GRIP - a high-performance architecture for parallel graph reduction.

Simon L Peyton Jones, Chris Clack, Jon Salkild, Mark Hardie
Department of Computer Science, University College London
Gower St, LONDON WC1E 6BT. (EMail: simonpj@uk.ac.ucl.cs)

Abstract

GRIP is a high-performance parallel machine designed to execute functional programs using supercombinator graph reduction. It uses a high-bandwidth bus to provide access to a large, distributed shared memory, using intelligent memory units and packet-switching protocols to increase the number of processors which the bus can support. GRIP is also being programmed to support parallel Prolog and DACTL.

We outline GRIP's architecture and firmware, discuss the major design issues, and describe the current state of the project and our plans for the future.

1. INTRODUCTION

Much current computer science research focuses on the parallel execution of a single program. Functional programming languages provide a particularly promising line of attack, since their freedom from side effects greatly relieves the constraints on parallel program execution. The natural implementation model for functional languages is **graph reduction,** but few parallel implementations of graph reduction have actually been constructed - the main exception being ALICE. **[Darli81a]**

It is highly desirable that "real" parallel implementations of functional languages should be built, since doing so is bound to reveal many new issues and clarify theoretical thinking on the subject. In addition, a high-performance cost-effective graph reduction machine would be of considerable use within and outside the research community.

In this paper we describe the architecture of a parallel graph reduction machine called GRIP (Graph Reduction In Parallel), which is under construction at University College London. GRIP is intended to provide high performance at moderate cost. A bus provides a low-cost low-latency access path to shared intelligent memories, and high performance is achieved by using the bus bandwidth effectively.

This approach means that, within its performance range, GRIP should provide more power for unit cost than more extensible parallel computers. Furthermore, its performance range represents a worthwhile improvement over uniprocessors: GRIP can employ almost a hundred MC68020 microprocessors. Our original performance target for a fully populated GRIP was one million reductions per second, but we now hope to considerably exceed this figure.

First we outline GRIP's system architecture, and its firmware architecture. These sections are followed by more detailed descriptions of the major hardware components, after which we discuss the simulation and diagnostics infrastructure, and report on the current project status. We assume that the reader has some familiarity with graph reduction. **[Peyto87a]**

2. SYSTEM ARCHITECTURE AND PACKAGING

In this section we describe GRIP's overall system architecture. We can think of almost any parallel reduction machine as a variation of the scheme shown in Figure 1. The Processing Elements (PEs) are conventional von Neumann processors, possessing some private memory. The Intelligent Memory Units (IMUs) hold the graph, and may or may not also contain a processor.

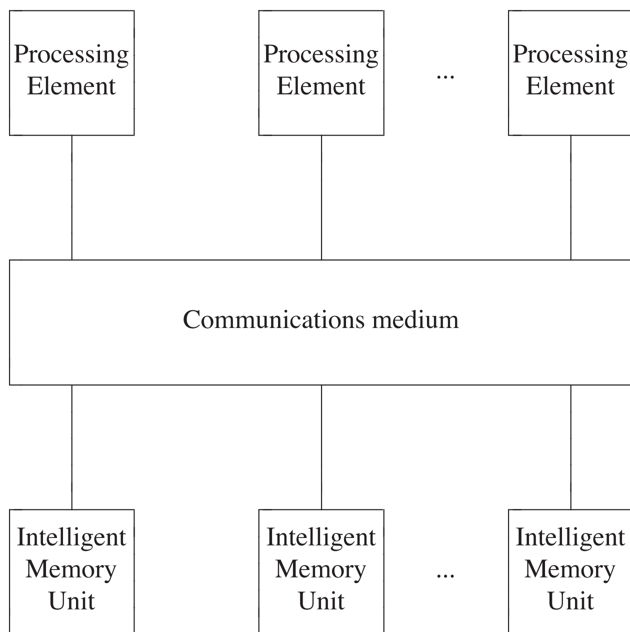


Figure 1. Physical structure of a parallel graph reduction machine

This rather bland-looking diagram actually covers a wide spectrum of machine architectures. The two major axes along which variations are possible are:

- The topology of the communications network.
- The intelligence of the IMUs.

We now discuss these issues and the decisions we have taken for GRIP.

2.1 The communications medium

In GRIP we have decided to give up extensibility in return for a dramatically improved cost/performance ratio by using a bus as the interconnection medium. The low-level machine architecture is largely centred on the requirement to reduce the bandwidth required from the bus, so as to allow a reasonable number of processors and memories to be connected to it.

The bus architecture is described in more detail below, and is based around the IEEE P896 Futurebus standard.

2.2 The intelligence of the IMUs

The amount of intelligence contained in the IMUs has a radical effect on the architecture. The extremes of the spectrum are:

- The IMUs provide only the ability to perform read and write operations to the graph. This results in a classical **tightly-coupled system**, where the graph is held in global memory, and every access to the graph by a PE requires use of the communications medium.

- The IMUs contain a sophisticated processor, each sufficiently intelligent to perform graph reduction unaided. The PEs are now vestigial, since they have nothing left to do, and can be discarded altogether.

  This results in a collection of processor/memory units connected by a communications medium, which is a classical **loosely-coupled system**. It is much cheaper for an IMU to access a graph node held in its own local memory than to use the communications medium to access remote nodes.

Though it is seldom pointed out, there is a continuous spectrum of possible architectures between these two extremes. To move from one extreme to the other we may imagine migrating functionality from the PEs into the IMUs.

To the extent that we can be sure of achieving locality of reference, it is desirable to move towards the loosely-coupled end of the spectrum, and to put local processing power in each memory unit. Since small chunks of computation are quite likely to be local, we have chosen to migrate a certain amount of functionality into the memories, by providing them with a microprogrammable processor optimised for data movement and pointer manipulation. This processor is programmed to support a range of structured graph-oriented operations, which replace the unstructured word-oriented read/write operations provided by conventional memories.

Since the IMUs service one operation at a time, they also provide a convenient locus of indivisibility to achieve the locking and synchronisation required by any parallel machine.

Thus, from an architectural point of view, GRIP's most unusual feature is that it occupies an intermediate point on the closely-coupled/loosely-coupled spectrum. Indeed, depending on the sophistication of the microcode in the IMUs, a range of positions on the spectrum is possible. The IMUs are described in more detail below.

2.3 The Processing Elements

The PEs are Motorola 68020 microprocessors, together with a floating point coprocessor, and as much private memory as we could fit without requiring the microprocessor's bus to be buffered (128k bytes at present, 512k bytes shortly). In this way, we have capitalised on the extraordinarily dense and cheap functionality provided by microprocessors.

2.4 Packaging

At first we planned to build PEs and IMUs on separate boards, so that we could "mix and match" to find the right ratio between their relative numbers. Subsequently, we realised that the scarcest resource was bus slots, and we should strive to maximise the functionality attached to each bus slot by combining PEs and an IMU on a single board.

It did not seem reasonable or possible to fit more than one IMU on each board, but the PEs are so simple

that we are able to fit four of them on each board. Each of these components is connected to the Bus Interface Processor (BIP), which manages their communication with each other and with the bus.

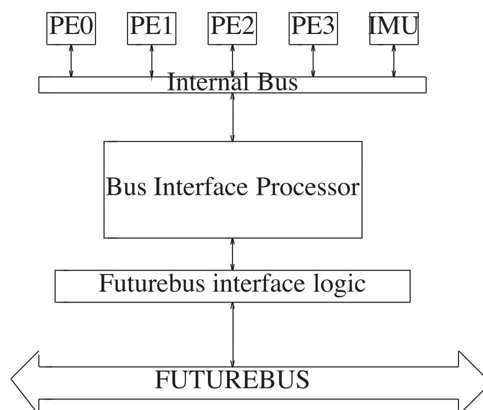


Figure 2. A GRIP board

Figure 2 gives a block diagram of a GRIP board. This (large) single board is then replicated as often as desired, up to a maximum of 21 on a half-metre Futurebus.

An added bonus of this architecture is that communication between a PE and its local on-board IMU is faster than communication with a remote IMU, and saves precious bus bandwidth. The design is not predicated on achieving such locality, but if locality can be achieved it will improve the performance significantly.

2.5 Host interfaces

Since GRIP is a research machine, we are not concerned with very high bandwidth input/output. GRIP is attached to a Unix host which provides all the file storage and input/output required. A 16-bit parallel DMA channel from the host is attached to an off-board extension of one of the PEs.

This host (an Orion) provides the development environment for all GRIP's firmware and software. In addition, each GRIP board is provided with a low-bandwidth 8-bit parallel diagnostics bus, which connects to a custom interface in the Orion. Through this bus all the hardware components on each board can be individually booted up and tested from diagnostic software running in the Orion.

3. FIRMWARE ARCHITECTURE

We now sketch the GRIP's firmware architecture, to provide a context for the detailed discussion of the hardware architecture which follows.

3.1 Overall organisation

One of the PEs is designated as the **System Manager**. It communicates directly with the host, and is responsible for global resource allocation and control within GRIP. All of the other PEs implement supercombinator reduction machines, and some may also support some input/output capabilities.

3.2 Task generation and distribution

The program to be executed is held as a graph, distributed across the IMUs, and is evaluated using supercombinator reduction.

The unit of concurrency is a **task**, which performs a sequence of reductions to reduce a particular sub-graph to weak head normal form (WHNF). **[Peyto87a]** A task may initiate a sub-task to evaluate a sub-graph whose value it will subsequently require; we call this **sparking** a sub-task. A task may be **blocked** when it attempts to access a graph node which is currently being evaluated by another task. The blocking mechanism attaches the blocked task to the blocking node; when evaluation of the node is complete, the blocked task can be resumed.

The pool of tasks awaiting execution or resumption is distributed over the IMUs, which are polled by PEs looking for work. In a heavily-loaded machine, this polling will cause little overhead, because a PE will almost always be successful in finding work, and because an IMU could return several tasks instead of just one. (A back-off strategy may be required when the machine is lightly-loaded, to prevent the bus being saturated with unsuccessful polls.)

There is room for considerable experiment here, to discover effective scheduling strategies and for choosing just which task should be returned to a polling PE. Various strategies have been suggested, such as "most recently sparked first", "least recently sparked first", "resumed tasks first", which may have a substantial effect on the overall performance. **[Sarge86a]**

We discuss most of the issues of this section in more detail in an earlier paper. **[Clack86a]**

3.3 Supercombinator representation and reduction

The supercombinators themselves are represented as a special sort of graph node, rather than being pre-loaded into all the processors. This is important because it allows GRIP to execute programs whose combined supercombinators occupy more memory than each PE has available, and because it allows uniform garbage collection of supercombinators. The latter property is particularly important if GRIP is to support multiple concurrent users, or functional operating systems which dynamically load up and discard sub-programs.

To avoid the cost of repeatedly fetching the same supercombinator, the PE firmware caches copies of the supercombinators it uses. In effect, therefore, the PEs dynamically load and retain the parts of the program that they are presently executing.

We anticipate implementing a variety of supercombinator representations. Initially, we have implemented a simple "template" representation, whereby the supercombinator is represented by a tree-structured template which is instantiated when the supercombinator is applied to its arguments. Explicit LET and LETREC nodes in the template control the construction of graphical instances. Though the template is implemented as a linked collection of nodes in an IMU, it can be sent over the bus in a single (long) packet, because the IMU knows that all of it will be required.

Work has already started on adapting the compiler technology of the Chalmers University G-machine **[Johns87a]** to compile supercombinators to sequential machine code, either for the 68020, or for some abstract machine which the 68020 interprets. This allows considerable scope for optimisation (see **[Peyto87a]** for a detailed discussion), and will produce considerable performance improvements over our initial implementation. Our overall view is that the graph serves as the communication and synchronisation mechanism between the concurrent activities, and should be used as little as possible by sequential threads of computation. Extending the G-machine model to a parallel machine seems to come much closer to this ideal.

3.4 Garbage collection

Initially we are using a stop-and-parallel-mark followed by concurrent-sweep garbage collection mechanism. When any IMU has run out of memory, the disappointed PE informs the System Manager, which tells all the other PEs to stop reduction and get ready for garbage collection. When all PEs are ready, the System Manager tells them all to begin marking.

Marking is carried out using a variant of the task-pool mechanism, whereby the IMUs each keep a pool of pointers to graphs which should be marked. This global pool provides a mechanism for spreading concurrency in the mark phase. Naturally, the IMUs have only a limited amount of space for this purpose, and when this is exhausted, the PEs revert to using the standard pointer-reversal algorithm to traverse the graph.

The only tricky bit is detecting when garbage collection is complete, since even one in-flight packet could contain a pointer to an arbitrarily large unmarked graph; we use a variant of one of Dijkstra’s algorithms to ensure correct termination without recourse to hardware broadcast mechanisms. **[Dijks85a]**

When marking is complete, reduction is resumed. In idle moments, the IMUs concurrently perform a linear sweep looking for unmarked cells, which they link into their freelist.

3.5 The IMU microcode

The IMU microcode implements a variety of operations on graph nodes, which directly support the reduction, synchronisation and blocking mechanisms described in our earlier paper. **[Clack86a]**

In addition, the IMUs have to support the task pool, scheduling algorithm, garbage collection mark and task pool operations, and subsequent concurrent scanning.

4. BUS ARCHITECTURE AND PROTOCOLS

In order to use the available bus bandwidth as efficiently as possible, we chose to use a packet-switched bus protocol. This in turn led us to design a packet-switching bus interface, called the Bus Interface Processor (BIP). We now explain the reasons for this choice, and describe the protocol and BIP architecture.

4.1 Packet switching

When using a conventional (circuit-switched) bus protocol, a processor that wishes to read a remote memory location first acquires the bus, then applies the address of the memory location, waits the access time of the memory, and finally reads the data off the bus. During the memory latency, the bus is not in active use, but cannot be used by any other processor. This latency may be relatively long, especially if the memory is intelligent.

In GRIP, therefore, the PE sends an instruction **packet** to the IMU, containing details of the operation the PE wishes to be performed, and then relinquishes the bus. While the IMU is processing the packet, the bus is free for other use. When the IMU has completed the operation, it sends a reply packet back to the originating PE.

PE-to-PE communication is also catered for.

4.2 Packet format

A packet contains one or more words, subject to some fixed maximum size (currently 256 words). Each word contains 33 data bits, and one "last-word" bit, which is used to identify the last word of the packet; the packet length is thereby defined implicitly.

The first word of a packet, called the **address word**, contains routing information, and is interpreted by the BIP; subsequent words are simply transferred without interpretation. In particular, the address word contains the following fields:

Field	Size
Board Address	5 bits
PE Address	2 bits
Opcode	5 bits
Other info	21 bits

The Board Address uniquely identifies the destination board. If the Opcode is zero, then the destination is a PE, and the PE Address identifies it; otherwise, the destination is the (unique) IMU on the destination board, and the PE address identifies the sending PE. On arrival at the destination board, the BIP automatically replaces the Board Address field with the board address of the **sending** board (which is known to all boards during data transfers).

The effect of these conventions is that it is particularly simple for an IMU to construct the address word of its reply to a PE, because the address word of an incoming request already contains the Board Address and PE Address fields of the sender.

The Other Info field is not interpreted by the BIP, but is used in PE-to-IMU transfers to indicate the address of the cell to be operated on; the Opcode indicates the operation to be performed. The packet format is depicted in the following figure:

Bit	33	32 31	30 26	25 21	20 0
	1	PE Addr	Opcode	Board Address	Other Info
	1	First data word			
	...	...			
	0	Last data word			

Figure 3. Packet format

4.3 The Bus Interface Processor

The BIP acts as a "Post Office" for the packet-switching system. It contains a fast buffer memory area which contains packets in transit, in which packets must be built before they are sent, and from which they may be read on receipt. The buffer memory is currently 8k words of 34 bits.

This buffer memory is divided into a number of fixed-size packet frames, whose size is a power of two. The memory can then be addressed by supplying a packet address as the most significant address bits, and a sub-packet address as the least significant address bits. There is a (strap-configurable) trade-off between the number of packet frames and their size, but the hardware imposes a limit of 8-bit packet and sub-packet addresses.

The BIP also manages an input queue of packets for each PE and the IMU, a queue of packets awaiting transmission over the bus, and a free stack containing empty packet frames. These queues contain packet **addresses** (not the packet data), which are used to address the packet data held in the buffer memory. Figure 4 gives a block diagram of the BIP organisation:

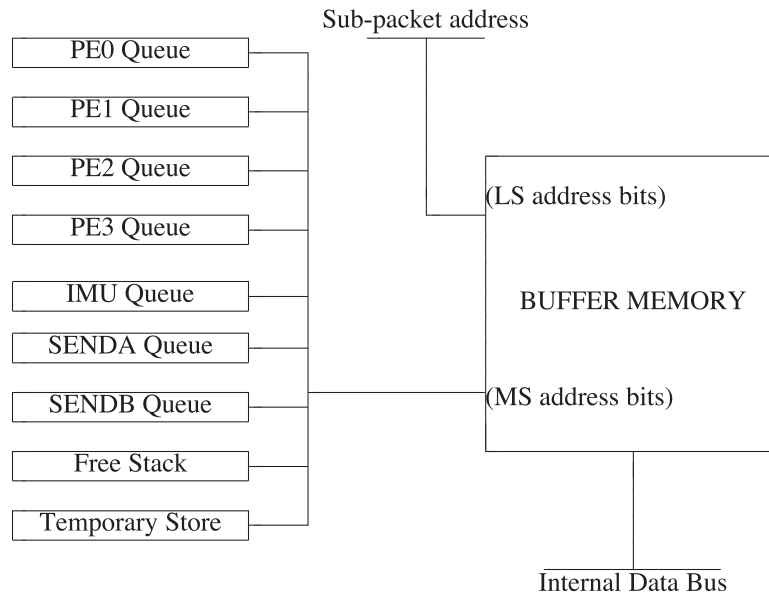


Figure 4. BIP block diagram

This organisation confers a number of advantages:

(i) When the board acquires the bus, any packets awaiting transmission can be sent out using the Futurebus's fast two-edge block-transfer handshake protocol, without the intervention of PEs or IMU.

(ii) If several packets are awaiting transmission (possibly to different destinations), they can be sent out end-to-end, thus amortising the cost of obtaining bus mastership.

(iii) If a PE is sending a packet to the IMU on the same board, or indeed to another PE on the same board, the BIP can simply transfer the packet (address) into the appropriate input queue. The PE sees a single uniform interface.

(iv) Packet addresses can be transferred from one queue to another in the same cycle as they are being used to address the buffer memory. Thus, for example, to send a one-word packet, a PE performs a single memory write cycle to

(1) claim a buffer from the Free Stack,

(2) load the data into it, and

(3) transfer it into the destination queue.

(v) There is considerable scope for ingenuity. For example, the BIP/IMU interface has a small state machine which prefetches the next word which the IMU will require. If the IMU's input queue is empty, this state machine falls into a "mouth-open" state. When the word it is waiting for is loaded into the BIP's buffer memory by (for example) the Futurebus receive logic, the state machine spots this fact, and loads the BIP/IMU interface latch with the data word as it goes by. This avoids the latency which would be caused if instead the IMU subsequently acquired BIP mastership and fetched the word out. This sort of thing is only possible because of the un-specialised nature of the internal bus.

There are a few complications. Firstly, when transmitting packets over the Futurebus, the destination board may not have an empty packet frame in which to store the incoming data. In this case, it signals this fact to the source board, which transfers the packet into the Resend Queue, instead of onto the Free Stack as would be the case for a successful transfer. the Send and Resend Queues swap roles when the send queue has been emptied - hence the titles "SendA" and "SendB" in the diagram.

Secondly, a PE may build a packet in stages, using a number of separate BIP cycles. The Temporary Store provides a place for the packet address to reside while the packet is being built.

Thirdly, the queues must be genuine FIFOs, not stacks, so that the system is guaranteed to maintain the order of packets sent from any particular sender to any particular recipient. This ensures that if a PE, for example, sends out several packets before receiving any of their replies, it can work out which reply corresponds to which of the packets it sent. The Free Stack has no such constraints, and a stack is cheaper to implement than a queue.

4.4 BIP implementation

The BIP is implemented as a fully asynchronous module, built largely out of PALs. It incorporates an asynchronous arbiter to allocate BIP mastership, and uses a simple Go/Done protocol to handshake with the current master.

The Futurebus interface logic, which handles the transmission and reception of packets over the Futurebus, is implemented as two separate asynchronous state machines, which act independently as BIP bus masters. Transmission of data over the bus is pipelined, so that one word can be transmitted over the bus while the next word is simultaneously fetched from the BIP.

5. INTELLIGENT MEMORY UNIT ARCHITECTURE

The Intelligent Memory Unit is a microprogrammable engine designed specifically for high-performance data manipulation operations on dynamic RAM. In this section we outline the internal architecture of the IMU and the model of the graph which it supports.

5.1 Data representation

The IMUs hold the **graph**, which consists of an unordered collection of **nodes**. Each node is represented by one or more **cells**, each of which occupies a contiguous area of RAM in the IMU.

The IMU can be programmed to support nodes of either fixed or variable size. If the node size is variable, then the programmer can choose to represent a node using a linked structure of fixed-size cells, or using a single variable-sized cell. There are many storage management and garbage collection issues here, but the point is that the IMU is sufficiently programmable to allow these trade-offs to be explored.

For the GRIP prototype we have chosen to use fixed-size nodes and cells. To give a more concrete idea of what the IMU is designed to do, we now give the details of this prototype representation. However, it should be emphasised that the representation is largely microprogrammed, so that much of the rest of this section describes firmware decisions rather than hardware constraints.

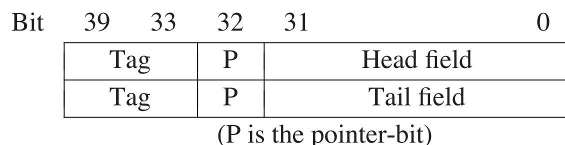
In the prototype representation, each cell contains some **tag bits** and two **fields** - a **head field** and a **tail field.** A field contains

(i) either

  a. a pointer, or

  b. a full 32-bit atomic data object.

(ii) a "pointer-bit", to distinguish one from the other.



Bit 39 33	32	31 0
Tag	P	Head field
Tag	P	Tail field

(P is the pointer-bit)

Figure 5. GRIP prototype cell representation

The cell is stored in two contiguous 40-bit words, each word containing a 33-bit field, and 7 tag bits. The tag information is split between the head and the tail, but since it will be treated as distinct sub-fields (garbage collection mark bits, reference count, cell type, etc), this does not seem to be a problem.

The hardware treats the top 5 tag bits (bits 35-39) specially by allowing them to control 32-way jumps. This allows very fast run-time case-analysis of cell types.

A pointer field is represented as follows:

Bit 32	31 26	25 21	20 0
1	Flags	Board Address	Cell Address

This representation is built into the hardware, which has to manipulate pointers directly. The 5-bit Board Address identifies the IMU which holds the cell, and the Cell Address identifies the cell within an IMU. This provides a 64M-cell address space (equivalent to 640M bytes).

The Flags field can be used for any purpose. For example, one bit could be used for a unique reference bit; **[Stoye84a]** one could be used to indicate that the graph pointed to was already evaluated; and so on. Like the top five bits of the word, the Flag bits can also be used to drive 32-way jumps, allowing fast case-analysis of the Flags field of pointers.

5.2 Data path

The requirement for rapid data movement has led us to design a rather unusual data path. Conventional microprogrammable machines usually consist of a dual-ported register file driving a two-input ALU, whose output is fed back into the register file, thus placing the (slow) ALU on the critical path. We have instead based the data path around a specially-programmed data multiplexer, built out of PALs, which provides two 40-bit registered outputs. Figure 6 gives the block diagram of the data path.

Taking the ALU off the critical path has allowed us to design to a cycle time of 64ns, and the two output

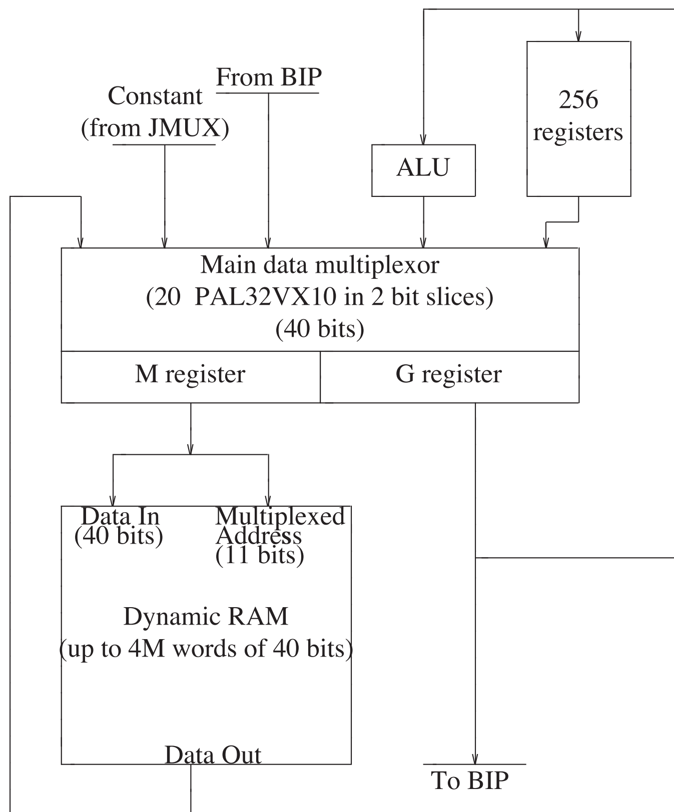


Figure 6. IMU data path

ports of the main multiplexer are extremely useful in overlapping memory operations with other data manipulation. We call the cycle time of the data section a **tick**, to distinguish it from the control section (which cycles at half the speed).

In any tick, the M and G registers can independently be loaded from any of the 5 inputs, or from each other. In addition, a variety of masking operations are available, using a mask stored in the register bank to control the merge of M or G with one of the inputs.

The register bank consists of 4096 40-bit registers, one of which may be read or written in any tick (but not both). This single-port nature has sometimes turned out to be an awkward constraint in microprogramming, but board area and speed preclude dual-porting them. A large number of registers are provided, because they are often used to hold long constants, thus allowing an 8-bit register address in the instruction to translate to a 40-bit constant.

The "Constant" input is actually driven by a 4-input 5-bit multiplexer (the **J mux**), whose output is replicated 8 times to cover 40 bits. Using a merge instruction under a mask from a register, a contiguous field of up to 5 bits can thus be inserted in any position in M or G. One input to the J mux is driven by a 5-bit constant field in the microinstruction; the other three select tag or flag fields in G and the dynamic memory output.

The 32-bit ALU is built out of two quad-2901 parts, and contains a dual-ported bank of 64 registers internally. It is relatively slow, and is cycled at half the speed of the rest of the data section. Its single input port reflects its lack of importance in the design, and it is up to the programmer to maintain data stable on this port through both ticks of an ALU cycle.

5.3 Dynamic RAM operation

The dynamic RAM is built out of Static Column parts, whose main control signals are Row Address Strobe (RAS), Chip Select (CS) and Write Enable (WE). The IMU is extremely closely-coupled to this RAM. Each tick is divided into three **sub-ticks** and, for each sub-tick, one bit in the microinstruction controls the state of RAS and CS. The microprogram thus defines a complete waveform for RAS and CS, with a resolution of one sub-tick (21.3ns), which allows the programmer complete freedom to exploit read-modify-write and within-page fast access cycles.

This flexibility is a unique feature of the GRIP IMU. It is also extremely simple to implement. The three RAS control bits, for example, are loaded broadside into a shift register at the beginning of each tick, and shifted along every sub-tick; RAS is driven directly off one of the shift register outputs. The CS and WE signals are controlled in a similar way, except that only one bit is needed for WE, because data timing constraints fix WE inactive during the first and last sub-ticks of each tick.

A single port from the M register suffices to keep the memory occupied:
(i) In the first tick the row address is supplied from M, RAS is taken active, and simultaneously the two halves of M are swapped over (this is another of the operations provided by the main data multiplexer).
(ii) In the second tick, the column address is now available on the output of M, and CS is taken active.
(iii) Data is available to be read at the end of the third tick, or data can be written, again from M.
Unless a write is taking place, RAS can be taken inactive during the third tick, without prejudicing any read in progress, ready to open an new cell in the next tick. Furthermore, the data just read can be used as the address to be accessed.

If, for example, the program is required to chase down a chain of pointers until some simple condition is met, this design allows the RAM to be cycled in 3 ticks (192ns), which is rather close to the RAM's specified minimum of 190ns. (The termination condition can, of course, be tested in parallel with accessing the next pointer, since the access can always be finished tidily if the condition is true.)

Of course, life is not always so easy, and in practice a programmer would be lucky to achieve a 100% duty cycle for the RAMs, but the close coupling between program and RAM offers considerable opportunities.

The RAM is protected by parity checking only, and no attempt at error correction is made. Refresh is carried out transparently to the microcode, but holds up the entire IMU while it is happening.

5.4 Control section

The IMU control section is conventional, except that it provides for 16-way and 32-way jumps. Figure 7 gives its block diagram.

There is one level of pipelining; the condition code (CC) and multi-way jump condition (J) are latched at the end of one cycle and used for conditional branching in the next. The sequencer is an extended 2910, with a 33-deep subroutine stack, and a 16-bit address path.

The control section cycles every two ticks, because it was impractical to make it cycle every tick without an unprogrammable degree of pipelining. It is for this reason that the control store is so wide. It contains a **cycle part**, which controls the sequencing; and two other parts of identical format, the **tick part** and the **tock part**, which control the data section during the the two ticks of the cycle.

The J latch output drives the least significant 5 bits of the Jump RAM, the other 8 address bits being supplied by the microinstruction to select a page in the Jump RAM. The Jump RAM translates this to an 8-bit address, which is fed to the sequencer, together with 5 further bits from the microinstruction. This

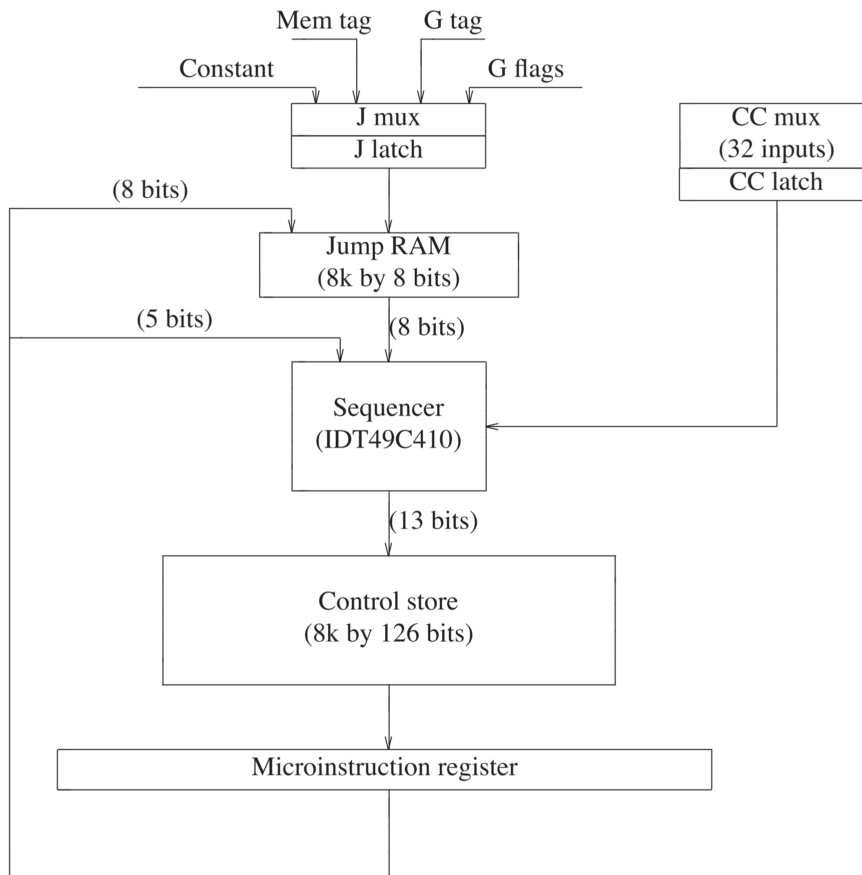


Figure 7. IMU Control Section block diagram

scheme allows 32-way jumps, provided they all land in the same 256-instruction page. Unconditional jumps are performed by forcing the J latch to zeros, having preloaded the bottom location of each Jump RAM page with the page address, so that the Jump RAM becomes an identity function. 16-way jumps are also provided (to save Jump RAM pages) by forcing the most significant bit of the J latch high or low.

6. DIAGNOSTICS AND TESTING

We have devoted considerable effort to developing high-quality disgnostic and simulation software, which runs in the host. In particular, the following software exists:

- An event-driven simulation of the parallel reduction algorithm. This provides approximate performance information and, more importantly, allows us to test and debug the parallel reduction machine. The simulator code is written rather carefully, so that the main bulk of it can simply be recompiled for the target PEs.

  Furthermore, the simulation produces a trace file of the data flowing into and out of each simulated IMU, which is used in testing the IMU microcode (see below).

- A microcode assembler for the IMU.

- A screen-based front panel IMU interface. This allows an IMU to be booted up and tested (using test script files), single-stepped, breakpointed, and so on. Multiple instances of this program can talk to separate IMUs simultaneously.

- An IMU simulator. This is presented to the user through exactly the same front-panel interface as the hardware IMU, which allows development and debugging of microcode independently of the hardware.

- A screen-based front panel BIP interface. This provides stepping and tracing facilities for the BIP, and can be used when commissioning the BIP or when tracing the flow of packets through the system.

7. PROJECT STATUS

Work on the first wire-wrap prototype is now well advanced, and we have built and tested an IMU and BIP, using the diagnostic software described above. A prototype pair of PEs are under construction, together with a second BIP, so we can configure a two-master Futurebus prototype. Once this is working we will lay out a circuit board, which we expect to have completed around the middle of 1987. A second prototype, with several boards, is scheduled for the late summer of 1987.

The project is supported by the UK Alvey Programme, and is being carried out with the collaboration of International Computers Limited and High Level Hardware Limited. Another Alvey project, FLAGSHIP, will be producing a DACTL (Declarative Alvey Compiler Target Language) compiler for GRIP. A third project based at Essex University is developing a parallel implementation of Prolog based on GRIP.

References

[Clack86a] Clack, CD and Peyton-Jones, SL, ''The four-stroke reduction engine,'', in *ACM Conference on Lisp and Functional Programming*, Boston (Aug 1986).

[Darli81a] Darlington, J and Reeve, M, ''ALICE - a multiprocessor reduction machine for the parallel evaluation of applicative languages'', pp. 66-75 in *Proc ACM conference on Functional Programming Languages and Computer Architecture* , ACM, New Hampshire (Oct 1981).

[Dijks85a] Dijkstra, EW, Feijen, WHJ, and Gasteren, AJM van, ''Derivation of a termination detection algorithm of distributed computations'', in *Control Flow and Data Flow - concepts of distributed programming*, ed. Broy, Springer-Verlag (1985).

[Johns87a] Johnsson, T, ''Compiling lazy functional languages'', PhD thesis, Programming Methodology Group, Chalmers University, Goteborg (1987).

[Peyto87a] Peyton-Jones, SL, *The implementation of functional programming languages*. 1987.

[Sarge86a] Sargeant, J, ''Load balancing, locality and parallelism control in fine-grain parallel machines'', Department of Computer Science, University of Manchester (Nov 1986).

[Stoye84a] Stoye, WR, Clarke, TJW, and Norman, AC, ''Some practical methods for rapid combinator reduction'', pp. 159-166 in *Proc ACM Symposium on Lisp and Functional Programming*, Austin (Aug 1984).