BIT-SLICE DESIGN OF A GRAPH REDUCTION PROCESSOR

Richard Pierre Vireday
B.S. Willamette University, 1986

A thesis presented to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirement for the degree
Master of Science
in
Computer Science and Engineering.

May 1986

The thesis "Bit-Slice Design of a Graph Reduction Processor" by Richard P. Vireday has been examined and approved by the following Examination Committee.

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

Shreekant Thakkar
Assistant Professor,
Dept. of Computer Science and Engineering

Robert G. Babb II
Assistant Professor,
Dept. of Computer Science and Engineering

A.C. (Kit) Bradley
Adjunct Assistant Professor,
Textronix Inc.

Dedication

This is affectionately dedicated to my parents, Pierre and Claire, who had the dream of all immigrants of a better life in America for themselves and their children.

Acknowledgements

There are many people that I would like to thank who helped me on this project both directly and indirectly. Mark Foster, who's been a good friend, cook, and latenight hacker for the past few years; Ananda Sarangi, for helping with the first stages of the G-Machine project; Ticky Thakkar, whose advice on board design helped tremendously; various other OGC CSE Students and faculty too numerous to mention; and of course, Dick Kieburtz whose guidance and enthusiasm directed my studies and this project.

I also have to thank my advisers at Willamette University, Mike Dunlap and Bill Braden, who never knew how much they scared a poor undergraduate when he had to try and justify an English/CS degree.

And finally Pam who, though justified, sometimes didn't suffer quite silently enough.

Table of Contents

List of Figures

Abstract

Bit-Slice Design of a Graph

Reduction Processor

Richard P. Vireday
Oregon Graduate Center, 1986

Supervising Professor: Richard B. Kieburtz

The G-Machine is an abstract architecture that to supports languages with graph computing models by utilizing software technologies to provide efficient graph manipulation on sequential machines. Graph computing models are different than those for standard imperative languages, and support for graph manipulation is generally lacking on machines designed for standard imperative languages.

The architecture is stack-based and manipulates graphs via pointers, a pointer stack, and tagged memories. The tags and pointer stack also help provide an effective method of performing "lazy" evaluation, a computing technique that allows for execution of many complex algorithms.

In this thesis, the abstract G-Machine Architecture has been redefined to provide a co-processor implementation. The heart of the G-Machine co-processor is the Program Execution Unit (PCU), a conceptually simple processor which can be implemented in a variety of ways. A design for the PCU is shown that is simple to build and program, utilizes existing technology, and provides complete support for the abstract G-Machine Architecture.

1. INTRODUCTION

Use of lazy functional languages is generally inhibited by the inefficiency of their execution, especially for very large programs. When Thomas Johnsson and Lennart Augustsson of the Chalmers University in Sweden built a compiler for a functional language, they developed a target machine model that can be implemented by simple translation of the target machine instructions into those for a conventional computer [Joh83]. They called this abstract machine model the G-Machine, because it uses graph reduction as its computation model.

The G-Machine Architecture is used in conjunction with the functional language LML (a derivative of ML [GMW79], but with Lazy Evaluation), and has proven to be an excellent tool to support languages with lazy evaluation. It provides faster execution of lazy functional languages on general purpose computers by a variety of optimizations and techniques. LML programs compiled for a VAX, using this abstract machine model, have run only 30% slower than similar Pascal programs executed on the same machine [Joh83].

The abstract G-Machine Architecture has been redefined with a view toward hardware implementation [Kie84, Kie85]. Providing an implementation proved more than just a matter of mapping abstract instructions into the host processor instructions, as is done with the LML compiler. As with any architecture, special hardware that supports the Architecture's method of accessing data can be used in order to provide efficient execution, especially for frequently occurring or "special" operations.

A G-Machine processor has been designed that supports the data structures and program execution model of the abstract G-Machine Architecture. The anticipated speedup in execution is expected to make larger functional programs practical and cost-effective to use. Larger here especially means both the execution times of programs, and large data structures (graphs). Although other processors for functional languages have been designed and built, this area remains largely experimental.

The design of a processor for the G-Machine Architecture had several goals. Among them was to provide feedback on the abstract architecture as well as to experiment with and clarify different design and implementation issues. While the first version of the design is adequate and highlights the particular merits of the G-Machine Architecture, the processor will require another design iteration before actual implementation.

This thesis presents a G-Machine processor and the rationale for its particular design aspects. An overview of graph reduction is presented in Section 2, followed in Section 3 by descriptions of the architecture. The G-Machine processor is presented in Sections 4 and 5 by showing it's instructions and hardware organization. Results from simulations of the processor are given in Section 6, along with an assessment of the design.

2. GRAPH REDUCTION

The fundamental model of computation employed in the G-Machine is graph reduction. Graph reduction is an implementation scheme for Church's theory of lambda calculus [Chu41], which uses lambda expressions to represent mathematical functions. The basic reduction rule is called β-reduction. β-reduction is the substitution of all occurrences of an argument (or parameter) by the actual parameter expression in an application of a lambda expression. To illustrate briefly, let a function $\theta = (\lambda$ x. x+1). This can be read as θ is defined by the rule on the right hand side, which has an argument of x. Applying θ with an argument (x) of say 5 yields the expression (5+1) which can then be evaluated.

2.1. Evaluation models

The first practical lambda evaluator was the SECD machine described by Landin [Lan64]. Like the G-Machine, the SECD machine is an abstract architecture intended for executing functional languages. Applicative expressions are represented as trees, with the leaf nodes labeled by identifiers. The identifiers name variables in the variable store, also known as an *environment*. The SECD name is derived from the four components of the machine.

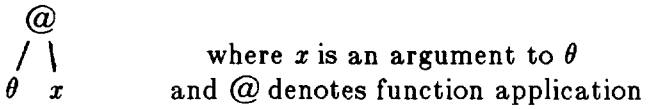
S	stack	to hold intermediate expressions during tree traversal
E	environment	for storage for variable bindings during execution
C	control list	the machine instruction store
D	dump	a stack to hold contexts during function calls

The stack **S** is used to hold parameters for each function call during the course of evaluation. If a function calls another, then the current stack environment and control list are "suspended" and a new environment is created before evaluation continues. Much of the complexity of the SECD machine comes from the construction of environments. For instance, if an expression contains free variables, then either substitute expressions must be created for the free variables, or the environments where the free variables are defined must be made available. Searching previous environments for the free variables when they are referenced can be done with a variety of methods. Regardless of the method chosen, it must be insured that the correct variable instance is referenced. This "environment passing" process, required in order to maintain free variables, can grow quite complex and creates a substantial amount of overhead in the SECD model.

The SECD machine represents a valuable model in terms of functional architectures since many machines use concepts that were first employed with the SECD architecture. One of those concepts is a fast expression stack. All reduction machines designed since the SECD machine have a stack to hold intermediate graph values during traversal. Although the computation rule was applicative rather than normal order, the basic design proved fruitful and stimulated much more work in the area of functional processors. Computation rules will be discussed further in Section 2.3.

One way to represent an expression in a computer is as a text (string). The string can be manipulated and rewritten much like employing paper and pencil to solve an equation. This approach to computation is known as *string reduction.* String reduction is different from graph reduction (Section 2.2 below) in that β-reduction can be done directly on strings if desired and the value of an expression is stored differently.

Rather than manipulating textual representations of expressions, another possible device is to have pointers to each of the components of an expression. The expression is then represented by a graph, which gives rise to a graphical way of viewing expressions. For instance, if we again use the function $\theta = (\lambda\ x.\ \ x+1)$, the graph to call this function can be represented as follows:



Graph reduction refers to a process in which expressions represented as graphs are evaluated by manipulation of the graphs. Since several arcs in a graph can point to a single element, multiple references to a sub-graph are possible. This is an important concept called *sharing.*

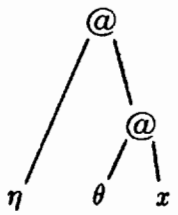
During the process of evaluation, the graph is mutated (reduced) using a set of rewrite rules until it cannot be transformed further. It is then said to have reached a normal form. The leaves of the graph are in normal form while the branches are application nodes, i.e. @ represent pointers to other elements of the tree. Normal forms usually are strings, numbers, pairs, and lists of normal forms. Generally they can be translated into a sequence of ASCII characters and printed. The rare exception would be where the result is an applicative expression, such as $(\lambda x.\lambda y.\ x+y)\ 3$ (where this example does not have all of its arguments).

The order in which a machine schedules reductions is known as its computation rule. Most expressions will have more than one redex[1], or possible point of reduction. While the final result should be independent of the order in which reduction steps are

[1] A redex is any REDucible EXpression i.e. in $\theta = (\lambda\ x.\ \ x+1)$, x is a potential redex on the left side of the rule.

carried out,[a] there can be a difference in the number of reduction steps used by different evaluators, and one may terminate while another fails to do so.

In *normal order reduction* the left outermost redex is always reduced first. With normal order reduction, an argument can be substituted into the body of a function without being evaluated first. For instance, in the graph below, the subgraph of the function θ is itself an argument to the function η.



When the application of η is evaluated, the resulting value is substituted into the top-most application node. Future references to this graph will find the result of the expression already evaluated.

Another is *applicative order*, in which no redex is reduced until all redexes internal to it have been reduced[1]. This has the possibility of not terminating in some cases for which a normal order evaluation will. A classic method used to show that applicative order terminates less often is to suppose that the argument to a function δ is a non-terminating sub-expression, and that $\delta = (\lambda\ x.\ \mathbf{C})$ where **C** is some constant expression. If we try to evaluate the argument **x** first, then δ will never terminate, whereas with normal order it will produce the constant expression **C**.

Graph reduction is usually done with normal order rather than applicative order evaluation. Together with the property that no sub-expression is evaluated more than

---

[a] Any reduction method should be consistent with the mathematical semantics of applicative expressions and should also have the Church-Rosser property which assures that regardless of the reduction method used, the normal form is unique.

[1] Known by some ½ wits as the "I will evaluate no redex before its time" order.

once, this method of computation is called *lazy evaluation.* The term lazy evaluation is due to P.Henderson and J.H.Morris III [HeM76], with the original idea now generally attributed to C.Wadsworth [GMW79].

2.2. Combinator Reduction

Many architectures designed to support β-reduction, combinator reduction, and string reduction all have in common the property that the **control** of execution (the selection of the next reduction step) is derived dynamically at each stage of the reduction sequence from the current form of the expression. Systems with this property are called *pure reduction* systems.

The combinator reduction architecture of Turner is a pure reduction system based on Curry's combinatory calculus. The combinatory calculus uses a translation technique called bracket abstraction to eliminate variables from lambda expressions, producing an applicative expression consisting solely of constant operators (combinators) and data. Reduction rules are defined for applications of each combinator but, as there are no free variables, combinator reduction does not involve substitution and there are no environments to be examined for possible values, as in the SECD machine. The basic combinators[2] are called S, K, I, B and C and have the following reduction rules:

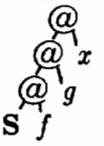
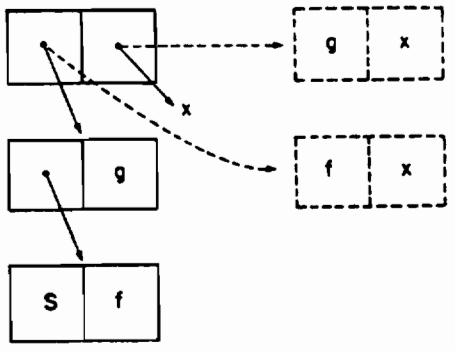
$\mathbf{S}\ f\ g\ x = f\ x\ (g\ x)$	$\mathbf{B}\ f\ g\ x = f\ (g\ x)$	*Example Graph*
$\mathbf{K}\ x\ y = f\ x\ (g\ x)$	$\mathbf{C}\ f\ g\ x = f\ x\ g$	
$\mathbf{I}\ x = x$		

Fig. 2.1 Basic SKI reduction rules

[2] Turner employs around 50 combinators, including the elementary arithmetic operators. Most of the others are used to help optimize reduction steps.

In the S-K machine, which is an implementation by Turner of his combinator reduction scheme, the programs are transformed into expressions containing the basic combinators S, K, I, B, C, as well as many others such that all variables are removed from the program body. The program is then a combinator expression which may be evaluated using normal-order graph reduction. This requires a stack to hold pointers to the expression, but does not require the overhead of multiple environments to store free variables.

A hardware implementation of the SKI scheme is the SKIM[1]. SKIM was built with conventional microprocessor components. Microcoded instruction sets are used to implement the combinator reduction. The instruction sets are very similar to Turner's S-K reduction machine. Programs are represented in SKIM by a graph built with memory cells that have two elements, which can effectively be thought of as the HEAD and the TAIL elements. The reducer uses normal order evaluation by scanning down the left-most branch of the program tree to find the operator at the leaf and to decide which reduction step to perform next. Since the reductions are fixed, they are fairly easy to program and execute quickly [Sto85].



S f g x = f x (g x)

Fig. 2.2 SKIM S expression and reduction

[1] Which stands for S K I Machine [CGM80].

As it scans down the list searching for a reduction rule, SKIM saves pointers to the nodes in an expression stack. When a reduction rule can be applied pointers are modified accordingly, as shown in the reduction of the **S** rule above. The dotted lines show the expression after **S** reduction.

2.3. Programmed Graph Reduction

In contrast to pure reduction systems, programmed graph reduction compiles the expression to be evaluated into a sequence of static instructions. Functions are represented by these compiled instruction sequences in a control memory. The graph mutations are performed as the machine operates under control of this compiled instruction stream, as in a conventional Von Neumann processor. The G-Machine is a programmed graph reduction system based on J.Hughes super-combinators approach [Hug82].

Hughes proposed a generalization of Turner's combinators, which he called *super-combinators.* Unlike the combinators in the SKI scheme, Hughes super-combinators may represent any λ-abstraction that contains no free variables. The combinators themselves are regarded as program defined constants. What this means is that super-combinators perform reduction like the SKI combinators, but they are not necessarily limited to fixed SKI type reductions. A super-combinator can be defined as any arbitrary reduction rule, composed perhaps of other super-combinators.

An advantage this has over classical combinator reduction is that it allows optimizing steps in the translation scheme. Unnecessary reductions can be eliminated at compile time.

In the LML Compiler[1], a program that contains lambda-abstractions is transformed into an expression without embedded lambda-abstractions, and a set of function definitions. This process is known as lambda-lifting and is analogous to Hughes' super-combinator abstraction scheme. Lambda-lifting, like the other process, ensures that only the variables that actually occur in an expression are bound in a closure, rather than binding the whole environment.

Several machines have been built to perform combinator reduction. Of these, perhaps the most successful have been NORMA [Sch85] and which is a normal-order evaluator for the combinatorial calculus, and SKIM [Sto85] mentioned earlier which is another combinatorial calculus engine. Berkling's GMD machine [Ber83] is a direct λ-calculus string rewriting engine.

The G-Machine is among the first machines directly utilizing programmed graph reduction (A second machine, the Categorical Abstract Machine [CCM85], is under development in France). The abstract G-Machine Architecture is described in the next section.

[1] When talking about the LML Compiler, both the original one from Sweden [Joh83], and one developed at OGC are referred to.

3. G-MACHINE ARCHITECTURE

The G-Machine is defined by a structure similar to Landin's SECD model. But, it has more components, and supports *both* applicative and normal-order computation whereas the SECD machine allowed only applicative order.

The main components of the G-Machine Model are

(C)	Control	memory for compiled functions
(P)	P-Stack	pointers into the Graph Memory
(V)	V-Stack	arithmetic and logical values
(G)	Graph Memory	storage for graphs
(D)	Dump	hold P & V stack values during function calls[1]
(E)	Environment	the current execution environment

Instructions for the abstract G-Machine architecture are called G-Codes. Detailed descriptions of all the G-Codes may be found in [Kie85] and [Kie84].

The external architecture can be expressed as an abstract register model (or sex-tuple) <C,P,V,G,D,E>, whose components are as described above. G-Codes are defined in terms of transformations on this abstract register model as the machine executes the instructions sequentially in the control stream (C).

For instance, a binary operation such as an ADD pops the top two values from the V-Stack, adds them together, and places the result onto the top of the V-Stack. This transformation is expressed as follows:

$$\langle \mathbf{ADD}.C, P, i_1.i_2.V, G, D, E\rangle \rightarrow \langle C, P, (i_1+i_2).V, G, D, E\rangle$$

[1] The Dump (D) is actually a holdover from early designs and is not directly implemented except for debugging and testing purposes.

The abstract register model has proven very useful for defining the architecture. Details can be refined further from these descriptions to provide an implementation description, sometimes to the RTL level. And, since there is a common abstract (data) base for every instruction, problems with documentation and changing instructions are reduced.

3.1. Memory Model

There are two main memories in the G-Machine, the Control (C) and the Graph (G). A third memory is the Dump (D). Each is logically distinct.

The C-Memory holds the G-Codes and any required arguments. This is the machine's program instruction store. The code store is a linear memory that is accessed conventionally with a program counter.

A location in the G-Memory (G) is a graph node, and is structured as shown in Fig. 3.1.

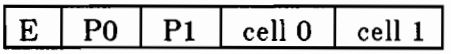


Fig. 3.1 Graph Memory Node

Each cell in a node is individually addressable. The **P** bits for each cell are tags that tell whether that cell is a pointer or not. The **E** bit says whether the whole node has been evaluated or not.

More tags may be added in different implementations, especially for such functions as garbage collection [Fos85]. However, the abstract G-Machine model needs only the **E** and **P** bits in order to execute.

A graph node can hold any of the following items:

(1) An individual scalar value (cell 0 only holds data)
(2) A pair of scalar values (both cells hold data)
(3) A function application where the cell 0 names a function in the control memory, and the cell 1 holds the number of arguments to that function
(4) A list constructor where each cell points to parts of the list (i.e. first cell points to the head, second to the rest)
(5) A hybrid pair where one cell holds a scalar value and the other holds a pointer to the other member of the pair.

The use of hybrid pairs was not in the original G-Machine model, but was added to provide for more efficient storage of scalar data imbedded in pairs.

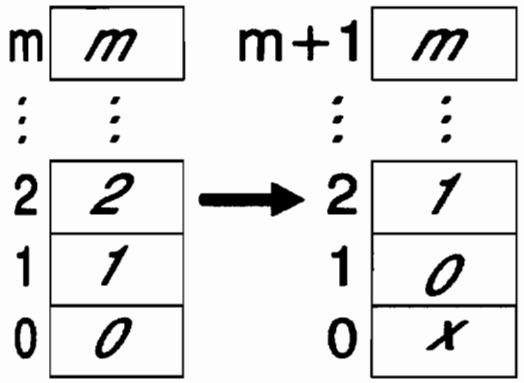
This type of graph node is not a unique representation of data memory. LISP has a similar definition [Hen80], with the **cons** and **cdr** views of list objects, as do other languages which have an *object oriented* view of data (cf. [UBF84]). The SKIM and NORMA machines [CGM80, Sch85] also have graph memories with two cells.

The Dump (D), can be thought of as a linear array of cells in which to put the stack contexts (P, and V stacks, and the C-memory control) during function calls. The need for the Dump for context switching has been eliminated in the Bit-slice machine by a different mechanism, described in Section 4.7.

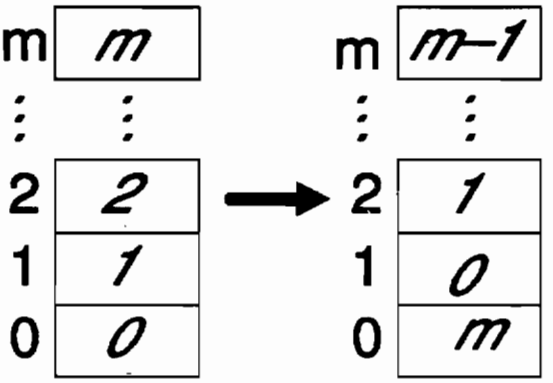
3.2. The Stacks

An overview of the stacks is given here. Detailed descriptions can be found in the G-Machine Architecture Handbook and [Joh83].

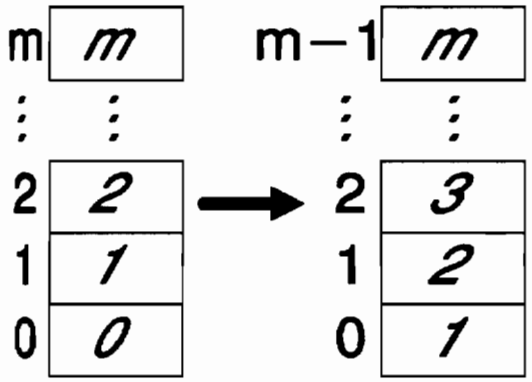
The G-Machine has two stacks in its architecture model: the P or Pointer Stack for graph manipulation, and the V or Value Stack for arithmetic operations. Both stacks have the normal push and pop stack operations, as well as the 3 other operations shown in Fig. 3.2. The top of the stack is considered element 0.



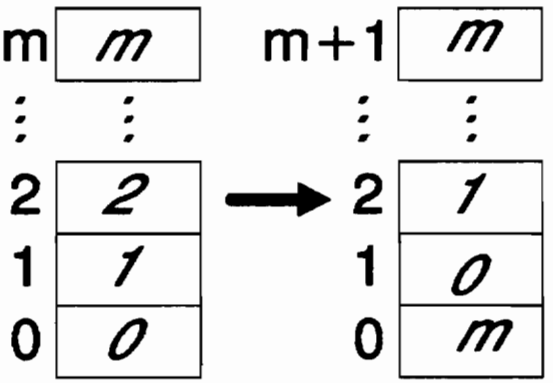
<(PUSH x).C,n_0...n_m.P,V,G,D,E>
→ <C.x.n_1...n_{m+1}.P,V,G,D,E>



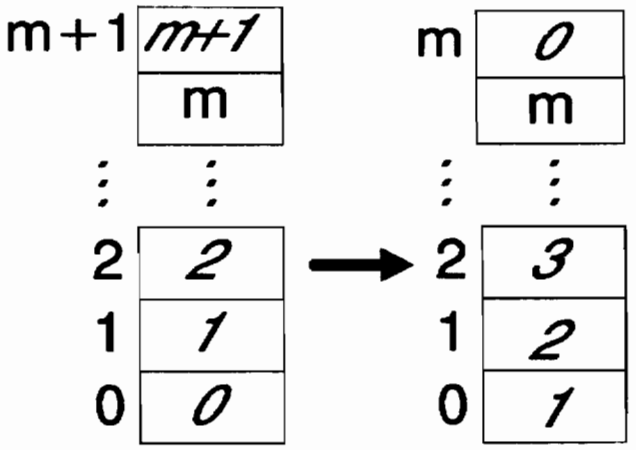
<ROTP m.C,n_0...n_{m-1}.n_m.P,V,G,D,E>
→ <C,n_m.n_0...n_{m-1}.P,V,G,D,E>



<POPP.C,n_0.P,V,G,D,E
→ <C,P,V,G,D,E>



<COPYP m.C,n_0...n_m.P,V,G,D,E>
→ <C,n_m.n_0...n_m.P,V,G,D,E>



<MOVEP m.C,n_0...n_m.n_{m+1}.P,V,G,D,E>
→ <C,n_1...n_{m+1}.n_0.P,V,G,D,E>

Fig. 3.2 G-Machine Stack Operations with Register Transfer Models

INSTRUCTION	OPERATION
PUSH	-- push a value onto the stack
POP	-- pop a value off of the stack
COPY *m*	-- bring element *m* to the top of the stack
ROT *m*	-- rotate the top of the stack to *m*'s place
MOVE *m*	-- overwrite element *m* with the top of the stack

The P-stack is a critical resource of the G-Machine. It holds the pointers to the current expression under evaluation and manipulation of the graph occurs on the P-stack. Proper implementation of the P-stack is critical because 20% to 25% of the G-Code in any particular program are P-stack operations [Sar84].

3.3. Arithmetic Operations

The abstract G-Machine Architecture has the following logical and algebraic operations defined for integers [Kie84, Kie85].

Arithmetic	Logical	Shifting
ADD	AND	ADJL
ADDC	OR	ADJR
SUB	NOT	CIRCL
SUBB	XOR	SHLA
MOD	XNOR	SHRA
MUL		SHRL
DIV		
INCR		
DECR		
NEG[1]		

The shifting operations are performed in a bitwise manner. For all operations that require two operands, the top two values are popped off of the V-Stack and the result of the operation is pushed back on. One operand instructions pop the top value, then push the result back on. The differences are shown in the following register transfer model.

[1] **NEGate is 2's complement, NOT is 1's complement.**

$<opc.C, P, i_1.i_2.V, G, D, E> \rightarrow <C, P, opc(i_1,i_2).V, G, D, E>$

where *opc* is one of ADD, ADDC, AND, OR, XOR, XNOR, DIV, MOD, MUL, SUB, SUBB

$<shft.C, P, i_1.i_2.V, G, D, E> \rightarrow <C, P, shft(i_1,i_2).V, G, D, E>$

where *shft* is one of SHLA, SHRA, SHRL, ADJL, ADJR, CIRCL

$<unary\text{-}op.C, P, i.V, G, D, E> \rightarrow <C, P, (result).V, G, D, E>$

where *unary-op* is one of NEG, NOT, INCR, DECR

The abstract G-Machine model does not specify any arithmetic precision requirements. But, because the LML Compilers[1] support 32 bit integers, it was decided that a goal of this design was to provide a 32 bit integer machine as well.

Regarding other types, Boolean variables are simply integers (0 for false, non 0 for true). Floating point numbers are not currently supported, as is indexing for arrays. Arrays in particular are a problem that functional languages and architectures have not found an adequate solution for as yet. The problem is not important in this discussion, and the reader is referred to these other texts [Ack82, ArT81, Bac78, Den80, Hen80] for an explanation of the problems surrounding arrays in functional languages.

3.4. Machine Operation

The G-Machine operates by sequentially executing the G-Codes from the Control Memory, transforming the state of the machine accordingly. Branches in the code and all addresses are Execution control of the G-Machine is done by the JMP and conditional jumps[2], CALLGLOBFUN, RET, and EVAL G-Codes.

---

[1] Both the Swedish and OGC compilers are 11/780 VAX-based.

[2] See Appendix C.

The procedure call and return mechanism is provided by the CALLGLOBFUN and RET. The other way to invoke a function is to EVAL it. EVAL is a complex G-Code instruction that reduces an application graph to its normal form. Taking the example of $\theta = (\lambda\ x.\ \ x+1)$ again,

in the expression θ 5, EVAL suspends the current context on the P-stack, performing the necessary stack pointer operations, and invokes the code for θ for evaluation.

When the function returns, the result (6) is written into the application pointer (shown as @) by the UPDATE instruction, and evaluation of the calling function would continue from the point where the expression was demanded.

A key point that must be stressed here is that graph nodes are only written to by the UPDATE instruction. Overwriting an evaluated node is erroneous and violates the semantics of lazy evaluation.

Complete definitions of all G-Codes supported in the abstract architecture can be found in [Kie84], with the newer abstract architecture described in [Kie85]. The next section describes the current design of a processor for the abstract architecture, and details the assumptions regarding instruction set organization and implementation.

4. PROCESSOR CONTROL UNIT DESIGN

The Processor Control Unit (PCU) of the G-Machine has gone through several design iterations. The first design proposal was based directly on the instructions in [Joh83]. This design was planned to be a single processor. But the results from functional simulation of the G-Machine [Sar84] and larger study of the architecture indicated that any proposed processor would have to be composed of distinct code processing and execution sections, as well as a separate graph memory section.

During the design and simulation of the G-Machine, two simulators were used.

Type	Purpose
Functional	Executed Abstract Architecture at an RTL Level
Bit-slice	Bit-slice PCU and ALU. Real components and timing. Simulated Instruction Fetch Unit.

G-Machine Simulators

The first simulator (Functional) was used to gain familiarity with the G-Machine Architecture, and to assist in refining the architecture for hardware implementation. The Bit-slice simulation was a proposed implementation begun in the fall of 1984. The term "Bit-slice" is something of a misnomer as portions of the design are bit-sliced, but not the whole processor. This could better be termed a custom hardware design.

4.1. Alternative G-Machine Designs

With the abstract G-Machine architecture as described in Section 3, it seems reasonable to assume that a general purpose micro-coded computer would lead to a reasonably efficient implementation of the G-Machine. This was considered several times

early in the project, but was not chosen for several reasons.

The Berkeley RISC project (and others such as the MIPS from Stanford [Hee81] have suggested that a relatively simple engine with a few carefully chosen instructions may perform as well as (or perhaps better than) general purpose micro-coded machines. In addition, the design and implementation costs of a RISC should be less than standard machines. One example claimed for this RISC approach occurs when modifications are proposed to the computation model on a RISC machine, i.e. an architecture such as the G-Machine with many complex operations. These modifications can occur almost entirely in software and are an implementation issue, rather than a basic machine design issue. Thus, the RISC approach to thinking of a processor was an influence to the design of the G-Machine PCU.

Yet there are disadvantages in adhering to a strictly RISC approach for the G-Machine. Primary among them is that the bandwidth necessary from the control memory to the processor is larger than that available. This is not only a problem with the G-Machine, but with all RISC-like CPU's. The code latency problems, rather than being directly part of the PCU, are the main responsibility of the separate Instruction Fetch and Translation Unit which performs code fetching, operand handling, and G-Code to PCU instruction translation.

Again, a point stressed in the RISC philosophy is to provide fast, direct hardware support for the most commonly occurring instructions, and to build more complex or less frequently used ones in software. Since the most common operations in the G-Machine are those associated with the pointer (P) stack, that was taken as the basic hardware measure in the execution cycle in the machine. A goal was to have all stack operations occur within one basic machine cycle. This limitation on the machine can

lead to a design with different, independent units, rather than a more tightly interlocked PCU if all operations are tied to the stack cycle.

4.2. G-Machine Processor Characteristics

The Bit-slice design is derived from the VLSI architecture described in the 1984 G-Machine Architecture Handbook [Kie84], and they share the following characteristics:

- RISC-like Instruction Set (84 instructions)
- 32 bit address/data words and ALU
- Data Types: Integer, pointer pair, mixed pair, byte, bit
- One cycle stack operations
- Separate Code and Data Memories
- Separate Execution (PCU) and Instruction Fetch Unit (IFTU)

Each of these points will be discussed in detail further on[1].

4.3. Memories and Host Support

As discussed in Section 3.1, there are two main memories in the G-Machine, the Control Memory (C) and the Graph Memory (G). Since the control memory is never written to by the G-Machine, the program must be loaded by the host processor. The control memory is organized and addressed by bytes (8 bits).

The Graph memory is accessed by addressing each cell of the graph. The lowest bit of the address is either 0 or 1 depending on which cell in the node is being addressed, as shown in Fig. 4.1 below.

[1]One point that should be noted is that the G-Machine PCU is not supposed to perform memory management. Another part of the G-Machine project is devoted to providing parallel garbage collection offloading this work from G-Machine proper [Fos85].

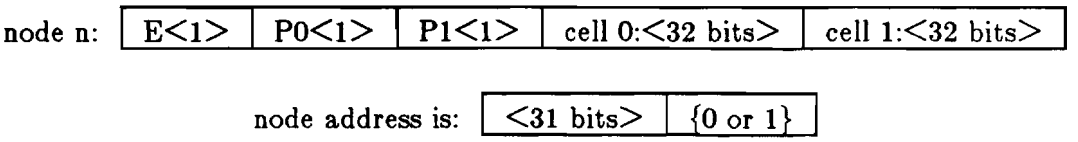


Fig. 4.1 Graph Memory Node and Addressing

NOTE: the number of address bits is implementation dependent.

Addresses to the G-Memory come via the Graph Address Register (GAR). The GAR is like a Memory Address Register (MAR) in many machines. It sits on the GBUS and always contains the current address for any G-Memory operation. A new address must be loaded into the GAR if another graph location must be read/written. This address is then used by the G-Memory for all subsequent graph operations. A new graph address is used by 1) loading the GAR with a new address, or 2) incrementing or decrementing the GAR. The lower bit of the address determines which cell is being accessed.

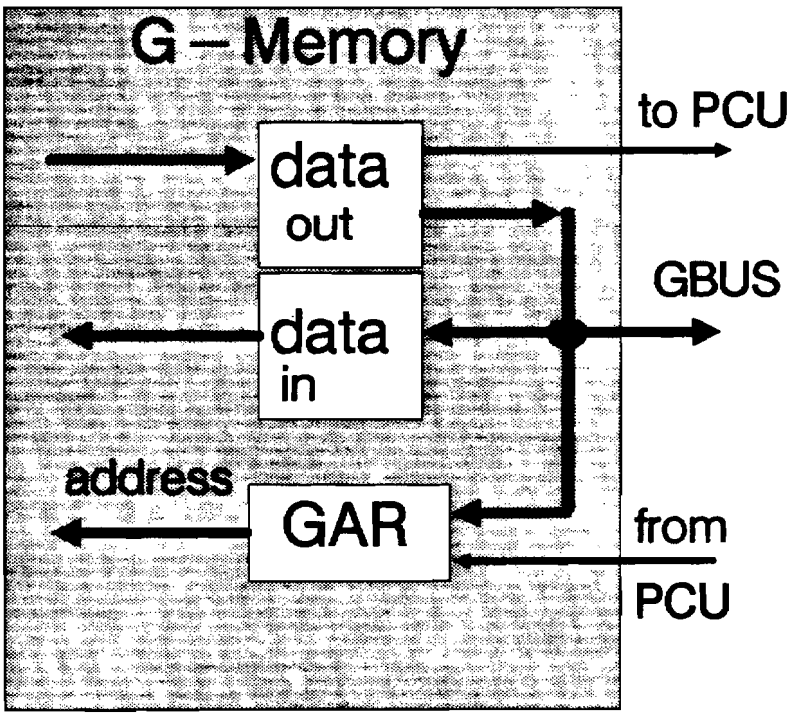
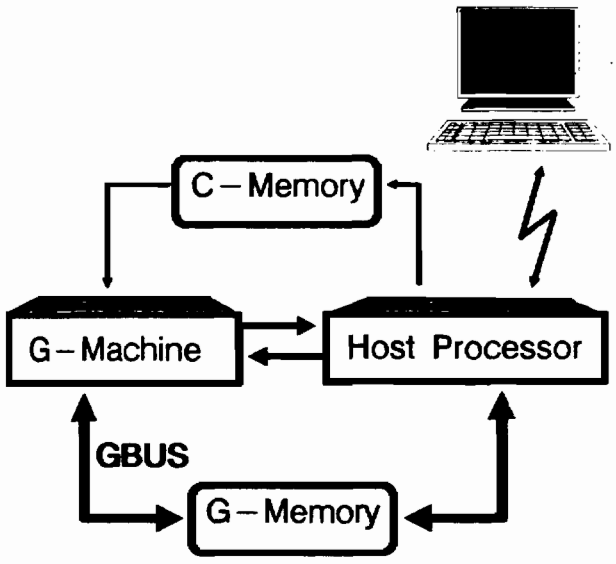
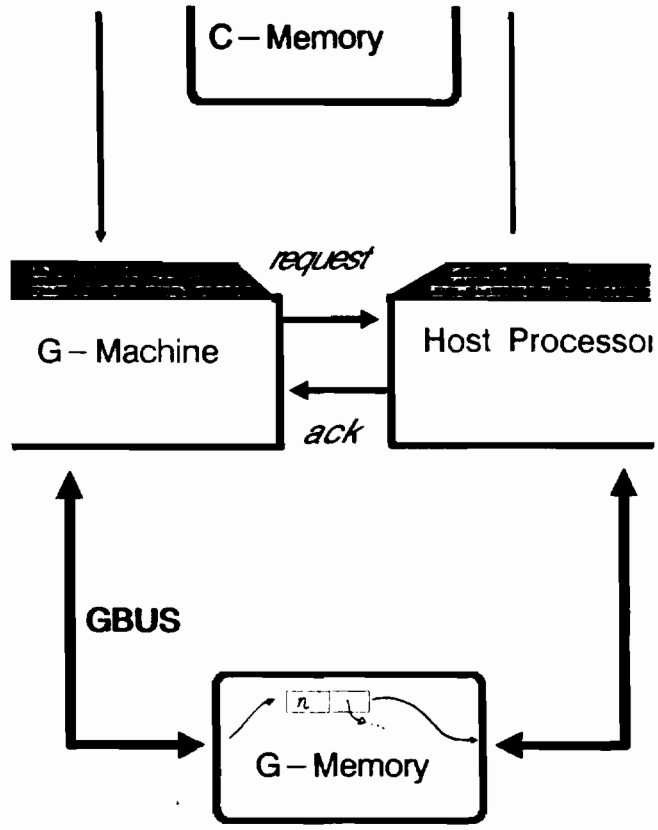


Fig. 4.2 Graph Address Register

In addition to loading the control memory, the Host processor also provides I/O services for the G-Machine via the Graph memory and may additionally provide garbage collection services for the G-Machine and a debugging environment. To request a Host service, the G-Machine signals the Host processor over dedicated lines, as shown in Fig. 4.3a. Both the C and G memories are dual-ported. Simultaneous access to particular memory locations is resolved at the memory level in favor of the G-Machine. Synchronization of data, to ensure correctness, is handled by the software of the G-Machine and the Host.



(a) Machine Implementation



(b) G-Machine Host Service Request

request node n:	E<1>	P0<1>	P1<1>	cell 0: request	cell 1: ptr or data

Fig. 4.3 Host to G-Machine interfaces

The type of service requested is defined in a graph node, the address of which the G-Machine can send on the GBUS bus in the Bit-slice PCU. A simpler scheme not used for this design is to assume a fixed location in the G-Memory as the request area [Kie85]. This then has the added advantage that the Host does not require access to the GBUS.

The G-Machine resumes operation when the Host processor signals it may continue with the *ack* line. In this way, the Host may perform functions without interrupting the G-Machine. Note that no specific machine has been designated as the host processor. Most any micro or mini-computer could serve, given the G-Machine's co-processor interface.

Another benefit that comes from having a host processor in this arrangement is that the G-Machine does not need to handle interrupts. This simplifies the design enormously.

4.4. Machine Instruction Sets

Rather than having a single processor do both the code fetch and program execution, a sub-processor, the Instruction Fetch and Translation Unit (IFTU), is used to translate G-Code into instruction sequences that are passed to the Processor Control Unit (PCU). The rational for this approach was expressed by R. Kieburtz.

> "An internal architecture is the specification for an implementation, and may reflect aspects of design that cater for an intended implementation -- its structure and its technology constraints. In order to achieve a clean separation between the external and internal architecture, the G-machine makes use of instruction translation from its external instruction set to an internal instruction set. This translation is performed at the earliest opportunity -- in the instruction fetch unit, at a stage of the instruction fetch pipeline." [Kie84]

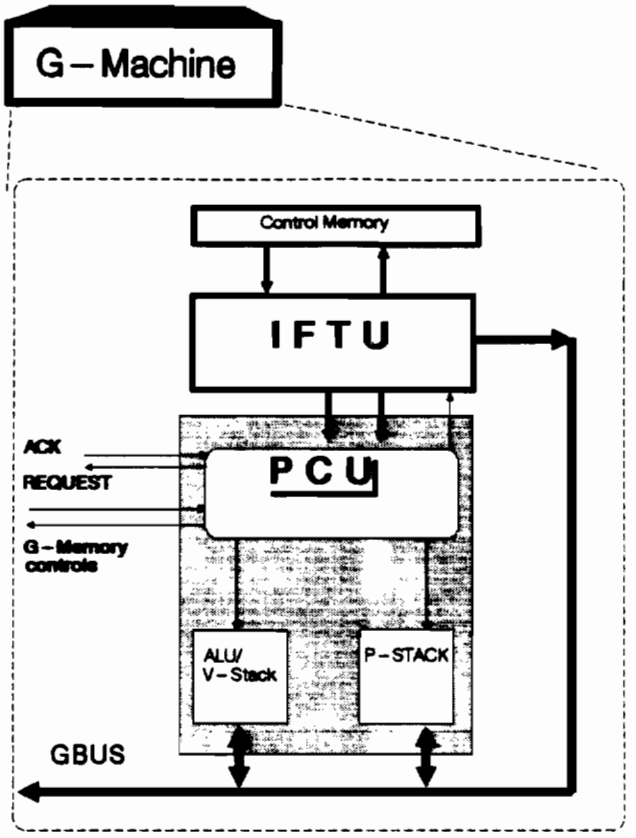


Fig. 4.4 Conceptual Diagram of G-Machine

The instructions are split between the two levels of abstract G-Codes (macro) and PCU (micro) instructions. The IFTU, the path between the levels, fetches the G-Codes from a memory shared with the host processor (see Fig. 4.3a). Each G-Code is expanded to the appropriate sequence of PCU instructions, and sent via a queue to the PCU. In many cases there is a one-to-one mapping of macro (G-Code) to micro (PCU) instruction.

The primary intent of this dynamic translation scheme was to prevent the PCU from being idle for a lengthy period of time. An additional benefit is to separate implementation details from the abstract architecture. Instruction sets can be changed, corrections made, and new G-Codes added easily using the basic abstract machine operation model. Compiler writers can have more function to create the instructions

they desire while being insulated from minor changes in hardware.

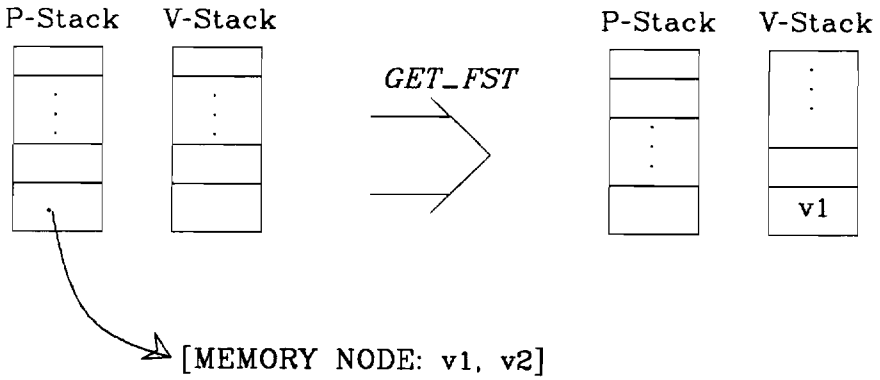
On most commercial machines, a new instruction is implemented by adding a new microcode routine. For the G-Machine, a new abstract instruction is implemented by creating a new sequence of PCU instructions. Since these instructions are not low level microcode, this reduces the problems with traditional microcode changes in terms of timing, AND the hardware knowledge that the person adding the instruction must have.

As an example of IFTU code expansion, we show a simple instruction (GET_FST) that fetches the first value in a graph node to the top of the V-Stack. This involves several steps:

(1) popping the address in the P-Stack to the memory [MVPG0];

(2) requesting a read from G-Memory [begin MVMV]
(3) wait for MEMory ACKnowledge (MEMACK) signal
(4) PUSH the datum read from G-Memory onto the V-Stack [finish of MVMV].

These steps can be described visually as follows:



-or-

$$\langle \mathbf{GET\_FST}.C,n.P,V,n{:}\{v_1,v_2\}.G,E,D\rangle \rightarrow \langle C,v_1.P,V,G,E,D\rangle$$

Fig. 4.5 GET_FST Operation

Each of the steps 1 and 2 above are actual PCU instruction that are expanded from the abstract **GET_FST** G-Code .

```
i072:        goto(GET_FST)$
              .
              .
              .
GET_FST:          MVPG0$
             iftuEND; MVMV$
```

The details of the actual G-Memory interface (MEMREQ and MEMACK signals) are part of the hidden structure of the MVMV instruction. This example is coded in N.mPc's metaMicro assembler [OrR80]. $ indicates the end of an instruction. *iftuEND* is an instruction to the translation unit that this is the end of a particular sequence and it can fetch the next G-Code. The labeled location i072 is in a jump table in which the IFTU uses the value of the G-Code instruction as an index. For sequences, such as **GET_FST** which have more than one PCU instruction, the jump table directs the IFTU to another location for the translation sequence. The IFTU is discussed in further detail in Section 5.

Given the translation structure described, we can consider there to be two types of instructions in the abstract G-Machine Architecture. ***Simple*** G-Codes are ones that have a directly corresponding instruction in the underlying PCU instruction set AND require only 1 IFTU cycle for translation; ***complex*** instructions require more than one IFTU translation cycle.

It should be noted that many of the PCU instructions involve the movement of data within the machine. These could be reduced by having a single move instruction, but is not. This is due to keeping the instructions in 0 and 1 address formats.

4.5. Processor Control Unit

Instead of choosing entirely custom chip solutions, or micro-coding a commercial machine to do G-Machine emulation, a somewhat different implementation of the G-Machine PCU is described. The primary guideline was to design a G-Machine processor using off-the-shelf components. To avoid getting immediately bogged down in hardware, a simulator with timing was built using the N.mPc system from Case-Western[1] [Str78]. This allowed for fairly easy design development and changes, as well as software development and debugging.

The VLSI RISC instruction set in the G-Machine Architecture Handbook was used as the basis for the Bit-slice Instruction Set. The Bit-slice processor was initially conceived of as an attempt to mimic the VLSI architecture with the IFTU and PCU divisions. At the time of the Bit-sliced processor's design, there were still many unanswered questions in terms of basic operations like I/O, memory access, arithmetic conditions and so forth. A new instruction set was designed in conjunction with S. Thakkar that included the specific details and instructions necessary for actual operations. This instruction set and full model definitions for the Bit-slice Machine are included in **Appendix C**.

A block diagram of the Bit-slice system is shown in Fig. 4.6. The Am2910 Microsequencer was chosen as the heart of the PCU [AMD83]. Besides being a stable and well known component, is quite flexible and would allow difficult instructions to be emulated easily. Another reason is that the Am2901 ALU Architecture would be used as the ALU, and the Am2910 fits naturally into using this particular ALU architecture. Other micro-sequencers might be equally suitable. This particular arrangement is a well pro-

[1] This was also a strong recommendation from our NSF reviewer.

ven design [MiB80]

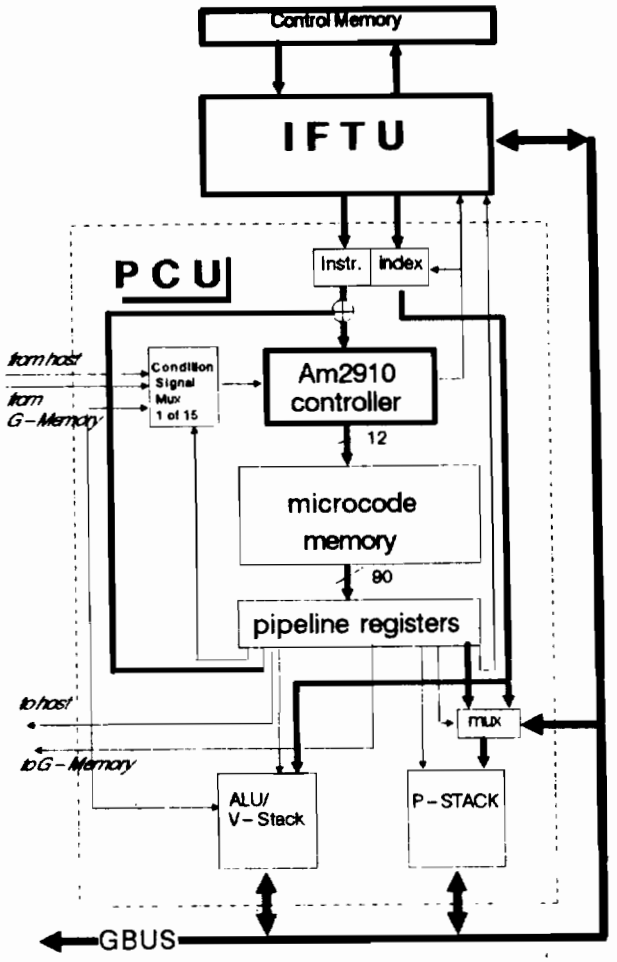


Fig. 4.6 Block Diagram of Bit-Slice System

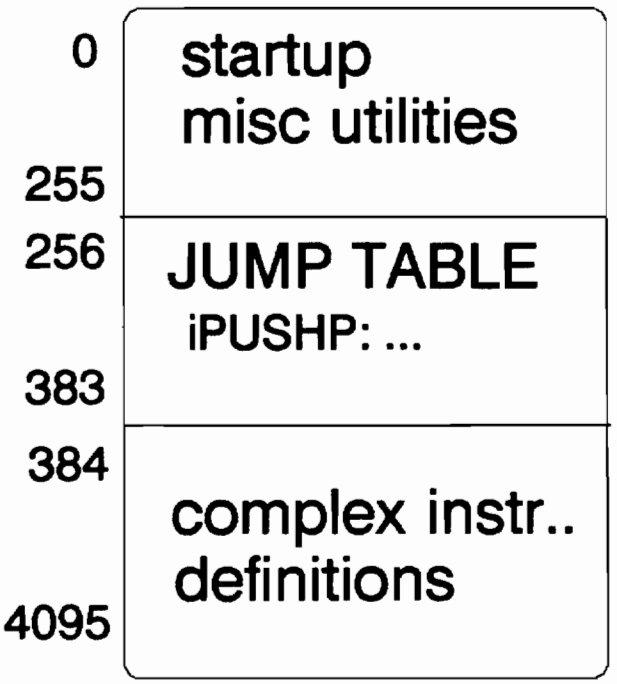
Using the Am2910 now adds a third level of programming to the design. In addition to the G-Code of the abstract Architecture, and the PCU instruction set, the Am2910 must be programmed as well.

Code	Memory
G-Code	*Control Memory (C)*
PCU Code	*IFTU Translation*
μ-code	*for the Am2910*

Code Hierarchy

This use of a conventional micro-control unit is different from the normal RISC approach of a hardwired central processor, and actually seems a step back from the original intention of the early G-Machine design proposal to a micro-coded design. However, the split in functionality shown in the early proposed design of separate IFTU and PCU units was followed. In fact, the programmed Am2910 PCU is very much a black box to the IFTU. The processors carry out simple operations and have well defined unit interfaces.

Instruction decoding is done differently with the Am2910 based PCU than in conventional micro-processors. A standard method is use a ROM to decode the micro-unit instructions. The ROM is placed in the instruction stream before the micro-processor. When a new instruction is fetched, the ROM decodes the operation code into an address in the μ-code that the micro-processor uses to fetch for it's next operation. This extra level of mapping is costly in terms of time and programming the extra ROM. In the Bit-Slice G-Machine, the 8-bit PCU instructions are passed to the processor, as shown in Fig 4.6, and are used **directly** as an address in a jump table in the μ-code.



μ-word organization

μ-op	instr	...*other bit fields* ...	na

Fig. 4.7 Micro-Controller Memory Organization

When the instruction in the jump table is fetched, the **μ-op** field contains Am2910 instructions. If the PCU instruction is completed, then the next PCU instruction is fetched. But if the PCU instruction requires more operations, then the next address field (**na**) is used to fetch the next μ-word from the complex instruction definitions.

This method is quite fast, especially as many PCU instructions usually only require one micro-instruction. A jump table approach was taken with the Series 37 HP 3000 [Ame85] for similar reasons of speed and simplicity. Elimination of the ROM decoding speeds up the circuitry and reduces the time it takes to get an address for the micro-code. Performance is not affected because the IFTU is operating in parallel to fetch the next instruction. The overlap eliminated the need for an extra decode cycle.

The timing scheme of the Bit-slice processor is based on four clock pulses, as shown in Fig. 4.8 below. The AMD clock chip (Am2925) provides these clock pulses from a

basic oscillation pulse.

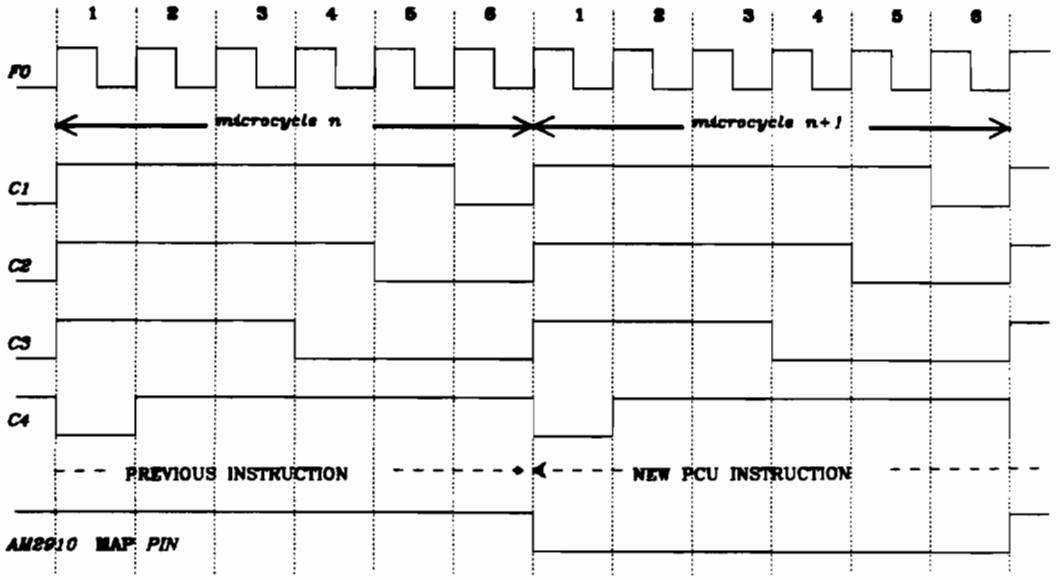


Fig. 4.8 Processor Clocks and Timing

One of the housekeeping aspects of the Host processor is the loading of the μ-code memory. This is performed by having the μ-code pipeline register loaded (via a serial shift line, or snake), and then writing to the μ-code memory, the reverse of normal operations. The μ-code memory address is controlled by bi-directional latches from the GBUS to the μ-code address lines from the Am2910, which are tri-stated (with the OE pin of Am2910) for initialization. The bi-directional latches also serve in debugging by enabling the host to watch the address lines. In addition, there are the two snake lines in the machine for debugging and initialization: one for the μ-code and one for the rest of the pipeline registers.

Testing of condition codes and other signals is implemented by a line fed into a 1-of-16 MUX [TTD84] and the output into the Am2910 CC (Condition Code) input. For instance, a Graph Memory read is controlled by waiting for the acknowledge line (MEMACK) from the Graph Memory. The PCU polls this line waiting for the signal, and can then read the value off of the GBUS when it becomes available. When the

PCU lowers the MEMREQ line, the value has been fetched and the Graph memory can stop driving the GBUS and the MEMACK line. Other condition codes come from the four ALU condition codes, described in the next section.

4.6. Bit-Slice ALU

In the Bit-slice Instruction Set, the most time critical instructions are the Arithmetic and the Control Transfer instructions. As mentioned in the previous section, the Am2901 ALU Architecture was chosen for the ALU unit. The 2901 ALU architecture naturally provides most of the instructions necessary for G-Machine ALU operations [AMD83], and can be programmed to do more complex operations such as multiply, divide and bit shifting. It also allows for easy testing of different conditions for the Control Transfer instructions[1].

Part of the problem in selecting an ALU was that there are not, and there may not be for some time, any single devices that easily performed like the G-Machine's abstract ALU. The 2901 was able to perform most of functionality save for having the V-Stack as a true dynamic shift register stack, like the P-Stack.

For this purpose, emulation of the abstract V-Stack consisted of an up/down counter (shown in the schematic in Fig. 4.9 below as the 74LS169A [TTD84]) that is used to access the components of the RAM in the ALU Slices. The RAM is treated as a fixed stack with a moving top.

---

[1] with the Am2904 Status and Shift Control Unit.

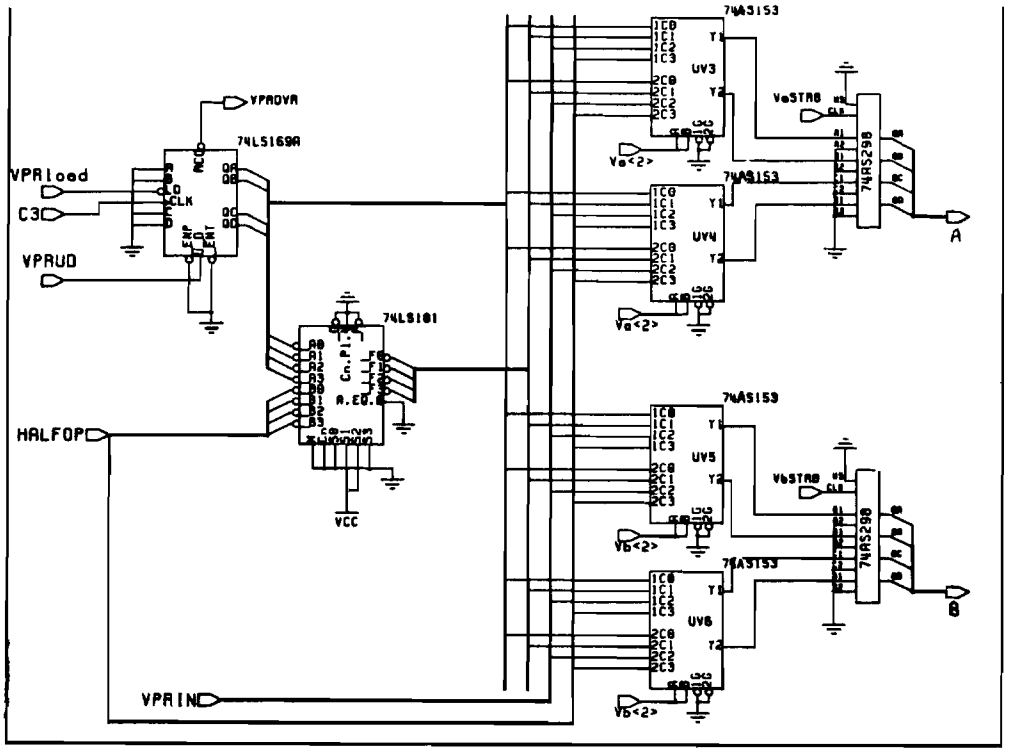


Fig. 4.9 Schematic of V-Stack pointer

In addition to the V-Stack pointer, several other inputs can be used to address the **A** and **B** inputs to the 2901 RAM. These are shown in the schematic of Fig 4.9 and include

- o the result of a small subtractor unit which is used for ROTATES and MOVES (the 74LS181)
- o the operand for the instruction currently being executed such as COPYV, ROTV, MOVEV, etc. (HALFOP input)
- o a value from the microcode (VPRIN input).

For the actual implementation, the IMI 4x2901 16-bit Arithmetic Units were chosen as the ALU Units[1].

[1] The 4x2901 models were used rather than original Am2901's simply because of the need for a 32 bit ALU. Two chips would be would be more reliable/testable/etc. than the 8 required with Am2901's. Also, the 4x2901 is about 20% faster.

The ALU RAM cells are organized as follows:

15	*D-Pointer*
14	-
	...
1	-
0	-

RAM location 15 is used as the *Dump Pointer* when the abstract instruction set uses the dump model. Otherwise, the V-Stack is allowed to wrap around to 0. Testing for wrap-around is provided by having the **VPROVR** signal from the up/down counter fed to the 1-of-16 PCU control MUX.

The V-Stack, although designed to be fast, consumes a major portion of the PCU μ-code and operation time. It should be built with the same principles as the P-stack, discussed in the next section, which would require a more custom designed ALU. The Bit-slice PCU essentially emulates the V-Stack by using the RAM on board the 2901 ALU slices.

Other ALU operations, in particular the shifts, multiplies and divides, are slow as they are performed using μ-code routines. A barrel shifter should be used to increase speed, and would fit in nicely with a less integrated ALU. For other multiplies and divides, perhaps a dedicated unit, like a floating-point co-processor, could be added.

Still, for basic integer operations such as ADD,SUB,INC and so forth, the ALU/V-Stack is very fast with the minimum time for any operation being 2 machine cycles. Figure 4.10 below shows the ALU timing in detail.

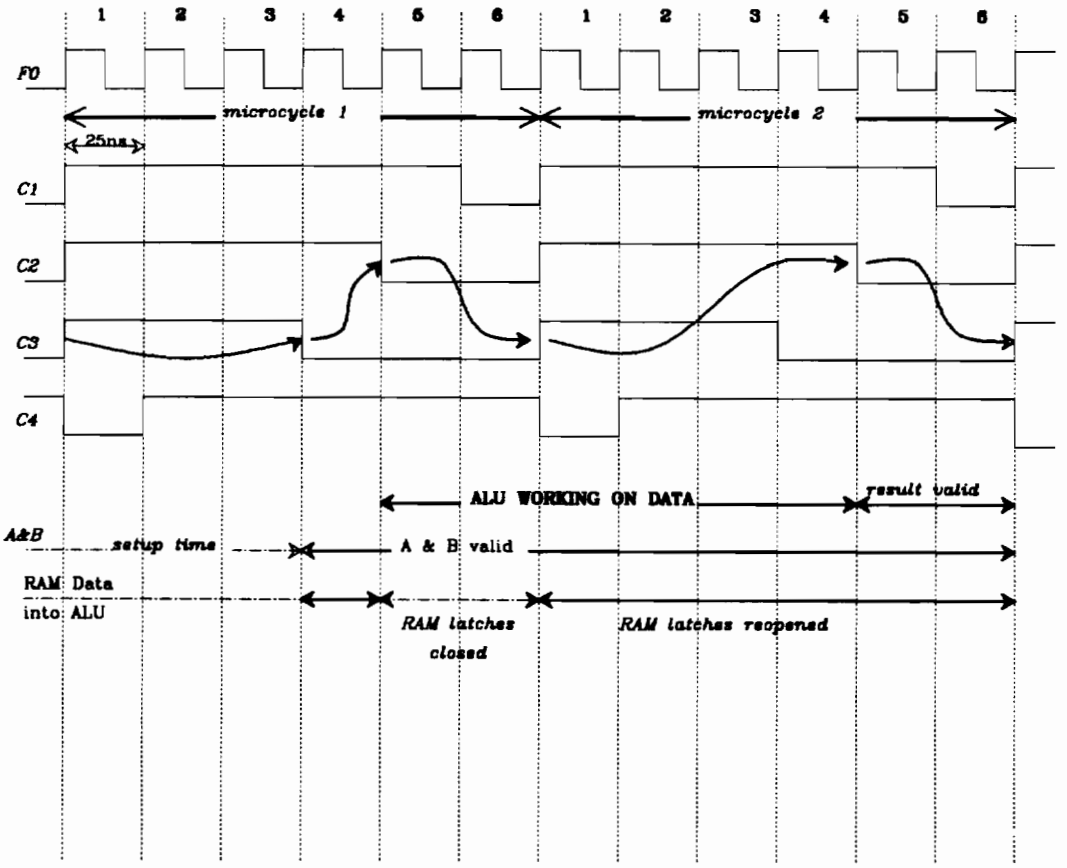


Fig. 4.10 Bit-slice ALU Timing Operation

During the first cycle, the A&B addresses to the ALU RAM's are computed. When C3 drops, the value from the RAM or other inputs to the ALU block in the slices are valid. The ALU circuitry then has 25ns before C2 drops to grab the data. The RAM latches reopen when C2 rises again. The ALU thus has from the C2 fall in μ-cycle 1 to the fall of C2 in μ-cycle 2 to compute (150ns). This is more than enough time for most ALU circuitry (the AMD 2901's require about 32ns). The final result is written back to the RAM in the last 50ns of cycle 2.

The condition codes for the ALU are stored in the Am2904 Status and Shift Control Unit (see Fig. 4.9). The 2904 is "designed to perform all the miscellaneous

functions which are usually performed in MSI around (a 2901) ALU." [AMD83] The four condition codes (Z,N,C,O)[1] for the Bit-slice G-Machine are kept in the 2904's μ-status register (μSR). The condition codes are set each time an arithmetic operation is performed or a new element is pushed onto the V-Stack. Jump instructions are defined in terms of whether particular bits are set accordingly. For instance, the iJNOTNEG instruction will branch only if the N bit is not set. The bits that each jump instruction tests are defined in **Appendix C** Sec. 3.7.

In addition to the μ-status register, the 2904 has a similar machine status register (MSR). The MSR is used to hold the current **E** and **P** bits (See Section 4.2 on Memories and Host support). The **E** bit is only set by an UPDATE instruction, and the **P** bit is set whenever a move from the P to the V-Stack occurs.

4.7. Bus Model

To provide data movement between the various components of the Bit-slice G-Machine, there is a local bus, the **GBUS**, that is connected to the following PCU and G-Machine units.

The GBUS is a 34 bit wide bus. 32 bits are for pointers and data, the upper two bits <33:32> are used for the graph node **E** and appropriate **P** bits respectively. The following table lists the machine units and how they are connected to the GBUS.

[1] Z = Zero bit, N = sign bit, C = Carry bit, O = overflow bit. See the Am2901 definition in [AMD83]

Units	GBUS bits	Connection to GBUS
ALU	0:31	at Y outputs and D inputs of 2901's
2904 Status Unit	32:33	Z,N,C,O outputs; Condition Codes, E and P bits are kept here
Graph Memory	0:33	All 34 bits†
P-Stack	0:31	in two slices, 0:15 and 16:31
P-Stack mux	0:3	lower 4 bits for indexed operations See Section 4.7 or Fig. 4.11
GAR	0:31	Graph Address Register
Literals Queue	0:31	in IFTU; for constants from the code See Section 5.
Host	all 34 bits	(for debugging)

† via a separate port on the G-Memory

This GBUS model allows the following logical data movements within the G-Machine:

(from)	(to)
• Constants in code	→ P and (via ALU) V Stacks
• Values in P&V Stacks	→ Graph Memory
• Values in P&V Stacks	→ GAR
• Values in P&V Stacks	→ Dump (Graph Memory)
• Values in V Stack	→ P-Stack index
• Values in Graph Memory	→ P&V Stacks
• Values from Graph Memory	→ Host processor

The G-Machine Bus (GBUS) in the Bit-slice engine is controlled at all times by the PCU; that is, the PCU directs the operation of every unit on the bus, with the exception of the G-Memory.

The PCU performs a G-Memory read/write operation by first placing the address of the desired graph node in the GAR, then raising the **MEMREQ** lines (MEMREQ consists of two signals lines. See **Appendix C** Section 4). During a read operation, the data is requested and the PCU waits until the **MEMACK** line from the G-Memory signals. The PCU then captures the data in the appropriate location, lowers the request lines, and continues.

During a write operation, the PCU raises the **MEMREQ** lines to signal a write and places the data onto the GBUS, holding it there until the G-Memory raises the **MEMACK** line signaling that it has captured the data. Note that this may halt the PCU while it waits for the G-Memory to perform an operation. The G-Memory must therefore latch the data as quickly as possible in order to avoid slowing the processor. For simulation purposes, the conceptual model of the G-memory operation assumed that the G-Memory would latch and respond to the PCU within the next cycle on a write, and within 2 cycles on a read. It takes less time on a write because the G-Memory can perform that operation while the processor does another. A design of the G-Memory [Ran86] would boost this to 4 cycles. See Section 6 and **Appendix D** for how this would affect the performance of the G-Machine.

A confusion arose on the Bus Model at first because the Architecture Handbook initially had a static bus [Kie84]. The PCU would write to the bus with one instruction, then read from it with another. Question: Would the bus be a register, or would two instructions have to be overlapped to execute at the same time? This issue was finally resolved by having transfers from one unit to another be individual instructions, as in many other machines. The instruction set for the Bit-slice design has separate instructions to move values from one location to another. These are the two move groups shown in the Instruction Set in **Appendix C**.

4.8. P-Stack Implementation

The stacks of the G-Machine are not simple stacks, as was shown in Section 3.2. Shifting the stacks, and providing the stack addressing required for such operations as COPY and ROTATE, would be quite expensive if performed via TTL hardware. In order to implement the complex stack operations efficiently, several custom VLSI P-

Stack chips were designed at OGC [BaR85, TaN85, Vir84]. Although the implementation description of the P-Stack has changed somewhat with the addition of the Backdoor (discussed later), for this design we can assume that the hardware for this unit is available. Having the stack available as an existing component made the design much easier, as shown in the P-Stack schematic of Fig 4.11.

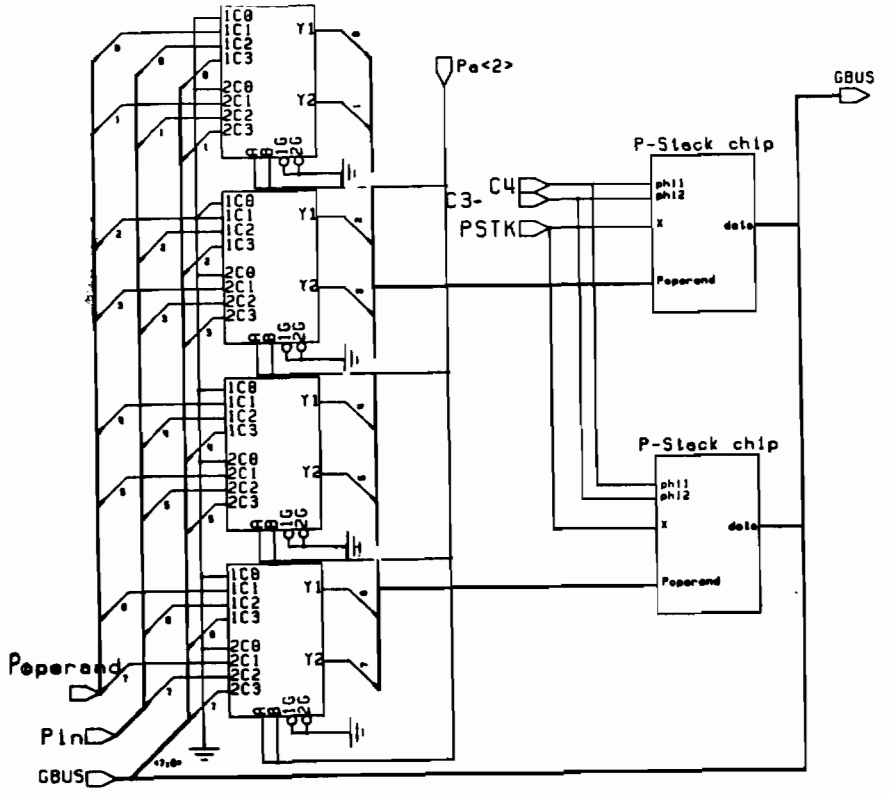


Fig. 4.11 P-Stack schematic

As shown in the schematic above, operands for the P-Stack (Poperand in the block) can come from

(1) G-Code (*Pin*),
(2) RISC codes (*Poperand*),
(3) microcode (*Pin*), or
(4) the GBUS (i.e to provide VROTP instruction index from ALU).

The idle operation of the P-Stack chips is a ROT(ate) 0. This was provided for by having the default operand (shown as C0) of each selector be 0.

4.8.1. P-Stack timing

One fundamental design goal was to ensure that the P-Stack operations take place in one basic machine cycle, however long such a cycle might be as many of the instructions in a program are the stack instructions. It has been observed that 20-25% of the G-Codes in the RTL simulator [Sar84] were stack instructions and an average of 23% in the Bit-Slice machine (ranging from 16% to 34%). In addition, *30%* of all Bit-slice PCU instructions were basic stack operations. This is one of the main reasons that custom P-Stack chips were built. Mimicking the functionality with TTL RAM's and counters would not be practical. It would have resulted in a much slower machine, or would have required a different clock scheme and higher performance (and more costly) TTL components to achieve the same speed as the custom P-Stack chip.

The basic functions of the P-Stack were discussed in Section 3.2. The P-Stack chips built by J.Bailey and L.Rankin at OGC [BaR85] perform all of the basic stack operations (PUSH,POP,COPY,ROT,MOVE), and were used as the basic hardware model. They have the following operating characteristics:

Stack Chip Characteristics

clock	two-phase non-overlapping
cycle time	min. 60 ns
data ready for a POP	start of phi2
data ready for a PUSH	start of phi2
width	16 bits
depth	24 words

Since the stack chips require a 2 phase clocking scheme not available with the standard PCU clocks, an extra clocking circuit is used that generates the requisite phi1 and ph2 signals. This circuit is shown in **Appendix D:** Bit-Slice G-Machine Schematics.

With the basic PCU cycle time of 150 ns, the stack chip easily fits into the G-Machine operational requirements. A later chip designed by M. Tamjidi and B.Nader [TaN85] also performed all of the stack operations, and had a backdoor as well (the backdoor is explained in Section 4.8). The timing diagram in Fig. 4.12 shows how the stack interfaces with the G-Bus for reading and writing data.

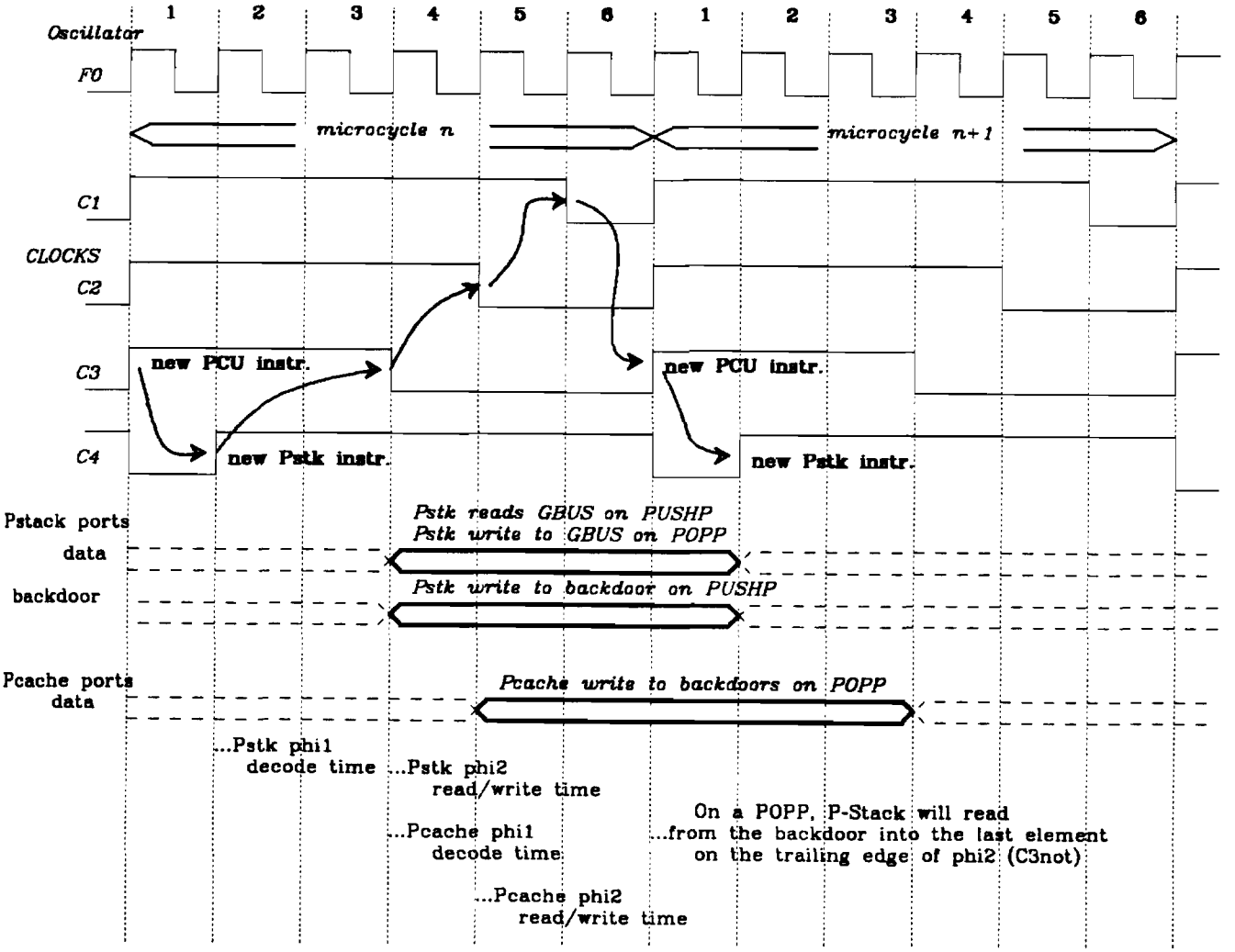


Fig. 4.12 P-Stack Bus Interface and Timing Diagram with Backdoor Operations

4.9. Context Switching

It was mentioned earlier that the Dump (D) memory was no longer a part of the design. The primary purpose of the Dump was to support *context switching*, which in the G-Machine model involves writing out the current contents of the P-Stack to the Dump and leaving the arguments to the new function on the top of the stack.

Context switching is a cumbersome and time consuming process, especially as opposed to that of a standard register based machine. The bottom of the P-stack must be rotated to the top, then written to the memory along with the information to restore it later. Context switching was noted to consume over 10% of the G-Code instructions and execution time in the RTL simulator [Sar84]. This was confirmed in the Bit-Slice simulation (10-14% average). In addition the number of PCU instructions used for context switching sometimes comprised up to *60%* of the instructions executed on some small programs with strict lazy evaluation (See **Appendix A**). This overhead can be accounted for by the fact that the test cases did not use very large functions, and so context switching tends to be the dominate activity. Since the cost of context switching is linear with respect to the number of function arguments, functions with more computational actions would see a decrease in the context switching overhead.

The context switching mechanisms described here are those developed to support the LML G-Machine instruction set where the basic problem is to suspend the current P-stack contents and construct a context on the stack for the newly invoked function. The stack is a finite resource, and so a problem arises regarding where to hold the contents until control has returned to that context.

The method initially used was to save the P-stack to a Dump Memory (D). For practical purposes, the high address end of the G-Memory was used as the Dump. This avoided the additional cost of another memory which would remain unused most of the time. If we consider the example where the top element is the argument to the new function, and the P-stack is currently ***m*** deep, then the elements numbered 1 thru ***m*** must be saved to the Dump. They are written out in the following order:

$$\langle C, n_0.n_1.n_2...n_m.P, V, G, E, D\rangle \rightarrow \langle C, n_0.P, V, G, E, n_1.n_2...n_m.D\rangle$$

The current G-Code Program Counter (*PC*) and the number of items saved (*items*) are then written to the Dump.

$$<C, P, items.V, G, [PC]+E, D> \rightarrow <C', P, V, G, E, items.PC.D>$$

This information is used on the function return to direct the restoration of the contents. Note that the count of the items saved comes from the V-Stack. This counter was incremented in the first step when the items were saved to the Dump. The Program Counter comes from the IFTU.

Restoration occurs in reverse order, leaving the RET instruction index element on the P-Stack.

$$<RET\ r.C, n_0 \ldots n_r.P, V, G, E, items.PC.n_1.n_2 \ldots n_m.D>$$
$$\rightarrow <C, n_r.n_1.n_2 \ldots n_m.P, V, G, [PC]+E, D>$$

where **r** is the index of the returned item on the stack

Rather than going through the overhead of writing out to the memory, and the complexities of rotating and managing the elements on the P-Stack, it would be nicer to just keep pushing new items onto the stack. This is where the "backdoor" and overflow stack described earlier come in.

The "backdoor" provides an elegant answer to the context-switching problem by having the P-stack overflow through a "backdoor" to a fast memory which serves as an overflow stack. This has proven in simulations to reduce the cost of context switching to be negligible in terms of execution time.

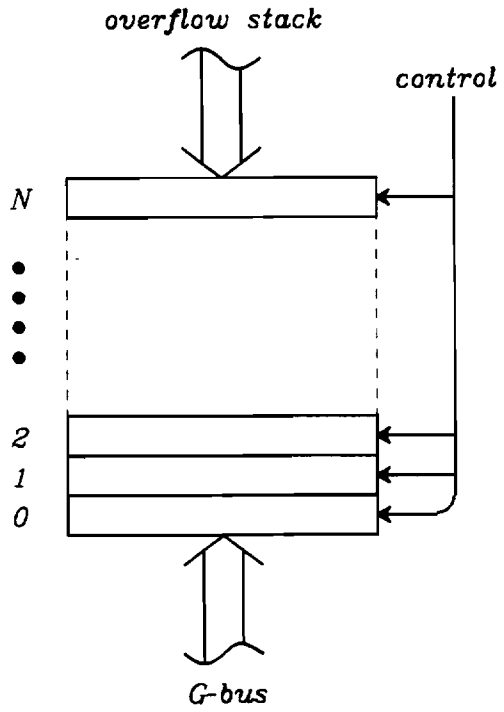


Fig. 4.13 P-Stack model with overflow

The overflow stack reduces the overhead of context switching as the information in the P-stack is not needlessly shuffled around. It also helps eliminate the problem of the stack overflowing. There are practical limits, of course, since the overflow memory is a hardware stack. For simulation purposes **only** 1K of overflow memory was allocated. This might be insufficient for larger recursive programs, but was more than enough for hardware test purposes.

The P-Stack overflow memory is provided in the simulator as a behavioral unit with a 1K overflow. Access to the backdoor memory from the P-Stack chip was shown in Fig. 4.12 (labeled as either backdoor, or 'pcache'), which shows the overall P-Stack interface. Assuming as before that r is the RET instruction index, then when a RET is performed, the machine looks as follows:

$$<\mathrm{C}, \mathrm{n}_r.\mathit{items}.\mathit{PC}.\mathrm{n}_1.\mathrm{n}_2 \ldots \mathrm{n}_m.\mathrm{P}, \mathrm{V}, \mathrm{G}, \mathrm{E}, \mathrm{D}> \rightarrow <\mathrm{C}, \mathrm{n}_r.\mathrm{n}_1.\mathrm{n}_2 \ldots \mathrm{n}_m.\mathrm{P}, \mathrm{V}, \mathrm{G}, [\mathit{PC}]+\mathrm{E}, \mathrm{D}>$$

The simplified context switching mechanism reduces the number of internal PCU instructions necessary for a context switch up to 10% for some small programs with lots of lazy evaluation. Execution time is correspondingly reduced upto 24% due to eliminating the reading and writing of values to the Dump (G-Memory). The tradeoff for achieving this increased performance is in having enough overflow stack memory to handle programs with large context depths.

4.10. PCU Design Summary

The Bit-Slice design closely follows the early design proposals of a simple PCU and separate IFTU. The IFTU is used to expand the control memory bandwidth to the RISC-like PCU. An Am2910 serves as the heart of the PCU, providing an efficient implementation similar in nature to the early PCU design.

A summary of the G-Machine Bit-slice PCU characteristics is given below.

Processor	Am2910-1 Microprogram Controller 90 bit control store word Micro-control store instr. time 150ns. Control Store size 4K (90 bit words) Writable Control Store (RAM)
Data path	32 bits
PCU Cycle	150 ns.
Clock	Crystal Controlled Am2925 (or equivalent)
ALU	32 bit Am2901 Architecture
Data Types	Integer, pointer pair, mixed pair, byte, bit
P-Stack	Custom CMOS VLSI min. 60 ns cycle time
Overflow Stack Cache	1K (for simulation)

Fig. 4.14 Bit-slice Processor Characteristics

5. G-MACHINE INSTRUCTION FETCH AND TRANSLATE UNIT

The Instruction Fetch and Translate Unit (IFTU) is a a primary component of the G-Machine. This unit fetches G-Codes from the Control Memory store, and translates them into PCU instructions. The intent of the IFTU is to achieve a low latency in the instruction pipeline when the instruction flow is interrupted by control transfer instruction. It also serves to increase the bandwidth of the instruction stream by dynamic translation of G-Code to PCU instructions.

This section describes in detail the IFTU/PCU interface developed for the Bit-slice G-Machine PCU, the simulated behavior of the IFTU, and attempts to predict the performance of an actual IFTU implementation. The primary focus of this design effort was the PCU, not the IFTU. As such, only a behavioral description of the IFTU was developed and not a complete hardware design as was done for the PCU.

5.1. PCU/IFTU Interface

The interface consists of four control signals from the PCU to the IFTU, and two 8 bit lines from the IFTU to the PCU:

PCU/IFTU Interface Signals		
PCU	**MAP**	PCU fetching next instr.
to IFTU	**JUMP**	Last PCU jump was taken
	NOJUMP	Last PCU jump was NOT taken
	LITERAL	Deposit top of Literal Queue to GBUS
IFTU	**PCU Instr<8>**	Next PCU instruction
to PCU	**Index<8>**	Next PCU instruction index

The **LITERAL** control line is disjoint in operation from the other PCU lines, as it has no effect on the instruction stream. The PCU can assert the **LITERAL** line at any time, using the GBUS to fetch a literal value from the IFTU Literals Queue. The other three signals control the IFTU operations.

PCU Signals to IFTU			
MAP	**JUMP**	**NOJUMP**	
0	0	0	PCU fetching next instruction/operand
↑[1]	0	0	PCU executing next instruction.
1	↑	0	Jump was taken by PCU. IFTU flushes buffers and translates next instr. sequence.
1	0	↑	Jump was not taken by PCU. IFTU can continue with current buffers.

Because the **LITERAL** line can be asserted immediately after the buffers are flushed, for example on an external jump, the value deposited will be invalid (garbage) because the Literals Queue is flushed as well. Care must be taken to see that this condition never occurs.

5.2. IFTU Pipe Stages

The two pipeline stages of the IFTU involve the fetching of bytecodes from the Control Memory (G-Codes) and the queue of instructions to the PCU. This is shown in the block diagram of the behavioral unit used for simulation.

[1] A ↑ indicates a rising edge.

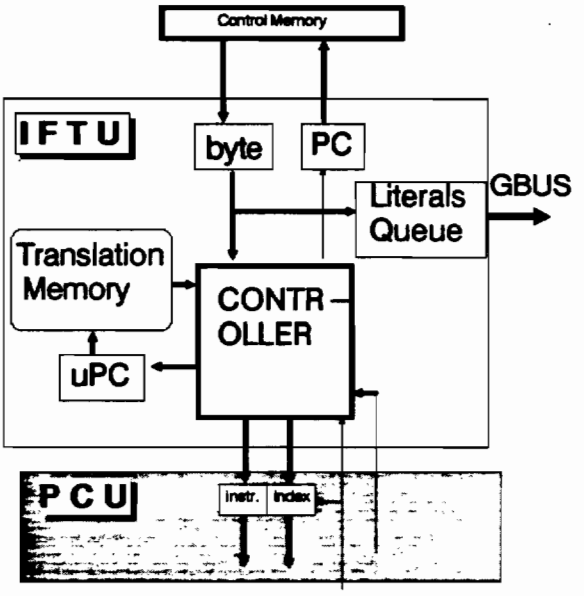


Fig. 5.1 Simulated IFTU Block Diagram

This block diagram of the IFTU shows the pipe stages as the PCU instruction queue and the G-Code instruction buffers. Each pipe stage is only 1 instruction deep in this model. The costliest operation occurs in the case when both the PCU queue and the external instruction buffers must be restarted. The PCU would sit idle waiting for the PCU instruction queue to fill up through both pipe stages.

One technique to help avoid idle PCU cycles due to G-Code jumps was designed by R. Kieburtz. This entails the use of multiple external instruction buffers, as shown in the figure of the IFTU below [Kie84, Kie85a]. If a conditional jump is encountered, the current active buffer continues on the current instruction path. Assuming the jump taken, and a second buffer is activated to fetch instructions from the taken path. If the jump is taken, the current buffer is switched to the path, otherwise control continues with the current buffer. Prefetching is performed only one jump ahead, to avoid the complexity of following many possible paths.

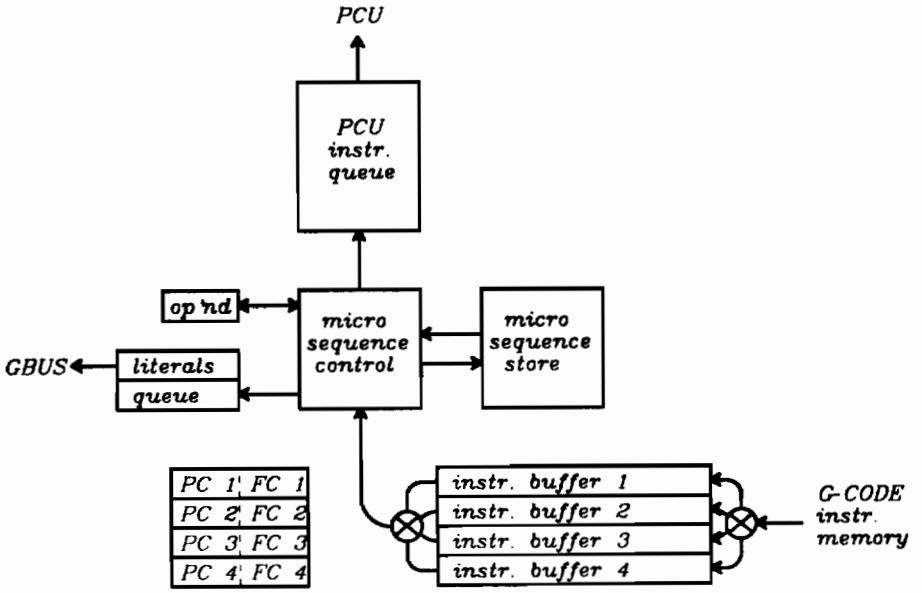


Fig. 5.2 Proposed IFTU Block Diagram

The restart time before the PCU receives new instructions becomes the time it takes to 1) switch to the new active buffer or fetch the proper G-Code to a new buffer; 2) translate the G-Code to PCU instructions; and 3) send the PCU instructions through the PCU instruction queue.

An obvious way to help reduce the restart time is to employ as many buffers as possible, so that when a switch occurs, there is no need to incur the cost of fetching the appropriate G-Code from the Control Memory. Four buffers are planned for, as shown in Figure 5.2. This number was based on the assumption that most selections by CASE instructions are small (see Section 5.3), and that they will generally tend to select the first few conditions. Should the assumptions not turn out to be true, the number of buffers would have to be increased in the eventual IFTU. For simulation purposes, only one external buffer was used.

The final IFTU pipeline stage is the PCU instruction queue which acts as a buffer, since not all PCU instructions take the same amount of time. The PCU queue is there-

fore a mechanism that the IFTU can use to keep ahead of the PCU while it performs other operations, such as absolute jumps and literal fetches.

5.3. CASE Support

Generalizing the external buffer scheme to provide several possible paths was supposed to allow direct support of a CASE instruction, i.e. selection based on a list of possible results. In contrast to the conditional jump, there is no way to know which path will be taken until the CASE condition is evaluated, so the translation sequence must be halted. Only after evaluation of the CASE condition would the IFTU switch control as soon as possible to the correct buffer.

A CASE instruction [Kie84] can be made possible via the following modifications to the design. First, if we consider the CASE as essentially a condition and a jump table of Control addresses, then the IFTU needs the index of the element to use as an address. This requires an index from the PCU back to the IFTU telling it which path to take. If the index is one of the already activated instruction buffers, then switching is fast, otherwise another fetch from the Control memory must be done in order to get the correct address in the CASE jump table.

The CASE instruction provided in the abstract architecture is somewhat different from the other instructions, in that it requires special handling by the IFTU.

$$<\text{CASE } m.a_0 \ldots a_i.C, P, i.V, G, [a_0{:}C_0, \ldots a_m{:}C_i]+E, D>$$
$$\rightarrow <C_i, P, V, G, E, D> \quad \{\text{if } 0 <= i < m\}$$

The index for the CASE comes from the V-Stack[1]. The IFTU can have special

[1] via a previous instruction that performed an operation and left the value there.

dedicated lines for the index, or it can read the GBUS for the index value. Because of the hardware support it requires, and because it does not fit well into the format of the other G-Machine instructions, the CASE instruction has been dropped from the new architectural description [Kie85a].

5.4. Code Translation

Translation of G-Codes to PCU instructions is an interesting problem, since there is more involved than a simple macro mapping. This is obvious from the examples of the CASE and EVAL instructions (See Section 3.4 for an explanation of EVAL). The IFTU must provide efficient translation of G-Code during multiple branchings, loops, and other external control flow operations, as well as support its own internal operations. As such, it is one of the most important elements in the execution performance of the G-Machine, in addition to the stacks and G-Memory.

In order to meet the functional performance requirements of the Bit-slice PCU, the behavioral simulation model of the IFTU, shown in Fig. 5.1, consists of:

(1) A translation sequence controller;
(2) A translation memory containing internal IFTU instructions and PCU instruction sequences;
(3) Access to/from Control Memory (C) and single external G-Code buffer;
(4) A PC to address the C Memory, and a μPC for addressing the translation memory;
(5) A stack of 5 words deep to hold the μPC on internal IFTU procedure calls;
(6) Two pipeline registers to the PCU for PCU instructions and indices;
(7) A 32 bit literals queue fed by the IFTU and connected to the GBUS.

The translation sequencer is a state machine with 7 instructions. These internal micro instructions enable the IFTU to perform branches, procedure calls and operand handling independently (asynchronously) of the PCU.

IFTUCODE		MEANING
000	iftuCONT	goto next instruction in sequence (default)
001	iftuEND	end of translation sequence. get another external instruction.
010	iftuJMP	jump absolute to location, use address/index as absolute address.
100	iftuINDEX	use the next literal on the literal queue as the index for this PCU instruction.
101	iftuJMPSTK	jump to next IFTU address, saving μPC on local jump stack.
110	iftuRTN	pop μPC from local jump stack.
111	iftuNULL	error. this is an unrecognized external instruction[1]

Fig. 5.3 IFTU Translation Sequencer Instructions

IFTU translation sequencer instructions are embedded with each instruction in the translation memory. Each memory location is provided with 1 byte for the PCU instruction, 1 byte for an index, and 3 bits for the IFTU instruction. This is shown in word 1 below.

word 1

	IFTU ADDRESS/INDEX	
IFTUCODE	PCU INDEX	PCU INSTRUCTION
3	8	8

word 2

<unused>	IFTU JUMP ADDRESS
3	16

Fig. 5.4 IFTU Translation Instructions

A second word (word 2) is used in the IFTU for local jump instructions. It uses the PCU Index and Instruction in word 1, and the next sixteen bits of word 2 as a jump transfer address, giving a full 32 bits for addressing the IFTU translation memory shown in Fig. 5.1. (This rather large addressing capability is not necessary, as was first

[1] The iftuNULL instruction is provided, but may not be available in an actual implementation.

thought. The actual implementation will only need the 16 bits of word 1.)

G-Code translation is accomplished using the G-Code as an index into a jump table in the translation memory. The reasons for this jump table method are similar to those for the the PCU μ-code: Efficiency and simplicity (no special translation hardware is required). See Section 4.5 concerning PCU μ-code.

G-Code Instructions that require only 1 PCU instruction are translated in 1 IFTU cycle and sent to the PCU queue. More complex G-Codes can be composed into multiple PCU instructions, and therefore start providing the PCU instruction queue in 2 IFTU cycles (1 to look up in the jump table, and another to jump to the location and fetch the instruction).

For simulation purposes, the behavioral model of the IFTU was written with only 1 level of pipelining for both the PCU instruction queue and the external G-Code buffers. A more detailed simulation based on the conceptual model in Figure 5.2 was not possible because the IFTU is still undergoing development, and the support for CASE instruction (Section 5.3) was not necessary.

This behavioral model allows analysis of the following aspects of code behavior and G-Machine operation.

(1) PCU instruction demand and execution speed
(2) G-Code to PCU code translation ratios
(3) minimum time to restart translation of PCU queue
(4) G-Code and IFTU jump instruction execution efficiency

The basic operation of the IFTU is as follows:

-1- Fetch G-Code into instruction buffer(s);

-2- Use G-Code as index into jump table;

-3- Fetch next translation instruction;

-4- Perform required IFTU instruction; default operation is to place PCU instruction and index into appropriate queues.

4a) If a jump instruction appears, wait till **JUMP/NOJUMP** is signaled and then start execution from the indicated path.

4b) If the jump taken was external as well, flush the G-Code buffers and restart G-Code PC in addition to actions in (4a).

4c) If the end of a instruction sequence, go to step 1. Perform default operation as well.

The events that cause the PCU to wait for instructions from the IFTU are G-Code and PCU code jumps[1], and G-Code instruction processing, such as for absolute jumps and literal fetches. G-Code jumps are the most important in terms of performance since they involve restarting the entire instruction pipeline. Internal IFTU jumps are more dependent on the efficiency of the particular implementation.

5.5. Jumps

All jump instructions, G-Code and PCU, test a condition code to determine whether a jump occurs or not. This means that a PCU conditional code jump costs the same as an external G-Code jump for testing and responding to the result, provided that the G-Code prefetch has been successful. There are two classes of PCU jumps: Internal jumps that affect only the translation sequence, and the external jumps that affect both the translation sequence and the G-Code Program Counter. Both types of jumps will be examined in this section.

---

[1] Absolute jumps in either the G-Code or the IFTU do not cause break like conditional jumps, but they do require IFTU processing time.

When a conditional jump instruction is detected, the IFTU continues translation based on the assumption that the jump will not be taken. If a second conditional jump instruction is detected, the IFTU stops producing PCU instructions until the first jump is evaluated. After the jump is evaluated by the PCU, the PCU signals to the IFTU either **JUMP** or **NOJUMP**. If **NOJUMP** is signaled, then the jump did not occur and processing continues. If **JUMP** is signaled, then the IFTU flushes the PCU instruction queue, and fetches the next appropriate G-Code instruction (on external jumps). Note that either a **JUMP** or **NOJUMP** signal MUST be sent to the IFTU, otherwise it will stop evaluating at the next conditional jump instruction.

To avoid the latency of stepping the PCU instruction through the entire PCU queue, it should be accelerated in some manner to the top of the queue. This acceleration will require special hardware, perhaps similar to that developed for the P-Stack. The PCU instruction queue acts as a buffer to keep the PCU busy while the IFTU fetches new G-Codes, literals or performs an unconditional jump. The amount of buffering necessary to provide in the PCU instruction queue is discussed further in Section 6.

5.5.1. External G-Code Jumps

When a restart of the G-Code buffers occurs, it is desirable to have instructions still in the PCU instruction queue. This instruction overflow on a jump is known as *spillover*. Specifically, spillover is a measure of how many instructions are still to be executed after the G-Code Program Counter has changed, but **before** a new G-Code instruction is requested.

How much spillover is enough? Assuming that each fetch from Control Memory takes a minimum of 1 IFTU cycle, the minimum time is four cycles (five for complex

instructions).

	CYCLE 1	CYCLE 2	CYCLE 3	CYCLE 4	CYCLE 5
IFTU	Flush Buffers	G-Code Fetched	Translate†	Pass to PCU	
PCU	**JUMP** signal	--	--	--	Execute

Fig. 5.5 Pipeline Restart Length

† Fetching the bytecode from the instruction buffer into the translate unit. This is the minimum time for simple instructions. Complex instructions require an extra cycle.

The average number of PCU instructions executed before a Program Counter change was **25.8** for the test programs, with an average spillover of **2.8.** Overall results of program executions can be found in **Appendix A**. Figure 5.6 shows the spillover taken with respect to the number of PCU instructions executed for several examples.

program	G-CODES	PCU CODES	SPILLOVER
ack 1,4	169	1244	2.5
fib 2	65	392	2.2
fib 6	494	3340	2.1
from 0-1	49	226	6.7

Fig. 5.6 Average Pipeline Spillover For Some Programs

The average spillover is not high, but there are very few PCU instruction queue restarts. On average, the IFTU unit normally provides 2.8 PCU instructions after the Program Counter changes and a new G-Codes is needed by the PCU.

5.6. IFTU Design Summary

For simulation purposes, the IFTU was assumed to deliver a new PCU instruction to the PCU within one cycle after the PCU instruction queue has been flushed. There-

fore, for one cycle during a jump, the PCU executed a NOP. This assumes that the IFTU is never obliged to wait for a G-Code from the Control Memory. An instruction cache or multiple buffers (such as those shown in Fig. 5.2) would make this non-deterministic prefetching possible.

These test programs are useful for extrapolating the the performance of the Bit-Slice G-Machine on very large programs. They characterized the major hardware portions of the design, as well as instruction execution and processing efficiency. Larger test cases were not possible due to the speed of the simulator. For larger tests, a simulator that was not so low-level as the N.mPc system provides would be desirable.

One important performance consideration for the IFTU is to determine to what degree the G-Code is expanded (the ratio of G-Code to PCU micro codes). On the average, there an expansion factor of **6.4** was observed in the programs tested. See **Appendix A** for more details.

Since the PCU is (potentially) capable of executing one instruction every cycle, the IFTU should also be able to deliver at least 1 instruction per cycle. When the IFTU cannot, the PCU instruction queue serves as a buffer.

The size of the PCU queue and other IFTU performance topics are discussed further in the next section.

6. BIT-SLICE PCU PERFORMANCE

The performance of the Bit-sliced PCU is compared against the other implementation of the abstract G-Machine Architecture by a comparison of G-Code programs executed under the Bit-slice PCU to the same programs run using the VAX-based LML compiler. This comparison is covered in Section 6.2 below.

Other performance factors of the Bit-slice PCU that are related to particular aspects of the design and not directly to the abstract architecture (i.e. the IFTU design, the Instruction Set and G-Code translation schemes) are discussed first. These provide a model of the machine and different measurements of the design's performance.

6.1. Design Performance Aspects

This section is divided into three sub-sections that cover the architectural performance aspects of the Bit-slice design in the areas of the stacks, the instruction set, and instruction execution speeds.

6.1.1. P-Stack

As a basic component of the G-Machine, the architecture demands the custom silicon P-stack [BaR85, TaN85, Vir84] for efficiency. As already discussed in Section 4.8, custom chips cut down on the number of components required to build a P-stack, and offer increased performance. This is clearly shown in the execution efficiency of the complex ROT(ate) and MOVE operations, where there is maximum factor of **18** improvement in the number of cycles per program when using a hardware P-Stack over

the VAX software emulation (see Fig. 6.1 below)

Test Program	Total ROT & MOVE Instr. Executed	Bit-slice Cycles	approx. VAX Cycles	approx. speedup
ack1,2	29	55	1044	19.0
ack1,4	114	214	3930	18.6
ack2,1	80	154	2706	18.2
fib2	29	55	1044	18.2
fib4a	92	178	3312	18.3
fib5	155	301	5580	18.4
fib6	260	506	9360	18.4
from	21	29	756	18.6
test20	12	20	432	18.6

Avg. maximum cycle factor over a VAX: 18.6
(VAX Cycles / Bit-Slice Cycles)

Fig. 6.1 ROT and MOVE Speedups

The ROT and MOVE instructions are most important because the other instructions (PUSH, POP, COPY) are fairly simple for most stacks to perform (or to emulate in software). ROT in particular can involve reshuffling large portions of the stack, and is a fairly common G-Code and PCU instruction. See Section 3.2 for a description of the P-stack operations.

The ROT instruction is modeled on in the VAX LML system by a loop. The number of loop executions depends on the index of the value being rotated. See **Appendix D** for a complete description of the comparison values and VAX ROT(ate) instruction sequence models.

6.1.2. IFTU and Code Throughput

How quickly each component operates and the speed of the entire machine determines the machine's throughput. The throughput of the Bit-slice design depends upon the following items of the IFTU and PCU:

— G-Code to RISC expansion ratio[1]
— PCU queue length and spillover[1]
— Instruction execution rates of PCU and IFTU

6.1.2.1. PCU Instruction Queue

The best measure of the IFTU's performance is how often the PCU has to wait for new instructions. The PCU instruction queue (see Figures 5.1 and 5.2) acts as a buffer to hold the next executable instructions. While the PCU is using the instructions in the queue, the IFTU can perform other operations independently. The number of instructions that the queue can hold is therefore based on how long the IFTU needs to perform these other operations, such as G-Code instruction prefetches, literal fetches from the code stream, and unconditional jumps.

Any code prefetching is handled in parallel to the main program stream, and should not slow instruction throughput. Literal fetches and unconditional jumps, on the other hand, have to fetch a value out of the instruction stream and place it into the Literals Queue and the G-Code Program Counter, respectively. The amount of time this can take is illustrated in Fig. 6.2 below.

[1] Already discussed briefly in Section 5 on the IFTU.

	number of IFTU cycles
PUSHLITERAL or JMP instruction	1
Get next 4 bytes (assuming 1 cycle C-Memory fetches see Sect. 5.6)	4
Get next G-Code instruction	1
Translate and send on (this is assuming the new instr. is not a PUSHLITERAL or JMP)	2
Total	8

Fig. 6.2 PCU Queue Length

As Fib. 6.2 shows, the size of the Bit-slice PCU instruction queue should be at least 8 instructions to allow the IFTU to fetch instructions with addresses.

6.1.2.2. Instruction Execution Rates

The PCU has an average execution rate of around **2.3**[1] cycles per PCU instruction, with bursts of 1 cycle instructions. Since the job of the IFTU to supply the PCU with instructions, the IFTU must have an appropriate bandwidth to the C-Memory.

The bandwidth (**BW**) from the IFTU to the C-Memory is determined by several factors as follows:

The expression (1—J) * (BW/b) gives the G-Code instructions/cycle

where

BW = bandwidth = number of G-Codes IFTU must fetch
b = average length in bytes of G-Codes
J = fraction of G-Code bytes that are addresses (4 bytes)

And so ((1—J) * (BW/b)) * TR gives the PCU instructions/cycle

[1] derived from simulations; number of cycles divided by number of instructions executed.

where

TR = translation ratio of G-Code to PCU instructions

Finally we have $L = ((1-J) * (BW/b)) * TR + N$

where

L = average length of a straight line PCU code sequence
N = number of PCU instructions to have in PCU instruction queue when the sequence is done

L gives the length of the average number of instructions that are performed without a jump or code break. Substituting values for the variables

$L = 26$ PCU instructions $= (.82 * (BW/2)) * 6.4 + 4$

and solving for BW gives us

BW = 8 bytes (on average) per PCU instruction sequence

Given that the Bit-Slice PCU machine executes at 2.3 cycles/PCU instruction, then the IFTU should provide at .17 bytes/cycle (2.3 * L) on average. The PCU operates then at a theoretical limit of 1.1 Mbytes/second:

(2.3 PCU cycles/sec) * (BW / (2.3 * 26)) = 1.1 Mbytes/second

Therefore, an estimate for the C-Memory bandwidth is around 900 nsec/byte for the IFTU:

(1 second/1.1 Mbytes) * 1ns = 900 nsec/byte

assuming that the IFTU operates at the same speed of 150ns as the PCU. This evaluation is inclusive of all G-Code programs performed, and include the EVAL instruction sequences.

6.2. Bit-Slice PCU vs. VAX-based LML

The Bit-slice PCU design is compared to that of the G-Machine model used by the LML Compiler on the VAX by seeing how the different G-Machine implementations support and execute the abstract G-Codes.

The comparison consists of matching the G-Code instructions to G-Code instructions, and this is done by comparing the number of cycles it takes for the Bit-slice PCU to perform the same G-Code operation as the VAX. However, this comparison is not possible due to the EVAL sequence. EVAL (see Section 3.4) is **the** complex G-Code instruction, which can be composed of many sub-instructions in an implementation. For instance, the Bit-slice PCU creates an EVAL by a sequence of instructions in the IFTU, where the VAX LML-Compiler uses a separate subroutine for EVAL.

If EVAL occurred rarely, it would be easier to account for. But, as it is the major operation that the G-Machine uses to provide Lazy Evaluation, it must be measured accurately. Therefore, instead of measuring cycles per G-Codes directly, a comparison is done between the Bit-slice PCU instructions and equivalent instructions on a VAX. This works well, since most G-Codes have a 1-to-1 correspondence with internal PCU instructions. And, since the PCU instructions simulated include all *internal* phases of EVAL[1] and the other complex G-Codes, (i.e. the sequence of Bit-slice instructions to perform a GET_FST or EVAL) the operations can be compared to the similar VAX sequence used to perform the G-Code operations.

For each instruction, both the total number of instructions executed and the number of cycles used to perform each instruction were computed. Where a PCU instruction did not have a counterpart in the VAX architecture, a sequence of

[1] such as the UNWIND phase of EVAL described in Section 3.4 and [Kie85a]

functionally equivalent instructions was used.

The VAX sequences are based on assembly code generated by the LML compiler, and a VAX 11/780 was chosen as the primary machine model for the number of execution cycles per instruction. The number of cycles for each instruction sequence are detailed in **Appendix D**.

One drawback to using the VAX as the comparison standard like this is the difficulty of including the cost of housekeeping activities and other support operations; for example, tag storage and manipulation. This would tend to give an instruction/speed advantage to the VAX, since not all of its operations are accounted for.

To be meaningful, the comparison of instructions is not based on speed, since the different components in each machine make sort of comparison difficult and the Bit-slice PCU has a 150ns cycle time where the VAX-11/780 has a 200ns cycle time. Instead, the comparison is based on the number of cycles each machine takes for a single PCU operation to be performed. By way of illustration, the MVMP (MoVe graph Memory to P-stack) instruction is outlined. MVMP takes from 3 to 5 cycles in the Bit-slice design, and from 6 to 7 in the VAX using the MOV Long with an autodecrement.[1]

MVMP 3-5 6-7 MOVL-

Cycle times for the VAX are sometimes approximate, especially those involving memory accesses are concerned. In the Bit-slice G-Machine, the memory was always assumed to be ready and able to respond to a read request during the next cycle. This gives the Bit-slice PCU a two cycle wait for a read. This is somewhat unrealistic, but

[1] shown as MOVL-. MOVL+ is an autoincrement. This is a shorthand for dealing with VAX instructions.

provides a simple multiplication factor for actual timing later. On the other hand, the cycle times for the VAX instructions are based on the VAX Hardware Handbook [DEC80], whose instruction and memory access times are imprecise.

It must therefore be emphasized that this comparison is based on the IDEALIZED BEST CASE TIMES of each machine's instruction.

Program	Instr.	Bit-slice Cycles†	approx. VAX Cycles	speedup
ack1,2	392	878-1104	2572-3613	2.93-3.27
ack1,4	1244	2735-3419	8094-11692	2.95-3.38
ack2,1	924	2015-2528	5878-8416	2.94-3.36
fib2	392	878-1104	2572-3613	2.94-3.35
fib4a	1196	2696-3390	7900-11149	2.94-3.33
fib5	2000	4514-5676	13213-18679	2.93-3.32
fib6	3340	7544-9486	22068-31229	2.93-3.31
from	226	481-628	1610-2286	2.94-3.32
test20	129	294-376	910-1304	2.94-3.32

Average speedup of G-Machine over VAX: 2.9-3.3

Figure 6.3 Bit-Slice G-Machine Speedup over VAX

† The range of values for the Bit-slice PCU comes from estimating Graph memory read/write of 5 and 2 cycles, respectively. For the VAX, accurate numbers for each instruction are not available, so the ranges reflect the possible times of each instruction. See **Appendix D** for the estimates of instruction times.

As shown in Fig. 6.3, the Bit-slice PCU has a speedup margin of around 3 over a VAX on just the number of machine cycles. Note that this includes all of the PCU housekeeping chores, but not all of the instructions that a VAX may perform for support. For both machines only the execution times of the instructions in the central processing unit are computed, except for the jump instructions of the Bit-slice PCU. This allows us to directly measure the performance of processor cores.

6.3. Overall Performance

Overall the IFTU is the critical path of any implementation. The PCU can be made so fast that it can go hungry at times waiting for the IFTU to fill the instruction pipes. In other architecture matters, the ALU of the Bit-slice PCU needs a slightly better design and hardware support, such as custom V-Stack chips like the custom P-Stack chips, in order to cut down on the number of components and increase the functionality. The ALU should also have more dedicated units (such as a barrel shifter) separate from the current integer ALU.

From the execution results presented here, it would seem that the Bit-slice G-Machine outperforms the VAX in the number of basic cycles used to execute similar programs. Architecturally, the Bit-slice design provides faster execution of the graph operations with direct support of the G-Machine stacks. Larger programs, such as functional language compilers or simulators would see the most benefits from these speedups.

This improvement in basic cycle speed is not enough by itself to justify building an actual machine, especially since the debugging and trace environments of a software system can be more detailed than those of a hardware system. The G-Machine becomes most useful in the handling of potentially larger programs than are currently possible today; that is, programs with graph memories of 100 megabytes or more [Fos85, Kie85b]. As stated in Section 4 on the development time of the Bit-slice PCU, The better execution performance of the Bit-slice PCU, coupled with fast development times and large graph memories make this design method attractive to build engines to handle larger programs and problems.

7. CONCLUSION

The design and structure of the G-Machine Bit-slice PCU has been described along with the simulation of the processor. The design process has provided invaluable feedback on the abstract G-Machine architecture and on an actual implementation. As shown through simulation, the Bit-slice G-Machine performs G-Code faster than a VAX, as might be expected due to the specific hardware support provided for G-Machine operations.

7.1. Further areas of study

As noted in Section 4.5, the PCU ALU requires far too many resources in terms of chips, μ-code and machine cycle time. The ALU is the longest path in the circuit to stabilize in the processor. Replacement of the Bit-slice ALU/V-Stack with a a more integrated design solution, perhaps similar to the P-Stack chip, would lower circuit time and simplify this portion of the processor enormously. It would also reduce the μ-code signals approximately 4 to 10 lines by removing the associated controls for the 8 chips that comprise the V-Stack pointers and ALU units to just two or three control signals.

Several other areas within the architecture still require more refinement, such as the Host processor interface and protocols with the Graph Memory. The Host interface to the G-Machine as currently defined (Section 4.2) is incomplete, but adequate for testing and refining the processor in its current state. All of the Host's actions need more physical definition before an actual implementation is built. A generic enough interface would allow a wide range current commercial processors, such as many micro-

computers, to act as the Host.

The IFTU is an area of the G-Machine that would appear to be the most complex in terms of hardware design. The fairly simple interface between the PCU and the IFTU proved to work well in simulation. By allowing the two units to operate asynchronously, the interface provides the potential for extra concurrent operations not explored by this research.

7.2. Results of Research

The data regarding the performance of this first G-Machine processor, the comparison of the Bit-slice PCU to that of a VAX, along with the experience of doing the actual design, have proven invaluable. The goals of this project were primarily to provide feedback on the abstract architecture as well as to experiment with and clarify different design and implementation issues for the G-Machine.

At the start of the project, the abstract architecture was not fixed, changing more than once in the early stages. Instructions were tried and verified in parallel with the development of the Bit-slice PCU hardware system, sometimes eliminating instructions (such as the CASE G-Code) or modifying the original hardware design. Because of the experimental nature of the G-Machine architecture, this was not an unusual occurance.

This project was also a first attempt by the author to design a large system, and to use and learn available board and system design tools and methodologies. In particular, the N.mPc simulation system from Case Western [Str78] used for simulation of the Bit-slice G-Machine model, and the Mentor Netlist Editor NETED for schematic capture of the physical design.

REFERENCES

[Ack82] Ackerman, W. B., "Data Flow Languages," *Computer*, February 1982.

[Ame85] Amerson, F. C., "Simplicity in a Microcoded Computer Architecture," *Hewlett-Packard Journal*, 1985.

[ArT81] Arvind and Thomas, R. E., "I-Structures: An Efficient Data Structure for Functional Languages," MIT/LCS/TM-178, 1981.

[Bac78] Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM*, vol. 21, 8 (1978), pp. 613-641.

[BaR85] Bailey, J. and Rankin, L., "Final Project Report," CSE522 Intro. to VLSI Design, Oregon Graduate Center, May, 1985.

[Ber83] Berkling, K., "Experiences with Integrating Parts of the GMD Reduction Language," in *VLSI Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1983.

[Chu41] Church, A., *The Calculi of Lambda-Conversion*, vol. Ann. Math. Studies, Vol.6, Princeton Univ. Press, Princeton, NJ, 1941. The information in this book was presented via lectures and other works cf. [Bac78, GMW79, Tur79] etc.

[CGM80] Clarke, T. J. W., Gladstone, P. J. S., MacLean, C. D. and Norman, A. C., "SKIM - the S, K, I reduction machine," *Proc. of 1984 ACM Lisp Conf.*, August 1980, pp. 128-135.

[DEC80] Digital Equipment Corp., VAX Hardware Handbook, 1980.

[CCM85] Cousineau, G., Curien, P. and Mauny, M., "The Categorical Abstract Machine," in *Functional Programming Languages and Computer Architecture*, vol. 201, J. Jouannaud (ed.), Springer-Verlag, September 1985, pp. 50-64.

[Den80] Dennis, J. B., "Data Flow Supercomputers," *Computer*, November 1980.

[AMD83] Advanced Micro Devices, Am2900 Family 1983 Data Book, 1983.

[FFK81] Fitzpatrick, D. T., Foderaro, J. K., Katevenis, M. G. H., Landman, H. A., Patterson, D. A., Peek, J. B., Peshkess, Z., Sequin, C. H., Sherbourne, R. W. and Dyke, K. S. V., "A RISCy Approach to VLSI," *VLSI Design*, 1981.

[Fos85] Foster, M. H., *Design of a List-structure Memory using Parallel Garbage Collection*, Masters thesis, Sept 1985.

[GMW79] Gordon, M., Milner, R. and Wadsworth, C., *Edinburgh LCF*, vol. 78, Springer-Verlag, 1979.

[HeM76] Henderson, P. and Morris, J. H., "A lazy evaluator," *Proc. of 1976 ACM ACM Symp. on Prin. of Programming Languages Conf.*, 1976, pp. 95-103.

[Hen80] Henderson, P., *Functional Programming: Application and Implementation*, Prentice Hall, Englewood Cliffs, NJ, 1980.

[Hee81] Hennessy, J. and et al., "MIPS: A VLSI Processor Architecture," *Proc of the CMU Conference on VLSI systems And Computations*, 1981, pp. 337-346.

[Hug82] Hughes, J. R. M., "Super Combinators: A New Implementation Method for Applicative Languages," *Proc. 1982 ACM Symp. on Lisp and Functional Languages*, August 1982, pp. 1-10.

[TTD84] Texas Instruments, The TTL Data Book, vol. 3, 1984.

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

[Kie84] Kieburtz, R. B., *G-Machine Architecture Handbook*, Oregon Graduate Center, 1984.

[Kie85a] Kieburtz, R. B., "A proposal for interactive debugging of ML programs," CS/E-85-006, Oregon Graduate Center, 1985.

[Kie85b] Kieburtz, R. B., *G-Machine Architecture Handbook*, Oregon Graduate Center, 1985.

[Lan64] Landin, P. J., "The Mechanical Evaluation of Expressions," *Computer Journal*, June 1964, pp. 308-320.

[MiB80] Mick, J. and Brick, J., *Bit-Slice Microprocessor Design*, MCGRAW, NY, 1980.

[OrR80] Ordy, G. M. and Rogers, L. A., metaMicro-Linking/Loader Utilities User's Manual, vol. Ver. 2.0, 1980.

[Ran86] Rankin, L., Private Communications, Oregon Graduate Center, April, 1986.

[Sar84] Sarangi, A., *Simulation and performance evaluation of a Graph Reduction Machine Architecture*, Masters thesis, July 1984.

[Sch85] Scheeval, M., *NORMA, a normal-order combinator reduction machine*, colloquium presented at Oregon Graduate Center, Jan., 1985.

[Sto85] Stoye, W., *Implementing Lazy Functional Languages*, colloquium presented at Oregon Graduate Center, Sept 1985.

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

[TaN85] Tamjidi, M. and Nader, B., "Final Project Report," CSE522 Intro. to VLSI Design, Oregon Graduate Center, December, 1985.

[UBF84] Ungar, D., Blau, R., Foley, P., Samples, D. and Patterson, D., "Architecture of SOAR: Smalltalk on a RISC," *11th Annual International Symposium on Computer Architecture*, 1984, pp. 188-197.

[Vir84] Vireday, R., "Final Project Report," CSE526 Testing and Diagnosis of VLSI Systems, Oregon Graduate Center, May, 1984.

Appendix A: Simulation Data

This Appendix shows the results of the programs used for testing†. Items in each table (i.e. # of PCUs, zCMEMREAD, etc.) are explained in the *Notes* following that table. All tests were run with the context switching to the Dump, unless otherwise noted. The first table lists the major results of the tests, and shows the number of instructions executed for each test. Items in this first table are referred to in calculations of subsequent tables.

	fib2	fib4a	fib5	fib6	from	test20†	ack2,1	ack1,2	ack1,4
# of G-Codes	65	182	299	494	49	21	144	172	169
# of PCUs	392	1196	2000	3340	226	129	924	1209	1244
Expansion Factor	6.03	6.57	6.68	6.76	4.61	6.14	6.41	7.02	7.36
Dev. from Mean	.37	.17	.28	.36	1.79	.26	.01	.62	.96
zCMEMREAD	142	379	616	1011	98	46	325	379	380
zIFTUREAD	486	1485	2484	4149	278	155	1194	1555	1597
# of CYCLES	910	2767	4627	7727	513	310	2088	2753	2796
	fib2	fib4a	fib5	fib6	from	test20	ack2,1	ack1,2	ack1,4

Notes	
# G-Codes	-- Total number of G-Codes executed
# of PCUs	-- Total number of PCU instructions exectuted
Expansion Factor	-- PCU/G-Codes
Dev. from Mean	-- Std. Deviation from the Expansion Factor
zCMEMREAD	-- Number of reads from the Control memory where bytecodes are stored. Reading is done a byte at a time.
zIFTUREAD	-- Number of reads from IFTU translation memory
# of CYCLES	-- Execution time of the program (in nanoseconds). This was made with a variety of assumptions on cycle time for different items. The lower bound (best case) was used for simulation.

G-Code expansion is discussed in Sections 4 and 6.

Mean G-Code to PCU Expansion Factor: 6.40
Std. Deviation for Expansion Factor: 0.74

† *test20* is included as a reference. It is a simple reduction of a function with two arguments, and is useful as a benchmark of the G-Code EVAL instruction. See **Appendix B: G-Code Programs** for a better description of *test20*.

Further execution data for the test cases executed with the Dump context switching mechanism.

	fib2	fib4a	fib5	fib6	from	test20	ack2,1	ack1,2	ack1,4
unnecessary iNOPs	13	40	67	112	11	2	48	57	61
iNOPs as a % of PCUs	3.3%	3.3%	3.3%	3.3%	4.8%	1.5%	5.1%	4.7%	4.9%
Graph READs	37	115	193	323	22	13	91	119	111
Graph WRITEs	28	85	142	237	22	11	55	75	83
Graph Nodes Allocated	19	49	79	129	23	12	30	38	46
Allocation Rates									
G-Codes	.29	.27	.26	.26	.47	.57	.21	.22	.27
PCU	.05	.04	.04	.04	.10	.09	.03	.03	.04
stackmax	6	6	6	6	6	5	9	9	9
PCU Stack Instructions	119	374	626	1046	65	36	392	119	407
% of PCU instr.	30.4	31.3	31.3	31.3	28.7	27.9	31.4	30.4	32.7
G-Code Stack Instructions	11	32	52	88	11	4	54	46	59
% of G-Code	16.9	17.6	17.7	17.8	22.4	19.0	31.4	31.9	34.9
	fib2	fib4a	fib5	fib6	from	test20	ack2,1	ack1,2	ack1,4

Notes	
unnecessary iNOPs	-- iNOPs that were an inefficiency in the translation scheme of the IFTU.
iNOPS as a % of PCUs	-- percentage of NOPs to total PCU instructions
Graph READs	-- Number of reads from Graph Memory to PCU
Graph WRITEs	-- Number of writes to Graph Memory from PCU
Graph Nodes Allocated	-- Number of nodes dynamically allocated in the Graph memory with the allocation rates versus the G-Codes and PCU instructions.
stackmax	-- maximum depth of the P-stack during execution
PCU Stack Instructions	-- total number of PCU P-stack instructions for each program
% of PCU instr.	-- total number of PCU P-stack instr. / total PCU instr.
G-Code Stack Instructions	-- total number of G-Code P-stack instructions for each program
% of G-Code	-- total number of G-Code P-stack instr. / total G-Codes

Average Graph Node Allocation Rates G-Code: 0.31 PCUs : 0.05

Further execution data. This table shows the jump statistics discussed further in Sections 4 and 5.

	fib2	fib4a	fib5	fib6	from	test20	ack2,1	ack1,2	ack1,4
JFUN/JMP's	0	0	0	0	0	0	3	3	3
G-Code Jumps	1	4	7	12	0	0	4	5	7
G-Code Not Taken	4	10	16	26	0	0	4	4	3
PCU Jumps	15	45	75	125	6	4	35	45	45
PCU NOT Taken	14	50	86	146	0	4	50	70	72
Total Taken	16	49	82	137	6	4	39	50	52
Total NOT Taken	18	60	102	172	0	4	54	74	75
Total Jumps Tested	34	109	184	309	6	8	93	124	127
% of PCUS(jmps)	8.6%	9.1%	9.2%	9.2%	2.6%	6.2%	10.0%	10.2%	10.2%
x 4 for Taken	64	196	328	548	24	16	156	200	208
x 3 for NOT Taken	54	180	306	516	0	12	162	222	225
Cycles for jumps	118	376	634	1064	24	28	318	422	433
% of Cycles	12.9%	13.5%	13.7%	13.7%	4.6%	9.0%	15.2%	15.3%	15.4%
% if 1 cycle jumps	4.1%	4.3%	4.4%	4.4%	1.2%	2.7%	4.5%	5.0%	5.1%
	fib2	fib4a	fib5	fib6	from	test20	ack2,1	ack1,2	ack1,4

Notes	
JFUN/JMPS	-- Absolute, or unconditional jumps in the G-Code.
G-Code Jumps	-- Number of external jumps taken. This is only for conditional jumps.
G-Code NOT Taken	-- Number of external jumps NOT taken.
PCU Jumps	-- Number of PCU jumps taken.†
PCU NOT Taken	-- Number of PCU jumps NOT taken.
Total Taken	-- Number of PCU AND external jumps taken.
Total NOT Taken	-- Number of PCU AND external jumps NOT taken.
Total Jumps Tested	-- Actual jump instructions (Taken/NOT Taken) tested.
% of PCUS(jmps)	-- Percentage of the code spent performing jumps.
x 4 for Taken x 3 for NOT Taken	-- Cost of cycles for each jump. 3 for jumps not taken, 4 for taken jumps.
Cycles for Jumps	-- Number of cycles taken for jumps
% of Cycles	-- Cycles for Jumps/# of Cycles. A percentage of the total time spent on performing jumps.
% if 1 cycle jumps	-- percentage of time if jumps were evaluated in 1 cycle. Computed by Total Jumps/(# of Cycles - Cycles for Jumps + Total Jumps)

The EVAL instruction is covered in Sections 3 and 4. This table shows the number of EVALs executed, and the percentage of executed code EVAL took. The '# PCU's per EVAL' is a constant around 50% for each of these programs because they are only calling one function repeatedly. For programs with a mix of larger functions the ratio will be less.

	fib2	fib4a	fib5	fib6	from	test20	ack2,1	ack1,2	ack1,4
# EVALS seq.s	7	19	31	51	2	1	19	23	24
% EVALs/GCODE	10.7%	10.4%	10.3%	10.3%	4.0%	4.7%	13.1%	13.3%	14.2%
# PCU's per EVAL	181	565	949	1589	18	54	478	640	721
% of PCU's(evals)	46.1%	47.2%	47.4%	47.5%	7.9%	41.8%	51.7%	52.9%	57.9%
	fib2	fib4a	fib5	fib6	from	test20	ack2,1	ack1,2	ack1,4

Notes

# EVALS seq.s	-- Number of EVAL instructions in G-Code
% EVALs/GCODES	-- EVAL instructions / total G-Codes
# PCU's per EVAL	-- Total number of PCU instructions per EVAL sequence
% of PCU's(evals)	-- Percentage of PCU instructions in EVAL sequence versus total PCU instructions executed

EVAL Performance for Tests

Spillover	fib2	fib4a	fib5	fib6	from	test20	ack2,1	ack1,2	ack1,4
# of PCU's per restart†	2.2	2.1	2.1	2.1	6.7	2	2.7	2.6	2.5

Average Spillover of pipeline: 2.8

(Avg. Spillover without *from*: 2.3)

The next table contains the same information as the first table of this Appendix, but for several tests run with the P-stack overflow context switching mechanism. See Section 4.9 for a description of the Context Switching mechanisms used in the G-Machine.

† # of PCU's per restart is the average number of instructions (or Spillover) executed after a change to the Program Counter. This tells how many instructions we can count on to be executed while a new G-Code is being readied for translation.

	test20	from	fib2
# of G-Codes	21	49	65
# of PCUs	117	218	316
PCU instr. reduced	10%	4%	20%
Expansion Factor	5.57	4.45	4.86
zIFTUREAD	142	268	391
Graph READs	11	18	27
Graph WRITEs	9	18	18
# of CYCLES	273	477	694
CYCLES reduced	12%	13%	24%
	test20	from	fib2

The next table is a comparison of the EVAL sequence in the different context switching models.

	test20	from	fib2
# EVALS seq.s	1	2	7
with Dump			
# PCU's per EVAL	54	18	181
% of PCU's(evals)	42%	8%	46%
with backdoor			
# PCU's per EVAL	46	20	133
% of PCU's(evals)	36%	9%	34%
	test20	from	fib2

As is shown from these tests, there is a large reduction in both the number of instructions necessary and the amount of time taken to do context switching with the overflow cache on the P-stack. The larger the program, the greater the benefits, especially for those programs like Fibonacci with heavy lazy evaluation.

The next table show the distribution of different sized G-Codes for the various programs.

	all test programs	A
1-byte	40%	37%
2-byte	42%	36%
4-byte	18%	26%

A = test20, from, and fib2 with backdoor overflow model

Appendix B: G-Code Test Programs

Listings of the G-Code run on the simulator.

```
--module              -- generated with R. Kieburtz's compiler
--      export ACK;
--
--      ACK = \x.\z. if x=0 then z+1
--      else if z=0 then ACK (x-1) 1
--      else ACK (x-1) (ACK x (z-1))
--end
CFUN Ack          2
CFUN sub          2
CINT one          1
CINT arg1         1       -- change as necessary
CINT arg2         2       -- 1,2  2,1  1,4 for testing
LDD 2000
ALLOC
PUSHCONST         arg2
PUSHCONST         arg1
GET_BYTE          0
CALLGLOBFUN       Ack     --Ack
GET_FST
GET_BYTE          73
MK_VAL_PR
SIGNAL
PUSHCONST         nil
SIGNAL

DEF_FUN           sub     -- makes P-stack do all the saving
EVAL 2
ROTP 1
EVAL 2
ROTP 1
GET_FST
GET_FST
SUB
RET_INT           0
END_FUN           sub


DEF_FUN           Ack
COPYP             0
EVAL 3
GET_BYTE          0
GET_FST
SUB
JNOT_ZERO         28
COPYP             1
EVAL 3
GET_FST
GET_BYTE          1
ADD
RET_INT           3
GET_BYTE          0
COPYP             1
EVAL 3
GET_FST
SUB
JNOT_ZERO         71
--      original values ----
PUSHCONST         one     -- 2
```

```
PUSHCONST       one     -- 3
COPYP           2
PUSHCONST       sub     -- 3
MK_APP
MK_APP
ROTP 1
MOVEP           2
MOVEP           0
JFUN 5          -- should be Ack
PUSHCONST       one     -- 4
COPYP           2
PUSHCONST       sub     -- 3
MK_APP
MK_APP
COPYP           1
PUSHCONST       Ack     -- 1
MK_APP
MK_APP
PUSHCONST       one     -- 5
COPYP           2
PUSHCONST       sub     -- 3
MK_APP
MK_APP
ROTP 1
MOVEP           2
MOVEP           0
JFUN 5          -- should be Ack
END_FUN         Ack
--JFUN jumps to an offset from the start of the def. of the current fun.
-- Bug in the way the assembler currently handles _CODEADDResses.
```

This is a non-linear Fibonacci. It does do tail-recursion, but it is hand optimized to set up the graphs before a recursive call.

```
-- Handcoded Fibonacci ProgramR.Vireday  Feb. 1985
--letrec fib n =
--      if n < 2 then 1
--      else                fib (n-1) + fib (n-2)
--in fib 4
-----------------------------------------------------
CFUN fib 1
CINT one 1
LDD 200
PUSHLITERAL     4
MK_VAL
PUSHCONST fib
MK_APP
EVAL 0
GET_FST
GETBYTE         73
MK_VAL_PR
SIGNAL

DEF_FUN fib     -- basepoint of jump addresses
EVAL 0
GET_FST
GET_BYTE        2
COPYV           1
SUB
JNEG 28         -- 28=compute
POPV
POPV -- clear the V-stack
PUSHCONST       one
UPDATE          1
RET  0

--LABEL compute
POPV
COPYV           0
GET_BYTE        1
SUB
MK_VAL
PUSHCONST fib
MK_APP
GET_BYTE        2
SUB
MK_VAL
PUSHCONST fib
MK_APP
GET_BYTE        2
CALLGLOBFUN     fib
ROTP 1
GET_BYTE        2
CALLGLOBFUN     fib
GET_FST
GET_FST
ADD
MK_VAL
UPDATE          1
RET  0
END_FUN fib
```

This is a handcoded **from** (or successor function). It is primarily straight-line code. Any EVAL's called should not have to work too hard to set up a graph.

```
-- from = \x. x.from(x+1)  in from 0.RPV
-- only does the first two items in the list
-------------------------------------------------
CFUN from 1
CINT zero 0

LDD   200
ALLOC
PUSHCONST zero
GET_BYTE          0
CALLGLOBFUN from
--PRINT
COPYP             0
FST   -- bring head of list in
GET_FST
GET_BYTE          73        -- character 'I' for Integer
MK_VAL_PR
SIGNAL

SND   -- set up next call
SND   -- get the pointer to the application argument
ALLOC
ROTP 1
GET_BYTE          0
CALLGLOBFUN from
--PRINT
COPYP             0
FST   -- bring head of list in
GET_FST
GET_BYTE          73        -- character 'I' for Integer
MK_VAL_PR
SIGNAL
PUSHCONST         0         -- signal a halt me boy
SIGNAL
NOP

DEF_FUN from
EVAL 0
COPYP             0
GET_FST
INCR
MK_VAL
PUSHCONST         from
MK_APP
ROTP 1
MK_PR
UPDATE            1
RET   0
END_FUN from
```

This is a simple test case that can be used for comparison of EVAL instructions. It builds a simple graph, then evaluates it. The answer, for completeness' sake, is 11.

```
-- test20 of the G-Machine simulator.  GCODE version
--      @                                  012 (P 010, P 06)
--     / \      Initial Graph to Evaluate   /          \
--    @   6                               010 (P 02, P04)  06 (E 6, -)
--   / \                                  /       \
-- Fadd  5                               02 (E 2, 31)   04 (E 5, -)
-- Build Evaluation Graph
CFUN Fadd 2
CINT five 5
CINT six  6
LDD 100
PUSHCONST five
PUSHCONST Fadd
MK_APP
PUSHCONST six
ROTP              1
MK_APP
EVAL 0
GET_FST
GET_BYTE          73
MK_VAL_PR
SIGNAL
SIGNAL            -- nothing on stack ie. '0', which is a signal to halt

DEF_FUN Fadd      -- optimized add.  Note that it destroys it's args
GET_FST
GET_FST
ADD
MK_VAL
UPDATE 1
RET 0
END_FUN Fadd
```

Appendix C: Bit-slice G-machine Instruction Set

CONTENTS

This section describes the PCU instruction set of the Bit-Slice G-Machine, the IFTU Translation instructions, the GAR and Literals Queue, and the Host Processor Interface. It gives a programmers eye view of the PCU and the IFTU translation unit.

This should be viewed as a supplement to the G-Machine Architecture Handbook [Kie84] in that it describes a particular G-Machine implementation. Descriptions of G-Codes can be found in the latest version of the Handbook.

G-Machine PCU
Bit-Slice Instruction Set
Version 2.1 (2/85)

1. INTERNAL INSTRUCTION FORMAT

SHORT FORMAT

OPCODE(F)
8

LONG FORMAT

OPCODE(F)	INDEX(X)
8	8

where INDEX is either the stack index or the argument for shift instructions.

2. INSTRUCTION SET

MS LS	000	001	010	011	100	101	110	111
0000	iNOP	MVPM	MVPG0	iPUSHP	iADD	iSHLL	iLZERO	iJZERO
0001	iCLRE	MVVM	MVPG1	iPUSHV	iADDC	iSHRL	iLNOTZERO	iJNOTZERO
0010	iSETE	MVDM	MVDG0	iPOPP	iADDL	iSHLA	iLNEG	iJNEG
0011	iCLRC	MVCM	MVDG1	iPOPV	iSUB	iSHRA	iLNOTNEG	iJNOTNEG
0100	iSETC	MVMP	MVPV	iCOPYP	iSUBB	iROT	iLCARRY	iJCARRY
0101	iINCD	MVMV	MVPC	iCOPYV	iSUBL	iGETBYTE	iLOVR	iJOVR
0110	iDECD	MVMD	MVVP	iROTP	iINC		iLJES	iJES
0111	iLDD	MVMC	MVCP	iROTV	iDEC		iLJEP	iJEP
1000	iINCG	iALLOC	MVGP	iMOVEP	iAND			
1001	iDECG		MVGD	iMOVEV	iANDL			
1010	iZERO		MVLP	iVROTP	iOR			
1011	iINTR		MVLV		iORL			
1100			MVLD		iXOR			
1101					iXORL			
1110					iNOT			
1111	iTRASH				iNEG			

84 simple instructions.

3. DESCRIPTION

The following description of the instructions is an abbreviated version of the standard abstract register description in the G-Machine Architecture Handbook. Nevertheless, it should be fairly self explanatory.

Meaning of Variables

BUS	G-Machine Internal Bus.
PC	Program Counter for C-Memory
DUMP_POINTER	Dump Pointer Register.
E,P	Eval and Pointer bits in tags registers.
GAR	Graph memory Address Register. Attached to BUS and Graph memory.
G_STORE	Graph Memory.

LITERAL_QUEUE	Literal Pipeline register from IFTU to PCU.
TOPS,TOVS	Tops of P and V stacks, respectively.
Z,N,C,V	Zero, Negative, Carry, Overflow status flags. Cin is the carry bit used in mathematical computations.

3.1. Special Control Instructions

iNOP	No operation.
iCLRE	Clear Eval.
iSETE	Set Eval.
iCLRC	Clear Carry bit.
iSETC	Set Carry bit.
iINCD	Increment DUMP_POINTER.
iDECD	Decrement DUMP_POINTER.
iLDD	Load the DUMP_POINTER from the literals queue.
iINCG	Increment GAR.
iDECG	Decrement GAR.
iINTR	Interrupt the HOST processor and halt.

3.2. Memory Transport Instructions

MVPM	[TOPS] → G_STORE. Write to G_STORE location pointed to by G_STORE ADDRESS REGISTER (GAR). Implicit iPOPP in this instruction.
MVVM	[TOVS] → G_STORE. Write to G_STORE location pointed to by GAR. Implicit iPOPV in this instruction.
MVDM	[DUMP_POINTER] → G_STORE
MVCM	[PC] → G_STORE
MVMP	[G_STORE] → TOPS. Read from G_STORE location pointed to by GAR. Implicit iPUSHP.
MVMV	[G_STORE] → TOVS. Read from G_STORE location pointed to by GAR. Implicit iPUSHV.
MVMD	[G_STORE] → DUMP_POINTER.
MVMC	[G_STORE] → PC.
iALLOC	[FREE_NODE_QUEUE] → TOPS. Allocate a cell in G_STORE.

3.3. Local Transport Instructions

MVPG0	[TOPS] → GAR. Access first node. Implicit iPOPP.
MVPG1	[TOPS] → GAR. Access snd node. Implicit iPOPP.
MVDG0	[DUMP_POINTER] → GAR. Access first node.
MVDG1	[DUMP_POINTER] → GAR. Access snd node.
MVPV	[TOPS] → TOVS. Implicit iPOPP, iPUSHV.
MVPC	[TOPS] → PC. Implicit iPOPP.
MVVP	[TOVS] → TOPS. Implicit iPOPV, iPUSHP.

MVCP	[PC] → TOPS. Implicit iPUSHP.
MVGP	[GAR] → TOPS. Implicit iPUSHP.
MVGD	[GAR] → DUMP_POINTER.
MVLP	[LITERAL_QUEUE] → TOPS. Implicit iPUSHP.
MVLV	[LITERAL_QUEUE] → TOVS. Implicit iPUSHV.
MVLD	[LITERAL_QUEUE] → DUMP_POINTER.

3.4. Stack Instructions

iPUSHP	[BUS] → TOPS (unused)
iPUSHV	[BUS] → TOVS (unused)
iPOPP	[TOPS] → BUS
iPOPV	[TOVS] → BUS
iCOPYP	Copies P-stack location X to TOPS.
iCOPYV	Copies V-stack location X to TOVS.
iROTP	Rotates the top of the P-Stack by X.
iROTV	Rotates the top of the V-Stack by X.
iMOVEP	Moves TOPS to location X in the stack.
iMOVEV	Moves TOVS to location X in the stack.
iVROTP	Rotates the P-Stack by the value in the top of the V-Stack. Pops the V-Stack.

3.5. Arithmetic and Logical Instructions

iADD	[TOVS] + [TOVS-1] → TOVS
iADDL	[TOVS] + [BUS] → TOVS
iADDC	[TOVS] + [TOVS-1] + Cin → TOVS
iSUB	[TOVS] - [TOVS-1] → TOVS
iSUBL	[TOVS] - [BUS] → TOVS
iSUBB	[TOVS] - [TOVS-1] - Cin → TOVS
iINC	[TOVS] + 1 → TOVS
iDEC	[TOVS] - 1 → TOVS
iAND	[TOVS] & [TOVS-1] → TOVS
iANDL	[TOVS] & [BUS] → TOVS
iOR	[TOVS] ¦ [TOVS-1] → TOVS
iORL	[TOVS] ¦ [BUS] → TOVS
iXOR	[TOVS] ⊕ [TOVS-1] → TOVS
iXORL	[TOVS] ⊕ [BUS] → TOVS
iNOT	1's complement of [TOVS] → TOVS
iNEG	2's complememt of [TOVS] → TOVS

3.6. Shift Operations

iSHLL	Logical shift left TOVS by X places.
iSHRL	Logical shift right TOVS by X places.

iSHLA	Arithmetic shift left TOVS by X places.
iSHRA	Arithmetic shift right TOVS by X places.
iROT	Rotate TOVS by X places.
iGETBYTE	Inserts X into a new TOVS.

3.7. Control Transfer Instructions

Instructions with a prefix of 'iL' work on local microcode. 'iJ' instructions work the same, but also cause the IFTU to take a jump (the last long address passed in is assigned to the macro PC).

ZERO	Jump if zero (Z).
NOTZERO	Jump if NOT zero ($\bar{Z}$).
NEG	Jump if sign bit is set (N).
NOTNEG	Jump if sign bit is NOT set ($\bar{N}$).
CARRY	Jump if Carry Set (C).
JOVR	Jump if OVeRflow Set (V).
JES	Jump if Eval Set (E).
JPS	Jump if Pointer Set (P).

4. GRAPH ADDRESS REGISTER

The Graph Address Register (GAR) is a register connected to the GBUS, and is used by the memory as the address register for memory reads and writes (see section 4). The low order bit <0> of the GAR is or'd with a signal from the micro-code memory in order to access the first and second elements of a graph node.

At present there is no hardware model for it, unlike the rest of the bit-slice implementation. (74ALS869's perhaps or some similar device could be used).

5. INSTRUCTION FETCH UNIT

The Instruction Fetch and Translation Unit (IFTU) performs pre-fetching of G-Machine Instructions, and translation of G-Machine External Architecture instructions to sequences of simple instructions which are 'fed' to the Processor Control Unit (PCU).

The operations of the IFTU are discuessed in Sections 5.1 and 5.4, and are repeated here for reference purposes.

signal	MAP	JUMP	NOJUMP	
	0	0	0	PCU fetching next instruction/operand
	↑[0]	0	0	IFTU translates the next instruction(s). If instruction is a jump, it saves the address in the literals queue.
	1	↑	0	Jump was taken by PCU. IFTU must flush buffers and get next translation.
	1	0	↑	Jump was not taken by PCU. IFTU can continue with current buffers.

In addition to the **MAP**, **JUMP** and **NOJUMP** signals as IFTU control, there are three control bits associated with each translation instruction. These are directives to the IFTU only, and are executed when the IFTU is translating the next instruction (the second case above).

IFTUCODE		MEANING
000	iftuCONTinue	goto next instruction in sequence
001	iftuEND	end of translation sequence. get another external instruction.
010	iftuJMP	jump absolute to location, use address/index as absolute address.
101	iftuINDEX	use the next literal on the literal queue as the index for this PCU instruction.
101	iftuEXJMP	take the next address from the control memory, and jump to it.
111	iftuNULL	error. this is an unrecognized external instruction

5.1. IFTU Instruction Format

5.1.1. Regular Instructions

	IFTU ADDRESS/INDEX	
IFTUCODE	PCU INDEX	PCU INSTRUCTION
3	8	8

The PCU INDEX, and PCU INSTRUCTION fields serve as an address in the micro-code for the iftuJMP instruction.

The iftuNULL instruction is a bonus at this point, and may not be necessary in an actual implementation. It is strictly for debugging purposes.

[0] A ↑ indicates a action occurs on the rising edge of this signal.

5.1.2. Control Transfer Instructions

The Control Transfer Instructions require two words for PCU and jump transfer address.

word 0

IFTUCODE	PCU INDEX	PCU INSTRUCTION
3	8	8

word 1

<unused>	IFTU JUMP ADDRESS
3	16

5.2. Literals Queue

The IFTU maintains a shift register queue for literals and indices that are detected when translating the control store. At present it is 32 bits wide, with a maximum of 4 elements allowed.

32 bit literals are deposited to the GBUS on a LOW → HIGH transition of the **LITERAL** strobe line.

The queue is flushed when the **JUMP** line is strobed. This is accomplished by clearing all of the presence bits associated with each element.

Appendix D: Bit-Slice PCU/VAX Instruction Times

This Appendix describes the values used to compute instruction times of the Bit-slice PCU and the VAX (see Section 6). In the table below, the VAX instructions in the last column show what instructions were used for comparison. The +'s and -'s in the VAX column indicate that an autoincrement/decrement is performed as well.

PCU Instructions	PCU Cycles	VAX Cycles	VAX Instructions
MVCP	2	6-7	MOVL-
MVDG0	2	5-6	MOVL
MVGD	2	5-6	MOVL
MVLP	2	6-7	MOVL-
MVLV	2	6-7	MOVL-
MVMC	3-5	5-6	MOVL
MVMP	3-5	6-7	MOVL-
MVMV	3-5	6-7	MOVL-
MVPG0	2	6-7	MOVL+
MVPG1	2	6-7	MOVL+
MVPG1	2	6-7	MOVL+
MVPM	4-6	6-7	MOVL+
MVPV	2	6-7	MOVL+
MVVM	4-6	6-7	MOVL+
iADD	2	6-9	ADDL2+
iALLOC	2	18-23	MOVL+;ADDL2,MOVAL--
iCLRE	3	6-9	BICL2
iCOPYP	1	6-7	MOVL-
iCOPYV	3	6-7	MOVL-
iDEC	2	4-7	DECL
iDECD	2	4-7	DECL
iDECG	2	4-7	DECL
iGETBYTE	2	1	MOVZBL-
iINC	2	4-7	INCL
iINCD	2	4-7	INCL
iINCG	2	4-7	INCL
iINTR	2	3	CHM
iJNEG	3-4	4-6	BLSSL
iJNOTZERO	3-4	4-6	BNEQ
iJZERO	3-4	4-6	BEQL
iLDD	3-4	5-6	MOVL
iLJES	3-4	4-6	BBSS
iLNOTZERO	3-4	4-6	BNEQ
iLZERO	3-4	4-6	BEQL
iMOVEP	1	7	MOVL+
iNOP	1	1	NOP
iPOPP	1	6	ADDL2
iPOPV	1	6	ADDL2

iROTP	1	17-36	ROTATE Sequence
iSETE	3	5-8	BISL2
iSUB	2	5-8	SUBL2
iVROTP	3	17-36	ROTATE Sequence
iZERO	1	3	CLRL

The VAX cycle times for each instruction were derived from the VAX Hardware Handbook [DEC80], with the following common assumptions:

— All VAX moves were memory to memory.

— Instruction fetch times for the VAX are not included in the number of cycles. Likewise, fetch times are included in the cycles for each PCU instructions, EXCEPT for the jump instructions. See Section 6.2 for a discussion of this.

— All PCU instruction cycle times were derived from the PCU microcode.

— Only those PCU instructions executed are used for comparison.

— All VAX arithmetic operations do not include time for testing of errors.

— For iALLOC instructions, it was assumed that a new node would always be available in the PCU.

— G-Memory accesses were assumed to take a minimum of 3 cycles, and a maximum of 5 based. This is feasible, as shown in a G-Memory design using some TI DRAMs chips [Ran86].

The ROT(ate) sequence following is intended to give some small measure of accounting for this instruction on the VAX. For indexes greater than 2, add 6-7 cycles. Note: during simulation, no iROT had an index greater than 2.

ROTATE SEQUENCES:

ROT 1		
	move PSTK addr register	1-2
	move top to register	4
	MOVL-	6-7
	MOVL-	6-7
	Total	17-20
ROT 2		
	move PSTK addr register	1-2
	move top to register	4
	MOVL-	6-7
	MOVL-	6-7
	MOVL-	6-7
	Total	29-36

The following table gives the minimum and maximum cycle speedups between the Bit-slice PCU G-Machine and the VAX. This is the same as that of Section 6.1.2, Fig. 6.1, which only shows the maximum cycle speedup.

Test Program	PCU Instr.'s Executed	Bit-slice Cycles	approx. VAX Cycles	approx. speedup
ack1,2	29	55	493-1044	8.96-18.98
ack1,4	114	214	1878-3930	8.81-18.49
ack2,1	80	154	1300-2706	8.68-18.16
fib2	29	55	493-1044	8.71-18.25
fib4a	92	178	1564-3312	8.73-18.35
fib5	155	301	2635-5580	8.74-18.41
fib6	260	506	4420-9360	8.74-18.44
from	21	29	357-756	8.81-18.59
test20	12	20	204-432	8.83-18.63

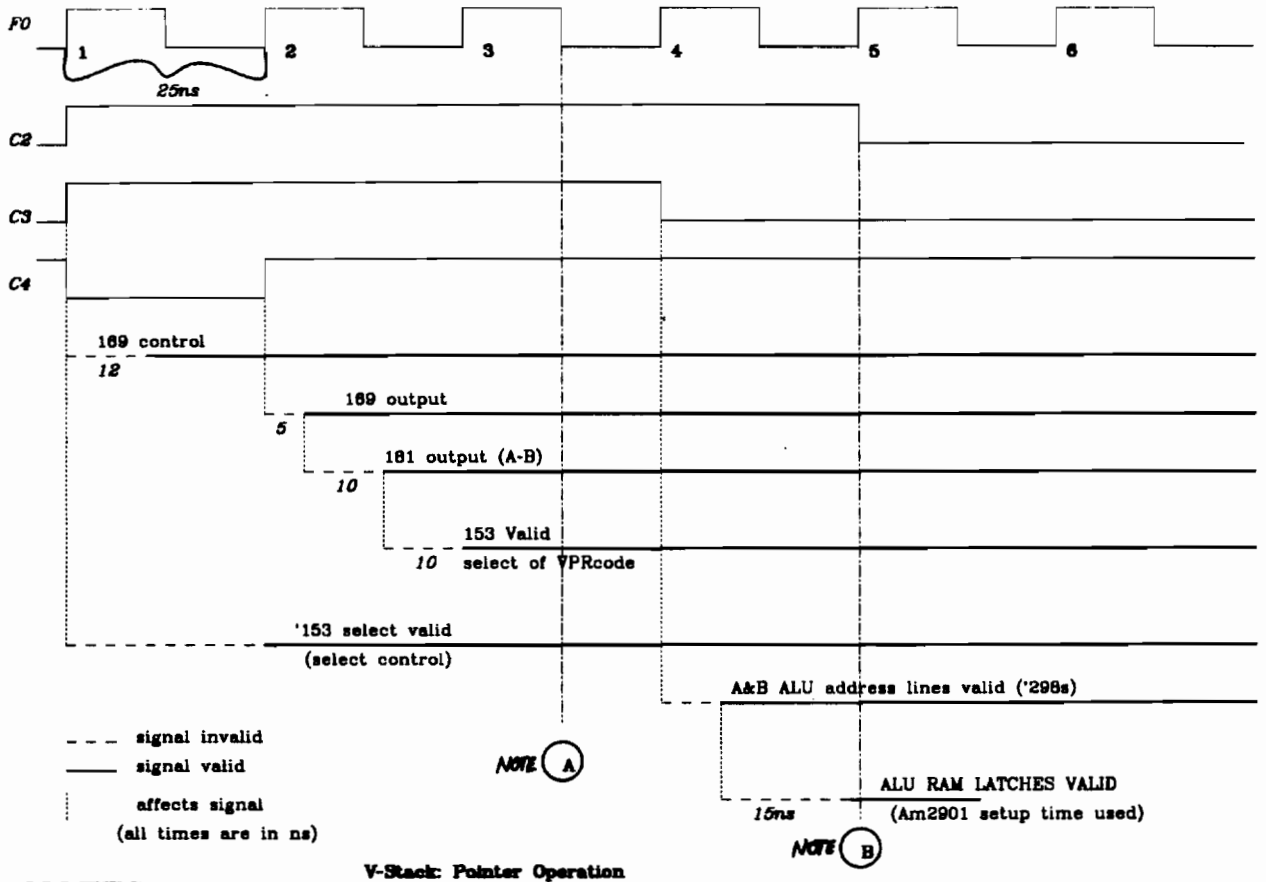
Average speedup from a VAX: 8.8-18.6

ROT(ate) and MOVE Speedups (see Fig. 6.1)

Appendix E: Bit-Slice G-Machine Schematics

The following pages contain the schematics for the Bit-slice G-Machine PCU. These were done on a Mentor Workstation with the NETED schematic editor.

The following figure shows the detailed timing of the V-Stack pointer circuit. Refer to Section 4.6 for a description of the V-Stack, and to the schematic pages 4 and 5 for a listing of the circuits.

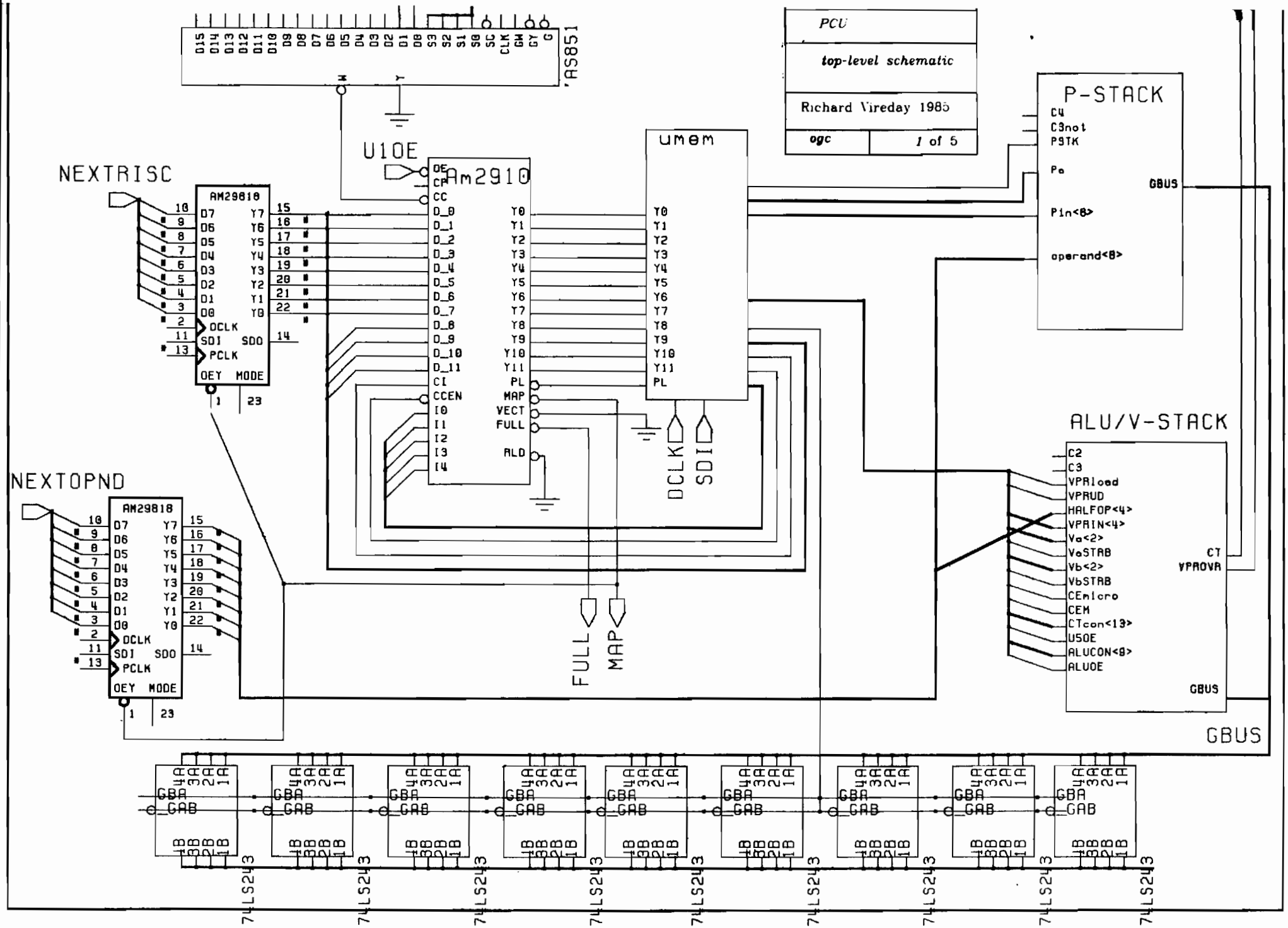


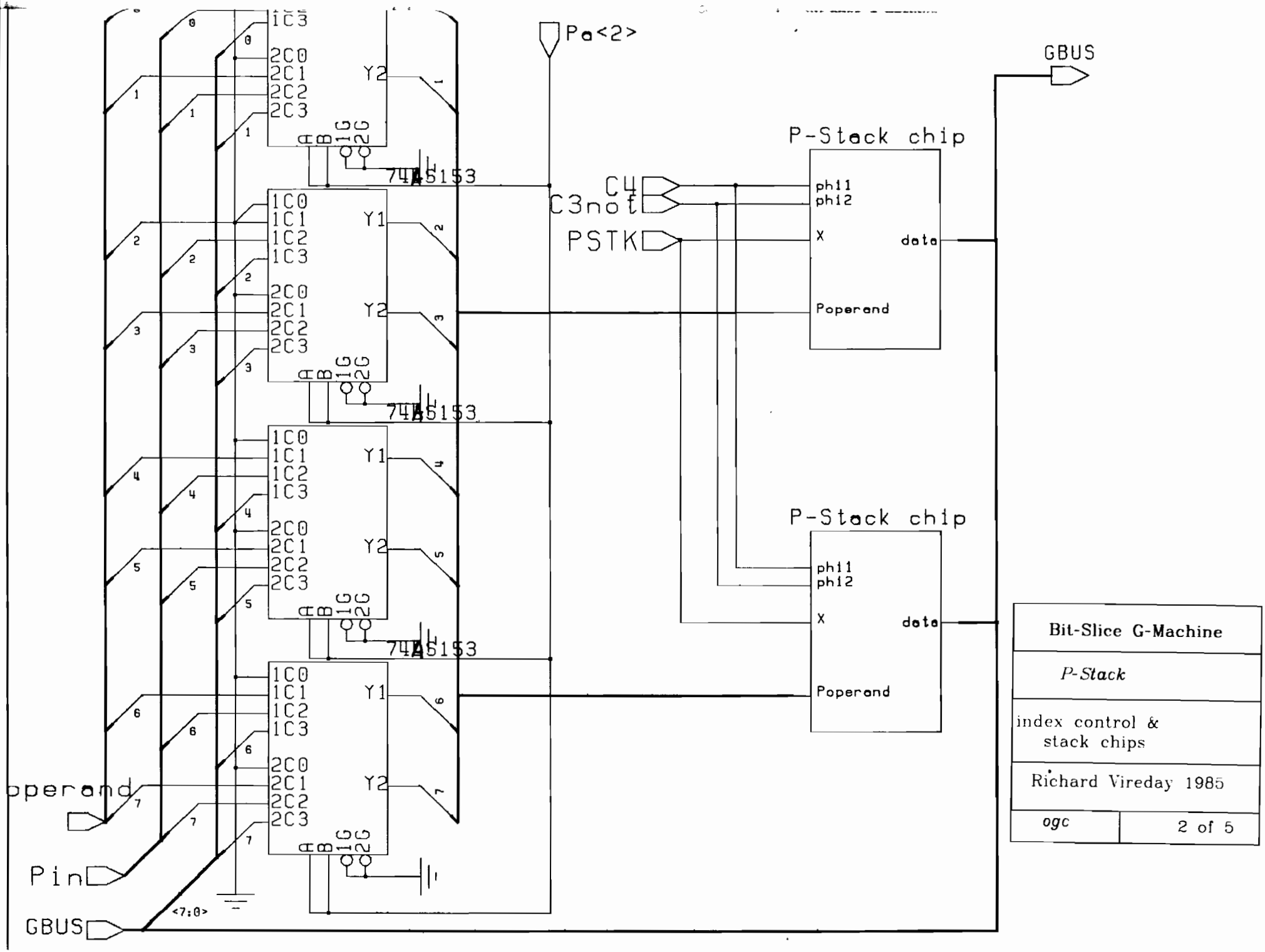
V-Stack: Pointer Operation

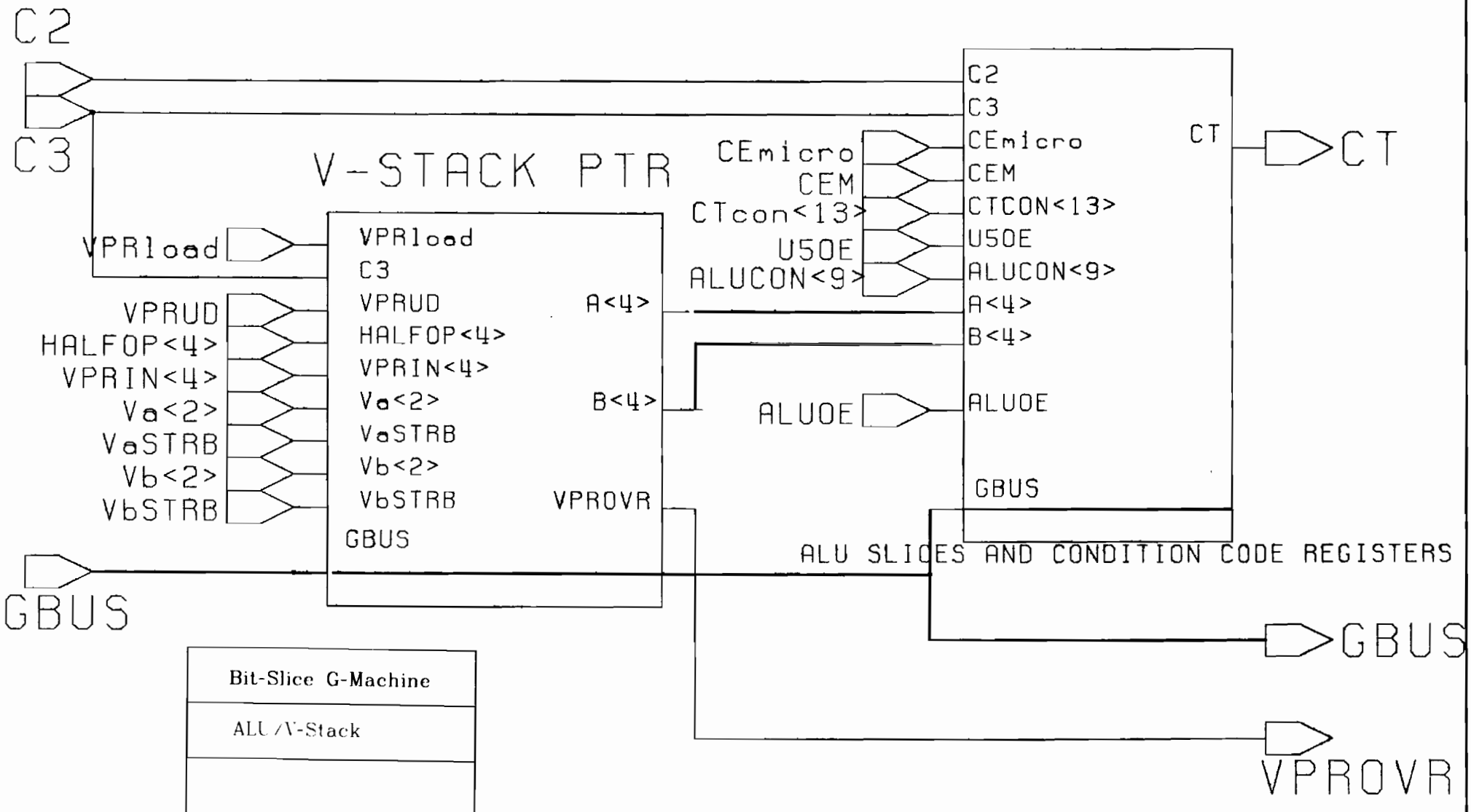
NOTES

A preferred time for '298 clock to drop. This is too close otherwise. When moved here, it gives an extra 10ns for the 2901 setup.

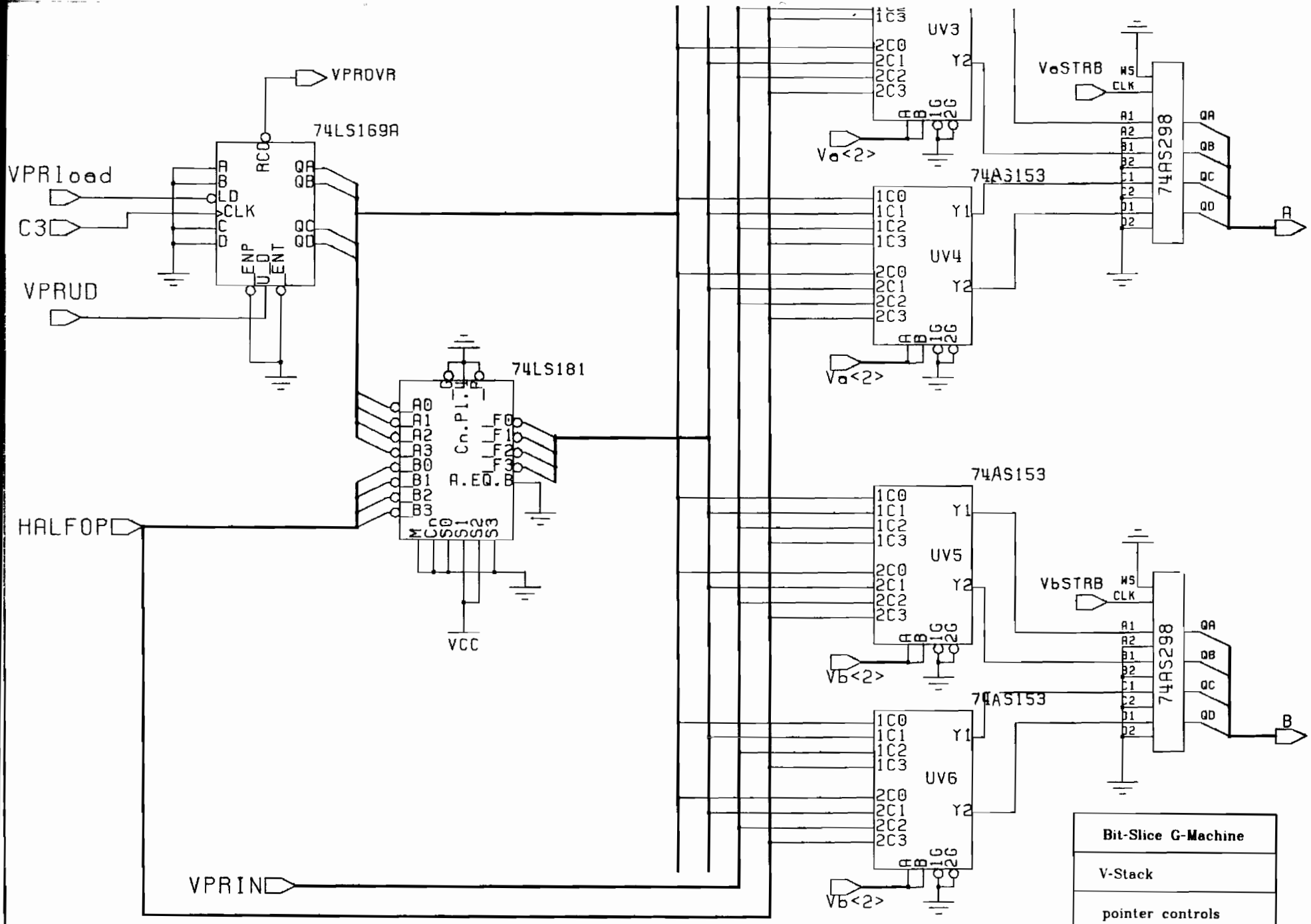
B RAM latches clos and ALU operation can begin.


PCU
top-level schematic
Richard Vireday 1985
ogc
1 of 5
'AS851
D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0 S3 S2 S1 S0 SC CLK GW GY G
W Y
U10E
NEXTRISC
AM29818
D7 D6 D5 D4 D3 D2 D1 D0
Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0
OCLK
SDI SDO
PCLK
OEY MODE
Am2910
OE CP CC
D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 D_10 D_11 CI CCEN I0 I1 I2 I3 I4
Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8 Y9 Y10 Y11 PL MAP VECT FULL RLD
UMEM
Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7 Y8 Y9 Y10 Y11 PL
DCLK
SDI
P-STACK
C4
C3not
PSTK
Pe
GBUS
Pin<8>
operand<8>
ALU/V-STACK
C2
C3
VPAload
VPAUD
HALFOP<4>
VPAIN<4>
Va<2>
VaSTAB
Vb<2>
VbSTAB
CEmicro
CEM
CTcon<13>
U5OE
ALUCON<8>
ALUOE
CT
VPAOVR
GBUS
NEXTOPND
FULL
MAP
GBUS
GBA
GAB
4A 3A 2A 1A
4B 3B 2B 1B
74LS243

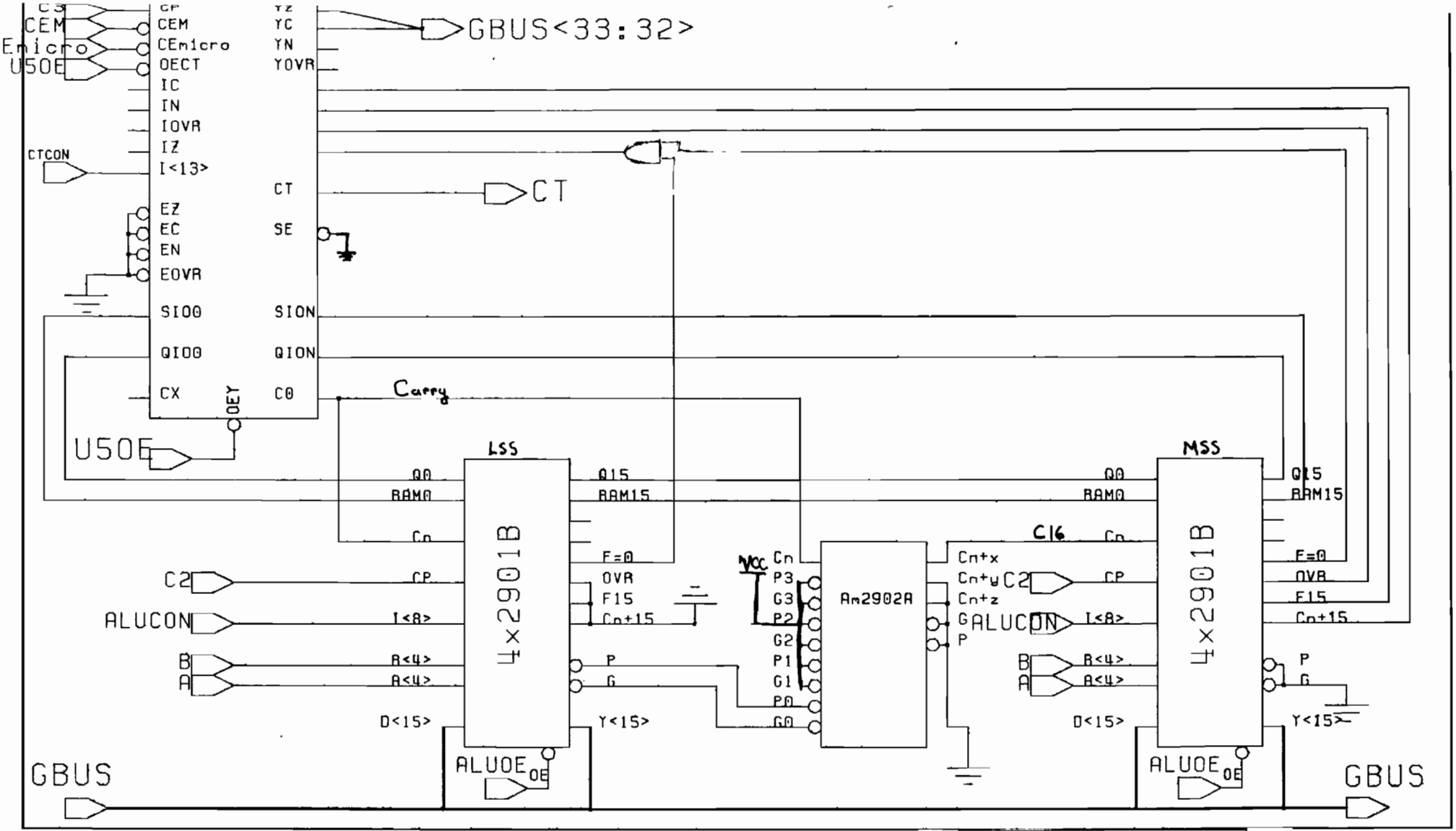

Pa<2>
GBUS
P-Stack chip
C4
C3not
PSTK
phi1
phi2
X
data
Poperand
74LS153
1C0
1C1
1C2
1C3
2C0
2C1
2C2
2C3
Y1
Y2
A
B
1G
2G
Operand
Pin
GBUS
<7:0>
Bit-Slice G-Machine
P-Stack
index control & stack chips
Richard Vireday 1985
ogc
2 of 5



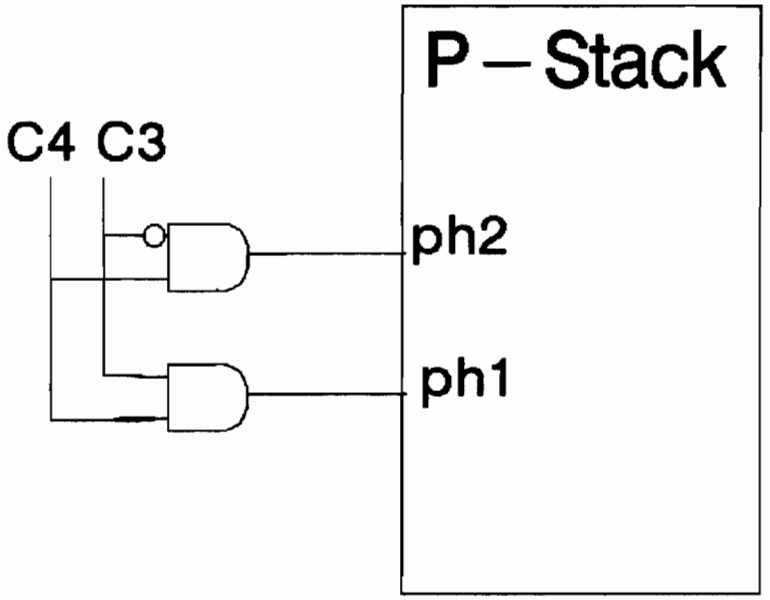
Bit-Slice G-Machine	
ALU/V-Stack	
Richard Vireday 1985	
ogc	3 of 5



Bit-Slice G-Machine	
V-Stack	
pointer controls	
Richard Vireday 1985	
ogc	4 of 5



Bit-Slice G-Machine	
ALU Bit-slice units	
Richard Vireday 1985	
ogc	5 of 5



P – stack clock circuit. Generates non – overlapping clocks from Bit – slice PCU system clocks

C4	C3	ph1	ph2
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

P – Stack Clock Circuitry	
Bit – Slice PCU	
R.Vireday	3/86

BIOGRAPHICAL NOTE

The author was born May 31, 1961 in Santa Monica, California. Two years later the family moved to Las Vegas, where he lived until graduating from Valley High School in 1979. He then entered Willamette University in Salem, Oregon in the fall of 1979.

After finishing the requirements for Bachelor of Science in English at Willamette, the author enrolled at the Oregon Graduate Center in January of 1983 as per requirements of an 3-2 advanced joint degree program between Willamette University and the Oregon Graduate Center. With the completion of the Masters degree from the Oregon Graduate Center, Willamette University will award him with the B.S. The author is the first such person to recieve a joint degree under this program.

The author has been married for less than a year to Pamela Rost, and they have several cats with whom they share their existence.

In March of 1984, the author accepted a full-time position with Intel Corporation in Oregon. After working in the Component CAD Design area for a year, he is engaged in the development of design software for programmable logic devices.