An INSTRUCTION FETCH AND TRANSLATION UNIT FOR THE G-PROCESSOR OF THE G-MACHINE

Shyue Ling Kuo
M.S. E.E. Bucknell University, 1980

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

January, 1988

The thesis "An Instruction Fetch and Translation Unit for the G-processor of the G-machine" by Shyue Ling Kuo has been examined and approved by the following

Examination Committee:

Richard B. Kieburtz, Thesis Advisor
Professor and Chairman
Department of Computer Science and Engineering

Robert G. Babb II
Associate Professor
Department of Computer Science and Engineering

Dan Hammerstrom
Associate Professor
Department of Computer Science and Engineering

Shreekant S. Thakkar
Sequent Corporation

Acknowledgements

I would like to express my sincere gratitude to Dr. Richard B. Kieburtz for giving me an opportunity to join G-machine group and work on the IFTU project, for his valuable guidance throughout the study, and patience during the course of this work.

I wish to thank members of my thesis committee for suggestions in preparing this thesis. My thanks also go to members of the G-machine group and members of the 1986 high performance VLSI class for their valuable suggestions and discussions.

Finally, I want to thank my husband Chi-Hwa Tsang for his encouragement, understanding, and support throughout my study at O.G.C.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

An Instruction Fetch and Translation Unit
for the G-processor of the G-machine

Shyue Ling Kuo
Oregon Graduate Center, 1987

Supervising Professor: Richard B. Kieburtz

The G-machine project is to study architecture support for the evaluation of functional language by programmed graph reduction. One area of interest is to build a high performance instruction fetch unit. In this thesis, the design, implementation and performance of an instruction fetch and translation unit (IFTU) for the G-machine is described. The IFTU is a microprogrammed sequential processing unit, it supports high level abstraction (G-code instruction set) and produces simple instructions (horizontal like microcode) for fast execution. Hardware support is provided for all control transfer G-code instructions to enable early instruction prefetching. Other IFTU features are : pipelined design, two way instruction prefetching, early decoding, delayed jump implementation and proper buffer placement.

A microsimulator of the IFTU has been built and integrated into the G-processor microsimulator. The IFTU's performance has been studied by running benchmark programs on the G-processor microsimulator.

1. INTRODUCTION

1.1. Introduction

The objective of this thesis is to study the performance of an instruction fetch and translation unit (IFTU) designed for the G-processor of the G-machine (Graph Reduction Machine). The G-machine is based on an abstract, stack architecture designed by Thomas Johnsson and Lennart Augustsson [Aug84] [Joh83] [Joh84] as the evaluation model for a compiler of LML, a functional programming language ML [Car84] with lazy evaluation rules. In the G-machine, expressions are represented as graphs and an expression is evaluated by successively applying reduction rules until it can not be reduced any more. The G-machine evaluates a functional program by programmed graph reduction, that is, by executing instructions which specify reduction steps, derived by compilation. An alternative to programmed graph reduction is interpretive β reduction (lambda calculus) [Ber75] or combinator reduction [Tur79] where the next reduction step is derived dynamically from an expression graph.

Architecture support for the evaluation of LML functional language based on the G-machine abstract architecture has being studied by Kieburtz and his research team members. A G-processor [Kie85] [Kie86] was designed as a result of Kieburtz's study. The G-processor provides hardware support for 1) graph traversal, 2) instruction fetch, 3) context switching, and 4) dynamic list structured memory. The G-processor consists of the following functional units : an instruction fetch and translation unit (IFTU), a processor controller unit (PCU), a value stack (V stack) with an ALU and a pointer stack (P stack), an address (A) register and a tag (T) register (see Figure 1.1). The G-processor is designed as a pipeline in which the IFTU feeds instructions to the execution unit. The execution unit is

internally pipelined, but that will not be discussed here.

The IFTU described here is designed as a pipeline (see Figure 1.2), it provides support for complex instruction set (G-code) and yet generates simple instructions (microcode) they can be rapidly executed by an execution unit.

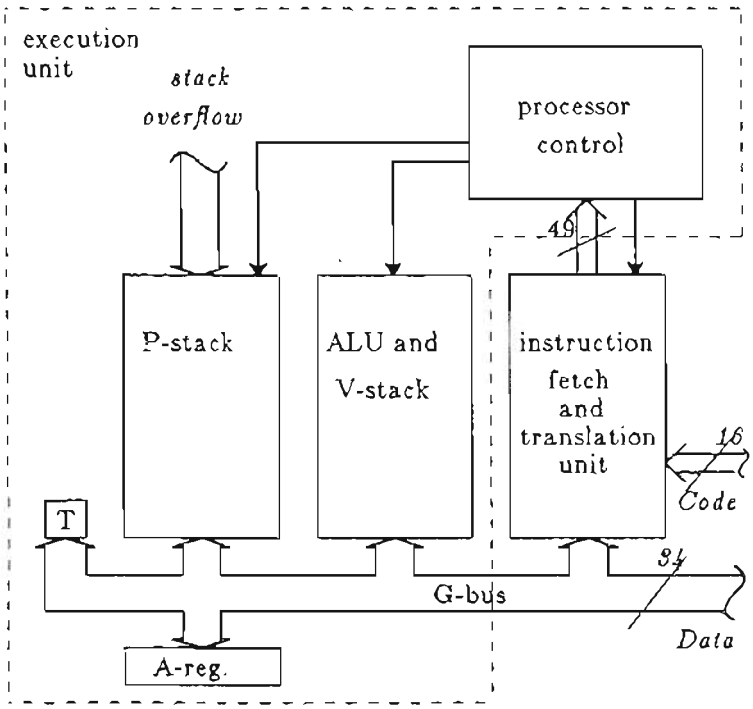


Figure 1.1 The G-processor block diagram [Kie85].

The IFTU design features are described as follows:

Microprogramming - The IFTU is designed with a translation unit [Bae80] with a micro store. The translation unit translates the fetched G-code instructions (output of the LML compiler) into sequences of microcode. Since most G-code instructions are translated into more than one microinstruction, this design reduces memory fetch bandwidth requirements.

Two Way Prefetching - A dynamic average of G-code to microcode expansion ratio in the range of 5-7 was observed from a previous simulation [Kie85]. Thus it is expected that there is some excess memory bandwidth to support more than one instruction stream. To make

use of the excess memory bandwidth, the IFTU is designed with two way prefetching. At a conditional branch instruction code from the normal and branch target instruction streams can both be prefetched into the IFTU. This design is intended to shorten the delay time after a branch is taken and to increase throughput. The effectiveness of this design depends on the amount of excess memory bandwidth available. In this thesis we investigate whether there is enough memory bandwidth to support two way prefetching.

Early Decoding - The IFTU partially decodes the fetched G-code to set up optimal instruction prefetching. The decoding feature also reduces G-code instructions to a more compact intermediate form (by removing literals and address constants) before sending them for translation. Early decoding has been used in Edge Computer [Ele87].

Hardware Support for Control Transfer Instructions - For an unconditional jump instruction, control transfer to the jump address is performed by the IFTU and no microcode is sent to the execution unit [Wil83]. For a conditional jump instruction, the IFTU fetches both the normal and branch target instruction streams. When a jump is taken, the IFTU flushes the instruction pipeline. When a jump is not taken, there is no interruption on the pipeline and only the branch target instruction buffer is flushed. Existing approaches that have been used in handling branch control transfer instructions include : branch targer buffers [Lee84] [Mor79] [Smi81], taken/not taken switches [Pat83] [Hai79], multiple instruction streams [And67], and loop buffers [Ibb82].

Hardware Support for Literal instructions - Literals are queued in a hardware FIFO queue before being passed to the execution unit. Therefore at the time the execution unit requests a literal, it may already be in the literals queue.

Delayed Jump - Delayed jump [Pas82] design is implemented both in the G-code and at the microcode level. A delayed instruction is placed after a conditional jump. The design is to keep the execution unit busy processing the delayed instruction while the IFTU is fetching

the next instruction from the stream selected by resolving the jump. The effectiveness of this design is studied in this thesis.

Buffers - Buffers are implemented in the IFTU wherever there is irregularity between an item being produced and consumed. Buffers in the IFTU are : a) an instruction queue to buffer assembled G-code instructions, b) a literals queue to buffer literals extracted from literal instructions, and c) two prefetch buffers to buffer instructions fetched from normal and branch target instruction streams.

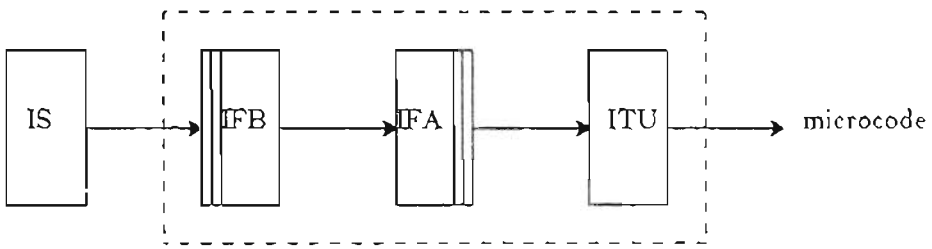


Figure 1.2 The instruction fetch and translation unit pipeline.

IFB - instruction fetch and buffer unit
IFA - instruction decoding and assembly.
ITU - instruction translation unit.
IS - the instruction store.

A microsimulator has been built for the IFTU and integrated with the execution unit microsimulator [1]. The IFTU simulation is done by running G-code programs compiled from LML source code programs on the G-processor microsimulator. The simulation studies the IFTU's throughput, prefetching ability, the elasticity provided by various buffers, and identifies performance parameters. Buffers sizes and memory fetch bandwidth have been varied in the simulation to search for an optimal IFTU configuration. The reasons why the IFTU is simulated at a detailed design level are :

[1] G-processor microsimulator was implemented for Professor Kieburtz by his students: Kuo, S. L. (IFTU) Farahmand-nia, S. and Rankin, L (execution unit).

1.) The IFTU simulator is a building block for the G-machine simulator.

2.) IFTU memory fetch bandwidth depends critically on the order of the G-code instructions being executed. It is difficult to accurately model G-code by a statistical simulation.

1.2. Background

An earlier design of the G-processor instruction fetch unit [Kie85] [LMO84] was a buffered pipeline with an effort to achieve low pipeline latency and high throughput. Latency was considered an important parameter because G-code compiled from LML source programs was expected to contain a relatively high incidence of case switch and conditional jump instructions and high incidence of rapidly executed stack instructions. Case and conditional jump instructions interrupted the normal sequence of instruction fetches and required restarting the pipeline. To minimize latency, four prefetch buffers were designed. This allowed up to four-way instruction prefetching to occur along the alternate instruction paths consequent to a conditional jump or a case switch (four way case) instruction. It was hoped that when a jump was taken, the next instruction would already be in one of the prefetch buffers.

The IFTU was designed with an 8 bits wide data path and the following functional organization. G-code is fetched from the instruction store into an instruction fetch buffer unit (IFB). This unit contains four instruction buffers. From the active instruction stream, a sequence of bytes is transmitted to an instruction assembly unit (IFA) which assembles instructions and their operands, and delivers them to the instruction translation unit (ITU). There G-code instructions are translated into sequences of RISC like microinstructions which are buffered by the instruction queue.

This version of the IFTU design has been simulated [Far85]. Simulations were tailored to observe the performance of conditional jump and case instructions. Statistical data were used in estimating the distribution of instruction types of an input G-code program. The number of microinstructions translated from each G-code instruction was estimated. It was assumed that the execution unit could execute a microinstruction in a single clock cycle. Simulation results showed that prefetching time made available by G-code-to-microcode

expansion was not adequate and suggested to increase the memory fetch bandwidth. Due to a lack of enough prefetching time, a case switch was frequently resolved by the execution unit before the target addresses could be fetched.

The IFTU architecture has also been studied by Thakkar and Hostmann [Tha86]. Instead of using prefetching buffers, a cache was used in their design. They studied the following IFTU configurations via statistical simulations:

1.) No prefetching along the branch target instruction stream.

2.) Explicit prefetching into the cache of the possible branch targets of conditional jump and case-switch instructions.

3.) Explicit prefetching with abort. In this configuration, the prefetching sequence was aborted if the branch was resolved by the execution unit before the sequence was completed.

Thakkar and Hostmann's study concluded that latency had little effect on throughput. They proposed using sequential prefetching with a small cache or using a large cache without any explicit prefetching in the IFTU design.

1.3. The Current Design

To widen memory fetch bandwidth, a 16 bit data path was proposed. Four way prefetch buffers were reduced to two way prefetch buffers. Case instructions were replaced with conditional jump and unconditional jump instructions by the compiler. This significantly simplified the IFTU design and increased memory fetch bandwidth made available to each enabled prefetch buffer.

When the ITU was designed with the capability of translating a microinstruction in a clock cycle, it was realized that there was no need to buffer microinstructions because the ITU was able to produce a microinstruction at a rate no slower than that the execution unit

could consume one. Instead, a buffer should have been placed between the IFA and the ITU due to the irregularity of a G-code instruction being produced for translation and a G-code instruction being translated. The production rate of G-code instructions for translation was irregular because unconditional jumps did not produce any executable G-code, conditional jumps might delay the following code sequence, and some executable G-code took longer to produce than others. Since G-code instructions expanded into different length of microcode sequences, it took variable length of time to translate different G-code instructions.

1.4. A RISC Design

Consequent to the result of this study and Hostmann's study [Hos88], a RISC IFU design was proposed [Kie87]. In the RISC design, G-code-to-RISC-code expansions are stored in the instruction store. A two way set associative cache is used to cache G-code and G-code expansion sequences. The RISC IFU performs the function of fetching G-code and macro expansion of G-codes to RISC codes.

2. THE IFTU DESIGN AND ORGANIZATION

The Instruction Fetch and Translation (IFTU) pipeline (see Figure 2.1) consists of the Instruction Fetch and Buffer Unit (IFB), the Instruction Assembly Unit (IFA) with a Literal Queue (LQ) and an Instruction Queue (IQ), and the Instruction Translation Unit (ITU) with a Micro Store. These units form a sequence of stages of an instruction pipeline that feeds an execution unit. G-code is fetched from the Instruction Store (IS) into the IFB. This unit contains two prefetch buffers, which can be enabled concurrently. This allows prefetching to occur along the normal and alternate instruction streams consequent to a conditional jump instruction or a function call. From the active instruction buffer, sixteen bit words of the instruction stream are sent to the IFA which assembles instructions and operands and delivers them to the IQ.

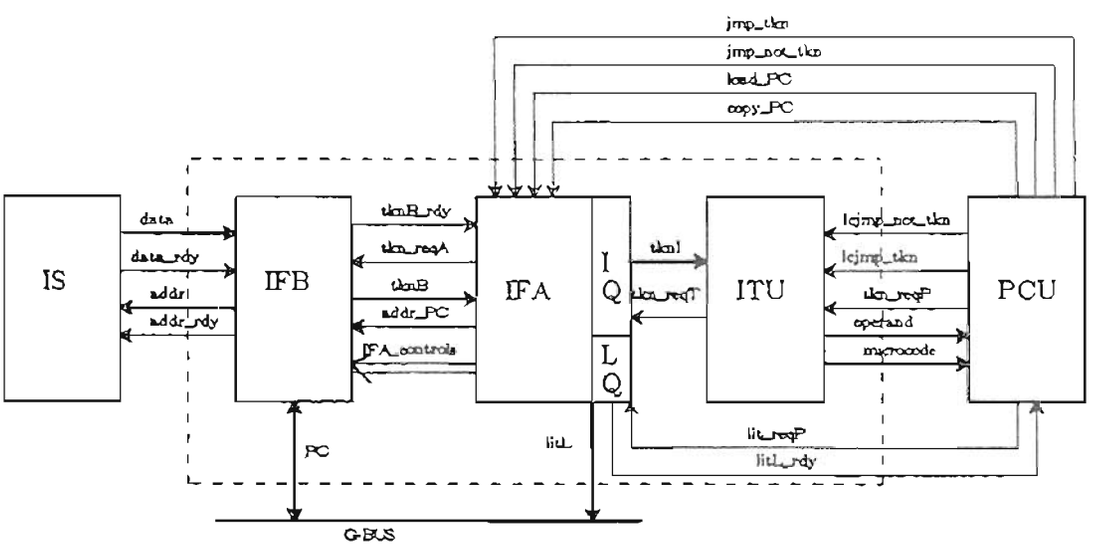


Figure 2.1 The Instruction Fetch and Translation Unit of the G-processor.

The IFA can continue to assemble instructions as long as the IQ is not full. The ITU translates each instruction into a sequence of microinstructions, and passes them to the execution unit. Functions performed by each IFTU pipeline stage are described in detail in the following sections.

2.1. Instruction Store (IS)

Using the address provided by the IFB, G-codes are fetched from the Instruction Store. The smallest addressable unit of the Instruction Store is a word (16 bits) The number of words fetched at one time is an IFTU parameter. Different fetch sizes have been simulated to determine their effects upon the IFTU performance. A fetch is initiated (fetch address is provided) only after the previous one is completed and only if there is space in the prefetch buffer. There is no two outstanding memory references at one time.

Different memory fetch latencies are also simulated to determine what is required for the peak IFTU performance.

2.2. Instruction Fetch Control and Prefetch Buffers (IFB)

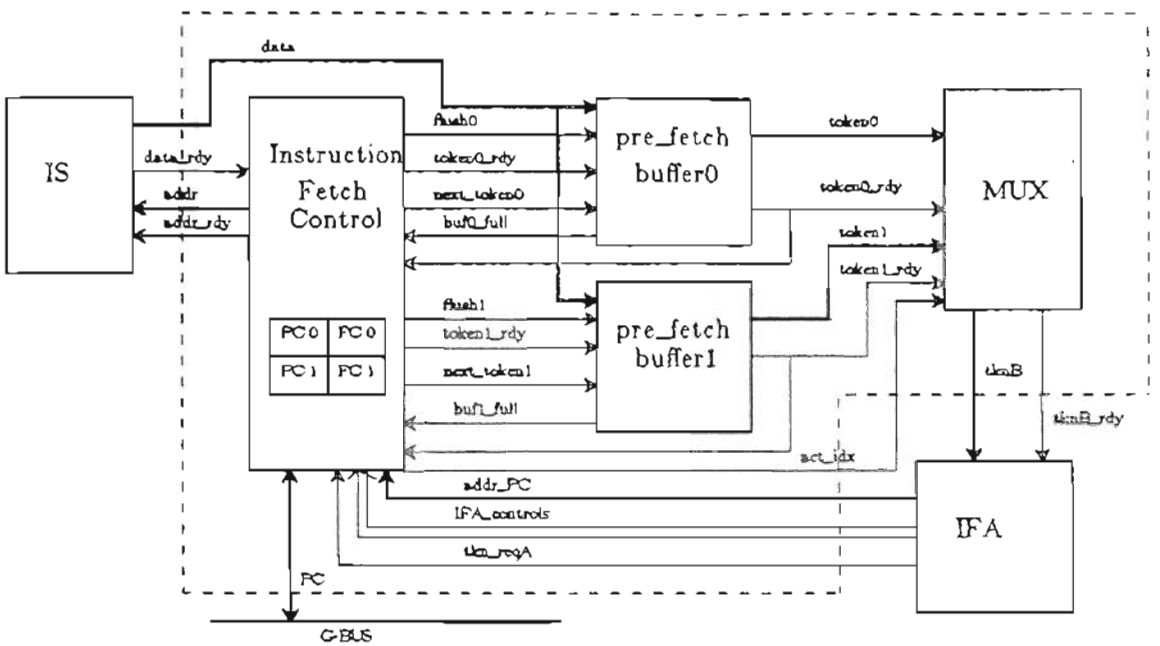


Figure 2.2 The Instruction Fetch Control and Pre-fetch Buffers.

The IFB maintains two prefetch buffers which are used to store the fetched G-code from the Instruction Store. For each prefetch buffer, the IFB maintains a fetch counter (FC) which contains the next memory fetch address and a program counter (PC) which contains the address of the next instruction to be processed by the IFA. An FC and a PC can be reset with an address extracted from a jump instruction of the fetched instruction stream or with the address provided via the G-Bus. The value of the active program counter (PC) can be loaded onto the G-Bus and saved in the P-stack. Both prefetch buffers may be enabled concurrently. This allows up to two way instruction prefetching. One of the prefetch buffers holds the normal instruction stream and is designated as the active prefetch buffer. From the active prefetch buffer, the prefetched G-codes are sent to the Instruction Assembly Unit for assembly. The other prefetch buffer holds the head of an alternate instruction stream

which may be activated consequent to a conditional jump or a function call.

When a conditional jump is resolved as taken, the IFB is responsible for flushing the current active prefetch buffer and activating the alternate prefetch buffer. Thus the alternate instruction stream becomes the normal instruction stream. Meanwhile, the IFTU pipeline is flushed. When a conditional jump is resolved as not taken, the IFB only flushes the alternate prefetch buffer, and there is no interruption of the IFTU pipe.

When both prefetch buffers are enabled, the IFB alternates memory fetch between the two prefetch buffers. This significantly reduces the memory bandwidth available to the normal instruction stream. Different prefetch algorithms have been implemented to compare the IFTU performances. Because most G-codes are expanded into a number of microinstructions, it was anticipated that there might be enough memory bandwidth to accommodate two instruction streams.

The Instruction Fetch control is responsible for initiating a memory fetch, receiving a memory's output, assigning an enabled prefetch buffer for the next memory fetch, incrementing the active program counter and responding to the control signals from the IFA. The functions preformed by the Instruction Fetch Control will be described in detail in the following sections.

2.2.1. Instruction Fetch Control

For each IFB buffers, there is a program counter register (PC), and a fetch counter register (FC). Each FC is guarded by a one-bit Enable/Disable flag which governs prefetching into the buffer, and by a one-bit Is_Disabled flag which is set when its corresponding buffer is to be flushed and there is an outstanding fetch in progress for the buffer. If the Is_Disabled flag has been set, it will be reset when the outstanding fetch is completed and the fetched data is discarded. A Fetch Index (f_idx) flag is used to indicate the FC register from which the next memory address is fetched.

2.2.1.1. Assign an Enabled Prefetch Buffer for the next Memory Fetch

Prefetch algorithms based on round robin scheduling was implemented with initial fetch priority given to either of the two instruction streams. The prefetch algorithm with initial fetch priority to buffer 0 is described in Table 2.1.

f_idx	previous f_idx	Buf0 full	Buf1 full	Buf0 enabled	Buf1 enabled
2	–	–	--	n	n
1	–	--	n	n	y
2	–	–	y	n	y
0	–	n	--	y	n
2	–	y	--	y	n
0	2	n	n	y	y
1	0	n	n	y	y
0	1	n	n	y	y
0	–	n	y	y	y
1	--	y	n	y	y

where

f_idx indicates the prefetch buffer for which the next memory fetch is initiated:

2 means no memory fetch will be initiated;

0 means the next memory fetch is for prefetch buffer 0;

1 means the next memory fetch is for prefetch buffer 1.

Table 2.1 Prefetch buffer index table.

2.2.1.2. Initiate a Memory Fetch

When a fetch is completed, the Instruction Fetch Control gets the address from the next enabled FC[f_idx] to start the next memory fetch. If none of the prefetch buffers is enabled or has space to store the memory's output, the Instruction Fetch Control will not initiate a fetch at this cycle. However the fetch condition will be checked in the following cycles. After a memory fetch is initiated, the fetch counter will be incremented. The size of the increment depends on the size of the memory fetch. The increment operation will be per-

formed concurrently with the memory fetch so that by the next memory fetch cycle, the memory fetch address is already available in the fetch counter.

2.2.1.3. Increment Program Counter

The Instruction Fetch Control increments the active PC by one whenever a word of data is transmitted to the IFA from the active prefetch buffer.

2.2.1.4. Respond to the IFA Control Signals

Responding to the control signals generated by the IFA, the IFB performs the following functions:

a) The IFB enables a prefetch buffer for prefetching G-code,

b) The IFB performs the control transfer of the active instruction stream and the alternate instruction stream.

c) The IFB flushes prefetch buffers.

d) The IFB loads the content of the active PC onto the G-Bus.

e) The IFB resets a PC and an FC with the address provided on the G-Bus or the address extracted from the fetched instruction stream.

Operations performed by the IFB for each IFA signal are described in detail in Appendix C.

2.2.2. Instruction Prefetch Buffers

There are two prefetch buffers in the IFB that function as first-in first-out (FIFO) queues. Data from the Instruction Store are stored in the buffers under the IFB control. Upon request the active prefetch buffer delivers its contents to the IFA in FIFO order and in 16 bit quantities.

To simulate fixed sized and variable sized memory fetches two types of prefetch buffers have been implemented. One buffer configuration is a fall through FIFO queue with a width of 16 bits. This type of buffer is used for simulating 16 bit fixed size memory fetches Responding to the token_ready signal from the IFB, the queue writes a token into the first available slot from the front of the queue. Responding to the next_token signal from the IFA, the queue puts its front item onto the output bus and asserts the token_ready signal to the IFA. The rest of the queue's items are subsequently shifted forward one slot. In case the queue is empty, the token_ready signal is set to false. The depth of a buffer is an IFTU parameter.

In another configuration, a buffer is designed to be two words deep. Each word has the size of a memory fetch. This type of buffer is used to simulate different memory fetch sizes. Responding to the token_ready signal from the IFB, the buffer puts the fetched data into one of the available words. Responding to the next_token signal from the IFA, the buffer delivers its contents in FIFO order and in the quantities of 16 bits to the output bus and asserts the token_ready signal to the IFA.

2.3. Instruction Assembly Unit (IFA)

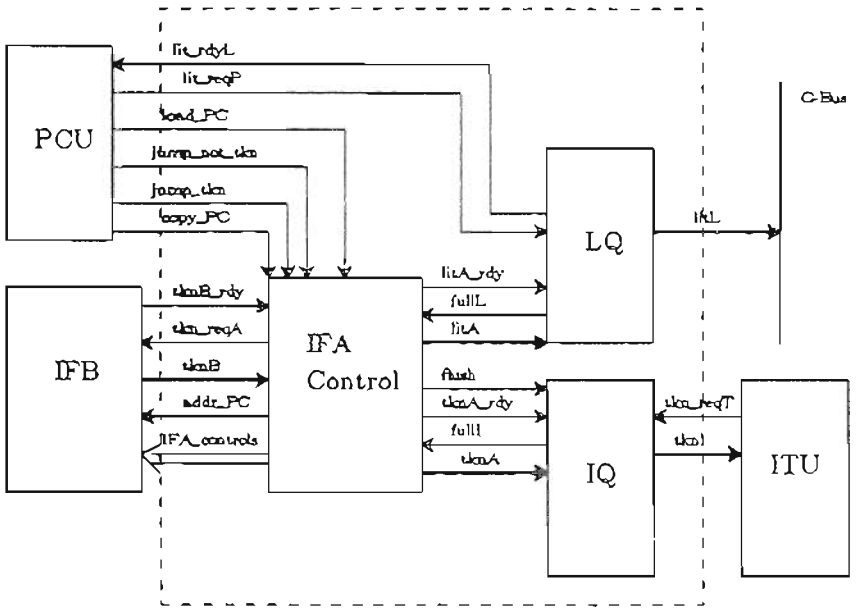


Figure 2.3 The Instruction Assembly Unit.

The IFA fetches two-byte words from the active prefetch buffer, assembles them into G-code instructions and delivers G-code instructions to the Instruction Queue (IQ). It also feeds an associated Literals Queue (LQ) where literal data fields extracted from instructions are stored. The IFA extracts addresses from jump instructions and sends them to the IFB to enable prefetching along an alternate instruction stream. It responds to signals from the PCU by requesting the IFB to activate an alternate instruction stream, to reset a PC and an FC and to enable/disable a prefetch buffer for prefetching. The IFA instructs the IFB and the ITU when to flush the IFTU pipeline.

The control of the IFA is described by a finite state machine. Based on the current IFA state, the signals (jump_tkn, jump_not_tkn , copy_PC, load_PC) from the PCU and the signal (token_rdy) from the active prefetch buffer, the IFA determines its next state. There

are eight types of G-code instructions. The way the IFA assembles each type of G-code instructions will be described in detail in section 2.3.2.

2.3.1. Component

The IFA consists of : 1) a four bit wide register holding the IFA states and 2) a Jump-Pending flag which, when set, indicates that a previous conditional jump instruction has not yet been resolved, and no following instruction that is a Conditional jump, Callglobfun, Return, or Eval can be assembled.

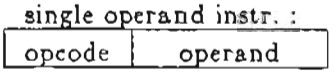
2.3.2. Functions

The IFA partially decodes G-code instructions to classify them into eight categories listed in Table 2.2. The following sections describe how each type of G-code instructions are assembled.

	Bytes taken from IFB	Opcode bytes passed to IQ	Operand bytes passed to IQ	Bytes passed to LQ or PC
1) single operand instr.	2	1	1	--
2) short literal	2	1	0	1
3) long literal	6	1	0	4
4) unconditional jump	4	0	0	3 (PC)
5) conditional jump	4	1	0	3 (PC)
6) callglobfun	4	1	0	3 (PC)
7) return	2	1	1	--
8) eval	2	1	0	--

Table 2.2 G-code instruction types.

2.3.2.1. Single Operand Instruction:



The IFA passes the opcode and the operand to the IQ.

2.3.2.2. Short Literal Instruction:

short lit instr. :

n	opcode	lit_1

LQ entry :

0	0	0	lit_1

The IFA passes the opcode to the output, and zero fills the output operand. The IFA then sends the output to the IQ. Meanwhile the low order byte of the LQ entry is filled with the lit_1 and the high order three bytes of the LQ entry are zero filled. The LQ entry is then sent to the LQ.

2.3.2.3. Long Literal Instruction:

long lit instr. :

n	opcode	-
n+1	lit_1	lit_2
n+2	lit_3	lit_4

LQ entry :

lit_1	lit_2	lit_3	lit_4

The IFA passes the opcode to the output and zero fills the output operand. The output is then sent to the IQ. Meanwhile the IFA fetches the next two words from the active prefetch buffer to form a long literal. The literal is subsequently delivered to the LQ.

2.3.2.4. Unconditional Jump Instruction:

uncond. jump instr.:

n	opcode	addr1
n+1	addr2	addr3

addr_PC:

addr1	addr2	addr3

After the IFA decodes an unconditional jump instruction, it fills the most significant byte of the addr_PC with addr1 and the two low order bytes of the addr_PC with the next fetched word (addr2 addr3). The IFA then sends the "unconditional_jump" signal to request the IFB to start fetching the normal instruction stream from the address provided on the addr_PC. For this type of instruction, no output instruction will be generated. Control transfer is performed within the IFTU pipeline.

2.3.2.5. Conditional Jump Instruction type

cond. jump instr.:

n	opcode	addr1
n+1	addr2	addr3

After the IFA decodes a conditional jump instruction, it passes the opcode to the output and fills the output operand with zero. It then sends the output to the IQ. Meanwhile the IFA fills the most significant byte of the addr_PC with addr1 and the two lower order bytes of the addr_PC with the next fetched word (addr2 addr3). The IFA then sends "load_alt" signal to request the IFB to start fetching the alternate instruction stream from the jump address provided on the addr_PC.

2.3.2.5.1. Delayed Branch Strategy The G-Machine compiler generates a delayed G-code instruction following a G-code conditional jump instruction. If there is no delayed G-code instruction that can be found, a G-code NOP is used as a delayed instruction. The IFTU ensures that the delayed G-code instruction is assembled before it processes the conditional jump taken or not taken. Thus it implements the "delayed branch" strategy [PaS82] to compensate for the "inertia" of an instruction pipeline in a RISC architecture. Because a conditional jump instruction requires the use of both the active and the alternate instruction streams, the Jump_Pending flag is set when a conditional jump instruction is assembled to prevent the subsequent conditional and unconditional jump, Callglobfun, Return, and Eval types of instructions, which require the use of an extra prefetch buffer, from being assembled.

The conditional jump taken or jump not taken signal from the PCU can arrive while the IFA is assembling single operand type instructions, short literal/long literal type instructions or while the IFA is waiting for the Jump_Pending flag to be reset. When a jump taken signal is received, the IFA sends "jump_taken" signal to request the IFB to activate the alternate instruction stream and to flush the active instruction buffer. The IFA also generates controls to flush the IFTU pipeline. When a jump not taken signal is received, the IFA sends the IFB to flush and disable the alternate instruction stream.

2.3.2.6. Callglobfun Instruction:

callglobfun instr.:

n	opcode	addr1
n+1	addr2	addr3

After the IFA decodes a Callglobfun type instruction, it passes the opcode to the output and fills the output operand with zero. The output is then sent to the IQ. Meanwhile the IFA fills the most significant byte of the addr_PC with addr1 and the low order two bytes of the

addr_PC with the next fetched word (addr2 and addr3). After the addr_PC is formed the IFA then sends "load_Alt and disable_Act" signal to request the IFB to start fetching the alternate instruction stream from the address provided on the addr_PC.

The IFA then stops the IFB from advancing the current active PC by not fetching any more G-code words. The IFA waits for a copy_PC signal from the PCU to send "copy_PC and jump_taken" signal to request the IFB to copy the current active PC onto the G-Bus so that it can be saved as the return PC then to activate the alternate instruction stream. If the addr_PC address is formed at the same clock cycle as the copy_PC signal is received, the IFA sends "copy_PC load_Alt and jump_taken" signal to request the IFB to perform the control transfer to the calling function address.

2.3.2.7. RET/RET-INT Instruction:

RET/RET-INT instr.:

n	opcode	-

After the IFA decodes a RET or a RET_INT instruction, it passes the opcode to the output and sends the output to the IQ. The IFA waits for a load_PC signal from the PCU to send "load_PC" signal to request the IFB to start fetching the normal instruction stream from the return address provided on the G-Bus.

2.3.2.8. Eval Instruction:

eval instr.:

n	opcode	-

When the IFA decodes an EVAL instruction, it passes the opcode to the IQ. The IFA also stops the IFB from advancing the active PC by not fetching any more words. The IFA waits for either a jump_not_taken signal or a copy_PC signal from the PCU. When the jump_not_taken signal is received, the IFA resumes to assemble the next instruction.

When a copy_PC signal is received, the IFA sends "copy_PC" signal to request the IFB to copy the current active PC onto the G-Bus so that it can be saved as the return address. The IFA then waits for a load_PC signal from the PCU to send "load_PC" signal to request the IFB to start fetching the normal instruction stream from the address provided on the G-Bus.

2.3.3. Literals Queue (LQ)

Literals fetched from the Instruction Store are stored in the LQ and delivered to the G-Bus upon the request of the PCU. When the IFA decodes a literal instruction, it extracts a literal from the instruction and formats it into a four-byte-wide literal entry.

The LQ is a FIFO queue. Responding to the lit_rdy signal from IFA, the LQ writes a literal into its first available slot from the front of the queue. The IFA ensures that there is a space in the LQ before raising the lit_rdy signal to deliver the input literal to the LQ. When the PCU decodes a literal instruction, it checks if any literal is in the LQ. If there is a literal in the LQ, the PCU then requests the LQ to put the next literal in FIFO order onto the G-Bus. The P-stack then absorbs the literal from the G-Bus.

2.3.4. Instruction Queue (IQ)

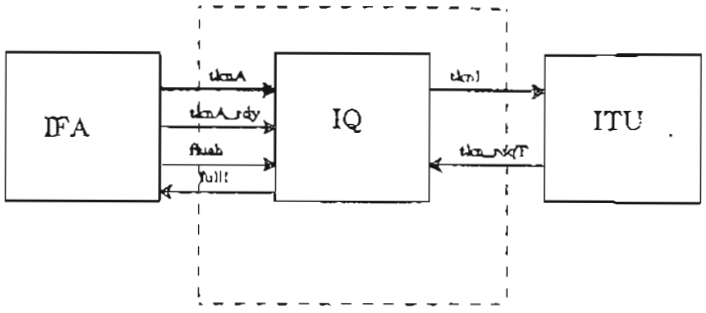


Figure 2.4 The Instruction Queue.

It takes variable number of clock cycles for the IFA to assemble a G-code instruction and for the ITU to translate a G-code instruction into microcode. To balance processing speeds of the IFA and the ITU, an Instruction Queue is placed between them as shown in Figure 2.4. This set-up increases the elasticity of the IFTU. The IQ is designed as a FIFO queue. Two byte wide G-code instructions from the IFA are stored in the IQ and delivered in FIFO order to the ITU upon request. The IFA checks if there is an available slot in the IQ before it sends an assembled G-code instruction to the IQ by asserting the tknA-rdy signal. Responding to the tknA_rdy signal), the IQ writes the assembled G-code instruction into its first available slot from the front of the queue. Whenever the ITU is ready for an input, it takes the tknI which is the front item of the IQ, and asserts the tkn_reqT control. This causes the rest of the queue items to shift forward one slot. When the ITU is ready for an input and the IQ is empty, the ITU will receive a virtual item which is defined as the G-code NOP instruction. In Chapter 5, performance results of the IFTU designed with and without an IQ will be presented.

2.4. Instruction Translation Unit (ITU)

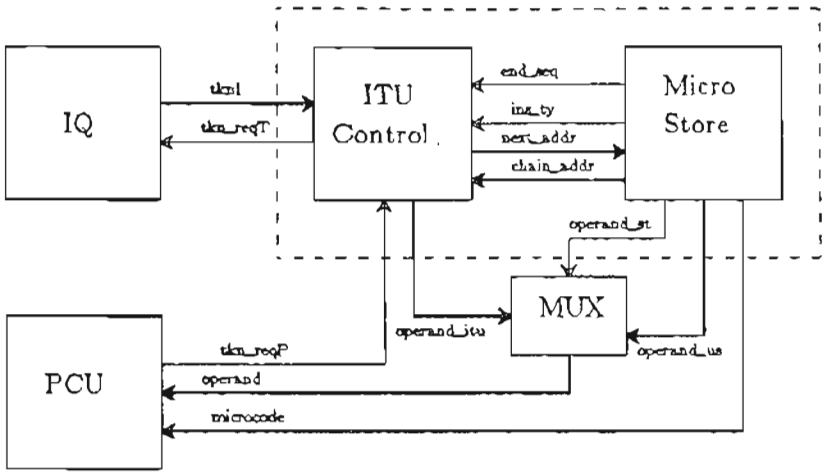


Figure 2.5 The Instruction Translation Unit.

The ITU shown in Figure 2.5 translates each G-code instruction fetched from the IQ into a sequence of microinstructions and passes these to the execution unit. The ITU is capable of generating one microinstruction at every clock cycle. It consists of a Micro Store in which all the G-code to microcode expansion sequences are stored (Appendix B). Data are stored in the Micro Store as fixed length consecutive words whose format is shown in Table 2.3.

microcode	operand_us	operand_st	chain_addr	end_seq	ins_ty
43 bits	5 bits	1 bit	12 bits	1 bit	1 bit

Table 2.3 Word format of the Micro Store.

where,

microcode : to be executed by the execution unit,

operand_us : the microcode operand from the Micro Store,

operand_st : specifies the source of a microcode operand
(from either the Micro Store or the ITU),

chain_addr : chained address of the next microinstruction,

end_seq : indicates an end of expansion sequence,

ins_ty : indicates the fetched microinstruction is a local conditional jump.

The address of the first microinstruction of an expansion sequence is given directly by the opcode of the G-code instruction being translated. The address of the next microinstruction of the sequence comes from a chained address fetched from the Micro Store with the preceding microinstruction. Microcode is sent to the execution unit directly from the Micro Store. A microcode sequence may contain local conditional jump instructions. Unconditional jumps are implicit in the chained address. G-code-to-microcode expansion sequences are designed such that a local conditional jump is followed by a delayed microinstruction. The ITU ensures that a delayed microinstruction [PaS82] is processed by the execution unit after a local conditional jump. This provides the ITU with a clock cycle in which to generate an address along the proper path after the decision is made on a local conditional jump.

2.4.1. The ITU Control

The ITU control is described as a finite state machine with five states as shown in Figure 2.6. The state machine is clocked by the next token request from the PCU and always starts in state S1 to translate a G-code instruction into a microcode sequence.

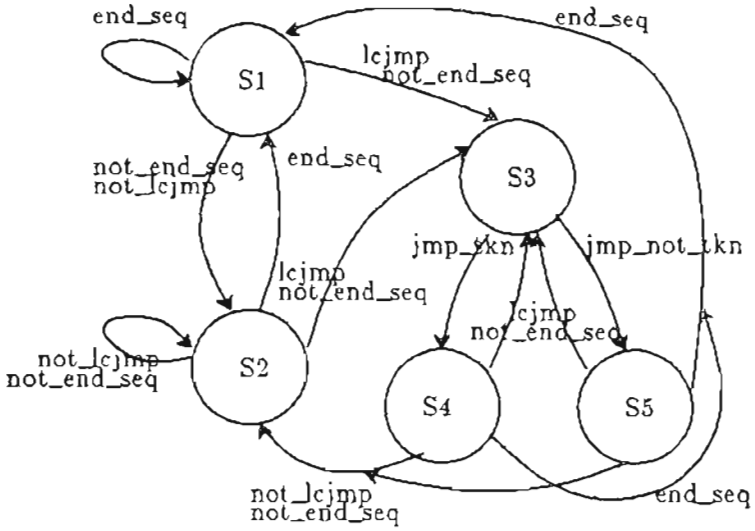


Figure 2.6 The State Diagram of the ITU Control.

2.4.1.1. State S1

In state S1, the ITU fetches the next G-code instruction from the front of the IQ. The instruction's opcode serves as the address of the first microinstruction of a microcode expansion sequence. It is passed to the Micro Store to fetch the next microinstruction. It is also passed to an incrementor to compute the address of the next delayed microinstruction. The G-code instruction's operand (operand_ITU) is sent to a multiplexer. Based on the control signal (operand_st) from the Micro Store, either this operand or the operand from the Micro Store is selected to be sent to the execution unit.

2.4.1.2. State S2

With an address supplied by the ITU, the Micro Store fetches a microinstruction. It also fetches the chained address of the next microinstruction. The chained address is sent back to the ITU. In state S2, the received chained address is passed to the Micro Store to fetch the next microinstruction. It is also passed to an incrementor to compute the address of the next delayed microinstruction.

2.4.1.3. State S3, State S4 and State S5

State S3 is entered when the previous fetched microinstruction is a local conditional jump. The address of the delayed microinstruction which is the output of the incrementor is sent to the Micro Store. The received chained address is stored in a register as the *jump address*. The ITU will stay in this state until jump taken or not taken signal is received.

When a jump taken signal is received, the ITU goes to state S4. In this state, the jump address is sent to the Micro Store to fetch the next microinstruction from the jump taken path. The jump address is also fed into the incrementor to compute the address of the next delayed microinstruction.

When a jump not taken signal is received, the ITU goes to state S5. In this state the received chained address is sent to the Micro Store to fetch the next microinstruction from the fall through path. The chained address is also fed into the incrementor to compute the address of the next delayed microinstruction.

2.4.2. Microcode Operand

The operand_st specifies the source of an operand field for the execution unit. An operand is either from the Micro Store or the ITU. An operand from the ITU is obtained from the operand field of a G-code instruction. It is fetched along with the corresponding

G-code instruction from the Instruction Store and is emitted for all the microcodes of the corresponding G-code-to-microcode expansion sequence. This allows the operand to be program dependent and compiler generated. An operand from the Micro Store is fetched along with the corresponding microinstruction. It is designed for a particular microinstruction of a G-code-to-microcode expansion sequence.

3. TIMING

The IFTU Clock is a two-phase non-overlapping symmetric clock. For each clock cycle, four triggering points are defined. They are ph1S, ph1E, ph2S, and ph2E as shown in Figure 3.1 where ph1S and ph2S correspond to the leading edges of phase 1 and phase 2 of the IFTU clock and ph1E and ph2E are defined to give finer granularity for ordering signals in simulation. The IFTU Timing is, therefore, described in terms of these triggering points.

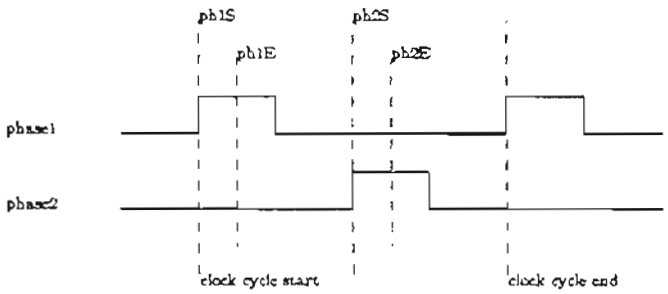


Figure 3.1 The IFTU clock.

Table 3.1 shows the timing of the input signals/data and output signals/data of the IFTU pipeline stages and the PCU. It takes at least three cycles and the memory fetch cycle time for the IFTU to fetch a G-code and deliver the first translated microinstruction.

	IFB Control	IFB Buffers	IFA Control	LQ
input signals/data valid at	ph1E	ph2S	ph1S	ph1E
output signals/data valid at	ph2S	ph1S	ph1E	ph2S

	IQ	ITU Control	Micro Store	PCU
input signals/data valid at	ph2S	ph1S	ph2S	ph1S/ph2S
output signals/data valid at	ph1S	ph2S	ph1S	ph1S

note: signals stay valid for a clock cycle duration.

Table 3.1 The timing of the interface signals/data of each IFTU pipeline stages and the PCU.

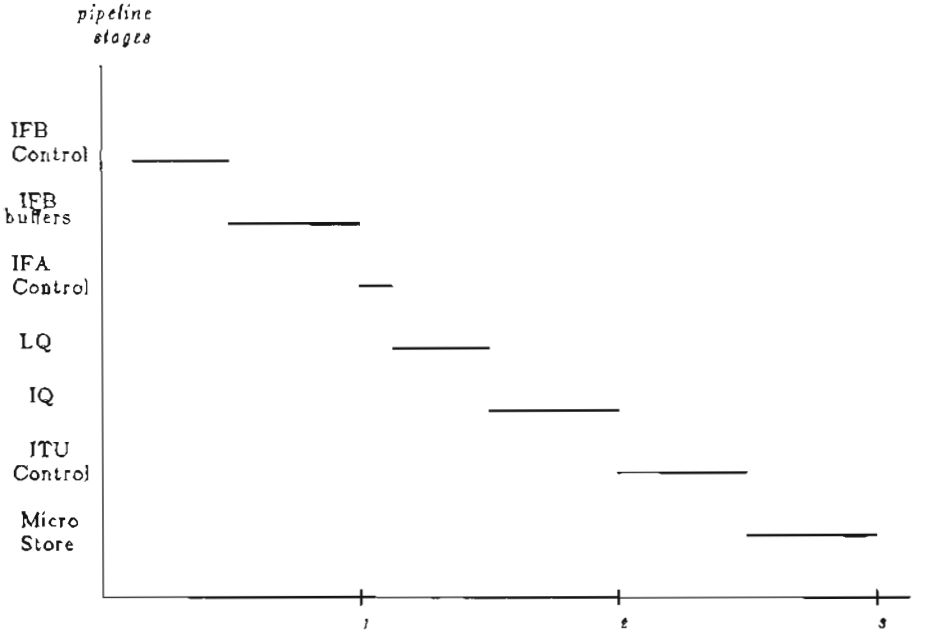


Figure 3.2 IFTU Timing

3.1. Communication via the G-Bus

Table 3.3 contains timing specifications for information exchanged between units via the G-Bus. For example, when the P-stack (one of the execution units) is ready to send or receive a PC, the PCU first sends a request (load_PC/copy_PC) to the IFA. The specifications of the load_PC/copy_PC along with other interface signals/data are listed in Table 3.2. The PC sent by the P-stack will be ready on the G-Bus at the subsequent clock cycle. Meanwhile the IFA is in the state of waiting for the load_PC/copy_PC signal. Once the signal is received, the IFA instructs the IFB to either pick up the PC from, or put the current active PC onto the G-Bus at the next clock cycle.

signals/data	sender	receiver	valid at	description
Operand	ITU	PCU	ph2S	instruction operand from the ITU
ljmp_tkn	PCU	ITU	ph1S	local conditional jump taken
ljmp_not_tkn	PCU	ITU	ph1S	local conditional jump not taken
next_tkn	PCU	ITU	ph1S	PCU request for the next microinstruction
litq_rdy	LQ	PCU	ph2S	Literals Queue is not empty
next_lit	PCU	LQ	ph1S	PCU request for a literal
load_PC	PCU	IFA	ph1S	load the content of G-Bus onto a PC and an FC
copy_PC	PCU	IFA	ph1S	copy the active PC onto the G-Bus
jump_tkn	PCU	IFA	ph1S	G-code conditional jump taken
jump_not_tkn	PCU	IFA	ph1S	G-code conditional jump not taken
microcode	Micro Store	PCU	ph1S	a microinstruction from the Micro Store

note: data/signals stay valid for a clock cycle duration

Table 3.2. The interface signals/data between the IFTU and the PCU.

When the P-stack is ready for a literal, the PCU sends the next_lit request to the Literals Queue. Upon receiving the next_lit signal, the literal at the front of the queue will be put onto the G-Bus. The P-stack can then pick up the literal at the next clock cycle.

triggering signal	data being put on the G-Bus	data sender	data receiver	data valid at
copy_PC	PC	IFB Control	P-stack	ph2S
load_PC	PC	P-stack	IFB Control	ph1S
next_lit	Literal	LQ	P-stack	ph2S

note: data on the G-Bus stay valid for a clock cycle duration.

Table 3.3. The communication between the IFTU and the PCU via the G-Bus.

4. IMPLEMENTATION

4.1. The G-processor Micro Simulator

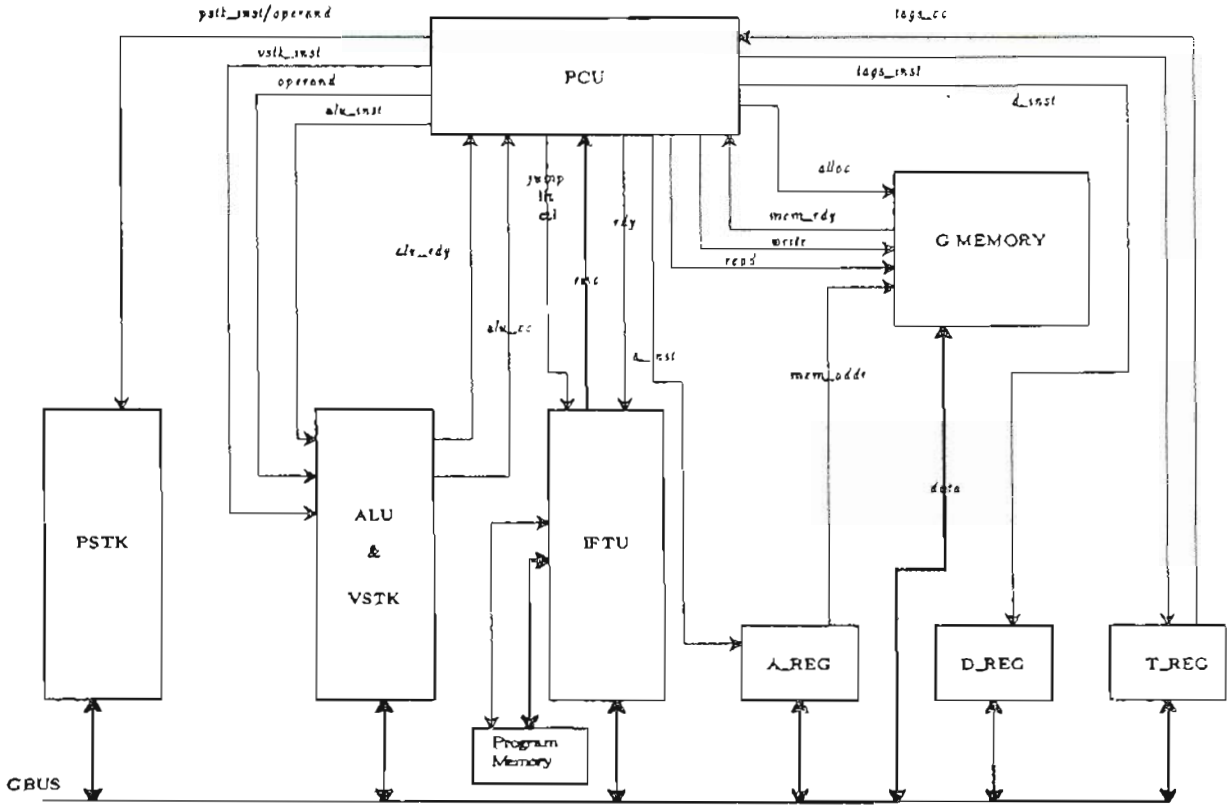


Figure 4.1 The G-processor microsimulator

The IFTU microsimulator has been implemented and integrated with a microsimulator of the execution unit to form the G-processor microsimulator as shown in Figure 4.1. A simulated Graph Memory and Instruction Store are also built. The execution unit consists of a Processor Control Unit (PCU) and the following functional units

1.) the P-stack, which holds pointers of the expression graphs.

2.) the ALU and V-stack, where algebraic functions are performed.

3.) the A (address) register,

4.) the T (tag) register.

The PCU dispatches a full set of control signals to each functional unit from an incoming microinstruction. The execution of a previous microinstruction can be overlapped with the next microinstruction as long as there is no conflict in resource requirement.

The G-processor microsimulator is coded in the ISP language of the N.2 simulation system [N2I84], which runs on the Vax 11/780 under Unix 4.3.

4.1.1. Instruction Store, Micro Store, and Graph Memory

The Instruction Store, the Micro Store and the Graph Memory are generated by the metaMicro assembler [N2M85] and Linking Loader [N2L85] tools of N.2 as simulated memories and are bound with the ISP defined memories by the initial declaration of the N.2 Ecologist [N2E85]. Building an N.2 simulated memory requires three files. They are : a metaMicro assembler description file in which the assembler instructions are defined, a linking loader file and an assembler source code file. The format of G-code instructions is specified in the metaMicro description file of the Instruction Store. The microcode instruction format is specified in the metaMicro description file of the Micro Store. The format of the graph memory is specified in the metaMicro description file of the Graph Memory. An IFTU's input program is used as an Instruction Store assembler source code file. Microcode expansion sequences are specified in Micro Store assembler source code file. The initial configuration of the Graph Memory is specified in the the Graph Memory assembler source code file.

4.1.2. G-code Instruction Type Table

The instruction decoding function performed by the IFA is implemented as a table look up in the IFTU microsimulator. A G-code instruction type table has been built as a metaMicro simulation memory. The opcode of a G-code instruction is used as the key to look up the table directly for the G-code instruction's type.

4.2. Run Time System

The steps of building a run time system are: 1.) Build the Instruction Store with the input program to be executed, 2.) Build the Graph Memory with the initial state of the G-Machine. 3.) Make sure that the Micro Store and the G-code instruction type table are present. 4.) Run the Makefile to build a G-processor microsimulator run time system.

The *initial* signal sets all the IFTU processes to their initial states and enables IFB buffer 0 for prefetching, starting from location zero of the Instruction Store. The system clock, which follows the initial signal, starts the execution of the G-Machine simulator.

5. PERFORMANCE

IFTU performance has been investigated by running four test programs through the G-processor microsimulator.

5.1. Measurement of the IFTU Performance

The measure of IFTU performance is PCU utilization defined as

$$PCU\ utilization = 1 - \frac{the\ number\ of\ PCU\ NOPs\ introduced\ by\ the\ IFTU}{the\ total\ number\ of\ execution\ cycles} \quad (5.1)$$

The IFTU generates a microinstruction according to the PCU's request. When the IFTU can not keep up with the PCU's request, it generates a NOP microinstruction instead. The total number of execution cycles is obtained by running a test program on the G-processor microsimulator up to a desired break point. The number of PCU NOPs is measured as the number of cycles that the PCU spends executing NOPs during the execution of the test program.

5.2. The Average Execution Unit Bandwidth

This section describes the relation between the average execution unit bandwidth and the memory fetch bandwidth. The average execution unit bandwidth is defined as the number of microinstructions that can be processed by the execution unit in a clock cycle. The average execution unit bandwidth has been measured to lie between 1/1.26 to 1/1.72 microinstructions/cycle [Kie87A] on various benchmark programs. Since the IFTU is designed as a pipeline, the amount of time it takes to execute microcode expanded from a G-code execution sequence plus NOPs (PCU NOPs) introduced by the IFTU can not be less than the amount of time required for the G-code sequence to be fetched from the Instruction Store.

$$(PCU\ NOPs + total\ microcode)/ExeBw \geq total\ G{-}code\ words/MemBw \qquad (5.2)$$

where

MemBw : memory fetch bandwidth, G-code words/cycle.

ExeBw : the average execution unit bandwidth, microinstructions/cycle.

Total G-code words consist of useful and not-useful G-code. Useful G-code instructions are those actually executed. There are intervals of time during the execution of the microcode expansion of EVAL or RET when there is no use in fetching G-code. During these time intervals it is certain that a control jump will be made, but the jump address is not yet available. These time intervals should be subtracted from the time available for fetching G-code. Not-useful G-code words are G-code words fetched during the above-mentioned time intervals and G-code words fetched for conditional jump instructions along not taken branches. After eliminating the above-mentioned time intervals and not-useful G-code words fetched from equation 5.2, it is reduced to:

$$(PCU\ NOPs + effective\ microcode)/ExeBw > effective\ G-code/MemBw \quad (5.3)$$

where

effective microcode : total microcode - EVAL or RET expansions.

effective G-code : useful G-code words fetched.

If the execution unit were running at its full speed, no PCU NOPs would be introduced by the IFTU.

$$effective\ microcode/ExeBw > effective\ G-code/MemBw \quad (5.4)$$

$$ExeBw < (effective\ microcode/effective\ G-code)\ MemBw \quad (5.5)$$

The PCU utilization predicted by Equation 5.4 will be compared with simulation results in section 5.4.1.

5.3. Test Programs

The test suite consists of the Iterated Jump program, the Nested Jump program, the strict Linfib program, and the From program. The LML codes and the compiled G-codes of the test programs are listed in Appendix A. The four test programs were chosen to exercise the instruction pipeline in as many efficient ways as possible. Table 5.1 lists *effective microcode/effective G-code* ratios of the test programs. The G-code execution profiles of the test programs are listed in Table 5.2. In the G-code execution profiles, G-codes are grouped into instruction types as described in section 2.3.1.2. The G-code execution profiles help in analyzing the performance of the test programs. They provide information in determining how frequently prefetching is enabled on both instruction streams.

	From	strict Linfib	Iterated/Nested Jump
effective microcode/effective G-code ratio	5.31	2.2	2.3

Table 5.1 The effective microcode/effective G-code ratios of the test programs.

	From	strict Linfib	Iterated/Nested Jump
single operand	0.675	0.7338	0.638
short literal	0.025	0.133	0.145
long literal	0.15	0.027	0.029
unconditional jump	0	0	0
conditional jump	0	0.0885	0.145
callglobfun	0	0.00885	0
return	0.025	0.00885	0.0145
eval	0.125	0	0.029

Table 5.2 The G-code execution profiles of the test programs.

The *Iterated Jump* program consists of a sequence of G-code conditional jump instructions, which are all resolved as jump taken, causing the IFTU pipe to be flushed frequently.

G-code	Words taken from Instruction Store	microcode sequence
GET_FST	1	iCOPYP iMOV_P_A iREADG_V
GET_BYTE	1	iMOV_L_V
SUB	1	iCMP iADDC
MOVE (0)	1	iMOVE_V
JNOT_NEG	2	iJ_NOT_ZERO
MOVE (0)	1	iMOVE_V

Table 5.3 G-codes executed between two conditional jump instructions in the Iterated Jump program.

Table 5.3. lists the G-codes executed in between any two conditional jump instructions and the corresponding microcode sequences. Simulation showed that it took the PCU 14 cycles to execute these microcode sequences. If it takes the Instruction Store one cycle to fetch a word of G-code, there are 7 extra cycles available to the IFB for prefetching. If the memory fetch cycle time for a word of G-code were greater than one cycle, then no excess memory bandwidth would be available for prefetching.

The *Nested Jump* program is similar to the Iterated Jump program except that the jump sequence is nested, and the conditional jump is never taken, thus the IFTU pipe is never flushed. Both the Nested Jump and the Iterated Jump programs have the same execution profile as shown in Table 5.2. The Nested/Iterated Jump execution sequence contains 14.5% conditional jump instructions and the *effective microcode/effective G-code* ratio is 2.3.

The *strict Linfib* is the strict version of the Linfib program which computes a Fibonacci sequence with some accumulator variables and a tail recursive linear-time algo-

rithm. It is shown in Table 5.2 that the strict Linfib execution sequence contains 8.85% conditional jump instructions which require prefetching along both the active and the alternate instruction streams. Consequently to maintain the performance of the active instruction stream it requires the memory fetch bandwidth to be doubled. The *effective microcode/effective G-code* ratio of the strict Linfib is 2.2 which is the lowest among all the test programs. Therefore, the strict Linfib is expected to have a poorer prefetching capability and be sensitive to the changes in memory fetch bandwidth comparing with other test programs.

The *From* program generates an infinite list of integers by lazy evaluation. There is no G-code instruction in the From execution sequence which will cause two instruction streams to be enabled for prefetching. The *effective microcode/effective G-code* ratio of the From program is 5.31 which is the highest among all the test programs. Therefore, the From program is expected to have the best prefetching capability and be less sensitive to the memory fetch bandwidth changes.

5.4. Tests Results

Parameters identified in the IFTU are :

1.) memory fetch time.

2.) memory fetch bandwidth.

3.) IQ depth.

4.) prefetch buffer depth.

5.) LQ depth.

6.) prefetch priority.

Tests have been designed to determine the effect of each IFTU parameter upon the IFTU performance. The test strategy is to first vary one parameter at a time while making all others as inert as possible.

5.4.1. The Effects of Memory Fetch Cycle Time

Two sets of tests have been set up to evaluate the effects of the memory fetch latency upon the IFTU performance. Large LQ, IQ and prefetch buffer sizes are used so that their effects on the IFTU performance are not significant. Memory fetch sizes in both tests are 16 bits. One set of tests uses round-robin prefetch scheduling with the initial fetch priority to the alternate buffer. The other set of tests uses round-robin prefetch scheduling with initial fetch priority to the active prefetch buffer. The test results are respectively summarized in Tables 5.4 and 5.5. Both tables show that as the memory fetch latency increases, the PCU utilization of every test program decreases.

The From program shows the least performance degradation with respect to the increase of the memory fetch cycle time. This is because the From program has a relatively higher *effective microcode/effective G-code* ratio allowing more time for prefetching and only one instruction stream is required to run the program. Because only one instruction stream is required in the execution of the From program, the different prefetch priority

choices have no effect on its test results. The effects of the difference in the prefetch algorithms on other test programs will be presented and discussed in the next section.

PCU utilization versus memory fetch cycle time IQ depth = 11, pre_fetch buffer depth = 4 LQ depth = 4, memory fetch size = 16 bits round robin prefetch scheduling with priority to the alternate buffer .						
	memory fetch cycle time					
	1 cycle	2 cycle	3 cycle	4 cycle	5 cycle	6 cycle
From program						
total execution cycles	383	394	409	440	481	525
PCU nops	23	30	40	70	97	122
PCU utilization	0.9399	0.9239	0.9022	0.8409	0.7983	0.7676
Iterated Jump program (jump taken)						
total execution cycles	321	332	383	464	548	633
PCU nops	32	39	86	163	253	344
PCU utilization	0.9003	0.8825	0.7755	0.6487	0.5383	0.4566
Nested Jump program (jump not taken)						
total execution cycles	288	321	386	468	553	639
PCU nops	14	43	104	182	272	363
PCU utilization	0.9514	0.8660	0.7305	0.6111	0.5081	0.4319
strict Linfib program						
total execution cycles	430	441	481	626	771	916
PCU nops	31	41	89	210	364	513
PCU utilization	0.9279	0.9070	0.8150	0.6645	0.5279	0.4350

Table 5.4 PCU utilization versus memory fetch cycle time

- with prefetch priority to the alternate buffer.

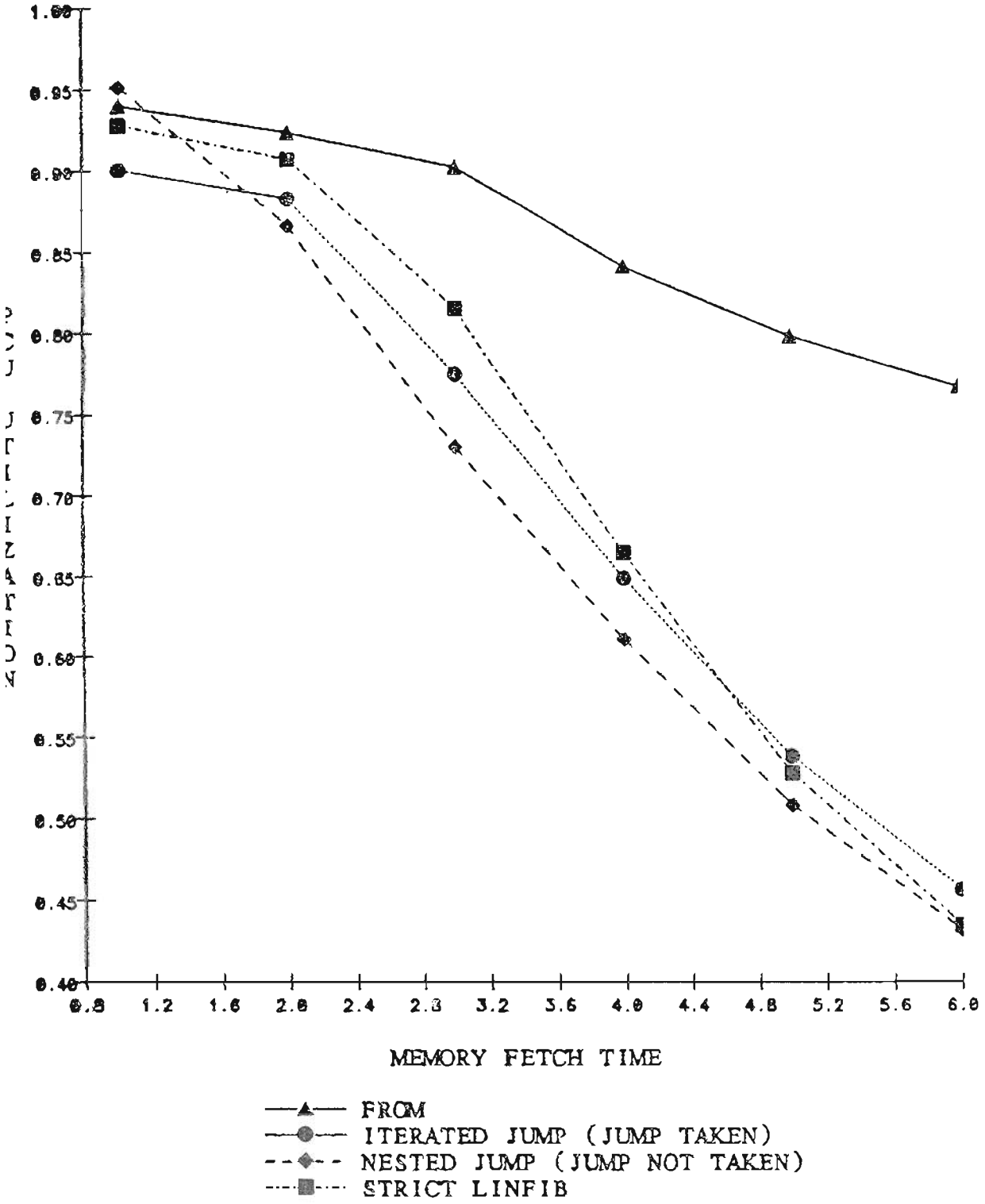


Fig. 5.1 PCU utilization versus memory fetch time - with prefetch priority to the alternate buffer.

PCU utilization versus memory fetch cycle time IQ depth = 11, pre_fetch buffer depth = 4 LQ depth = 4, memory fetch size = 16 bits round robin prefetch scheduling with priority to the active buffer							
		memory fetch cycle time					
	normalized	un-normalized					
	1 cycle	1 cycle	2 cycle	3 cycle	4 cycle	5 cycle	6 cycle
From program							
total execution cycles	383	383	394	409	440	481	525
PCU nops	16	23	30	40	70	97	122
PCU utilization	0.9582	0.9399	0.9239	0.9022	0.8409	0.7983	0.7676
Iterated Jump program (jump taken)							
total execution cycles	170	321	338	410	500	593	678
PCU nops	20	32	45	114	199	298	398
PCU utilization	0.8824	0.9003	0.8669	0.7220	0.602	0.4975	0.4130
Nested Jump program (jump not taken)							
total execution cycles	150	288	306	362	436	513	591
PCU nops	0	14	28	80	150	232	315
PCU utilization	1.0	0.9514	0.9085	0.7790	0.6560	0.5478	0.4670
strict Linfib program							
total execution cycles	390	430	441	505	658	811	964
PCU nops	20	31	41	113	242	404	561
PCU utilization	0.9487	0.9279	0.9070	0.7762	0.6322	0.5018	0.4180
		normalized					
Nested Jump program (jump not taken)							
total execution cycles		150	160	210	280	350	420
PCU nops		0	10	60	130	210	280
PCU utilization		1.0	0.9375	0.7143	0.5357	0.4	0.3333

Table 5.5. PCU utilization versus memory fetch cycle time - II.
(Data listed in the first column and the last row of Table 5.5 have been normalized by subtracting initial bias. The rest of data are not normalized.)

For the Nested and the Iterated Jump programs, microcode expanded in between two conditional jump instructions allows enough time for G-code to be prefetched into both prefetch buffers (refer to section 5.3 Iterated and Nested Jump test programs) when the memory fetch cycle time is one cycle. Therefore, in theory there should be no PCU NOPs introduced by the IFTU when a conditional jump is not taken. However the test results of

the Nested Jump program listed in Table 5.5 show that when the memory fetch cycle time is one cycle, the PCU executes 14 NOPs instead of zero NOP. The discrepancy in the value of PCU NOPs is accounted for by the code sequence which initializes the program and the codes, such as EVAL and RET, which start and end the program. In order to determine the cause of those PCU NOPs of the Nested Jump program and the effect of initialization code of other programs, test results have been normalized by subtracting the initial bias from test results. The initial bias is obtained by extrapolating the values of PCU NOPs, obtained by running Nested Jump program with 10, 20, 30, and 40 conditional jumps, to obtain estimated time for a trivial program (zero conditional jump). The normalized test results of the Nested Jump program at different memory fetch cycle time cycles are listed in Table 5.5. Also included in Table 5.5 are the normalized test results of other test programs at one cycle memory fetch cycle time. As can be seen in Table 5.5 that there is no PCU NOP after normalization for the Nested Jump program for one cycle memory fetch cycle time. The above-mentioned 14 PCU NOPs is caused by the initial bias.

The normalized PCU utilizations for the Nested Jump Program shown in Table 5.5 are plotted in Figure 5.3 as a function of memory fetch cycle time. Also included in Figure 5.3 is the predicted PCU utilization derived from Equation 5.4 as a function of memory fetch cycle time for the average execution unit bandwidth of 1/1.26 and 1/1.72 microinstructions/cycle. It can be seen in Figure 5.3 that the simulation data are consistent with the predicted PCU utilizations when the average PCU bandwidth is in between 1/1.26 and 1/1.72 microinstructions/cycle.

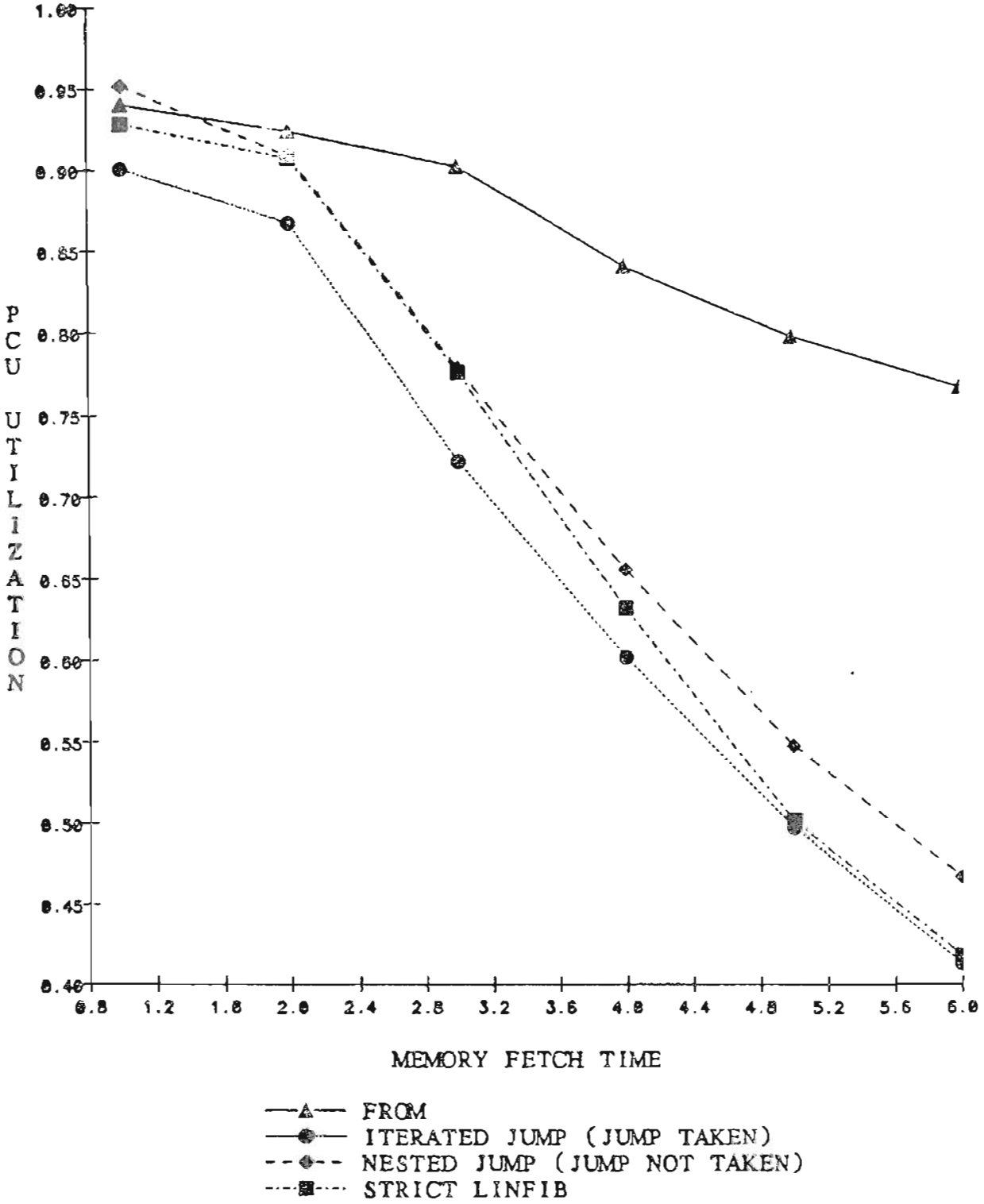


Fig. 5.2 PCU utilization versus memory fetch time - with prefetch priority to the active buffer.

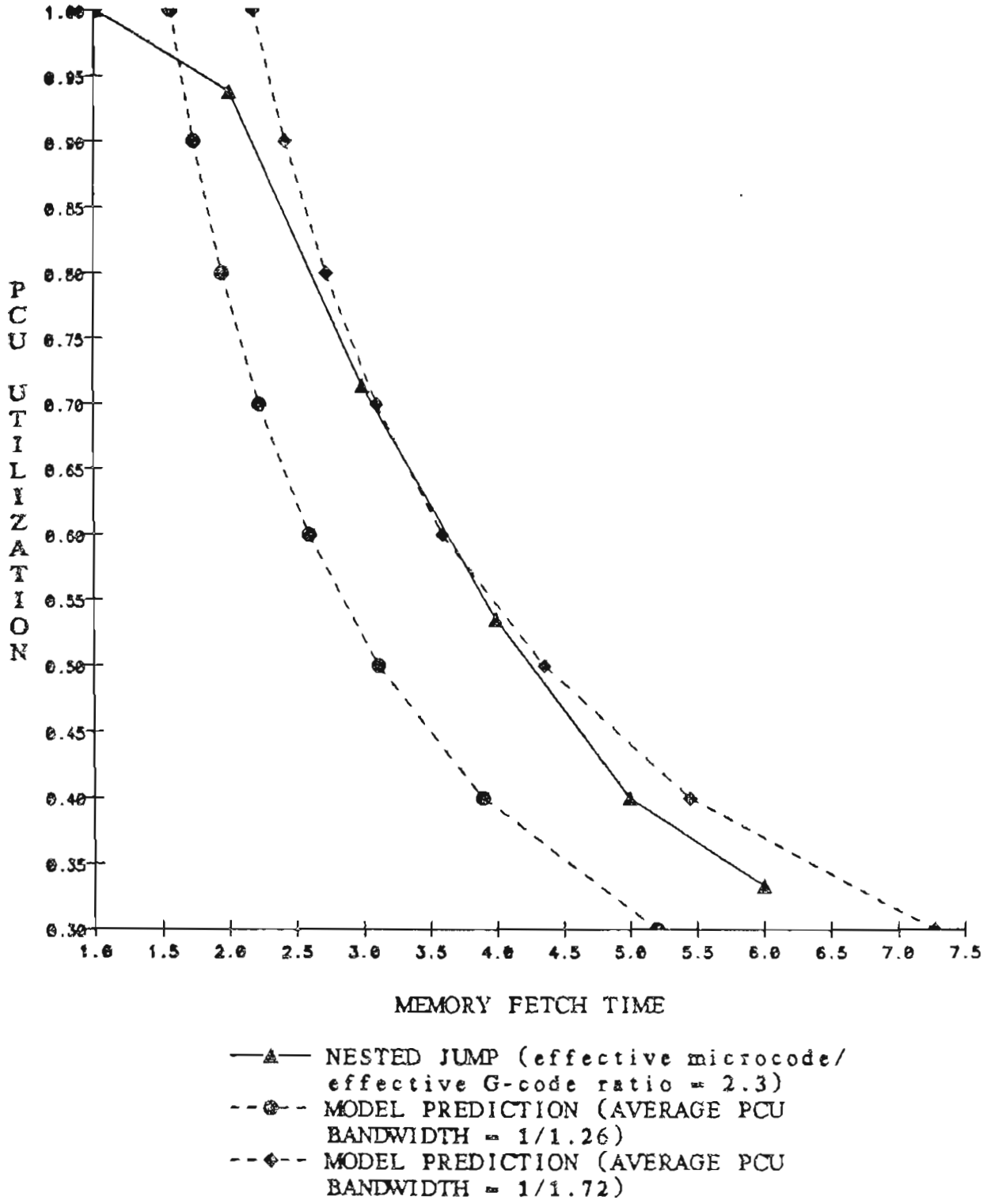


Fig. 5.3 Effects of memory fetch time on PCU utilization.

5.4.2. The Effects of the Prefetch Algorithm

Table 5.4. shows that when the memory fetch cycle time is greater than or equal to two cycles, the Iterated Jump program, which causes the IFTU pipe to be flushed frequently, has a better PCU utilization than the Nested Jump program which does not cause the IFTU pipe to be flushed at all. Since simultaneous prefetching can not be done for both the active and the alternate instruction streams, the memory fetch bandwidth for each stream is reduced by 50% when the alternate buffer is enabled. When there is little excess bandwidth available to dedicate for prefetching the alternate/normal stream, the priority given to one stream or the other can have more important effects than the pipe flush latency. The prefetch algorithm used in Table 5.4 is based on round-robin scheduling with the initial fetch priority assigned to the alternate buffer. When an alternate buffer is enabled for prefetching, the next memory fetch will be initiated for the alternate buffer. Table 5.5 shows that when the initial fetch priority is assigned to the active buffer, the PCU utilization for the Nested Jump program which does not cause the IFTU pipe to be flushed is improved.

It is also shown in Table 5.4 that if the memory fetch cycle time is one cycle, the Nested Jump program has better PCU utilization than the Iterated Jump program. Since there is excess memory bandwidth for prefetching, the effects of the pipe flush latency become significant.

5.4.3. The Effects of the Memory Fetch Bandwidth

The PCU utilization as a function of memory fetch bandwidth for all the test programs are summarized in Table 5.6. The first column of Table 5.6. lists the IFTU performance when the memory fetch bandwidth is 32 bits/cycle. The other three columns list the IFTU performance when the memory fetch bandwidth are 32 bits/2 cycle, 48 bits/3 cycles, and 64 bits/4 cycles. Table 5.6. shows that with a bigger memory fetch size and a longer memory latency, similar PCU utilization can be achieved as with a faster memory fetch cycle time and a smaller memory fetch size.

PCU utilization versus memory fetch bandwidth
IQ depth = 11, prefetch buffer depth = 4,
LQ depth = 4
round robin prefetch scheduling with priority to active buffer

	memory fetch size / memory fetch cycle time			
	32 bit/1 cycle	32 bit/2 cycle	48 bit/3 cycle	64 bit/4 cycle
From program				
total execution cycles	383	386	391	394
PCU nops	23	26	31	34
PCU utilization	0.9399	0.9326	0.9207	0.9137
Iterated Jump program (jump taken)				
total execution cycles	321	323	325	328
PCU nops	32	34	36	39
PCU utilization	0.9003	0.8947	0.8892	0.8811
Nested Jump program (jump not taken)				
total execution cycles	288	290	292	294
PCU nops	14	16	18	20
PCU utilization	0.9514	0.9448	0.9384	0.9320
strict Linfib program				
total execution cycles	430	434	438	442
PCU nops	31	35	39	43
PCU utilization	0.9279	0.9194	0.911	0.9027

Table 5.6. PCU utilization versus memory fetch bandwidth.

5.4.4. The Effect of the IQ Depth

Table 5.7 summarizes the test results of PCU utilizations as a function of the IQ depth. The extra cost if the IQ is not included in the IFTU design is proportional to the number of G-code instructions being executed. For example, there were 50 G-code instructions in an execution sequence which translates into 100 microinstructions. It took 1 cycle for each microinstruction to be executed, and the IFTU latency was 3 cycles. The total execution time would be 102 cycles. If the cost for each G-code in the absence of the IQ were two extra cycles, then total execution time would nearly double to 202 cycles. This would degrade total performance by 50%. With no IQ in the IFTU design, Table 5.7. shows that the From program has the least performance degradation among all the test programs. This is because the From program has a relatively high *effective microcode/effective G-codes* ratio. Since the rest of the test programs have relatively low *effective microcode/effective G-code* ratios, they suffer bigger performance degradations when the IQ is not in the IFTU design. There is also a noticeable performance degradation for all the test programs when the depth of the IQ decreases from two to one. The relative advantage gained by extending the IQ depth to be greater than two is negligible for all the test programs. Therefore, an IQ depth of two is considered an optimal value.

PCU utilization versus IQ depth prefetch buffer depth = 4, LQ depth = 4 memory fetch size = 16 bits, memory fetch cycle time = 1 clock cycle round robin prefetch scheduling with priority to active buffer					
	IQ depth				
	0 deep	1 deep	2 deep	3 deep	11 deep
From program					
total execution cycles	433	394	383	383	383
PCU nops	78	38	23	23	23
PCU utilization	0.8199	0.9036	0.9399	0.9399	0.9399
Iterated Jump program (jump taken)					
total execution cycles	386	335	321	321	321
PCU nops	110	46	32	32	32
PCU utilization	0.7150	0.8627	0.9003	0.9003	0.9003
Nested Jump program (jump not taken)					
total execution cycles	364	309	288	288	288
PCU nops	102	35	14	14	14
PCU utilization	0.7198	0.8667	0.9514	0.9514	0.9514
strict Linfib program					
total execution cycles	554	477	430	430	430
PCU nops	180	77	31	31	31
PCU utilization	0.6751	0.8386	0.9279	0.9279	0.9279

Table 5.7. PCU utilization versus IQ depth.

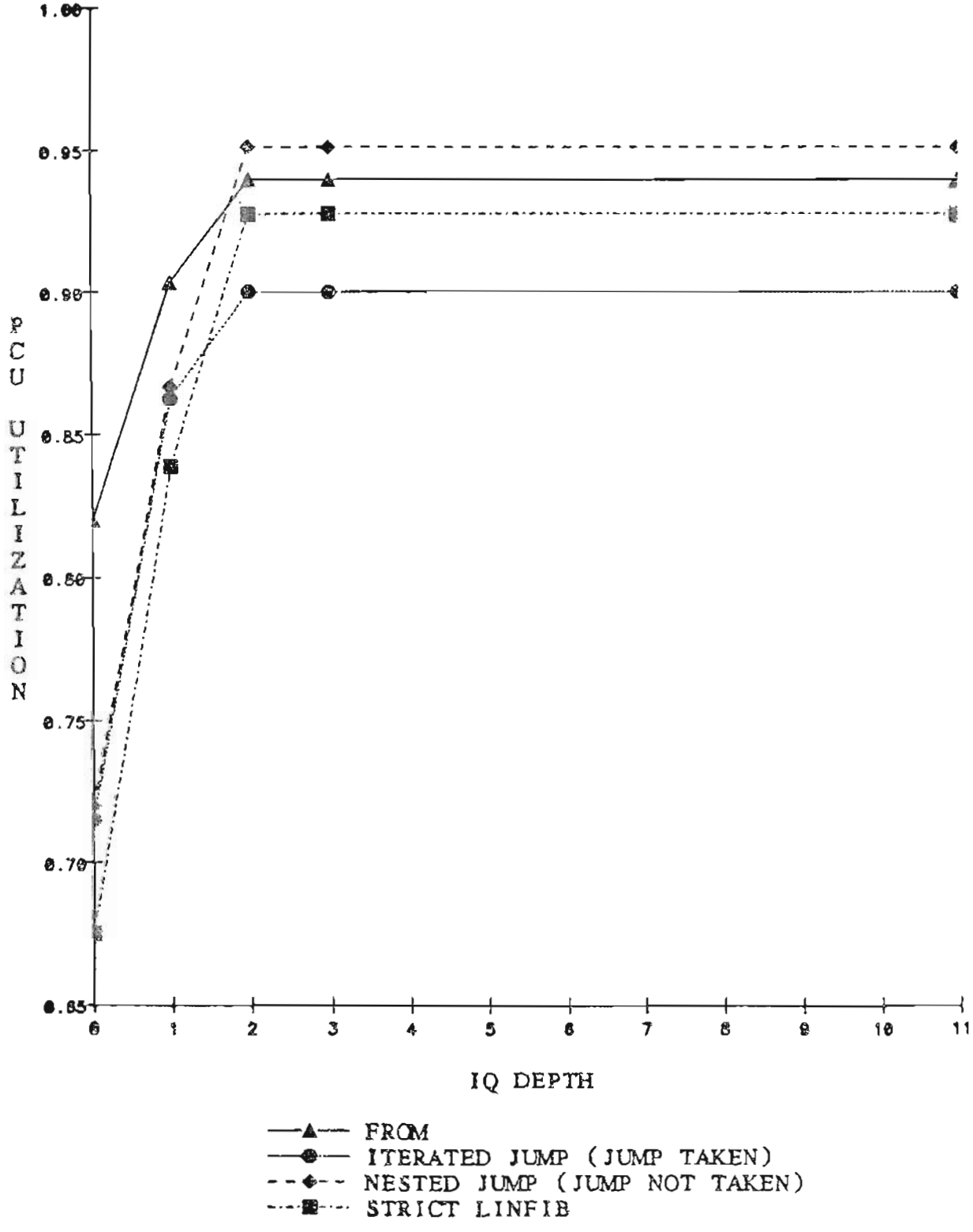


Fig. 5.4 PCU utilization versus IQ depth.

5.4.5. The Effects of the Prefetch Buffer Depth

Two sets of tests have been run to study the effects of the prefetch buffer depth on the PCU utilization. One set of the tests has a large IQ so that the effect of the IQ depth is not significant. The other set of the tests has no IQ in the IFTU. Their test results are respectively summarized in Tables 5.8 and 5.9. Both tables show that the IFTU design with a two word deep prefetch buffer delivers close to optimal PCU utilization for all the test programs. When the prefetch buffer depth is only one and the buffer is full, the IFB has to wait until the buffer becomes empty before it can initiate the next memory fetch. Consequently some memory bandwidth is wasted.

PCU utilization versus Prefetch buffer depth IQ depth = 11, LQ depth = 4, memory fetch size = 16 bits, memory fetch cycle time = 1 clock cycle round robin prefetch scheduling with priority to active buffer				
	prefetch buffer depth			
	1 deep	2 deep	3 deep	4 deep
From program				
total execution cycles	394	383	383	383
PCU nops	30	23	23	23
PCU utilization	0.9239	0.9399	0.9399	0.9399
Iterated Jump program (jump taken)				
total execution cycles	327	321	321	321
PCU nops	34	32	32	32
PCU utilization	0.8960	0.9003	0.9003	0.9003
Nested Jump program (jump not taken)				
total execution cycles	296	288	288	288
PCU nops	18	14	14	14
PCU utilization	0.9392	0.9514	0.9514	0.9514
strict Linfib program				
total execution cycles	442	430	430	430
PCU nops	39	31	31	31
PCU utilization	0.9118	0.9279	0.9279	0.9279

Table 5.8. PCU utilization versus Prefetch buffer depth.

PCU utilization versus prefetch buffer depth LQ depth = 4, IQ depth = 0, memory fetch size = 16 bits, memory fetch cycle time = 1 clock cycle round robin prefetch scheduling with priority to active buffer			
	prefetch buffer depth		
	1 deep	2 deep	4 deep
From program			
total execution cycles	448	433	433
PCU nops	85	78	78
PCU utilization	0.8103	0.8199	0.8199
Iterated Jump program (jump taken)			
total execution cycles	392	386	386
PCU nops	112	110	110
PCU utilization	0.7143	0.7150	0.7150
Nested Jump program (jump not taken)			
total execution cycles	370	364	364
PCU nops	104	102	102
PCU utilization	0.7189	0.7198	0.7198
strict Linfib program			
total execution cycles	565	554	554
PCU nops	188	180	180
PCU utilization	0.6673	0.6751	0.6751

Table 5.9 PCU utilization versus prefetch buffer depth - with no IQ.

5.4.6. The Effects of No IQ and One Deep Prefetch Buffer

The test results of PCU utilizations as a function of memory fetch cycle time for the IFTU design with no IQ and one deep prefetch buffer are summarized in Table 5.10. Table 5.10 shows that PCU utilization degradations range from 12% to 28% comparing with respective data in Table 5.5. The strict Linfib program shows the worst PCU utilization degradation.

PCU utilization versus memory fetch cycle time LQ depth = 4, IQ depth = 0, prefetch buffer depth = 1, memory fetch size = 16 bits round robin prefetch scheduling with priority to active buffer				
	memory fetch cycle time			
	1 cycle	2 cycle	3 cycle	4 cycle
From program				
total execution cycles	448	468	497	531
PCU nops	85	97	116	140
PCU utilization	0.8103	0.7927	0.7666	0.7363
Iterated Jump program (jump taken)				
total execution cycles	392	412	467	554
PCU nops	112	128	179	262
PCU utilization	0.7143	0.6893	0.6167	0.5271
Nested Jump program (jump not taken)				
total execution cycles	370	397	464	552
PCU nops	104	127	190	274
PCU utilization	0.7189	0.6801	0.5905	0.5036
strict Linfib program				
total execution cycles	565	588	662	771
PCU nops	188	204	263	366
PCU utilization	0.6673	0.6531	0.6027	0.5253

Table 5.10 PCU utilization versus memory fetch cycle time - with no IQ and one deep prefetch buffer.

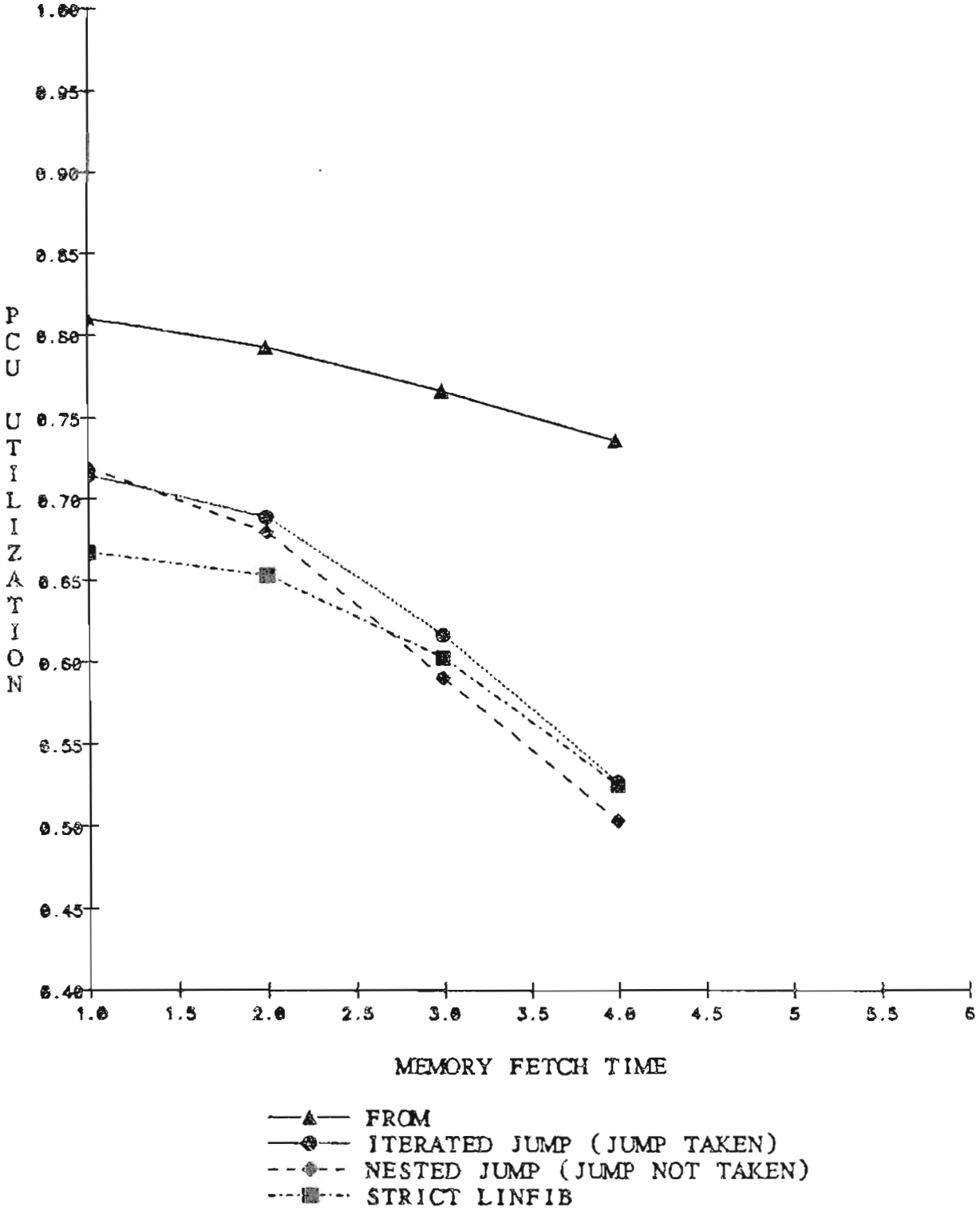


Fig. 5.5 PCU utilization versus memory fetch time - with no IQ and one deep prefetch buffer.

5.4.7. A Close-to-Optimal Design

Test results presented so far show that a close to optimal IQ depth, and prefetch buffer depth are two. Since there are rarely more than two consecutive literal instructions in a G-code program, the LQ depth of two is considered to be sufficient in an IFTU's design. With IQ and prefetch buffers depths of two, and four deep LQ, the PCU utilization as a function of memory fetch bandwidth is summarized in Table 5.11. This table shows that PCU utilization varies from 88.3% to 95%. It also shows that wider memory fetch size can compensate for longer memory fetch cycle time.

PCU utilization versus memory fetch bandwidth IQ depth = 2, prefetch buffer depth = 2, LQ depth = 4 round robin prefetch scheduling with priority to active buffer				
	memory fetch bandwidth			
	32 bit/1 cycle	32 bit/2 cycle	48 bit/3 cycle	64 bit/4 cycle
From program				
total execution cycles	383	386	391	394
PCU nops	23	26	31	34
PCU utilization	0.9399	0.9326	0.9207	0.9137
Iterated Jump program (jump taken)				
total execution cycles	321	323	325	327
PCU nops	32	34	36	38
PCU utilization	0.9003	0.8947	0.8892	0.8838
Nested Jump program (jump not taken)				
total execution cycles	288	290	292	294
PCU nops	14	16	18	20
PCU utilization	0.9514	0.9448	0.9384	0.9320
strict Linfib program				
total execution cycles	430	434	438	442
PCU nops	31	35	39	43
PCU utilization	0.9279	0.9194	0.911	0.9027

Table 5.11. PCU utilization versus memory fetch bandwidth
- with a close-to-optimal IFTU design.

5.5. Factors that Affect the IFTU Performance - Discussion

The maximum performance that the IFTU can deliver is limited by certain aspects of its implementation; these limitations are intrinsic. However the performance degradation due to some of these limitations may not be visible, if the program executed by the IFTU has certain characteristics such as a high *effective microcode/effective G-code* ratio. The factors that affect the IFTU performance are discussed below:

5.5.1. Memory Fetch Bandwidth

The IFTU throughput can be restricted by the memory fetch bandwidth. Simulation results (see Tables 5.4 and 5.5) show that the IFTU performance decreases with decreasing memory fetch bandwidth. Memory fetch bandwidth can be improved by increasing the memory fetch size, or decreasing the memory fetch cycle time. The IFTU reaches its peak performance when the memory bandwidth is one G-code word/G-processor clock cycle.

5.5.2. Prefetching

The IFTU is designed with a prefetching capability. There are two prefetch buffers. One is for the alternate instruction stream and the other is for the normal instruction stream. Prefetching is possible because most G-code instructions translate into more than one microinstructions giving the IFTU added time to prefetch G-code. Simulation results of this study have shown that the IFTU is capable of prefetching G-code into both prefetch buffers without noticeable degradation in the active instruction stream throughput, when the memory fetch bandwidth is one G-code word/G-processor clock cycle.

5.5.3. Pipeline Elasticity

The Instruction Queue (IQ) has been placed between the IFA and the ITU to buffer the assembled G-code instructions from the IFA. This allows G-code instructions to be pre-assembled, giving an optimal usage of the time allowed by the G-code-to-microcode

expansions.

5.5.4. The IFTU Pipeline Latency

The IFTU pipeline latency is defined as the number of cycles it takes the IFTU to generate its first output for an input G-code program. The IFTU pipeline latency, which is equal to the memory fetch latency plus three cycles, is caused by the cumulative delay of the pipeline stages. The latency is one of the intrinsic limitations of the IFTU and is independent of the program being executed.

5.5.5. Buffers

The IFTU contains an IQ, an LQ and two prefetch buffers. Close to optimum IFTU performance is obtained when the depth of the IQ and both prefetch buffers are equal to two. The performance degradation can be as much as 28% if the IQ is not in the IFTU design. The performance degradation resulting from decreasing the buffer depth from two to one is 3.5 % to 9 % for the IQ and 1 % for the prefetch buffers. Since there are no more than two consecutive literal instructions in most programs, it is sufficient to design the LQ to be two deep.

5.5.6. Prefetch Algorithm

The prefetch algorithm is based on the round robin scheduling. It specifies the buffer to which the next memory fetch is initiated when the alternate buffer is first enabled. When there is very little or no excess memory bandwidth for prefetching and both prefetch buffers are enabled, the effects of the initial fetch priority of the prefetch algorithm becomes significant.

5.5.7. Effective Microcode/Effective G-code Ratio

When an input program has a high *effective microcode/effective G-code* ratio, it allows the IFTU more time to prefetch and to assemble G-codes ahead. Consequently, better IFTU performance is obtained.

5.5.8. Conditional Jump Instruction

The IFTU design gives an effective way to handle the G-code conditional jump instruction. When the IFA decodes a conditional jump instruction, it enables the alternate instruction buffer for prefetching. The IFA also ensures that a delayed instruction is assembled before it processes a jump taken or not taken. The delayed instruction keeps the PCU busy while the IFA processes a jump taken or not taken. When a jump not taken signal is received, there is no interruption on the IFTU pipe. When a jump taken signal is received, the IFTU pipeline is flushed and the alternate instruction stream is made the active instruction stream. The number of cycles that the PCU spends executing NOPs due to the IFTU pipe being flushed depends on the number of cycles being masked by the microcode expanded from the delayed G-code instruction. It takes the IFTU one clock cycle to flush the pipe and three cycles plus memory fetch cycle time to fill up the pipe. A delayed G-code instruction which expands into four microinstructions will mask all the latency if there is no cost for memory fetch(es). Although the flush IFTU pipe latency is an intrinsic IFTU limitation, it can be compensated by the program being executed.

5.5.9. Unconditional Jump Instruction

Control transfer of a G-code unconditional jump instruction is within the IFTU. It costs the IFTU two cycles to assemble an unconditional jump instruction and one cycle to perform the control transfer. However, these cycle delays may not be visible to the execution unit because the G-code instruction flow is buffered by the IQ.

5.5.10. RET/RET_INT Instruction

A RET or a RET_INT G-code instruction mandates that the subsequent instruction stream to start with the return address. After the return address is provided, it takes the IFTU four cycles (one cycle to load the FC and three cycles to fill up the IFTU pipeline) and the memory fetch cycle time to generate the next microinstruction. The microinstruction, which enables the return address to be loaded, occurs at the end of the RET/RET_INT expansion sequence. Therefore, very little time can be used to prefetch and assemble G-code instructions ahead. Redesign of the microcode sequence for RET/RET_INT instructions could mask this latency.

5.5.11. Literal Instruction

When the IFA decodes a literal instruction, it puts the literal's operand into the LQ. It may take the IFA multiple cycles to assemble a literal instruction. These latencies may not be visible to the execution unit because they are absorbed by the execution of G-code instructions buffered in the IQ. When the PCU is ready to dispatch a literal instruction, the corresponding literal may be already in the LQ.

5.5.12. Delayed G-code Instruction

When a delayed G-code instruction is not a NOP, the IFTU designed with the delayed instruction may deliver better performance than the IFTU designed without the delayed instruction. Better performance will occur when the PCU is able to execute microcode expanded from the delayed G-code instruction immediately after the execution of the conditional jump instruction. When a delayed G-code instruction is a NOP, the IFTU designed with the delayed instruction may deliver poorer performance than the IFTU designed without the delayed instruction. Poorer performance will occur when the PCU has to execute the delayed NOP instruction even though a jump is resolved as not taken.

5.5.13. Delayed Micro Instruction A delayed microinstruction is executed after a local conditional jump microinstruction. This gives the ITU a clock cycle time to generate the address of the next microinstruction from either the jump taken or not taken path. If the PCU does not execute a delayed microinstruction it will have to wait for a clock cycle. Therefore, an IFTU design incorporating the delayed microinstruction will enhance the IFTU performance.

6. CONCLUSION

An instruction fetch and translation unit has been designed for the G-processor. A microsimulator of the IFTU has been built and integrated with a microsimulator of the G-processor execution unit. The IFTU performance was evaluated by running benchmark programs compiled from the LML compiler on the G-processor microsimulator. The IFTU was designed as a buffered pipeline with G-code prefetching capability. The amount of time available for G-code prefetching was provided by G-code to microcode translation performed by a translation unit. Two prefetch buffers were designed to accommodate normal and jump target instruction streams. Hardware support was provided for literals G-code and for all control transfer instructions.

Simulation results showed that an IFTU designed with buffers (they are an instruction queue, a literals queue and prefetch buffers) of two words deep and with memory fetch bandwidth of one G-code word per clock cycle of the G-processor delivered close-to-optimal performance. Relative peak performance of the IFTU in terms of the PCU utilization was from 90% to 96% from the simulations results. Significant performance degradation was observed if the instruction queue was missing in the IFTU design. The IFTU was able to prefetch G-codes into both prefetch buffers without noticeable performance degradation when memory fetch bandwidth was one G-code word/cycle. In order to achieve the required memory fetch bandwidth, it is suggested to widen memory fetch size or to use a cache. Simulation results showed that by widening memory fetch size, optimal IFTU performance can be achieved with longer memory fetch cycle time.

7. FUTURE WORK

7.1. RET/RET-INT

An improvement can be made in the IFTU latency on a RET or a RET_INT by making the following changes: 1.) Code those micro-instructions, which cause the return address to be sent to the IFTU, at its earliest possible location in a RET/RET_INT expansion sequence. This allows prefetching to start from the return address as early as possible. However, in the present design, the return address is provided at the end of a RET/RET_INT expansion sequences. 2.) Add the micro-instructions, which cause a jump taken signal to be sent to the IFA, at the end of a RET/RET-INT sequence to signal the end of sequence. In the present design, the end of the RET/RET_INT expansion sequence is implied with the receiving of a return address. 3.) It is also necessary to include the design of expecting a jump taken signal for a RET/RET-INT sequence in the IFA. The improvement on a RET/RET-INT is expected to reduce a latency from four cycles plus memory fetch latency to two cycles.

7.2. Delayed G-code Instruction

Implementing a delayed G-code instruction in the IFTU design may or may not improve the IFTU performance. But it requires a more complex IFA design, because:

1.) The IFA has to ensure that a G-code delayed instruction is assembled before it can process a conditional jump taken or not taken.

2.) The IFA has to ensure that a G-code delayed instruction is a valid instruction.

An alternative is to give up the G-code delayed instruction design. In this case the PCU has to wait a cycle after the execution of a conditional jump instruction to

ensure that the IFTU pipe will be flushed if the conditional jump instruction is resolved as a jump taken. This can be done by adding a microcode NOP at the end of a G-code conditional jump expansion sequence. Currently, a G-code conditional jump translates into one microcode conditional jump instruction.

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).

[Aug84]
Augustsson, L., "A Compiler for lazy ML," *Proc. ACM Symp. on LISP and Functional Programming*, Austin, Texas, August 1984.

[Bae80]
Baer Jean-Loup, "Computer Systems Architecture", Computer Science Press Inc. 1980 pp 334-371.

[Ber75]
Berkling, K., Reduction languages for reduction machines, *Proc. IEEE Int. Sympos. on Computer Arch.*, pp 133-140, Jan. 1975.

[Car84]
Cardelli, L., "Compiling a functional language, "*Proc. of 1984 ACM Conf. on Lisp and Functional Programming*, 1984, pp.208-217.

[Chip86]
Billing, K., Chao, P., Dobos, L., Farahmand-nia, S., Hammerstrom, D., Kuo, S L., Nguyen, N., Rankin, L., *The G-Machine IFTU Chip Microarchitecture Specification* Dept. of Computer Sc. & Eng., Oregon Graduate Center, June 1986.

[Ele87]
Manuel Tom "Getting mainframe power out of a CISC supermicro," *Electronics*, September 3, 1987, pp 66-68.

[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

[Hai79]
Hailpern, B. T. and Hitson, B. L., "S-1 Architecture Manual", Tech. report STAN-CS-79-715, Computer Systems Lab., Stanford University, 1979.

[Hos88]
Bill Hostmann, "An Examination of Designs for the Instruction Pipeline of the G-machine", M.S. Thesis, Dept. of Computer Sc. & Eng., Oregon Graduate Center, to be published, 1988.

[Ibb82]
Ibbett, R. N., The architecture of High Performance Computers, (Macmillan Press 1982), 1982, Springer Verlag.

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

[Joh84]
Johnsson, T., "Efficient compilation of lazy evaluation," *Proc. of 1984 ACM SIGPLAN Notices Conf. on Compiler Constr*, June, 1984

[Kie85]
Kieburtz, R. B., "The G-machine: a fast, graph-reduction evaluator, "*Proc. of IFIP Conf. on Functional Prog. Lang. and Computer Arch.*,
Nancy, 1985.

[Kie86]
Kieburtz, R. B., *G-machine architecture handbook*, Dept. of Computer Sc. & Eng., Oregon Graduate Center, 1986.

[Kie87A]
Kieburtz, R. B., "Performance evaluation of a G-machine implementation, in *Graphic Reduction*, R. B. Keller and J. H. Fasel (ed.), Springer-Verlag, LNCS series, 1987.

[Kie87B]
Kieburtz, R. B., "A RISC Architecture for Symbolic Computation", Tech. Rep. CS/E 87-001, Oregon Graduate Center, Beaverton, OR, 1987 pp. 17-19.

[LMO84]
Lampson, B. W., McDaniel, G. and Ornstein, S. M., An instruction fetch unit for a high preformance personal computer, *IEEE Trans. on Computers C-33*, No. 8 (Aug. 1984), pp. 712-730.

[Lee84]
Lee, J. K. F. and Smith, A. J., "Branch Prediction Strategies and Branch Target Buffer Design, *IEEE Computer Magazine*, vol. 17, 1 (1984).

[Mor79]
Morris, D. and Ibbett, R. N., The MU5 Computer System, 1979, Springer Verlag.

[N2M85]
ENDOT, "N.2 metaMicro User's Manual", version 1.10 5/17/85.

[N2L85]
ENDOT, "N.2 Linking/loader user's manual", version 1.06, 5/16/85.

[N2E85]
ENDOT, "N.2 Ecologist user's manual", version 1.03, 5/10/85.

[N2I84]
Case Western Reserve Univ., and ENDOT, "N.2 ISP' user's manual", version 1.00, 3/1/84.

[N2U85]
ENDOT, "N.2 Utilities user's manual", version 1.04, 1/8/85.

[Pat82]
Patterson, D. A. and Sequin, C. H., "A VLSI RISC," *Computer*, vol. 15, 9 (1982), pp. 8-18.

[Pat83]
Patterson, D. A., Garrison, P., Hill, M., Lioupis, D., Nyberg, C., Sippel, T. and Dyke, K. V., "Architecture of a VLSI Instruction Cache for a RISC", *Proc. ACM SIGARCH symp. on Computer Architecture*, June 1983.

[Ran86]
Rankin, L. J., Kuo, S, L., *Micro-Instruction Dispatch and Decoding*, Dept. of Computer Sc. & Eng., Oregon Graduate Center, 1986.

[Smi81]
Smith, J. E., "A Study of Branch Prediction Strategies," *Proc. ACM SIGARCH Symp. on Computer Architechture*, May 1981.

[Tha86]
Thakkar, S. S., Hostmann, W. E., "An Instruction Fetch Unit for A Graph Reduction Machine", *Computer Arch. News Vol. 14, No. 2* June 1986, pp 82-91.

[Tur79]
Turner, D. A., New implementation techniques for applicative languages, *Software - Prac. & Exper. 9*, No. 1 (Jan. 1979), pp 31-49

[Wil83]
Wilkes, M. V., "Keeping jump instructions out of the pipeline of a RISC-like computers", *Computer Arch. News 11*, No. 5 (Dec. 1983), pp. 5-7.

APPENDIX A : TEST PROGRAMS

I. Test Programs in LML format

1. Iterated Jump

jump sequences are iterated and conditional jumps
are always taken.

```
let
  foo n =
     if n<1 then n
     else if n<2 then n
     else if n<3 then n
     else if n<4 then n
        .
        .
        .
     else if n<38 then n
     else if n<39 then n
     else n
in foo 10
```

2. Nested Jump

jump sequences are nested and conditional jumps
are is never taken.

```
let
  foo n =
     if n>1 then
     if n>2 then
     if n>3 then
     if n>4 then
        .
        .
        .
     if n>26 then
     if n>27 then
     n
     else n
     else n
        .
        .
        .
     else n
```

```
      else n
in foo 10
```

3. From
generate an infinite list of integers.

```
letrec
   from x = x from (x + 1)
in from 2
```

4. strict Linfib
generate a Fibonacci number with some accumulator variables and a tail recursive linear-time algorithm.

```
letrec
   fib x y n =
      if n = 0 then y
      else fib y (x + y) (n - 1)
in fib 0 1 2
```

II. Test Programs in G-code format:

Iterated Jump Program
Jump sequences are iterated and conditional jumps are always taken.

```
begin
        PUSHCONST (6)
        PUSHCONST (4)
        MK_APP
        EVAL
L0:
        EVAL
        GET_FST     (0)
        GET_BYTE    (1)
        SUB    (1)
        MOVEV       (0)
        JNOT_NEG    (L24)
        MOVEV       (0)
        COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
L24:    GET_FST     (0)
        GET_BYTE    (2)
        SUB    (1)
        MOVEV       (0)
        JNOT_NEG    (L46)
        MOVEV       (0)
        COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
L46:    GET_FST     (0)
        GET_BYTE    (3)
        SUB    (1)
        MOVEV       (0)
        JNOT_NEG    (L68)
        MOVEV       (0)
        COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
          .
          .
          .
          .
L816:   GET_FST     (0)
        GET_BYTE    (38)
        SUB    (1)
```

```
        MOVEV       (0)
        JNOT_NEG    (L838)
        MOVEV       (0)
        COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
L838:   GET_FST     (0)
        GET_BYTE    (39)
        SUB     (1)
        MOVEV       (0)
        JNOT_NEG    (L860)
        MOVEV       (0)
        COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
L860:   COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
end$
```

Nested Jump Program

Jump sequences are nested and conditional jumps are never taken.

```
begin
        PUSHCONST (6)
        PUSHCONST (4)
        MK_APP
        EVAL
L0:
        EVAL
        GET_BYTE    (1)
        GET_FST     (0)
        SUB     (1)
        MOVEV       (0)
        JNOT_NEG    (L596)
        MOVEV       (0)
        GET_BYTE    (2)
        GET_FST     (0)
        SUB     (1)
        MOVEV       (0)
        JNOT_NEG    (L588)
        MOVEV       (0)
        GET_BYTE    (3)
        GET_FST     (0)
        SUB     (1)
        MOVEV       (0)
```

```
        JNOT_NEG    (L580)
        MOVEV       (0)
          .
          .
          .
        GET_BYTE    (26)
        GET_FST     (0)
        SUB    (1)
        MOVEV       (0)
        JNOT_NEG    (L396)
        MOVEV       (0)
        GET_BYTE    (27)
        GET_FST     (0)
        SUB    (1)
        MOVEV       (0)
        JNOT_NEG    (L388)
        MOVEV       (0)
        COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
L388:   COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
L396:   COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
          .
          .
          .
          .
L580:   COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
L588:   COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
L596:   COPYP(0)
        UPDATE_POLY         (2)
        POPP
        RET
end$
```

From Program
generate an infinite list of integers

```
begin
L36:
        PUSHCONST   (8)
        PUSHCONST   (2)
        MK_APP
Loop:   EVAL
        COPYP(0)
        FST
        EVAL
        POPP
        SND
        JMP     (Loop)
L0:
        COPYP(0)
        PUSHCONST   (4)
        MK_APP
        PUSHCONST   (2)
        MK_APP
        COPYP(1)
        UPDATE_PR   (3)
        POPP
        RET
L22
        EVAL
        GET_FST     (0)
        GET_BYTE    (1)
        ADD     (1)
        MOVEV       (0)
        POPP
        RET_IN
end$
```

strict Linfib
generate a fibonacci number with some accumulator
variables and a tail recursive linear-time algorithm.

```
begin
        ALLOC
        ALLOC
        ALLOC
        GET_BYTE (2)
        CALLGLOBFUN (L80)
L0:
        GET_FST     (2)
        GET_BYTE    (0)
        SUB     (1)
        JNOT_ZERO   (L24)
```

```
        MOVEV       (0)
        COPYP(0)
        UPDATE_POLY     (4)
        POP2
        POPP
        RET
L24:    GET_FST     (2)
        GET_BYTE    (1)
        SUB    (1)
        JNOT_ZERO   (L48)
        MOVEV       (0)
        COPYP(1)
        UPDATE_POLY     (4)
        POP2
        POPP
        RET
L48:    GET_FST     (2)
        GET_BYTE    (1)
        SUB    (1)
        MK_VAL
        MOVEP       (2)
        COPYP(1)
        GET_FST     (1)
        GET_FST     (2)
        ADD    (1)
        MK_VAL
        MOVEP       (2)
        MOVEP       (0)
        JMP    (L0)
L80:
        PUSHCONST   (8)
        MOVEP       (1)
        PUSHCONST   (10)
        MOVEP       (0)
        PUSHCONST   (12)
        JMP    (L0)
end$
```

APPENDIX B : MICRO STORE

The Micro Store is implemented as an N.2 simulated memory. Data are stored in the Micro Store as fixed length words. The following file specifies G-code-to-microcode expansion sequences.

```
!-------------------------------------------------------------------------
!
!       The metaMicro assembler source code file
!       for building the Micro Store of the IFTU of
!       the G-processor. In this file, all the
!       G-code-to-microcode expansion sequences are defined.
!
!       1. 1st instr. of a micro sequence (except those micro sequences
!          which contain only one risc instruction) always have
!          addr_ty parameter = 1, next_addr parameter = a label,
!          as shown in the example, label eval_s.
!       2. argument list from left to right are defined as follows:
!
!       1. end_seq          0       - not end of micro seq
!                           1       - end of micro seq
!       2. oprnd_ty         0       - operand from ITU
!                           1       - operand from micro store
!       3. operand          xxxxx   - 5 bit operand from micro store to PCU
!       4. pre_ins_ty       1       - current instr. is a conditional jump
!                           0       - current instr. is not a cond. jump
!       5. addr_ty          0       - next instr. address from .+1
!                           1       - next instr. address from a label
!       6. next_addr        .+1     - go to next logical addr
!                           label   - jump to label
!
!
!       Author:  Boris Agapiev, Siroos Farahmand-nia,
!                Shyue Ling Kuo and Linda Rankin. 1986
!
!-------------------------------------------------------------------------
!
include risc.m$
begin
                    iNOP            (1,0,0,0,0,0)
                    iIFTUNOP        (1,0,0,0,0,0)
ALLOC:              iALLOC          (1,1,0,0,0,0)

XFER_V_P:           iMOV_V_P        (1,1,0,0,0,0)

PUSHCONST:          iMOV_L_P        (1,1,0,0,0,0)
```

```
PUSHLITERAL:    iMOV_L_V        (1,1,0,0,0,0)

!.=0203$
!INSERT_BYTE:

POP2:           iPOPP           (0,1,0,0,1,pop2_s)

POP4:           iPOPP           (0,1,0,0,1,pop4_s)

POP8:           iPOPP           (0,1,0,0,1,pop8_s)

POP16:          iPOPP           (0,1,0,0,1,pop16_s)

SIGNAL:         iSIGNAL         (1,0,0,0,0,0)

COPYP:          iCOPYP          (1,0,0,0,0,0)

COPYV:          iCOPYV          (1,0,0,0,0,0)

MOVEP:          iMOVEP          (1,0,0,0,0,0)

MOVEV:          iMOVEV          (1,0,0,0,0,0)

POPP:           iPOPP           (1,1,0,0,0,0)

POPV:           iPOPV           (1,1,0,0,0,0)

ROTP:           iROTP           (1,0,0,0,0,0)

NROTP:          iNROTP          (1,0,0,0,0,0)

ROTV:           iROTV           (1,0,0,0,0,0)

VROTP:          iMOV_V_PCU      (0,1,0,0,1,vrotp_s)


ADD:            iADD            (1,0,0,0,0,0)

ADDC:           iADDC           (1,0,0,0,0,0)

AND:            iAND            (1,0,0,0,0,0)

!.=015$
!DIV:           iDIV            (1,0,0,0,0,0)

!.=016$
!MOD:           iMOD            (1,0,0,0,0,0)

MUL:            iMUL            (1,0,0,0,0,0)

OR:             iOR             (1,0,0,0,0,0)
```

```
SUB:            iCMP            (0,0,0,0,1,sub_s)

SUBB:           iCMP            (0,0,0,0,1,subb_s)

NOT:            iCMP            (1,1,0,0,0,0)

NEG:            iCMP            (0,1,0,0,1,neg_s)

INCR:           iINCR_V         (1,1,0,0,0,0)

DECR:           iDECR_V         (1,1,0,0,0,0)

CIRC:           iCIRCL          (1,1,0,0,0,0)

SHLA:           iSHLA           (1,1,0,0,0,0)

SHRA            iSHRA           (1,1,0,0,0,0)

SHRL:           iSHRL           (1,1,0,0,0,0)

JNOT_NEG:       iJ_NOT_NEG      (1,1,0,0,0,0)

JNEG:           iJNEG           (1,1,0,0,0,0)

JNOT_ZERO:      iJ_NOT_ZERO     (1,1,0,0,0,0)

JZERO:          iJZERO          (1,1,0,0,0,0)

JCARRY:         iJCARRY         (1,1,0,0,0,0)

JOVR:           iJOVR           (1,1,0,0,0,0)

J_IF_PTR:       iJ_IF_PTR       (1,1,0,0,0,0)

J_NOT_PTR:      iJ_NOT_PTR      (1,1,0,0,0,0)


FST:            iMOV_P_A        (0,1,0,0,1,fst_s)
SND:            iMOV_P_A        (0,1,1,0,1,snd_s)
GET_FST:
                iCOPYP          (0,0,0,0,1,gfst_s)
GET_SND:
                iCOPYP          (0,0,0,0,1,gsnd_s)
GET_BYTE:
                iMOV_L_V        (1,0,0,0,0,0)

MK_APP:         iALLOC          (0,0,0,0,1,mkapp_s) ! allocate a new node

MK_PR:          iALLOC          (0,0,0,0,1,mkpr_s)  ! allocate a new node

MK_VAL:         iALLOC          (0,0,0,0,1,mkval_s) ! allocate a new node
```

```
MK_V1_PR:       iALLOC          (0,0,0,0,1,mkv1pr_s)! allocate a new node

MK_VAL_PR:      iALLOC          (0,0,0,0,1,mkvalpr_s)       ! allocate a new node

CALLGLOBFUN:    iINCR_V         (0,0,0,0,1,callgf_s)  ! increment # of args.

EVAL:           iCOPYP          (0,1,0,0,1,eval_s)    ! copy pointer to graph

UPDATE:         iCOPYP          (0,0,0,0,1,update_s)

UPDATE_PR:
                iSET_EVAL       (0,1,1,0,1,updpr_s)

UPDATE_V1:      iSET_EVAL       (0,1,1,0,1,updv1_s)
UPDATE_V_PR:    iSET_EVAL       (0,1,1,0,1,updvpr_s)

UPDATE_POLY:    iCOPYP          (0,0,0,0,1,updpoly_s)

RETURN:                 iROTP           (0,1,2,0,1,return_s)  ! fetch excess

RET_IN:         iROTP           (0,1,2,0,1,ret_in_s)  ! fetch excess


!start body of micro codes
!-------------------
set

!POP2   iPOPP           (0,1,0,0,1,pop2_s)
pop2_s: iPOPP           (1,1,0,0,0,0)

!POP4   iPOPP           (0,1,0,0,1,pop4_s)
pop4_s: iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (1,1,0,0,0,0)

!POP8   iPOPP           (0,1,0,0,1,pop8_s)
pop8_s: iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (1,1,0,0,0,0)

!POP16  iPOPP           (0,1,0,0,1,pop16_s)
pop16_s:
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
```

```
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0. +1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (0,1,0,0,0,.+1)
        iPOPP           (1,1,0,0,0,0)

!VROTP  iMOV_V_PCU      (0,1,0,0,1,vrotp_s)
vrotp_s: iNOP   (0,1,0,0,0,.+1)
                iNROTP  (1,1,0,0,0,0)

!SUB:   iCMP            (0,1,0,0,1,sub_s)
sub_s:  iADDC           (1,0,0,0,0,0)

!SUBB.  iCMP            (0,1,0,0,1,subb_s)
subb_s: iADD            (1,0,0,0,0,0)

!NEG:   iCMP            (0,1,0,0,1,neg_s)
neg_s:  iINCR_V         (1,1,0,0,0,0)

!FST:   iMOV_P_A         (0,1,0,0,1,fst_s)
fst_s:  iREADG_P        (1,0,0,0,0,0)

!SND:   iMOV_P_A         (0,1,1,0,1,snd_s)
snd_s:  iREADG_P        (1,0,0,0,0,0)

!GET_FST:
!       iCOPYP          (0,0,0,0,1,gfst_s)
gfst_s: iMOV_P_A        (0,1,0,0,0,.+1)
        iREADG_V        (1,0,0,0,0,0)

!GET_SND:
!       iCOPYP          (0,0,0,0,1,gsnd_s)
gsnd_s: iMOV_P_A        (0,1,1,0,0,.+1)
        iREADG_V        (1,0,0,0,0,0)



!MK_PR: iALLOC          (0,0,0,0,1,mkpr_s)  ! allocate a new node
mkpr_s:
        iSET_EVAL       (0,1,1,0,0,.+1)     ! set the is_evaluated bit in the T-register
        iROTP           (0,1,1,0,0,.+1)     ! exchange two pointers at top of P-stack
        iCOPYP          (0,1,1,0,0,.+1)     ! copy pointer to newly allocated node
        iMOV_P_A        (0,1,0,0,0,.+1)
        iWRITEG_P       (0,0,0,0,0,.+1)     ! store first element from P
        iROTP           (0,1,1,0,0,.+1)     ! exchange two pointers at top of P-stack
```

```
        iCOPYP          (0,1,1,0,0,.+1)     ! copy pointer to newly allocated node
        iMOV_P_A        (0,1,1,0,0,.+1)
        iWRITEG_P       (1,0,0,0,0,0)       ! store second element from P


!MK_VAL: iALLOC         (0,0,0,0,1,mkval_s) ! allocate a new node
mkval_s:
        iSET_EVAL       (0,1,1,0,0,.+1)     ! set the is_evaluated bit in the T-register
        iCOPYP          (0,1,0,0,0,.+1)     ! copy pointer to newly allocated node
        iMOV_P_A        (0,1,0,0,0,.+1)
        iWRITEG_V       (0,0,0,0,0,.+1)     ! store first element from V
        iZERO           (0,0,0,0,0,.+1)
        iCOPYP          (0,1,0,0,0,.+1)
        iMOV_P_A        (0,1,1,0,0,.+1)
        iWRITEG_V       (1,0,0,0,0,0)


!MK_APP: iALLOC         (0,0,0,0,1,mkapp_s) ! allocate a new node
mkapp_s:
        iSET_EVAL       (0,1,0,0,0,.+1)     ! clear the is_evaluated bit in the T-register
        iROTP           (0,1,1,0,0,.+1)     ! exchange two pointers at top of P-stack
        iCOPYP          (0,1,1,0,0,.+1)     ! copy pointer to newly allocated node
        iMOV_P_A        (0,1,0,0,0,.+1)
        iWRITEG_P       (0,0,0,0,0,.+1)     ! store first element from P
        iROTP           (0,1,1,0,0,.+1)     ! exchange two pointers at top of P-stack
        iCOPYP          (0,1,1,0,0,.+1)     ! copy pointer to newly allocated node
        iMOV_P_A        (0,1,1,0,0,.+1)
        iWRITEG_P       (1,0,0,0,0,0)       ! store second element from P


!MK_V1_PR: iALLOC       (0,0,0,0,1,mkv1pr_s)! allocate a new node
mkv1pr_s:
        iSET_EVAL       (0,1,1,0,0,.+1)     ! set the is_evaluated bit in the T-register
        iCOPYP          (0,1,0,0,0,.+1)     ! copy pointer to newly allocated node
        iMOV_P_A        (0,1,0,0,0,.+1)
        iWRITEG_V       (0,0,0,0,0,.+1)     ! store first element from V
        iROTP           (0,1,1,0,0,.+1)     ! exchange two pointers at top of P-stack
        iCOPYP          (0,1,1,0,0,.+1)     ! copy pointer to newly allocated node
        iMOV_P_A        (0,1,1,0,0,.+1)
        iWRITEG_P       (1,0,0,0,0,0)       ! store second element from P


!MK_VAL_PR: iALLOC      (0,0,0,0,1,mkvalpr_s)       ! allocate a new node
mkvalpr_s:
        iSET_EVAL       (0,1,1,0,0,.+1)     ! set the is_evaluated bit in the T-register
        iCOPYP          (0,1,0,0,0,.+1)     ! copy pointer to newly allocated node
```

```
        iMOV_P_A        (0,1,0,0,0,.+1)
        iWRITEG_V       (0,0,0,0,0,.+1)     ! store first element from V
        iCOPYP          (0,1,0,0,0,.+1)     ! copy pointer to newly allocated node
        iMOV_P_A        (0,1,1,0,0,.+1)
        iWRITEG_V       (1,0,0,0,0,0)       ! store second element from V


!UPDATE:                iCOPYP          (0,0,0,0,1,update_s)
update_s:
                iCOPYP          (0,1,1,0,0,.+1)
                iMOV_P_A        (0,1,0,0,0,.+1)
                iREADG_P        (0,0,0,0,0,.+1)
                iROTP           (0,1,1,0,0,.+1)
                iMOV_P_A        (0,1,0,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_P       (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,1,0,0,.+1)
                iREADG_P        (0,0,0,0,0,.+1)
                iCOPYP          (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,1,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_P       (1,0,0,0,0,0)
!
!UPDATE_V1:     iSET_EVAL       (0,1,1,0,1,updv1_s)
updv1_s:
                iCOPYP          (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,0,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_V       (0,0,0,0,0,.+1)
                iCOPYP          (0,0,0,0,0, +1)
                iMOV_P_A        (0,1,1,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_P       (1,0,0,0,0,0)
!
!UPDATE_V_PR:   iSET_EVAL       (0,1,1,0,1,updvpr_s)
updvpr_s:
                iCOPYP          (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,0,0,0, +1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_V       (0,0,0,0,0,.+1)
                iCOPYP          (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,1,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_V       (1,0,0,0,0,0)
!
!UPDATE_PR:     iSET_EVAL       (0,1,1,0,1,updpr_s)
updpr_s:
                iCOPYP          (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,0,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
```

```
                iCOPYP          (0,1,0,0,0,.+1)
                iWRITEG_P       (0,0,0,0,0,.+1)
                iCOPYP          (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,1,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iPOPP           (0,0,0,0,0,.+1)
                iWRITEG_P       (1,0,0,0,0,0)
!
!UPDATE_POLY:   iCOPYP           (0,0,0,0,1,updpoly_s)
updpoly_s:
                iCOPYP          (0,1,1,0,0,.+1)
                iMOV_P_A        (0,1,0,0,0,.+1)
                iREADG_V        (0,0,0,0,0,.+1)
                iLJ_IF_PTR      (0,0,0,1,1,is_ptr)
                iNOP        (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,0,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_V       (0,0,0,0,1,second)
is_ptr:         iMOV_V_P        (0,0,0,0,0,.+1)
                iROTP           (0,1,1,0,0,.+1)
                iMOV_P_A        (0,1,0,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_P       (0,0,0,0,0,.+1)
second:         iMOV_P_A        (0,1,1,0,0,.+1)
                iREADG_V    (0,0,0,0,0,.+1)
                iLJ_IF_PTR      (0,0,0,1,1,ptrtoo)
                iNOP        (0,0,0,0,0,.+1)
                iCOPYP          (0,1,0,0,0,.+1)
                iCOPYP          (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,1,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_V       (0,0,0,0,0,.+1)
                iPOPP           (0,0,0,0,1,fin)
ptrtoo:         iMOV_V_P        (0,0,0,0,0,.+1)
                iCOPYP          (0,0,0,0,0,.+1)
                iMOV_P_A        (0,1,1,0,0,.+1)
                iTRASH      (0,0,0,0,0,.+1)
                iWRITEG_P       (0,0,0,0,0,.+1)
fin:            iNOP            (1,0,0,0,0,0)

! Micro-sequence for CALLGLOBFUN
!
!CALLGLOBFUN:
!       iINCR_V         (0,0,0,0,1,callgf_s)   ! increment # of args.
callgf_s:
        iCOPYV          (0,1,0,0,0,.+1)             ! copy it
        iZERO           (0,0,0,0,0,.+1)             ! V-top = 0
        iMOV_V_P        (0,0,0,0,0,.+1)             ! excess : = 0
        iMOV_V_PCU      (0,0,0,0,0,.+1)
        iNOP            (0,0,0,0,0,.+1)
        iNROTP          (0,0,0,0,0,.+1)             ! put excess below arguments
```

```
        iMOV_PC_P       (0,0,0,0,0, +1)             ! store PC
        iMOV_V_PCU      (0,0,0,0,0,.+1)
        iNOP            (0,0,0,0,0,.+1)             ! for ope. to get ready
        iNROTP          (1,0,0,0,0,0)               ! put PC below arguments
        !iNOP           (0,0,0,0,0,.+1)


! Microsequence for RETURN
!
!RETURN:iROTP           (0,1,2,0,1,return_s) ! fetch excess
return_s:
        iMOV_P_V        (0,0,0,0,0,.+1)             ! and put it onto V-stack
        iSHLA           (0,1,0,0,0,.+1)             ! set cond. codes
        iLJZERO         (0,0,0,1,1,cont)     ! jump if it is 0
        iNOP            (0,0,0,0,0,.+1)
        iROTP           (0,1,1,0,0,.+1)             ! fetch PC
        iMOV_P_V        (0,0,0,0,0,.+1)             ! and store it on the V-stack
        iROTV           (0,1,1,0,0,.+1)             ! restore excess
! Unwind code (modified) starts here
        iCOPYV          (0,1,0,0,0,.+1)             ! copy arg count
        iMOV_P_V        (0,0,0,0,0,.+1)             ! move arity from P to V
        iCOPYV          (0,1,0,0,0,.+1)             ! copy it
        iMOVEV          (0,1,2,0,0,.+1)             ! and save it
        iCMP            (0,0,0,0,0,.+1)
        iADDC           (0,0,0,0,0,.+1)             ! excess:=arg.count-arity
        iLJNEG          (0,0,0,1,1,suspend)  ! branch if result is < 0
        iNOP            (0,0,0,0,0,.+1)
        iMOV_V_P        (0,0,0,0,0,.+1)             ! save excess
        iROTV           (0,1,3,0,0,.+1)             ! fetch saved PC
        iMOV_V_P        (0,0,0,0,0,.+1)             ! store PC
        iROTP           (0,1,2,0,0,.+1)             ! rotate pointer to fun node
        iMOV_P_A        (0,1,1,0,0,.+1)             ! load PC with entry point
        iREADG_PC       (0,0,0,0,0,.+1)
        iCOPYV          (0,1,0,0,0..+1)             ! copy arity
        iINCR_V         (0,0,0,0,0,.+1)             ! V-top := arity+1
        iCOPYV          (0,1,0,0,0,.+1)             ! copy it
        iMOV_V_PCU      (0,0,0,0,0,.+1)
        iNOP            (0,0,0,0,0,.+1)
        iROTP           (0,0,0,0,0,.+1)             ! rotate redex node
        iCOPYP          (0,1,0,0,0,.+1)             ! and copy it
        iDECR_V         (0,0,0,0,0,.+1)
rewind: iROTV           (0,1,1,0,0,.+1)             ! bring loop index to top
        iDECR_V         (0,0,0,0,0,.+1)
        iLJZERO         (0,0,0,1,1,loop_exit)
        iNOP            (0,0,0,0,0,.+1)
        iMOV_P_A        (0,0,0,0,0,.+1)             ! fetch argument pointer
        iREADG_P        (0,0,0,0,0,.+1)
        iROTV           (0,1,1,0,0,.+1)
        iCOPYV          (0,1,0,0,0,.+1)             ! copy it
        iMOV_V_PCU      (0,0,0,0,0,.+1)
        iNOP            (0,0,0,0,0,.+1)
```

```
          iROTP          (0,0,0,0,0,.+1)              ! rotate next app. node
          iNOP           (0,0,0,0,1,rewind)
loop_exit: iMOV_P_A      (0,0,0,0,0,.+1)
          iREADG_P       (0,0,0,0,0,.+1)
          iNOP           (0,0,0,0,1,rexit)
cont:     iROTP          (0,1,1,0,0,.+1)              ! rotate PC
          iMOV_P_PC      (0,1,0,0,1,rexit)    ! load PC with entry point
suspend:  iADD           (0,0,0,0,0,.+1)              ! restore arg. count
          iROTV          (0,1,1,0,0,.+1)              ! get saved PC
          iMOV_V_PC      (0,0,0,0,0,.+1)              ! and restore it
loop.     iPOPP          (0,0,0,0,0,.+1)
          iDECR_V        (0,0,0,0,0,.+1)
          iLJZERO        (0,0,0,1,1,rexit)
          iNOP           (0,0,0,0,0,.+1)
          iNOP           (0,0,0,0,1,loop)
rexit:    iNOP           (1,0,0,0,0,0)

! Micro sequence for RET_INT
!
!RET_IN: iROTP           (0,1,2,0,1,ret_in_s)  ! fetch excess
ret_in_s:
          iPOPP          (0,0,0,0,0,.+1)              ! destroy it
          iROTP          (0,1,1,0,0,.+1)              ! fetch PC
          iMOV_P_PC      (0,0,0,0,0,.+1)              ! and restore it
          iCOPYP         (0,1,0,0,0,.+1)              ! copy the update ptr
          iMOV_P_A       (0,0,0,0,0,.+1)
          iTRASH         (0,0,0,0,0,.+1)
          !iMOV_P_A      (0,0,0,0,0,.+1)
          iWRITEG_V      (0,0,0,0,0,.+1)              ! write integer
          iZERO          (0,0,0,0,0,.+1)
          iCOPYP         (0,1,0,0,0,.+1)              ! copy the update ptr
          iMOV_P_A       (0,1,1,0,0,.+1)
          iTRASH         (0,0,0,0,0,.+1)
          !iZERO         (0,0,0,0,0,.+1)
          !iCOPYP        (0,1,0,0,0,.+1)              ! copy the update ptr
          !iMOV_P_A      (0,1,1,0,0,.+1)
          iWRITEG_V      (1,0,0,0,0,0)                ! write 0
! Microsequence for EVAL
!
!EVAL:   iCOPYP          (0,1,0,0,1,eval_s)   ! copy pointer to graph
eval_s:   iMOV_P_A       (0,1,0,0,0,.+1)              ! fetch first word
          iREADG_P       (0,0,0,0,0,.+1)
          iLJ_IF_EVAL    (0,0,0,1,1,eout)     ! return if graph is already evaluated
          iZERO          (0,0,0,0,0,.+1)              ! set V-top to 0
eunwind:
          iCOPYP         (0,1,0,0,0,.+1)              ! copy pointer to function part
          iINCR_V        (0,0,0,0,0,.+1)              ! increment arg. count
          iMOV_P_A       (0,1,0,0,0,.+1)              ! fetch word
          iREADG_P       (0,0,0,0,0,.+1)
          iLJ_IF_EVAL    (0,0,0,1,1,efun)             ! jump to a function case
          iNOP           (0,0,0,0,1,eunwind)  ! continue to unwind
```

```
efun:     iCOPYV          (0,1,0,0,0,.+1)              ! copy arg. count
          iMOV_P_V        (0,1,0,0,0,.+1)              ! move arity from P to V
          iCOPYV          (0,1,0,0,0,.+1)              ! copy it
          iMOVEV          (0,1,2,0,0,.+1)              ! and save it
          iCMP            (0,0,0,0,0,.+1)
          iADDC           (0,0,0,0,0,.+1)              ! excess:=arg count-arity
          iLJNEG          (0,0,0,1,1,esuspend) ! branch if result is < 0
          iNOP            (0,0,0,0,0,.+1)
          iMOV_V_P        (0,1,0,0,0,.+1)              ! save excess
          iMOV_PC_P       (0,1,0,0,0, +1)              ! store PC
          iROTP           (0,1,2,0,0,.+1)              ! rotate pointer to fun node
          iMOV_P_A        (0,1,1,0,0,.+1)              ! load PC with entry point
          iREADG_PC       (0,0,0,0,0,.+1)
          iCOPYV          (0,1,0,0,0,.+1)              ! copy arity
          iINCR_V         (0,0,0,0,0,.+1)              ! V-top := arity+1
          iCOPYV          (0,1,0,0,0,.+1)              ! copy it
          iMOV_V_PCU      (0,1,0,0,0,.+1)
          iNOP            (0,0,0,0,0,.+1)
          iROTP           (0,1,0,0,0,.+1)              ! rotate redex node
          iCOPYP          (0,1,0,0,0,.+1)              ! and copy it
          iINCR_V         (0,0,0,0,0,.+1)
erewind:  iROTV           (0,1,1,0,0,.+1)              ! bring loop index to top
          iDECR_V         (0,0,0,0,0,.+1)
          iLJZERO         (0,0,0,1,1,eloop_exit)
          iNOP            (0,0,0,0,0,.+1)
          iMOV_P_A        (0,1,1,0,0,.+1)              ! fetch argument pointer
          iREADG_P        (0,0,0,0,0,.+1)
          iROTV           (0,1,1,0,0,.+1)
          iCOPYV          (0,1,0,0,0,.+1)              ! copy it
          iMOV_V_PCU      (0,1,0,0,0,.+1)
          iNOP            (0,0,0,0,0,.+1)
          iROTP           (0,1,0,0,0,.+1)              ! rotate next app. node
          iNOP            (0,0,0,0,1,erewind)
eloop_exit: iMOV_P_A      (0,1,1,0,0,.+1)
          iREADG_P        (0,0,0,0,0,.+1)
          iNOP            (1,1,0,0,0,0)
esuspend: iRESUME         (0,0,0,0,0,.+1)              ! send JMP_NOT_TAKEN to IFU
          iADD            (0,0,0,0,0,.+1)              ! restore arg. count
lesuspend: iADD   (0,0,0,0,0,.+1)              ! restore arg. count
eloop:    iPOPP           (0,0,0,0,0,.+1)
          iDECR_V         (0,1,0,0,0,.+1)
          iLJZERO         (0,1,0,1,1,eend)
          iNOP            (0,1,0,0,0,.+1)
          iNOP            (0,1,0,0,1,eloop)
eout:  iPOPP        (0,0,0,0,1,eexit)
eend:     iNOP            (1,1,0,0,0,0)
eexit:    iRESUME                 (1,0,0,0,0,0)              !for sending JMP-NOT_TAKEN to IFU
tes
end$
```

APPENDIX C : IFA SIGNALS

Operations performed by the IFB for each signal from the IFA are described as follows:

1. "jump_taken" signal

The jump_taken signal informs the IFB to make the alternate instruction stream the active instruction stream. Operations performed by the IFB are 1) disable the active buffer for pre-fetching, generate control to flush the active buffer, 2) set the Is_Disabled flag for the active buffer if there is an outstanding fetch for the active buffer, and 3) switch between the active and the alternate indexes.

2. "jump_not_taken" signal

The jump_not_taken signal informs the IFB to disable the alternate buffer for pre-fetching, to flush the alternate buffer and to set the Is_Disabled flag for the alternate buffer if there is an outstanding fetch for the alternate buffer.

3. "copy_PC" signal

The copy_PC signal informs the IFB to copy the content of the active PC onto the G-BUS so that it can be saved as a return PC.

4. "load_PC" signal

The load_PC signal informs the IFB to reset the active PC and the active FC with the address provided on the G-BUS. The IFB also generates the controls to inform the active pre-fetch buffer to flush its content and to set the Is_Disabled flag if there is an outstanding fetch for the active buffer.

5. "load_Alt" signal

The load_Alt signal informs the IFB to enable an alternate instruction buffer for pre-fetching. The IFB will reset the alternate PC and the alternate FC with the address fetched

by the IFA via the addr_PC, and enable the alternate buffer for pre-fetching.

6. "unconditional_jump" signal

The unconditional_jump signal is sent to the IFB when the IFA decodes an unconditional jump instruction. Responding to this signal, the IFB will reset the PC of the normal instruction stream with the unconditional jump address. The operations performed by the IFB are 1) reset the active PC and the active FC with the jump address fetched by the IFA, 2) generate control to inform the active pre-fetch buffer to flush its content, and 3) set the Is_Disabled flag for the active buffer if there is an outstanding fetch for the active buffer.

7. "load_Alt and disable_Act" signal

The "load_Alt and disable_Act" signal informs the IFB to reset the alternate PC and the alternate FC with the address fetched by the IFA, to enable the alternate FC for pre-fetching, and to disable the active buffer from pre-fetching.

8. "copy_PC, load Alt and jump_taken" signal

The "copy_PC, load Alt and jump_taken" signal requests the IFB to copy the active PC onto the G-BUS so that it can be saved as a return PC, to reset the alternate PC and the alternate FC with the address fetched by the IFA, to enable the alternate FC for pre-fetching then to disable the active FC, and to generate the control to flush the active buffer, to set the Is_Disabled flag if there is an outstanding fetch for the active buffer, and to switch between the active index and the alternate index.

9. "copy_PC and jump_taken" signal

The "copy_PC and jump_taken" signal requests the IFB to copy the active PC onto the G-BUS and then to disable the active buffer from pre-fetching, to generate the control to flush the active buffer, to set the Is_Disabled flag for the active buffer if there is an outstanding fetch for the active buffer, and to switch between the active index and the alternate index.

Biographical Note

The author was born in 1951, in Taipei, Taiwan, Republic of China. She received her B.S. degree in Engineering Science from the National Cheng-Kung University in 1974. In the following year, the author worked as a Research Assistant in the Industrial Technology Research Institute, Hsin-Hchu, Taiwan where she maintained EAI 590 hybrid computer.

In 1975, the author was enrolled in the Graduate School of the Electrical Engineering Department of Bucknell University. She received her M.S. degree in Electrical Engineering and worked as a Software Engineer for six years before she moved to Oregon with her husband in 1983.

In 1984, the author was entrolled as a part-time graduate student in the Computer Science Department of Oregon Graduate Center. During the course of the graduate study at OGC, the author's first child, Yunn-Huey Tsang, was born in 1987. The author completed the requirements for the degree of Master of Science in Computer Science and Engineering in December, 1987.