associated with the program counter may specify a complex operation (for example a multiply

operation). The rational behind microprogrammed instruction pipelines is based on the observations that a complex operation is completely specified by a sequence of primitive operations. Each primitive operation is associated with a microinstruction which contains information to specify the primitive operation uniquely. Microprograms, built up from microinstructions and stored in a microinstruction control store between the instruction fetch and execute units, are executed by the execution unit and carry out the sequence of primitive operations which make up the function specified by the complex instruction. Microinstructions are fetched from the microinstruction control store much as complex instructions are fetched from instruction memory. Additional logic for a microsequencer is required to direct the fetching and execution of the microprograms.

Several benefits can be ascribed to incorporating a microprogrammed control unit into a design:

1. Regularity. Microprogramming permits an orderly approach to control design.
2. Flexibility. Implementation of alternative instruction sets is possible by simply modifying the microinstructions. Additionally, new instructions can be added to an existing set of microinstructions to increase functionality.
3. Reduced memory access delays. A fetched instruction expands into a sequence of microinstructions fetched from the microinstruction control store thus reducing the need to access instruction memory and incurring the associated overhead.
4. Reduced software complexity. Microprogramming can implement complex instruction sets which contain a high level of semantic content.

Microprogramming does not come without it's disadvantages:

1. Longer Pipeline. A microsequencer and microinstruction store are required and add cost overhead to the design.

2. Performance. A hardwired design could be faster than a microprogrammed design built from the same technology since the former does not have the overhead of fetching and decoding microinstructions.

3. Microprogramming serves to minimize the effects of memory access delays when most instructions expand into a sequence of microinstructions. However, bursts of instructions which do not expand into sequences of microinstructions will suffer the full effect of the instruction memory access delays.

An example of a microprogrammed instruction pipeline can be found in the IBM 360 computer family [Fag64]. A microprogrammed instruction pipeline was used since "it is the only method known by which an extensive instruction set may be economically realized in a small system." [Ste64].

3.1.2 Reduced Instruction Set Computer (RISC) Pipelines

When the instruction-usage patterns of assembly-language programs are examined, the statistics reveal that compilers favor the simpler instructions and make almost no use of the elaborate and complex instructions available in an extensive instruction set [Hen85]. Given this perspective, computer designers have begun creating instruction pipelines based on the small instruction set that dominate computation. The instructions of a RISC are fetched from a high performance memory hierarchy and executed on fast, hardwired execution units. By removing the microinstruction control store and replacing the complex instruction set with a

simplified instruction set, RISC machines have achieved surprising performance [Hen85], [Kat83], [Mah86], [Pat85], [Rad83], [Tay86], [Ung84].

3.2 Instruction Pipeline Design 1

The first design examined for the instruction pipeline of the G-machine is based on a design proposed by Kieburtz [Kie84],[Kie85]. This design will hereafter be referred to as Design 1. Design 1 follows along the lines of a traditional microprogrammed instruction pipeline. The microinstruction control store contains microprograms for executing the G-machine's complex operations such as EVAL (which evaluates a graph).

Design 1 (Figure 3.2) consists of the following pipeline units: the Instruction Fetch Unit (IFU), the Literals Queue (LQ), the Instruction Translation Unit (ITU), the Microinstruction Queue (MIQ) and the Processor Control Unit (PCU).

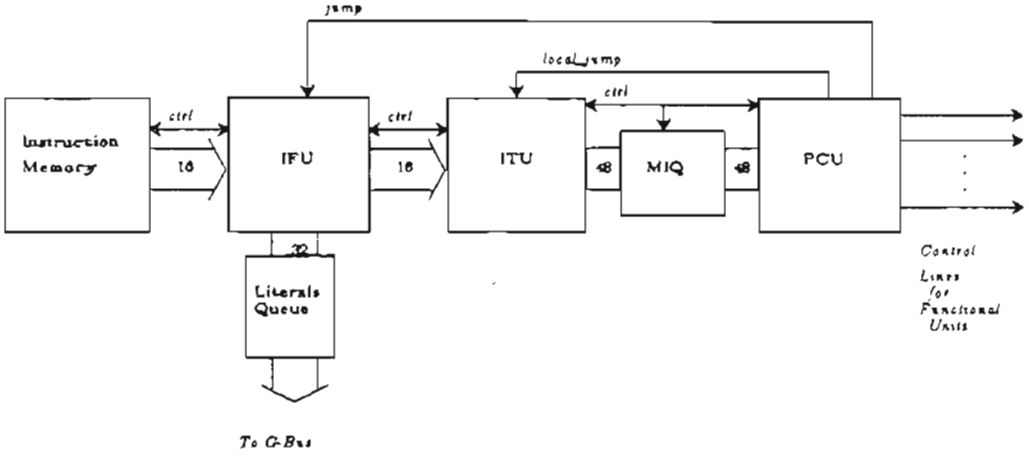

Figure 3.2. IP of Design 1

3.2.1 The Instruction Fetch Unit (IFU)

The Instruction Fetch Unit is the first stage of the instruction pipeline. The IFU directs the fetching of instructions and maintains the Program Counter (PC). Instruction formats consist of: 2 bytes for simple instructions, 4 bytes for jump, case-switch and branch instructions, and 6 bytes for long literals. An instruction word returned from instruction memory is partially decoded by the IFU to determine if additional fetches are required to assemble the instruction. If additional bytes are required, they are fetched; the instruction is assembled and, depending on its type, is sent to the ITU or if its a literal, to the LQ for subsequent access by the functional units.

Two types of branches are possible: conditional and unconditional. In the case of an unconditional branch, immediately after decoding and assembling the branch instruction the IFU modifies the PC according to the instruction and begins fetching from this branch target. For a conditional branch instruction, the IFU assembles the conditional instruction and sends it on for evaluation by the PCU. The IFU then continues to fetch instructions along the branch-not-taken path. Only one level of pending conditional instruction is supported. If the IFU encounters a subsequent conditional instruction (while awaiting the evaluation of a previously queued conditional instruction), it suspends its fetching operation and blocks awaiting the evaluation of the previous conditional (i.e., whether the branch is taken or not).

If the PCU signals (via the *jmp* control signal) that a pending conditional branch is taken, instructions in the pipeline pending execution must be flushed and instructions from the *jump to* address are fetched. If the PCU signals that the pending conditional branch is not taken, there is no change in the instruction fetch

stream.

Branches are implemented such that an instruction immediately following the branch can be executed before the branch takes effect (a *delayed branch*[Pat85]). In the case of a conditional branch instruction, the compiler may not be able to schedule an instruction in this delay slot that can be executed in both the taken-branch and not-taken-branch cases. For these cases, a NOP instruction must be inserted.

3.2.2 The Instruction Translation Unit (ITU)

Once the IFU has assembled an instruction, the instruction is sent to the ITU. The ITU contains a control store of microinstructions. Each microinstruction is 48 bits wide and consists of a horizontal set of explicit control signals for the functional units. The ITU identifies an instruction as either a *simple* instruction or a *complex* instruction. Simple instructions, such as the stack manipulation operations, translate directly into single microinstructions. Complex instructions, such as EVAL (the instruction to evaluate a graph), translate and expand into sequences of microinstructions requiring the use of several functional units. The ITU serves to reduce the instruction memory bandwidth requirements since complex instructions which expand into a sequence of microinstructions, minimize the need to access instruction memory.

Within a microinstruction sequence associated with a complex instruction there may be conditional branches. These *local* branches are handled by the microsequencer of the ITU as directed by the PCU. Once a local branch (local to ITU/PCU) has been resolved, the ITU accesses its microinstruction control store

with the new "jump to" address and continues to direct the sequencing of microinstructions from this new control store address.

3.2.3 The Microinstruction Queue (MIQ)

Ideally, the simple microinstructions from the ITU should be executed one per cycle, but this may not be possible in all cases. For example, the memory-bound instructions such as a READ take more than one cycle to complete. If the PCU receives another READ operation immediately following a READ operation, the PCU must await completion of the first READ before beginning the second. The MIQ serves to buffer the microinstructions between the ITU and PCU when the PCU requires more than one cycle to complete a microinstruction operation. The MIQ can shift-in and shift-out a value every clock cycle. The MIQ can also be flushed and a new value shifted to the head of the queue in one cycle, hence reducing the latency associated with a branch instruction.

3.2.4 The Processor Control Unit (PCU)

The PCU receives microinstructions from the ITU via the MIQ. Since the microinstructions need little decoding or distribution logic, the main task of the PCU is to dispatch these microinstructions in the form of control signals to the respective functional units.

Since the operations carried out by the functional units can be overlaped, the PCU need not be completely synchronous. A semaphore associated with a functional unit is set when a microinstruction, which requires that functional unit, is dispatched by the PCU. When the microinstruction completes, the functional unit clears the

semaphore. Each microinstruction carries a set of resource bits that indicate which functional units are required for its execution. Instruction dispatch is controlled by using the resource bits from the microinstruction as a mask over the semaphores of the corresponding functional units. When the conjunction of the masked semaphores is zero, the instruction can be dispatched. Using an asynchronous PCU accommodates alternative hardware implementation strategies (faster graph memory, etc.) and allows overlap in the dispatching of control signals to the functional units. This technique referred to as *Scoreboarding* of functional units has been used successfully in other high performance computers (CDC 6600 [Tho64]). Scoreboarding allows the scheduling of available functional units with subsequent instructions if an instruction has an associated delay due, for example, to a READ from Graph Memory (which may take up to 3 cycles to complete) or an ALU operation (which may take up to 3 cycles to complete). The units scoreboarded are the P-stack, G-memory, the ALU, and the literals queue.

In the next section, a simulation environment for examining the performance of Design 1 is described.

4. A SIMULATION ENVIRONMENT FOR EXAMINING DESIGN 1

In this section, the simulation environment and the simulation assumptions for examining the performance of Design 1 are described.

4.1 Performance Measures

The performance metrics used in this study to examine pipeline performance are throughput and latency. *Throughput* is the rate at which items are processed to completion. In these simulations *throughput* is measured as:

$$\frac{\textit{Number of instructions executed}}{\textit{Simulated time}}$$

Latency is the time for one item to traverse the entire pipeline when it is otherwise empty. The *average latency* of the pipeline, due to a branch instruction, is measured as:

$$\frac{\textit{Total time pipeline is idle due to a branch}}{\textit{Total number of branches taken}}$$

Idle time for the instruction pipeline is used in the average latency measure instead of idle time for the execution unit, because instructions with literals may follow a branch and these are never seen by the execution unit.

4.2 The Simulation Environment

A simulator which models the instruction pipeline of Design 1 has been developed to examine performance under a range of assumptions and configurations. The simulation models of the pipeline units (i.e. the IFU, the ITU, etc.) are implemented as communicating concurrent tasks executing independently of each

other. The models are written in C and use the **Interwork** [Int86] Concurrent Programming Toolkit to manage task scheduling and communication. The models provide a "black-box" description of a pipeline unit; the input/output structures of each unit and the associated control lines are global variables to which other units (tasks) have access. The body of the C program makes up the functional description of what the unit does with its input/output structures and control lines.

The notion of concurrent tasks provides the necessary timing information relative to overall pipeline performance since a delay in one unit is reflected in the throughput of the overall pipeline simulation. By using a tool such as **Interwork**, it is possible to keep the simulations at a high level of abstraction. The alternative was to use a low level register transfer level language such as N.2.[1] The higher level of abstraction allows for smaller simulation programs and a fast turnaround time for simulation runs. A modification to the simulation can be made at the functional level by simply changing a few lines of code within the program description of a unit; as a result, the effect of the modification can be quickly determined. While the overall simulation is at a high level, the timing and signals between the various units is accurately modeled to obtain performance information.

Since the units execute as cooperating concurrent tasks, it becomes difficult to verify that the simulation is accurately modeling the design, since the available tools provide little support for debugging concurrent tasks. Thus, it became necessary to develop a tool to examine the state of one task relative to other tasks. This tool

1. N.2 is a trademark of Endot, Inc

took the form of a windowing system developed using the **curses** library [Arn]. The windowing system divides the terminal screen into equal sized sub-windows with one sub-window per pipeline unit. Each unit has the ability to communicate with its particular sub-window through special window function calls developed specifically for these simulations (the calls are placed in the C program which describes the unit). When a task makes a call to a window function, the window manger (through which all calls are managed) concatenates the simulation time onto the task message. The message is then displayed in the appropriate sub-window. This windowing system not only provides a tool for debugging and verifying the performance of the simulator, but also provides insights into bottlenecks and resource contention issues. Figure 4.1 shows an example of the windowed display with a few messages from the different pipeline tasks.

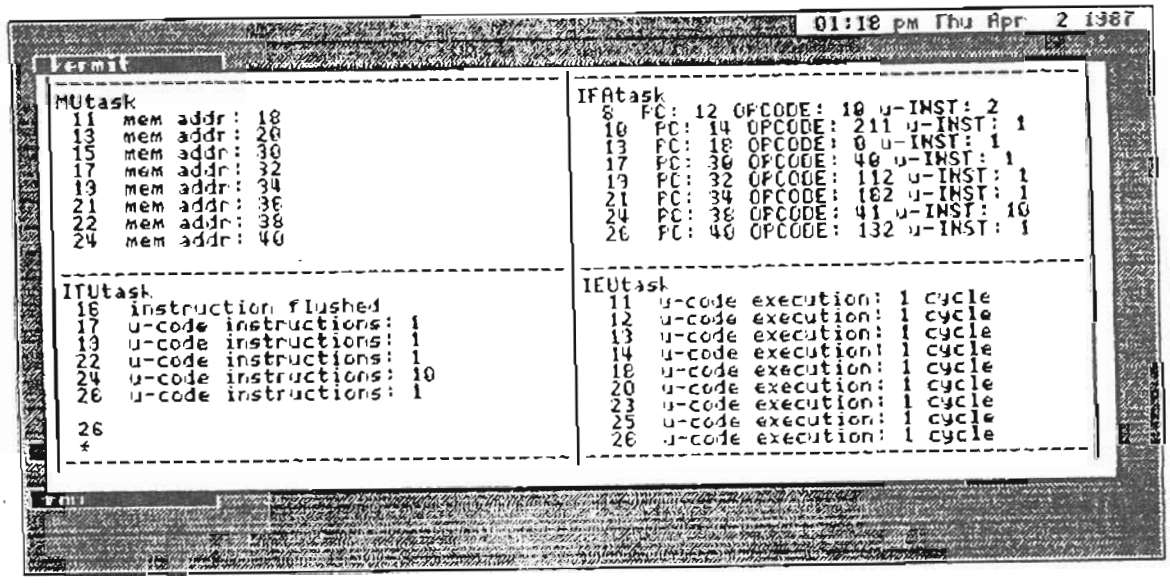

Figure 4.1 Windows for Debugging and Verification of Simulation.

The name of the task is in the upper left hand corner of the window (**MUtask** = memory task, **IFAtask** = Instruction Fetch Unit task, **TUtask** = Instruction Translation Unit task, **IEUtask** = Instruction Execution Task (PCU)). The

numbers along the left edge of the sub-windows are the simulation times associated with messages from the tasks. Thus, it is possible to determine what state each of the pipeline tasks is in at simulation time 13. The asterisk in the lower left hand corner indicates the number of instructions queued on the MIQ at the time indicated above the asterisk. For every instruction added to the MIQ, an asterisk is added to the display, conversely for every instruction removed from the front of the MIQ by the PCU, an asterisk is removed from the display. The asterisk display provides an insight into the dynamic behavior of the MIQ activity. The windowing tool has proven to be very valuable for debugging and verifying the simulation.

4.3 Global Simulation Assumptions

When the simulations to examine the performance of Design 1 were conducted, limited data was available to characterize the execution of large programs on the G-machine. In order to model the execution environment of the instruction pipelines the following assumptions have been made. These assumptions apply to all the configurations simulated. Assumptions specific to only one configuration are included in the section describing that configuration.

1. A synthetic workload is used to drive the Design 1 simulations. The workload, which is the arrival of the different instruction types fetched from Instruction Memory, is a sequence based on a mapping from a random vector (a sequence of random numbers) to a distribution that represents an instruction type mix. The references to the different instruction types are distributed as follows:

Instruction Type Distribution	
short	40%
single operand	10%
short literal	10%
long literal	10%
unconditional jump	10%
conditional jump	10%
m-way case-switch	10%

Table 4.1 Instruction Type Distribution.

The instruction reference pattern is intended to give the simulation a reasonable approximation of a typical LML program execution environment and provide a basis for comparing the performance of the different hardware configurations. This distribution is based on the observed distribution of instruction types by Sarangi [Sar84]. Other instruction workloads which contained a higher percentage of short instructions and case-switch instructions were also used. (The relative performance of the different configurations remained constant across different instruction mixes used.)

2. An assumption is made concerning the instruction execution rate of the PCU. Not every instruction dispatched by the PCU to a functional unit executes to completion in 1 cycle. Certain ALU and READ operations require up to 3 cycles to complete. Simulations on the G-machine for small test programs [Sar84] show that 15-25% of the instructions dispatched by the PCU require a full 3 cycles to complete. These simulations assume that 20% of the instructions execute in 3 cycles while the balance of the instructions execute in 1 cycle.

3. The expansion of a fetched instruction into a sequence of microinstructions, for subsequent dispatch to the functional units by the PCU, occurs in the ITU.

Since information concerning the number of microinstructions executed per fetched instruction was not available when these simulations were conducted, some assumptions needed to be made. For these simulations, the number of microinstructions generated by the ITU is based on a uniform probability distribution of between 1 and 25 instructions. This approximation was arrived at by a static count of the average number of microinstructions generated by a fetched instruction as referenced in The G-machine Programmers Guide [Kie84].

4. The LML compiler uses delayed branching for scheduling instructions associated with a conditional branch. A delayed branch instruction follows a conditional branch instruction such that an instruction from before the conditional branch instruction is moved to fill one of the delay slots following the conditional branch instruction.

5. The length of the Microinstruction Queue is fixed at length 8. Queues of different length have been simulated. In the simulations in which the length of the queue was varied, it was observed that little if any performance gain was realized from a queue of length greater than just 1 or 2 words. The simulations showed that the length of the queue is not as important as simply having a mechanism (e.g even a one word wide shift register) to act as a buffer between units.

6. The delay associated with an instruction memory access is assumed to be 5 cycles. For the simulations which incorporate an instruction cache, a 1 cycle penalty is assumed for an instruction cache access [Hua84].

The next section describes the particular configurations simulated and the simulation results.

5. THE PERFORMANCE OF DESIGN 1 AND ITS VARIATIONS

The performance of Design 1 is examined using the simulation environment described in the previous section. Selected design enhancements are incorporated into the design and the relative performances are compared.

The primary reason for pipelined instruction execution is increased performance over a non-pipelined design. An N-fold speedup promised by an N-stage pipeline is seldom realized due to the irregular flow of instructions through the pipeline. To maintain a steady flow of instructions in a pipeline two problems have to be resolved. First, instruction memory access time may be so long that a request by the first stage of the pipeline (the instruction fetch unit) may not be satisfied soon enough by the instruction memory hierarchy to maintain the flow. Second, a change in expected instruction sequence, caused by a branch instruction, invalidates the contents of the pipeline; the pipeline must be flushed and refilled with instructions from the branch target address. The problem of delays due to branches is related to the problem of memory access time since the penalty for a branch instruction will depend on the time to fetch the target instruction from memory. This section examines a range of techniques for Design 1 which reduce these delays and consequently increase performance.

5.1 Reducing Delays Associated With Instruction Memory Access

The delays associated with instruction memory access are due to the difference between the rate that instructions can be delivered to the instruction fetch unit from the instruction memory, and the rate that the delivered instructions are executed. The following techniques have proven to be effective in reducing instruction memory

access delays.

5.1.1 Instruction Caches

Instruction caches make up one part of the overall concept of *storage hierarchies*, where faster, but less dense memories are placed closer to the fetch stage of the instruction pipeline, and the slower, but more dense (and larger) memories back them up. An instruction cache is a small, fast buffer with an access cycle time that is intermediate between that of the execution cycle time and the access cycle time of the low speed, inexpensive dynamic RAMS that are used to build the main instruction store.

Most programs exhibit *locality of reference*, which means that instruction references over a short period tend to access either a previously requested instruction or an instruction nearby. If the caching hardware buffers the most recently accessed instructions in the instruction cache then the property of locality of reference implies that subsequent instruction requests will be available in the cache, i.e., a cache *hit*. A cache hit reduces the delay associated with an instruction memory access since the effective delay will be the access cycle time of the cache rather than the longer access cycle time of the main instruction store. Actual cache hit rates depend on the design and organization of the instruction cache as well as the actual instruction mix. Hit rates greater than 90% have been shown to be possible using instruction caches as small as 4096 words [Smi82] [Smi83].

The performance of Design 1 has been simulated with an instruction cache added between the Instruction Store and the Instruction Fetch Unit. The instruction cache is organized as a byte addressable fully associative cache using a modified

FIFO replacement strategy. Other cache replacement strategies were examined, but the simulations showed little difference when comparing *relative* performance of the pipeline configurations as a function of the cache replacement strategies. An assumption is made for the cache hit and miss costs: in the case of a cache hit, a one cycle cost is incurred; in the case of a cache miss, a 5 cycle cost is incurred [Hua84].

The relative throughput and latency of Design 1 with a 1024 word cache using the synthetic instruction workload is given in Tables 5.1 and 5.2. As can be expected, an instruction cache has a large effect on overall pipeline performance. Cache size has a significant effect on both the throughput and latency. Several cache sizes (124 words, 256 words, 512 words, and 1024 words) were simulated [Tha86]; only the results for the 1024 word cache are given here for comparing the performance of Design 1 with and without an instruction cache.

	Relative Throughput
No instruction cache	1.00
With 1024 word cache	1.24

Table 5.1 Throughput Comparison With An Instruction Cache

	Average Latency (Cycles)
No instruction cache	6.5
With 1024 word cache	4.5

Table 5.2 Latency Comparison With An Instruction Cache

The results show that a 1024 word cache yields a 24% increase in throughput and a 30% decrease in latency. Even a simple instruction cache can significantly reduce

memory access times and the instruction memory bandwidth requirements. In addition to reducing the delays associated with instruction memory access, the latency results show that the instruction cache also serves for reducing the delays associated with branches. The instruction cache buffers the most recently accessed instructions and the locality of reference implies that subsequent instruction requests will be available in the cache. For the case of a loop, the instruction cache often contains the instructions associated with the loop hence eliminating the need to access main instruction memory. The low cost of CMOS static RAMS makes an instruction cache an easily justified addition for the performance gains demonstrated here.

J.E. Smith [Smi87] has noted that an instruction cache can provide the functionality associated with additional hardware enhancements. For example, Smith asserts that the need for Branch Target Buffers (BTBs) [Lee84] is eliminated with a sufficiently large cache, since the chances of a cache hit on the branch target are high enough to not warrant BTBs.

These simulations assumed only a very simple cache organization and replacement policy. Improved performance over that observed in these simulations could be possible by tailoring an instruction cache to the specific referencing patterns associated with LML programs.

5.1.2 Implicit Instruction Prefetching

The delay associated with instruction memory access can be reduced by taking advantage of the fact that most instructions are fetched sequentially. By fetching more than 1 instruction from instruction memory at a time it is possible to *implicitly*

fetch an instruction in advance of the instruction being required. These implicitly fetched instructions are held in an instruction buffer until required. On subsequent instruction requests, the instruction buffer is accessed before making a request to the lower levels of the instruction memory hierarchy. For sequential instruction execution, the delays due to memory accesses are reduced since the next instruction will be available in the instruction buffer. Prefetching multiple instructions into an buffer allows a continuous sequence of instructions to be supplied to the instruction pipeline at a rate roughly approximating the instruction execution rate. The Fairchild Clipper [Far86] uses implicit prefetching into an instruction buffer to obtain its 33 ns. cycle time.

To simulate the performance of Design 1 using implicit prefetching, the organization of the instruction cache is modified such that each line in the cache is 4 bytes wide. The instruction data path is expanded to 32 bits wide such that 4 bytes are returned to the IFU per instruction memory request. A 4 byte instruction buffer is added to the IFU to hold the additional fetched instructions. The assumption is made that an instruction can be fetched from the instruction buffer and decoded in one cycle.

The performance of Design 1 using implicit prefetching is shown in Tables 5.3 & 5.4.

	Relative Throughput
No instruction cache	1.00
With 1024 word cache	1.24
With 1024 word cache, and implicit prefetching	1.53

Table 5.3 Throughput Comparison With An Instruction Cache And Implicit Prefetching.

	Average Latency (Cycles)
No instruction cache	6.5
With 1024 word cache	4.5
With 1024 word cache and implicit prefetching	4.5

Table 5.4 Latency Comparison With An Instruction Cache And Implicit Prefetching

Implicit prefetching improves the overall throughput, since for a consecutive sequence of instructions, it is likely that the next instruction will be found in the instruction buffer and a fetch to the lower levels of the instruction memory hierarchy will be unnecessary. No improvement is realized in pipeline latency, since implicit prefetching does nothing to reduce the delays associated with branches.

5.1.3 Explicit Instruction Prefetching

Like implicit prefetching, explicit prefetching takes advantage of the assumption that instruction references will typically be sequential. Explicit prefetching refers to initiating the fetch of successive instructions into an instruction cache during the execution of a previous instruction. Prefetching results in the next instruction being available in the cache *before* it is accessed. The Fairchild Clipper [Far86] uses instruction prefetching to achieve a 96% cache hit rate from its

instruction cache.

In order to do explicit prefetching of instructions, a *Remote Program Counter* (RPC) [Pat83] is required to direct the prefetching operation. The RPC addresses the instruction cache for instruction i while instruction i-1 is being executed. If instruction i is not in the instruction cache, it is requested from the instruction memory and brought into the cache ready for access when instruction i-1 completes execution.

Prefetching does have its drawbacks. A conditional branch instruction i will always initiate a prefetch of word i+1. If the branch is taken, then the prefetch was not needed, and the proper instruction must be fetched. If the prefetch is still in progress when the branch address is known, then the prefetching can increase the time for the branch to be taken. In addition, this unused prefetch represents an additional request to the memory system and thus increases the instruction/memory bandwidth requirements.

In the following simulations, the pipeline of Design 1 is modified to include explicit instruction prefetching. 4 bytes are brought into the instruction cache per prefetch operation. A remote PC is added to the cache controller to direct the fetching of instructions into the cache independent of the IFU. The relative performance of incorporating explicit prefetching into Design 1 is given in Tables 5.5 and 5.6.

	Relative Throughput
No instruction cache	1.00
With 1024 word cache	1.24
With 1024 word cache, and implicit prefetching	1.53
With 1024 word cache, implicit prefetching and explicit prefetching	1.57

Table 5.5 Throughput Comparison With An Instruction Cache , Implicit Prefetching And Explicit Prefetching.

	Average Latency (Cycles)
No instruction cache	6.5
With 1024 word cache	4.5
With 1024 word cache and implicit prefetching	4.5
With 1024 word cache, implicit prefetching and explicit prefetching	4.5

Table 5.6 Latency Comparison With An Instruction Cache ,Implicit Prefetching And Explicit Prefetching.

Explicit prefetching improves throughput in the same way as implicit prefetching *i.e.* for a consecutive sequence of instructions it is likely that the next instruction will be found in the instruction cache (due to prefetching) and a fetch to the lower levels of the instruction memory hierarchy will be unnecessary. No improvement is realized in pipeline latency since explicit prefetching does nothing to reduce the delays associated with branches. While the performance improvements for explicit prefetching are not as dramatic as those for an instruction cache and implicit instruction prefetch, the little extra hardware required (remote PC) would justify its incorporation into a design, especially where the instruction cache hit rate is a

critical performance parameter.

5.2 Reducing Delays Associated With Branches

A fundamental disadvantage of pipelining is the delay incurred due to branches which stall or require flushing of the pipeline. Since branches constitute anywhere from 15-30% of the instructions executed on typical machines [Lee84], these delays can have a significant effect on overall performance. Both hardware solutions and architectural changes have been proposed to overcome these delays [Lee84] [McF86]. This section examines branch target prefetching for reducing branch delays.

For a conditional branch instruction, the instruction fetch unit must fetch either the next sequential instruction or the branch to instruction. If the next sequential instruction is fetched and the conditional branch is taken then a delay occurs while the branch to instruction is fetched and the pipeline is restarted.

5.2.1 Multiple Instruction Streams

A brute force solution to this problem is to replicate the initial stages of the pipeline (i.e., multiple instruction streams) so that both the succeeding instruction and the potential branch target can be fetched during the evaluation of the conditional instruction. This approach gives rise to the following problems:

1. Successive or additional conditional branch instructions may enter the initial stage of the pipeline before the target for the first branch instruction has been resolved.

2. The cost of replicating significant parts of the pipeline can be substantial.

3. The control logic of such a scheme can be rather complex.

Multiple instruction streams have been evaluated for use with Design 1 [Far85]. It was envisioned that the LML compiler could generate up to 4-way case switch instructions [Kie85] (current versions only generate 2-way conditionals). Up to four different instruction streams could be prefetched. Fetches were handled in round-robin sequence for each instruction stream; only one instruction stream was decoded. Simulation results for the multiple instruction streams showed poor performance.

5.2.2 Branch Target Prefetch

As an alternative to multiple instruction streams, it is possible to provide a minimum amount of logic to prefetch just the branch target of a conditional branch instruction. The extra logic computes and directs the fetch of the branch target. If the conditional branch is taken, the branch target can be immediately accessed with a minimum amount of pipeline delay. The IBM 360/91 uses branch target prefetching to prefetch a double word target[And67].

Branch target prefetching has one shortcoming. If the branch target prefetch is still in progress when the branch address is known, then the prefetching of the branch target actually increases the time for the branch to be taken. For example, in a 2-way conditional branch, a branch target prefetch to branch target 1 begins while the conditional branch instruction is being evaluated. After fetching branch target 1, a fetch is initiated for branch target 2. If the conditional instruction is resolved and the branch is taken to branch target 1, then the prefetch of branch target 2 is unnecessary overhead which increases pipeline latency. Branch target prefetching is useless for a short instruction pipeline since a branch will necessarily be resolved before the branch target prefetching is complete.

The pipeline of Design 1 has been simulated with branch target prefetching ,*i.e.*, prefetching into the instruction cache the branch target of a conditional branch instruction during the evaluation of the conditional branch instruction.

Four Branch Target Address Registers (TARs) are added to the IFU of Design 1 to hold the addresses of the potential branch targets, indicated by the conditional branch instruction, as well as some additional logic to direct the fetching of the instructions associated with the TARs into the instruction cache. Upon resolution of the conditional branch by the PCU, the PC takes on the address of the indicated TAR and makes a request for the associated instruction. An instruction cache hit is assured since the instructions associated with the TARS have been fetched into the instruction cache during the branch evaluation.

The performance of Design 1, when branch target prefetching is combined with the techniques for reducing the delays associated with instruction memory access, is given in Tables 5.7 and 5.8.

	Relative Throughput	
	Without Branch Target Prefetch	With Branch Target Prefetch
No instruction cache	1.00	N.A.
With 1024 word cache	1.24	1.26
With 1024 word cache, and implicit prefetching	1.53	1.54
With 1024 word cache, implicit prefetching and explicit prefetching	1.56	1.45

Table 5.7 Throughput Comparison With An Instruction Cache, Implicit Prefetching, Explicit Prefetching and Branch Target Prefetching.

	Average Latency (Cycles)	
	Without Branch Target Prefetch	With Branch Target Prefetch
No instruction cache	6.5	N.A.
With 1024 word cache	4.5	2.5
With 1024 word cache and implicit prefetching	4.5	2.5
With 1024 word cache, implicit prefetching and explicit prefetching	4.5	4.0

Table 5.8 Latency Comparison With An Instruction Cache ,Implicit Prefetching, Explicit Prefetching and Branch Target Prefetching.

The simulations with an instruction cache and implicit prefetching realized a significant latency reduction (44%) with branch target prefetching. However, throughput improves only slightly for the corresponding configurations. It can be observed from the above results that a significant reduction in pipeline latency has only a slight effect on overall throughput for the pipeline of Design 1. This can be attributed to the small percentage of conditional branch instructions which are taken, causing pipeline latency.

An explicit prefetch and a branch target prefetch both require access to instruction memory, one prefetch must block awaiting the completion of the other. The simulations with explicit prefetching show reduced throughput (of 7%) when branch target prefetching is added. Likewise, latency is reduced by only 11% when branch target prefetching is added to the configurations with explicit prefetching as compared to a 44% reduction without explicit prefetching. Hence, the combination of branch target prefetching with explicit prefetching causes instruction memory access

delays associated with memory contention with a resulting reduction in performance.

Additional simulations were conducted to examine the effect of aborting the branch target prefetch upon completion of the conditional branch evaluation [Tha86]. While the results were similar to those obtained for not aborting the branch target prefetch, some increase in pipeline latency was realized and a slight increase in throughput was seen.

It becomes evident from the above results that the additional logic required to do branch target prefetching is not justified since the performance gains are slight even though the latency is significantly reduced. Additionally, branch target prefetching should not be combined with explicit instruction prefetching, since the instruction memory contention actually causes reduced performance.

5.3 Summary Of Results

Design 1 and variations thereof have been simulated to examine which configurations reduce the delays associated with instruction memory access and branches. The simulations show that an instruction cache along with implicit instruction prefetching with an instruction buffer provide the most significant performance improvement. Additional incremental performance gains can be realized by incorporating explicit instruction prefetching. Branch target prefetching decreases the latency of the pipeline but does little to improve the overall pipeline performance. Branch target prefetching, when combined with explicit instruction prefetching, actually degrades pipeline performance due to instruction memory access contention by the two prefetching schemes.

Incorporating design enhancements into the pipeline of Design 1 increases the overall operational and implementation complexity of the pipeline. In the following sections, an alternate, simplified design for the instruction pipeline of the G-machine will be examined.

6. AN ALTERNATE DESIGN FOR THE PIPELINE OF THE G-MACHINE

The previous sections have examined, via simulation, variations on a design for the instruction pipeline of the G-machine. The simulations show that improved performance is possible by adding architectural support for reducing the delays associated with instruction memory access and branches. The cost of the additional support is increased design complexity in terms of design, implementation and operation. The following sections set out to establish if a pipeline design that is less complex than that of Design 1 can provide similar performance and functionality.

6.1 A RISC Instruction Pipeline

Design 2 proposed in this section for the instruction pipeline of the G-machine does not use a microprogrammed control unit. The design is motivated by the recent successes demonstrated by RISC designs as well as the results from the previous sections that show that the delays associated with instruction memory access can be sufficiently reduced to the point that instruction memory bandwidth need not be a performance bottleneck. The decision to use a microprogrammed versus a RISC pipeline depends on whether or not similar performance can be achieved by a RISC pipeline compared to that of a microprogrammed pipeline.

Additional motivation to examine a simplified pipeline comes from the work done on the Dragon processor at the Xerox Palo Alto Research Center (XPARC) [Mon85]. The origins of the Dragon are in the XPARC Dorado processor [Pei83] which uses a highly pipelined microprogrammed instruction pipeline. The Dorado's IFU alone contains six pipeline stages and requires a full printed circuit board to implement. The designers of the IFU admit that the design is too complex and that

it has ended up being implemented as "logical steel wool". The full instruction pipeline of the Dorado is spread across six printed circuit boards! In contrast, the Dragon is implemented as a RISC-like processor and requires only four custom VLSI chips!

In a RISC design, the instruction fetched is the instruction executed; there is no microinstruction store which translates the instruction into a sequence of microinstructions as in Design 1. A RISC instruction pipeline for the G-machine should be the simpler of the two pipeline designs in terms of design execution and implementation, however, instruction memory bandwidth limitations may require that a microinstruction store (as in Design 1) is required. For convenience the RISC design discussed in the following sections will be referred to as Design 2.

The instruction set for the G-machine is well matched for execution on a RISC style instruction pipeline since: a) only *load* and *store* instructions access memory, b) only simple addressing modes are necessary, c) it is possible to design an instruction format which does not cross word boundaries, making it possible to reduce decode/assembly time and, d) the functions currently executed by a sequence of microinstructions can be migrated to compile time implementations; the microinstructions are packaged into subroutines fetched from the memory hierarchy. The notion of migrating what would normally be a sequence of microinstructions to compile time implementations called *millicode* has been successfully carried out on the Hewlett-Packard series of Spectrum Processors [Cou86].

Design 2 consists of a simple 2-unit instruction pipeline. A block diagram of Design 2 along with the control and data lines which connect the different units of

this pipeline design are shown in Figure 6.1.

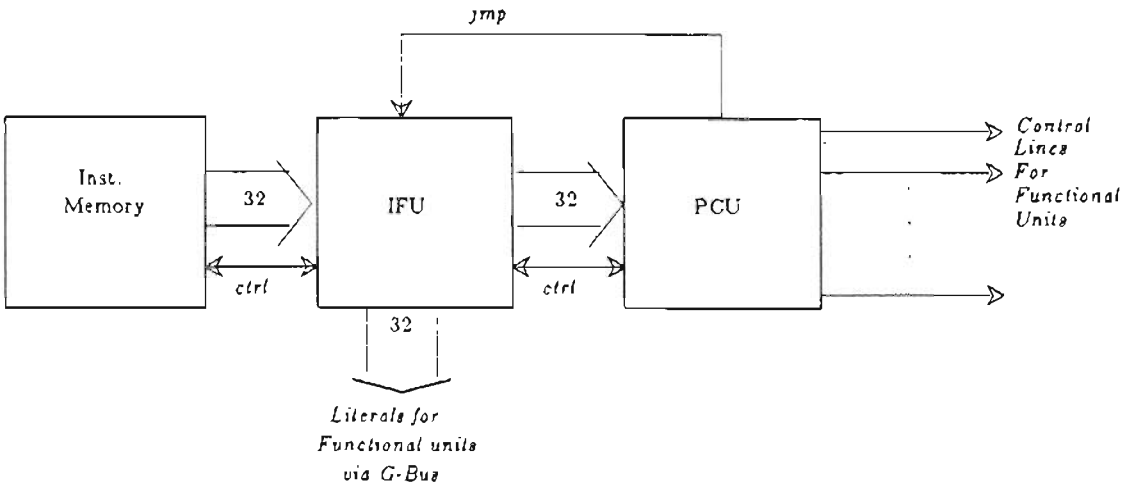

Figure 6.1 Instruction Pipeline of Design 2

The Literals Queue (LQ), the Microinstruction Queue (MIQ) and the Instruction Translation Unit (ITU) of Design 1 are not required in Design 2. Since all the instructions are executed in hardware, the instruction word must be wide enough to simplify the state machine which generates the control signals to all of the functional units. For Design 2, an instruction size of 32 bits (4 bytes) is assumed for all instructions. A wider instruction size could be used, it is only a matter of instruction path width between memory and the Instruction Fetch Unit. The assumed instruction format is simple and does not cross word boundries, in conformity with the definition of a RISC [Pat85].

6.1.1 Functional Description of Pipeline Stages

The Instruction Fetch Unit (IFU): The Instruction Fetch Unit for Design 2 performs the same basic functions as the Instruction Fetch Unit for Design 1. The

primary difference lies in the decode/assembly of the instructions. Design 2 requires that the instructions are all of one format (4 bytes wide) and that no assembly is required. The instruction is decoded in one cycle, then executed by the PCU. There is no need for the Literals Queue in this design, since literals associated with an instruction will be accessed immediately by the functional units. A single 32 bit wide shift register serves as a 1 word wide FIFO buffer between the IFU and PCU.

The Processor Control Unit (PCU): Each clock cycle, the PCU generates a set of control signals directly from the fetched instruction. Like the PCU of Design 1, the different functional units are scoreboarded. Unlike the PCU of Design 2, the additional logic associated with a microsequencer is not required, nor are the additional control signals to the ITU for local jumps (jumps which occur within a microsequence).

A microcoded pipeline achieves its performance by providing a method whereby an instruction, which is fetched from the instruction memory hierarchy, is expanded into an instruction sequence by a translation unit. A simplified instruction pipeline such as Design 2 may not be able to provide the instruction memory bandwidth required by the Processor Control Unit. The simulations described in the following sections compares Design 1 and Design 2 to examine whether Design 2 can provide the performance and functionality equal to that of Design 1.

7. THE SIMULATION ENVIRONMENT FOR COMPARING DESIGNS 1 AND 2

The performance of the instruction pipeline of Design 1 is compared with the performance of the instruction pipeline of Design 2. Like the previous simulations for Design 1, these simulations are conducted using the **Interwork** tool. The simulations are set up to investigate whether the simplified instruction pipeline of Design 2 can provide similar performance to that of the microcoded pipeline of Design 1.

Since the time that the initial set of simulations on Design 1 were conducted, instruction traces of executing LML programs have become available. While the simulations examining the performance of Design 1 were conducted with statistical workloads, the simulations for comparing Design 1 and Design 2 are conducted using these instruction traces. Hence a different simulation environment and set of assumptions is required.

7.1 Simulation Workloads

The simulations of the performance enhancements for Design 1 used a workload of an instruction reference pattern based on a mapping of random numbers to an assumed instruction type distribution. A synthetic workload was used since instruction traces for large LML programs did not yet exist. The simulations for the comparison of Design 1 and Design 2 are based on instruction traces of executed LML programs.

Subsequent to the Design 1 simulations, a macrosimulator for the G-processor was developed [Kie87]. The macrosimulator does not simulate the execution of each microinstruction but it does provide timing information for performance evaluation

of the overall instruction pipeline. It is possible to collect instruction traces of executing LML programs from the macrosimulator. The macrosimulator is based on the instruction pipeline of Design 1, *i.e.*, a microprogrammmed pipeline.

Instruction traces collected from a series of LML programs executed on the macrosimulator are used to drive the following simulations. Six LML source programs have been used to generate the instruction traces. The LML programs [Appendix A] *Ackermann*, *Tak* (the Takeuchi function) and *Fibonacci* are used since they generate a large number of recursive function calls. *Towers of Hanoi* is a list-processing example that performs little arithmetic but generates many function calls. In addition the LML programs *Primes*, and *Factorial*, have been used. While these programs are not exhaustive, they do tend to represent a possible mix of instruction reference patterns that the G-machine might encounter in actual programs.

The macrosimulator provides (in the form of a trace file) instruction reference patterns by maintaining a record of the program counter associated with each executed instruction. Associated with each program counter location is the instruction opcode and the number of microinstructions executed in carrying out the instruction function.

The IFU of Design 1 accesses lines in the instruction trace file sequentially, using the PC of the trace file as the location from which the next instruction is to be requested from the memory hierarchy. The ITU stage of the simulation uses the information in the trace file to determine the number of microinstructions that are to be executed by the PCU.

Since the macrosimulator which provides the instruction traces is based on the

microprogrammed pipeline of Design 1, the instruction traces need to be modified for execution on Design 2. In Design 2, the "microinstructions" are assumed to be part of the compiler instruction set. The instruction trace files are modified to reflect the effective movement of microinstructions back into the instruction memory hierarchy. The microinstruction are packaged into instruction sequences fetched from the instruction memory hierarchy, rather than a microinstruction store. Hence the workload for Design 2 is based on an in line expansion of instructions which would be functionally equivalent to microinstructions.

7.2 Simulation Assumptions For Input Parameters

The following assumptions have been made in the process of simulating the throughput of Design 1 and Design 2:

1. The previous simulations of Design 1 show that significant performance improvement can be expected by incorporating an instruction cache into the instruction memory hierarchy. For these simulations, the IFU fetches instructions from a 2-level instruction memory hierarchy. The first level is an instruction cache, the second is the main instruction memory. This organization is used since it is the easiest form of memory hierarchy to implement and it is well understood. Several manufacturers of memory components have inexpensive parts and controller chips for such systems. By changing the state machine (in the simulation) which does the cache management, it is possible to abstract several different memory organizations onto the cache to support instruction fetching.

2. There is little information available on cache hit rates for LML instruction

streams. As a result, several cache hit rates are used (input parameters) for the simulation runs. The cache hit rates used in the simulations range from 90% up to 100%. These assumptions are based on work done on other instruction caches [Smi83], [Pat83], [Fai85].

3. Analysis of test programs, run on the macrosimulator, show that the PCU can execute an instruction (on the average) every 1.3 to 1.5 cycles [Kie87]. For example: 80% of instructions take 1 cycle to execute and 20% of instructions take 3 cycles to execute for an average execution rate of 1.4 cycles. The occurrence of a 3 cycle instruction is based on a random variable calculated using the *random()* function. While this does not model the actual sequence of instruction dispatches exactly, it does give an approximation of performance. For these simulations the input parameter for the average execution rate of the PCU ranges from 1.3 to 1.5 cycles.

7.3 Performance Measure

The primary motivation for the simulations is to establish whether or not Design 2 can provide equivalent instruction throughput to that of Design 1 without using a microinstruction store. Throughput for these simulations is defined in the same way as for the previous simulations.

8. COMPARING THE THROUGHPUT OF DESIGNS 1 AND 2

The throughput of Design 1 and Design 2 executing 6 different LML programs is compared using the simulation assumptions and input parameters described in the previous section. The inputs to the simulations are:

1. Instruction cache hit rate. Range: 90 to 100%.
2. Average cycles per instruction. Range: 1.3 to 1.5 cycles. This is the average time which the PCU consumes in executing an instruction (for either design).

Results from the simulations are shown in Figure 8.1. The results represent the composite throughput of the designs executing the trace files described in the previous section. The x-axis is the cache hit rate. The y-axis is the normalized throughput. The results for each set of input conditions are normalized to the lowest throughput configuration, in this case: Design 1 with a 90% cache hit rate and a 1.5 cycle average PCU instruction execution rate. There are three groups of curves: one group for each of the assumed PCU instruction execution rates (1.3,1.4 & 1.5 cycles).

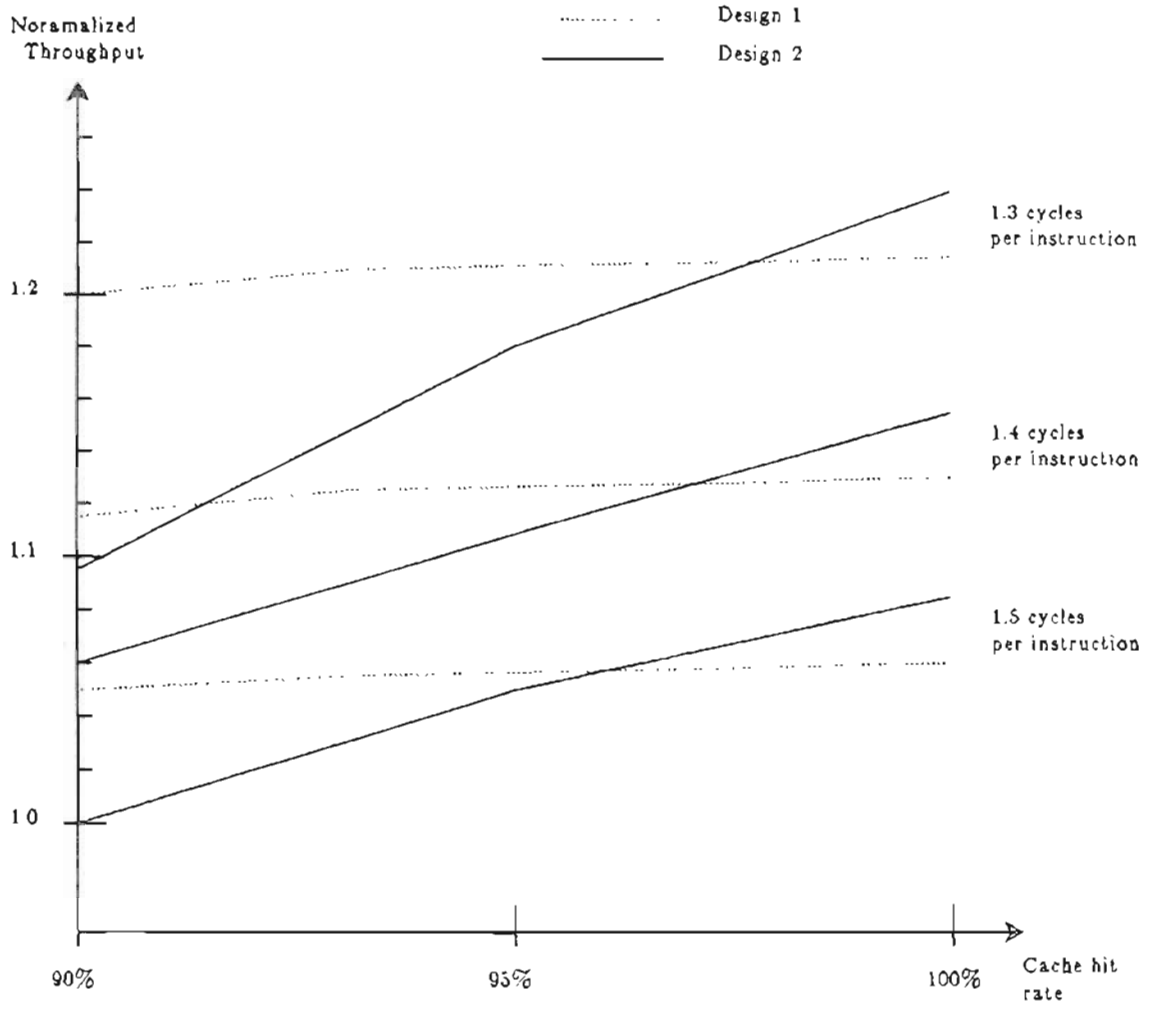

Figure 8.1 Relative Throughput of Design 1 versus Design 2

Two observations can be made from examining these results. First, the throughput of Design 2 (no microinstruction store) is quite sensitive to the cache hit rate while that of Design 1 remains fairly constant across a range of rates. As the average PCU instruction execution rate goes up (i.e. it consumes instructions faster), the demands on the memory hierarchy goes up. However, even for the worst case (cache hit rate: 90%, PCU instruction execution rate: 1.3 cycles/instruction) the performance difference between the two designs is only 10% Increasing the cache hit rate to 95% brings the worst case difference between the designs to about 2.5%.

The other observation concerns the crossover point of the curves for Design 1 and Design 2. As the cache hit rate increases, so does the throughput of Design 2, to the point that for certain cache hit rates (>97%)the performance of Design 2 actually exceeds the performance of Design 1. This can be attributed to three causes:

1. If an instruction in Design 1 expands into a large number of microinstructions, the IFU stalls once the interstage buffer between the IFU and ITU is filled. No further instructions are fetched/pre-fetched pending the completion of the instruction with the large expansion ratio.

   Even when the length of the inter-stage buffer between the IFU and ITU was extended to an 8 word queue, similar results were achieved since the time to move an instruction through the ITU and PCU to execution is usually greater than the time it takes to fetch a subsequent instruction (assuming a cache hit). Hence it is sufficient to buffer a least one instruction beyond that which is being executed by the ITU/PCU.

2. If the instruction, which expands into a large number of microinstructions, is a branch instruction, then the IQ and the IFU must be flushed and the pipeline must be restarted (a 3 cycle penalty), even with the compiler providing delayed branch instructions.

3. The instruction pipeline of Design 1 has an extra pipeline stage: the ITU. Each instruction fetched incurs an extra cycle delay during the translation of the instruction into a microinstruction sequence.

9. CONCLUSION

Enhancements for one design (Design 1) of the instruction pipeline for the G-machine have been simulated to evaluate the effect each enhancement has on the overall performance of the pipeline. The enhancements result in increased performance, however, the pipeline implementation becomes significantly more complex.

Design 2, a RISC design for the instruction pipeline, is proposed to decrease the complexity of the pipeline. The main drawback to a RISC style pipeline is that it does not enjoy the benefits of having a microprogram store feeding the PCU. hence the instruction memory bandwidth requirements of the RISC pipeline may limit it's performance. The results of Section 8 show that the Design 2 can provide similar throughput to that of Design 1. It should therefore be possible to design a RISC instruction pipeline for the G-machine patterned after the pipeline of Design 2. Such a pipeline would provide similar throughput to that of Design 1 while while being less complex to implement. Design 2 is less complex than Design 1 since:

1. There is no need for the ITU of Design 1.
2. The additional logic associated with a microsequencer as well as the additional control lines such as those for local and non-local branches are not required.
3. The Microinstruction Queue and the literals Queue can be eliminated since the pipeline units will operate in lock-step.
4. The IFU is simpler since it has simple instructions types which do not require assembly.

While Design 2 is less complex than Design 1, it does place heavy demands on the instruction memory hierarchy. Since the pipeline units operate in lock-step, the delays associated with instruction memory access become critical. The simulations have shown that a cache hit rate of at least 90% is required. Instruction cache hit rate is directly related to the size of the cache; hence a fairly large instruction cache is required. Recent advances in the cost, density and speed of static RAMs have contributed to the ability to design high-performance instruction caches to support the speed of a RISC processor. The design enhancements studied for Design 1 for reducing the delays associated with instruction memory access should also be applicable to Design 2. A large high speed cache along with instruction prefetching and an instruction buffer can provide the needed instruction memory bandwidth of Design 2.

The RISC instruction pipeline of Design 2 can provide the functionality of Design 1 and should be easier to implement. Equivalent throughput can be achieved by providing an efficient instruction caching mechanism which is attractive to implement when compared to having to implement the additional custom logic required for Design 1.

9.1 Areas For Further Research

Several areas should be explored further. An instruction cache obviously provides a significant performance improvement. In the simulations comparing Design 1 and Design 2, the cache hit rate was an input parameter. Studies should be conducted on what cache organization provides the best performance for the instruction reference pattern of LML programs. Two way and four way set-

associative caches have been proposed. The four way set-associative cache may provide the best performance for LML programs since LML programs typically are made up of calls to small functions.

In the simulations comparing Design 1 and Design 2, the microinstruction routines of the Instruction Translation Unit were simply expanded in line with the instructions generated by the compiler (much like macro expansion). In line expansion of the microinstruction routines increases the size of the program significantly. The larger the program size, the larger the instruction cache will have to be to support hit rates in the 90 to 95% rate. One alternative to in line expansion is to "lock-down" these routines in the instruction cache and call them as subroutines as is done in the Hewlett-Packard Spectrum processor [Cou86]. Since the routines are not recursive and require no parameters, the Instruction Fetch Unit could perform such simple subroutine calls with no significant overhead. With a cache hit guaranteed by locking down these subroutines in the instruction cache, the performance of Design 2 will easily approximate that of Design 1. Treating these routines as subroutines should also keep the program sizes of Design 2 in line with the program sizes of Design 1 with the associated savings in cache sizing.

9.2 Final Thoughts

The design of an instruction pipeline is a multi-dimensional problem. Design enhancements which individually increase performance may provide poor results when combined together as in the case of using explicit instruction prefetching and branch target fetching with the instruction pipeline of Design 1.

The design of an efficient instruction pipeline requires an in-depth knowledge of

both the dynamic behavior of the programs to be executed (which may or may not be available) as well as the implementation issues. Experimentation via simulation seems to be the best available way to narrow in on a design which provides the best possible performance for the least amount of complexity.

REFERENCES

[And67] Anderson, D.W., Sparacio, F.J. and Tomasulo, R.M., "The Model 91: Machine Philosophy and Instruction Handling", *IBM Journal of research and Development*, vol. 11, Jan 1967.

[Arn] Arnold, K.C.R.C, *Screen Updating and Cursor Movement Optimization: A Library Package*, Computer Science Division, Department of Electrical Engineering and Computer Science, Univ. of California, Berkely.

[Cou86] Coutant, D.S., Hammond, C.L., Kelley, J.W, "Compilers for the New Generation of Hewlett-Packard Computers", *Hewlett-Packard Journal*, vol. 37, no.1, Jan 1986.

[Fag64] Fagg, P., Brown, J.L., Hipp, J.A., Doody D.T., Fairclough J.W., Greene J., "IBM system/360 Engineering", FIProceedings AFIPS FJCC, pt. 1 vol. 26, 1964, pp 205-231.

[Far85] Farahmand-nia, S., Kuo, S.L. and Rankin, L., *Simulation of the G-machine Instruction Fetch and Translation Unit*, Dept. of Computer Sc. & Eng., Oregon Graduate Center, May 1985.

[Far86] *Introduction to the CLIPPER Architecture*, Fairchild Corporation.

[Hua84] Huang, K.H., Briggs, F.A., in *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984, pp 98.

[Hen85] Hennessy, J.L., "VLSI RISC Processors", *VLSI Systems Design*, October 1985, pp 22-32.

[Int86] *Interwork Concurrent Programming Toolkit*, User's Manual, Version 1.1, Block Island Technologies, Portland, Oregon, 1986.

[Joh83] Johnsson, T., *The G-machine -- an abstract architecture for graph-reduction*, Dept. of Computer Sciences, Chalmers Univ. of Technology, Gothenburg, 1983.

[Kat83] Katevenis, M.G.H., *Reduced Instruction Set Computer Architecutres for VLSI*", PhD dissertation, University of California, Berkeley, October 1983.

[Kie84] Kieburtz, R.B., "The G-machine: A Fast, Graph Reduction Evaluator", *Proc. of IFIP Conf. on Functional Prog. Lang. and Computer Arch.*, Nancy, 1985.

[Kie85] Kieburtz, R.B., *Control of the Instruction Fetch Unit*, Oregon Graduate Center, April, 1985. Unpublished document.

[Kie86] Kieburtz, R.B., *G-Machine Architecture Handbook*, Oregon Graduate Center, Feb 20, 1986.

[Kie87] Kieburtz, R.B., *Performance Measurement of a G-machine Implementation* , Oregon Graduate Center, Feb, 1987.

[Lee84] Lee, J.K.F, Smith, A.J., "Branch Prediction Strategies and Branch Target Buffer Desgin", *Computer*, Vol 17, No. 1, January, 1984.

[McF86] McFarling, S., Hennessy, J., "Reducing the Cost of Branches", *Proc. of the Thirteenth Annual Symp. on Comp. Arch.*, 1986, pp 396-403.

[Mah86] Mahon, M.J., Lee, R.B., Miller, T.C., Huck, J.C. and Bryg, W.R., "Hewlett-Packard Precision Architecture: The Processor", *Hewlett-Packard Journal*, Vol 37, No. 8, Aug 86, pp 4-21.

[Mon85] Monier, L., Sindhu, P., "The Architecture of the Dragon", *Proc. of the 12th Annual Symp. on Comp. Arch*, 1985, pp. 118-121.

[Mor83] Morris, D., Ibett, R.N., *The MU-5 Computer System*, Springer-Verlag, 1979

[Pat83] Patterson, D.A., Garrison, P., et. al., "Architecture of a VLSI Instruction Cache for a RISC", *Proc. of the Tenth Annual Symp. on Comp. Arch*, ACM, 1983, pp 108-116.

[Pat85] Patterson, D.A., "Reduced Instruction Set Computers", *Comm. of the ACM* , Vol 28, No. 1, Jan 1985, pp 8-20.

[Pei83] Pier, K.A., "A Retrospective on the Dorado, A High-Performance Personal Computer", *Proc. of the Tenth Annual Symp. on Comp. Arch.*, ACM, 1983, pp 252-269.

[Rad83] Radin, G., "The 801 minicomputer", *IBM J. Res. Dev.*, May 1983, pp 237-246.

[Ran86] Rankin, L., "A Dual-Ported Real Memory Architecture For The G-Machine", M.S. Thesis, Oregon Graduate Center, July 1986.

[Sar84] Sarangi, A., "Simulation and Performance Evaluation of A Graph Reduction Machine Architecture", Masters Thesis, Oregon Graduate Center, 1984.

[Smi81] Smith, J.E., "A Study of Branch Prediction Strategies." *Proc. of the Eighth Annual Symp. on Comp. Arch* , pp 135-148, 1980.

[Smi82] Smith, A.J., "Cache Memories", *Computing Surveys*, Vol 14, No. 3, Sept. 1982, pp 473-530.

[Smi83] Smith, J.E., Goddman, J.R., "A Study Of Instruction Cache Organizations

and Replacement Policies", *Proc. of the Tenth Annual Symp. on Comp. Arch*, 1983, pp 132-137.

[Ste64] Stevens, W.Y., "The Structure of System/360, Part II: System Implementations," *IBM System Journal*, vol. 3, no. 2, 1964, pp 136-142.

[Str78] Straubs, R., *ISP Users's Manual*, Case-Western Reserve University, 1978.

[Tay86] Taylor, G.S., Hilfinger, P.N., Larus, J.R., Patterson, D.A., Zorn, B.G., "Evaluation of the SPUR Listp Architecture", *Proc. of the Thirteenth Annual Symp. on Comp. Arch.*, 1986, pp 444-452.

[Tha86] Thakkar, S.S., Hostmann, W.E., "An Instruction Fetch Unit for A Graph Reduction Machine", *Proc. of the Thirteenth Annual Symp. on Comp. Arch.*, 1986.

[Tho64] Thornton, J.E., "Parallel Operation in the Control Data 6600", *Proc. AFIPS FJCC*, vol 26, pr. 2, 1964, pp. 33-40.

[Veg84] Vegdahl, S.R., "A Survey of Proposed Architectures for the Execution of Functional Languages", *IEEE Transactions on Computers*, Vol C-33, No. 12, Dec 1984, pp 1050-1071.

APPENDIX A: LML PROGRAMS

Ackermann

```
letrec

        A x z =  if x=0 then z+1
                                 else if z=0 then A (x-1) 1
                                 else A (x-1) (A x (z-1))
in A 2 2
```

Tak

```
letrec
        tak x y z=
            if ~(y < x) then z
            else        tak (tak (x-1) y z)
                            (tak (y-1) z x)
                            (tak (z-1) x y)
in tak 10 7 5
```

Towers

```
let move x y =  10 * x + y
and
        hanoi f s t n =
        if n = 1 then |move f s|
        else
        (hanoi f t s ( n - 1 )) @ (move f s . hanoi t s f ( n - 1 ))
in hanoi 1 2 3 5
```

Primes

```
letrec
        from x =  x . from (x+1)
and
        filter p seq =
       case seq in
        (a.rest) :
                  if a%p ~= 0 then a.filter p rest
                else filter p rest
        end
and
         sieve seq =
  case seq in
        (p.rest) :
                p.sieve (filter p rest)
        end
and
        cnthd n (h.l) = if n <= 0 then [] else  h . cnthd (n-1) l
in cnthd 10 (sieve (from 2))
```

Fibonacci

```
letrec
        fib n =  if n < 2 then n else fib (n-1) + fib (n-2)
in fib 10
```

Factorial

```
letrec
        fact n =  if n=0 then 1 else (n * fact (n-1))
in fact 10
```

APPENDIX B: CONVERTING THE INSTRUCTION TRACES FOR DESIGN 2

The first group of simulations examined the performance of Design 1 when a range of enhancements are incorporated into the design for decreasing the delays associated with instruction memory access and branches. When these simulations were conducted, the LML compiler was not yet complete and only limited information existed about the instruction referencing patterns of an LML program executing on the G-machine. Since limited information was available, assumptions had to be made and a statistical workload was developed based on these assumptions. The statistical workload was then used for driving the simulations (as described in Section 4).

Prior to the second set of simulations comparing the throughput of Design 1 and Design 2, a macrosimulator was developed by Boris Agapiev for executing compiled LML programs. The macrosimulator uses a model of the instruction pipeline similar to that of Design 1 *i.e.* a microcoded instruction pipeline. The macrosimulator has been instrumented to output (to a trace file) the following information during the execution of a program.

1. The value of the PC associated with each memory reference.
2. The opcode associated with each instruction executed.
3. The number of microinstruction executed in carrying out the fetched instruction.

The LML programs listed in Appendix B have been executed on the macrosimulator to obtain their instruction traces. These instruction traces have been used to as the workload for the simulations comparing the throughput of Design 1 and Design 2.

The first 15 entries of the trace file for the *factorial* program are listed below. The first column is the PC, the second column is the opcode and the third column is the number of microinstructions executed.

```
6 94 4
8 58 3
10 132 1
12 10 2
14 211 1
18 0 1
30 40 1
32 112 1
34 182 1
38 41 10
40 132 1
42 229 10
6 94 49
56 94 4
58 58 3
```

Since Design 2 is not a microcoded pipeline, for equivalent functionality the operations performed by the microinstructions must be directed by instructions fetched from the instruction memory. For the purpose of comparing the the throughput of the two designs, the microinstructions were expaned in line with the instructions fetched from the instruction memory.

In Line Expansion of Microinstructions

In order to preserve the instruction referencing pattern of the programs, the value of the PC in the instruction trace files had to be modified to represent the in line expansion of microinstructions. In the following discussion the trace files from the macrosimulator will be refered to as Trace 1 files while the modified traces containing the in line expansion of the microsequences will be refered to as Trace 2 files. Trace 2 files contains:

1. The PC for the first instruction for a block of instructions. A block of instructions is defined by a sequence of instructions from Trace 1 with the last instruction of the block being a branch instruction.

2. The number of instructions executed sequentially from the starting PC. This number is arrived at by summing the number of microinstructions associated with a block of instructions in Trace 1.

3. The branch type of the instruction terminating the block. Three branch types are defined: Conditional, Unconditional, and EVAL.

4. The address of the location of the next block to be executed.

5. The number of instructions executed for an EVAL.

As an example, the first few lines of the modification of the Trace 1 file for *Factorial* is given below.

Trace 1		Trace 2	Comments
6 94 4	} -------->	6 4 3 81 4	(94 = EVAL, a conditional branch)
10 132 1	}		
12 10 2	} -------->	81 8 1 123 0	
14 211 1	}		(211 = JMP_NOT_ZERO)
18 0 1	}		(NOP for delayed branch)
30 40 1	}		
32 112 1	}		
34 182 1	} -------->	123 24 2 6 0	
38 41 10	}		
40 132 1	}		
42 229 10	}		(229 = CALL_GLOB_FUNC)
6 94 49	} -------->	6 49 3 171 49	(94 = EVAL)
56 94 4	} -------->	171 4 3 246 4	(94 = EVAL)
58 58 3	}		
60 132 1	}		
62 10 2	} -------->	246 23 2 81 0	
64 48 1	}		
66 105 16	}		(105 = RETURN_INT)
8 58 3	}		
10 132 1	}		
12 10 2	} -------->	81 8 1 123 0	
14 211 1	}		(211 = JMP_NOT_ZERO)
18 0 1	}		(NOP for delayed branch)
30 40 1	}		
32 112 1	}		
34 182 1	} -------->	123 24 2 6 0	
38 41 10	}		
40 132 1	}		
42 229 10	}		(229 = CALL_GLOB_FUNC)