

Using a Dataflow Multiprocessor for Functional Logic Simulation

D. Abramson

Division of Information Technology
C.S.I.R.O.
c/o Department of Communication and Electronic Engineering
Royal Melbourne Institute of Technology
124 Latrobe St
Melbourne, Australia, 3000

ABSTRACT:

This paper discusses the implementation of a functional simulator for digital systems. The simulator is designed to run on a dataflow multiprocessor, which allows concurrent execution of logic functions. The paper proposes a model for the simulator and discusses an actual simulator package. Some results are presented which illustrate the speedup attained as more processors are added.

1. Introduction

Digital simulator packages have been used to test the functionality of hardware designs before the hardware is constructed. In this way, many design errors can be eliminated before prototype hardware is produced. Unfortunately many simulators are too slow to exercise real problems.

A number of accelerator hardware units are available which dramatically increase simulator performance. Unfortunately, such units are quite expensive, and can only be used for simulation purposes. This paper discusses the design of a simulation package which can execute on a general purpose multiprocessor. The aim of using a multiprocessor is to increase the performance of the simulation system. The advantage of this approach is that the hardware can be used for other tasks, possibly at the same time. The multiprocessor architecture under consideration is the dataflow model as first proposed by Dennis [1] [2] [3] and as used by TI's DDP [4], the LAU architecture [5], Arvind's research group at MIT [6,7], Manchester University [8], the DDM architecture [9], the EDFG system [10] and SIGMA-1 [19].

2. Dataflow Multiprocessors

Dataflow machines are multiprocessors which execute parallel program graphs rather than sequential programs. The order of execution of the nodes in the graph (or instructions) is determined by the availability of their operands rather than the strict sequencing of instructions in a von Neumann machine. Consequently, the program statements are executed in a non-deterministic manner, and concurrency is obtained if more than one node executes at the same time. Figure 1 shows a sample dataflow graph for a boolean expression and figure 2 shown a model for the hardware required to execute such dataflow programs. In this hardware, the program graph is distributed to the processing elements so that the computation of

A and B can proceed at the same time as C and D. The results of any computation are sent from the processor which holds the source node to the processor which holds the destination node. When the results arrive at the destination processor they wait in the matching unit until all of the operands for the destination node are ready before the next result is computed. Thus the **or** is performed when both A and B and C and D have been computed and the final **and** is computed once the **or** has completed.

There are currently two main classifications for dataflow architectures, *static* and *dynamic*. In the static dataflow model only *one* token (or instruction operand) is allowed on a program arc at any time. In the dynamic model many tokens are allowed on arcs, and their order is determined by special tag fields. A variant of the static model, called *queued static*, allows tokens to form ordered queues on the inputs to nodes. This queuing removes the one-token-per-arc restriction. A good overview of these architectures can be found in [14]. The architecture being considered for this simulation work is a combination of the *queued static* and *dynamic* architectures. In either the static mode or the dynamic mode multiple tokens are allowed on the inputs of a node. If the tokens do not contain a tag value then they are stored in an ordered queue. In this case the token at the head of the queue is consumed when a token arrives on the other input of the node. If they contain a tag value then multiple tokens are placed on a node input without any order, and are only consumed when a token with an equivalent tag arrives on the other input. If multiple tokens of the same tag occur on an input then a queue is maintained for each separate tag value. These matching rules allow the hybrid architecture to operate either as a static machine (with queuing) or as a dynamic machine, or both static and dynamic. The hybrid architecture attracts the advantages of both schemes and avoids most of their disadvantages. It is further discussed in [20]. The simulation system discussed in this paper only uses the hardware in a static mode.

3. Mapping Combinational Logic onto Dataflow Graphs

The simplest and most obvious method for simulating digital hardware using a dataflow multiprocessor is to translate a functional description of the logic into a dataflow graph. The hardware may then be exercised by streaming data tokens through the graph and observing the results. For example, the graph shown in Figure 1 can be exercised by sending data tokens for the signals A, B, C, D and E. The result token value indicates the value which will be produced by a corresponding hardware unit.

In general, any network of logic can be translated directly into a dataflow graph. Logic signals can be transmitted as BOOLEAN data values. The dataflow multiprocessor can offer speed improvements over a uniprocessor system because many nodes may execute in parallel, as they do in a real hardware system. Combinational logic can only be exercised in this manner when a complete set of input tokens is provided. Moreover, it is possible to determine when the hardware has finished when an output token arrives. This simple model fails to simulate sequential logic with storage devices, combinational logic which uses feedback, or blocks of logic which communicate asynchronously. In this situation an input to a gate (or device) may change state, which requires the logic to immediately calculate a result without waiting for the other inputs to change. However, the dataflow model only allows a node to produce a result when all of its inputs are present. Thus, combinational logic devices cannot be directly translated into dataflow graphs as described above. Some examples are illustrated in Figures 3(a), 3(b) and 3(c). In Figure 3(a) the bottom flip-flop is only clocked once, whereas the top one is clocked twice. Thus, the **and** gate only receives one token on the bottom input, and can only produce one output value. The token on the top input is left pending. In Figure 3(b) the values on the top inputs require matching signals on the bottom inputs. However, the bottom signals cannot be produced until a complete set of inputs has been formed. Thus the circuit will remain deadlocked. In figure 3(c) this problem is generalised across multiple communicating sequential machines. These problems occur because dataflow machines are event driven whereas real hardware circuits are state driven.

A solution to this problem involves placing a section of graph, called an *input guard*, on each input of a section of combinational logic. The *input guard* has an input, an output, a request signal and a demand signal. If a new value arrives on the input then the value is forwarded to the output, and the demand signal is asserted. When the request signal is asserted the *current* value of the input is transmitted on the output. Thus, the *input guard* requests the old values of all other inputs when a new value arrives on its input and forwards the new value. A simple example is shown in Figure 4. If a token A arrives on Input 1, the **and** gate is activated with the new token A and the old value of Input 2 rather than waiting for a new value to arrive on Input 2. In this case the input guard on input 1 generates a *demand* signal for the stored value on input 2. This demand signal is interpreted as a *request* signal for the input guard on input 2. Similarly, the *demand* signal from input guard 2 is interpreted as a *request* signal for the guard on input 1. In this way, the **and** gate only ever receives matched pairs of input tokens, and thus will always produce an output.

The use of *input guards* generates much more token traffic when a signal arrives on both inputs simultaneously, because each *guard* will request a new value from the other. Such traffic can saturate the simulation because the output of the logic function will pass on the multiple tokens to the next section of logic. This propagating effect can be removed by placing a section of graph, called an *output guard* on each output of combinational logic. This *guard* only transmits values when the logic level has changed. In this way only state transitions are passed from one gate to another, and the effect of multiple input transitions is diminished. Consequently, an input only receives a new value if it is different from the old value. An example is shown in Figure 5.

An example of how multiple inputs are handled is shown in Figure 6. The demand signals for all other inputs are connected to the request signal of any given input. Multiple connections are produced by using a *non-deterministic merge* of the demand signals, that is, they all send tokens to the same graph node.

The logic required for input and output guards is quite simple, and is shown in Figure 7. The input guard makes use of a special node called a **storage node**. A storage node can be loaded with a new value by sending a token to the **write** input. When a token is sent to the **read** input the previously stored value is emitted. The stored value is emitted whenever requested until it is *overwritten* by a new token value on the **write** input. The guard works by sending the new input value to a storage node. It is written into the node, and then subsequently requested. The order of the **write** and the **read** is guaranteed by the static queued architecture of the RMIT dataflow machine. The input value is also sent to the demand output. If a request arrives then it is sent to the **read** input of the storage node.

The output guard compares the new value on its inputs with the previously held value. The new value is sent to both inputs of an **not equal** node. If the node indicates a match then the input is sent to the output. The graph must be *primed* with an initial value, shown as the black token in Figure 7. After this initial priming, any token which arrives on the input is sent to both inputs of the not-equal node. The new value is thus compared with the primed value. The primed value is then replaced by the new value.

The combination of *input and output guards* reduces the amount of token traffic when a circuit is in steady state, but increases the amount of traffic when a state change occurs. If a section of logic is in steady state, then the output graphs remember the current values of the signals and do not retransmit them unless necessary. In complex digital systems only a small portion of the hardware is active at any time, thus this optimisation should reduce the token traffic significantly. For example, in a computer system only some register values are changed in any machine cycle, thus the hardware devoted to the other registers remain inactive. However, when a change does occur, the logic is required to calculate a new value for each new arrival of an input value. Thus, if n variables change state, the combinational logic is activated n times, rather than one. The effect of these multiple transitions should not be significant because the output graphs will only transmit if a change of output value occurs.

4. Mapping memory devices onto dataflow graphs

4.1 Registers

Almost all combinational logic can be mapped easily onto boolean dataflow graphs. It is possible to implement register type devices directly from feedback circuits. However, a more efficient implementation makes use of **storage nodes**, as described earlier. A two bit positive edge triggered register is shown in Figure 8. Storage nodes are used to hold the current values of each bit. When the clock signal becomes true the storage values are read out, and are sent to the output guards. The output guards only transmit those values which have changed. The clock signal need only be compared with **true** because a new clock value will only be received when it has changed state. Thus, the storage nodes are only read when the clock makes a transition from false to true. When it returns to false, the storage nodes are not read.

4.2 Memory devices

Large read-write memories can be constructed from arrays of registers, however, such an implementation would be extremely inefficient for large memories. Another storage

mechanism, called the *object store*, is available in the RMIT dataflow machine for holding large structures. The object store effectively occupies one or more processor slots in the multiprocessor and provides access to large memory arrays. The object store is further described in [1]. For the purposes of this paper it is only necessary to know that structures are manipulated by special object store access nodes. A structure may be read with the **SSR node**, and may be written with the **SSW node**. In Figure 9 we show the implementation of a conventional random access memory. The address is stored in a number of storage nodes, and only need be read when the **rd** signal is asserted. At this time, either the current data values on the **Din** lines are stored into the object store, or the object store is read, and the results sent via a set of output guards to **Dout**. Storage nodes are used to hold the current **Din** values. The address is passed to the read or write nodes by **pass if true (PIT)** and **pass if false (PIF)** nodes.

5. Detecting Stability in the Graph

The main problem with the use of input and output graphs is that it is difficult to determine when the logic has reached steady state. For example, consider a large combinational circuit which receives a new set of input values (or a partial set). If the output of the combinational logic changes state then a token will be emitted at some later stage. However, if the logic does not change state then a value will never emerge. Consequently, it is very difficult to determine when it is safe to change the inputs. Such information is needed during simulation to determine when the next input vector can be read, or when to change state on a clock signal.

A solution to this problem involves counting some time interval since the last *activity* detected in the logic. After this time interval has elapsed then the logic is assumed to be in steady state. This approach differs from conventional logic simulators because it is not possible to examine the state of all signals at any point in time. With such information available, conventional simulators need only iterate until the signals do not change.

The timeout mechanism is implemented by sending all signal values in the simulation to a central **timing node**. The timing node only emits a token when a pre-defined time period has expired after the **last** token has been received. Such a timing node is similar in operation to a conventional mono-stable circuit. The major disadvantage of this approach is that the network traffic is increased. However, even a very complex hardware unit, such as a central processor, address translator and cache memory built by the author, only contained about 2000 different signals. Also, only a small proportion of these signals change value after each clock step. Thus, the overhead of the stability checking mechanism should not be significant. The mechanism is shown in Figure 10.

6. Implementation Considerations

6.1 Where to place input guards

The input guards need only be applied to each section of combinational logic which does not include any internal feedback. Deciding which integrated circuits should have their inputs guarded can only be determined by examining the logic and detecting feedback. The addition of sequential circuits complicates the task significantly. Even if a large amount of combinatorial logic is detected it is not clear what the optimal size is. If guards are placed on too large a section then the multiple input guard firing will cause excessive token traffic. If the guards are placed on too small a section then they will dominate the amount of computation performed. Consequently, the approach

used in this paper is that the inputs of all *integrated circuits* should contain guards. This approach allows integrated circuits to be connected safely without any circuit analysis. The disadvantage is a small overhead on each circuit. When dataflow code is written for a new integrated circuit the designer can decide where to place the input guards. A program is available for inserting the guard code at pre-defined places to simplify the process. This program is described later in the paper.

6.2 Lazy versus Eager Evaluation of logic functions

Dataflow languages usually allow a choice of lazy or eager evaluation strategies. This choice is useful when writing programs which model integrated circuits. For example, in modelling an arithmetic and logic circuit the designer has the choice of computing all of the function values regardless of the function selector value, and then choosing the correct one (eager evaluation), or using the function selector value to determine which function to evaluate (lazy evaluation). In general, eager evaluation will produce a faster solution. However, when the resources required exceeds those available, the eager evaluation strategy may be slower than the lazy scheme because it performs many computation which will never be required. When an integrated circuit is modeled the designer must choose the appropriate strategy.

6.3 Word oriented devices

When modelling word oriented devices, such as arithmetic units and memory systems, the designer has the choice of constructing the device from single bit units, or packing the data into a machine word. The advantage of using machine words is that they are more efficient because the hardware provides direct support for arithmetic operations. The major disadvantage is that they require an entire word to arrive effectively as one token. This may not necessarily be the case when various bits come from different sections of asynchronous hardware. The use of single bit logic simplifies the construction of complex interconnection patterns because no further circuit analysis is required. All of the simulation studies presented in this paper have modelled word oriented devices at the single bit level.

6.4 Bus transmitters and receivers

The use of output guards means that an input to a gate only changes when the logic level changes state. Thus, it is possible for an input to safely remember the old value. This assumption is only valid when an input is only driven from one source. When a tristate, or open collector, bus is being modelled an input cannot assume it is only connected to one source. This is only one bit of a 16 bit bus changes, the inputs on the bus cannot assume that they are the same as the last time they changed. The solution to this problem is to remove the output guards from bus driver circuits. Thus, they always transmit values even if they have not changed. The inputs then always receive a new set of values.

7. A Prototype Simulator

7.1 Simulator Overview

In this section we describe the structure of a simulation system currently being constructed at RMIT, Melbourne. The system consists of a number of programs which build a dataflow program representing the hardware under investigation. A library of integrated circuit models is first built. Some of these modules are "hand-crafted" in a dataflow language called DL1 [18].

Others are compiled from PLA or PROM definitions. A netlist extracted from schematic circuits is used to determine which modules to include in the final dataflow program. The netlist information is used to connect the components together. Finally the dataflow program is loaded into the RMIT dataflow machine for execution. Test vectors or simulation commands are read from a file, and results are written to a trace file. Optionally, test vectors can be compiled into the dataflow program itself. It is expected that this software environment will be completed by the end of 1987. This tool environment is shown in Figure 11.

7.2 Building component models

Each integrated circuit is modelled by a dataflow *subgraph* written in DL1. DL1 is a single assignment language which permits subgraph definitions and invocations. A subgraph definition contains a list of input parameters and a list of output parameters (and their types). A convention is used for denoting the pin numbers for the various input signals. DL1 is augmented by a meta-syntax for describing which inputs should share input guards. For example, the following subgraph attaches an input guard to pins 1 and 2, a separate guard to pins 4 and 5, a separate guard to pins 10 and 9, and a separate guard to pins 13 and 12. Output guards are attached to pins 11, 8, 6 and 3. The & symbol in an input list indicates that an input guard is required. Input guards which have the same number following the & are connected in the manner described previously. The symbol & in the output list denotes that an output guard should be attached.

```
subgraph
  7400 (pin1&1,pin2&1,pin4&2,pin5&2,
pin10&3,pin9&3,pin13&4,pin12&4:boolean)
  -> (pin3&,pin6&,pin8&,pin11&:boolean);
begin
  pin1 and pin2 -> pin3;
  pin4 and pin5 -> pin6;
  pin10 and pin9 -> pin8;
  pin13 and pin12 -> pin11;
end;
```

7.3 Programmable Logic devices and Read only memories

It is possible to code programmable devices in DL1 in the same way as predefined components. However, a separate program is available for translating PLA or PROM equations into DL1 programs. The PLA programmer accepts input for the PAL assembler PALASM2 [21]. The PROM generator accepts input which contains a starting address and data values. As described earlier PALs are represented as subgraphs containing the boolean equations. PROMs are represented as arrays in the object store.

7.4 Building a simulation program

A commercial schematic capture package is used to generate a net list and component list. The component list is used to determine which DL1 subgraphs to include in the simulation program. The subgraph which models the integrated circuit is found and copied into the DL1 source. Each time that component type is used in the hardware design a subgraph call is planted. The netlist is used to connect the inputs and outputs of the subgraph calls. Each signal is simply code as a new DL1 variable. DL1 variables need not be declared before use. A standard input/output interface is included into the program before it is compiled and loaded into the dataflow machine.

8. Some Experimental Results

In this section we present some simulation results. Since the RMIT dataflow machine is currently being constructed, some sample DL1 programs have been run under an event driven simulator for the architecture.

8.1 Simulation Environment

The architecture is simulated taking into account the amount of network traffic, the varying execution times for individual nodes and the time taken to perform a match operation. The simulations were based on the execution times expected for the 68020 multiprocessor implementation of the machine with 128 processing elements. The floating point computation times are for a 68881 co-processor. The matching store retrieval times are set for those expected performance from a well designed hash table implementation.

A number of performance graphs are generated by the simulator. Graph 1 shows the number of tokens in the machine. The top trace shows the total number of tokens in the communication network and queues together with those stored in the matching store. Graph 2 shows the fraction of execution time devoted to the function evaluation, queue read time, queue write time and matching function. The queue read and write times indicate the amount of time spent transmitting tokens between processors. This graph clearly shows the amount of time spent on the matching process in relation to the other activities in the data-flow machine. Graph 3 shows the number of elements active at any time. The bottom trace shows the minimum activity level during the sampling period and the top trace shows the maximum activity level during the sampling period. The plot in the right top corner of the graphs shows processor activity plotted against time for each processor in the system. Each active processor is marked as a black horizontal line. If the processor is inactive then white space is shown. The processor number is held on the Y axis and time along the x axis. This plot is particularly useful for evaluating the workload distribution algorithm because "hot-spots" show as black areas on the graph.

The hardware being modelled is a 32 bit arithmetic and logic unit, with a barrel shifter and pre-scaler, as shown in Figure 12. This design has been taken from the circuits of a real computer system [22]. A number of simulations were run. These include the ALU exercised with 3 data sets, 72 data sets and 99 data sets from machine traces. In all of the simulations the test vectors are read in before any simulations are attempted. In this way the I/O time can be separated from the actual processing time. The test vector read time can be seen on the activity traces as the periods of very low activity.

8.2 Effect of Timeout Values

In this section we show the effect of using different timeout values. The graphs in Figure 13 show the ALU exercised with 3 test vectors with the TIMEOUT value set to 5 mSeconds. Each test vector can be seen clearly on the activity traces. The processor activity increases whilst the vector is processed, and dies down when complete. The next test vector is not released until the timeout period has expired. In Figure 14 the TIMEOUT value is set to 3 mSeconds. Here the activity trace clearly shows 3 data sets. However, in Figure 15 the TIMEOUT value has been set to 2 mSeconds. This period is clearly too small because only two data sets are visible on the activity traces.

8.3 Number of Data Sets

Figures 16 and 17 show simulation runs with 72 and 99 data sets respectively. The activity traces show that the processor array is well utilised. The 72 data sets were set up as random test data. They included both logical and arithmetic functions. Addition operations were tried for various carry propagation cases. The 99 data sets are taken from real machine traces generated whilst the processor was running some microcode diagnostics. These also show good processor activity. It should be noted that all of these dataflow graphs have been statically allocated to the 128 processor machine. They show a good work load distribution.

8.5 Speedup Characteristics

In Figure 18 we show the effect of restricting the number of processors available to the dataflow graph. The trace shows the characteristic form for a dataflow machine. Close to ideal speedup is achieved until the problem runs out of concurrency, where increased processor numbers give diminished return. It should be noted that these results are only for the 32 bit ALU path of the processor.

9. Conclusion

In this technical report we have shown how digital logic can be modelled on a dataflow multiprocessor. Because the *firing* rules for dataflow instructions require all operands to be present before the instruction can execute some extra logic is required in order to model digital devices. The extra logic takes the form of input and output *guards* applied to the inputs and outputs of integrated circuits.

This approach shows that a multiprocessor should be able to provide significant speedups over uni-processor simulators. We are in the process of constructing a tool set which will help component model construction as well as allowing real designs to be simulated on a dataflow machine.

We have presented a number of simulations on a small logic design. There are still a number of unresolved issues which require further investigation. When the tool set is complete we plan to simulate an entire computer system for which we have circuit diagrams and test programs. This study should illustrate whether the input-output guard arrangement is effective in a large simulation. This work will be reported at a later stage in a separate technical report.

Acknowledgements

The author wishes to acknowledge the support of the project team, and in particular the help of Dr Greg Egan and Mr Mark Rawling who have provided much assistance with all aspects of this work. The software environment is being constructed by Simon Wail. The Parallel Systems Architecture Project at RMIT is being supported by the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) under an Information Technology joint research project.

References

- [1] D.A. Abramson, G.K. Egan, M. Rawling and A. Young, "The RMIT Data Flow machine: The Architecture", Department of Communication Engineering Technical Report, TR112061R, Royal Melbourne Institute of Technology, Melbourne, 1987.
- [2] Arvind and D.E. Culler, "Data-flow Architectures", Laboratory for Computer Science, MIT technical report MIT/LCS/TM-294, 1986.
- [3] Arvind and R.A. Iannucci, "A Critique of Multiprocessing Von Neumann Style", Proc 10th Ann. Int'l Symp. Computer Architecture, Stockholm, June 1983, pp 426-436.
- [4] Arvind and K.P. Gostelow, "The U-Interpreter", Computer, Vol 15, No 2, Feb 1982, pp 42-50.
- [5] M Cornish, D.W. Hogan, and J.C. Jensen, "The Texas Instruments Distributed Data Processor", Proc. Louisiana Computer Exposition, Lafayette, La., March 1979, pp 189-193.
- [6] A.L. Davis, "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine", Proc 5th Ann. Symp. Computer Architecture, New York, 1978, pp 210-215.
- [7] J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor", Proc 2nd Ann. Symp. Computer Architecture, New York, May 1975.
- [8] J.B. Dennis, G.A. Broughton, and C.K.C. Leung, "Building Blocks for Data Flow Prototypes", Proc 7th Ann. Symp. Computer Architecture, LaBoule, France, May 1980.
- [9] J.B. Dennis, G.R. Gao, and K.W. Todd, "Modeling the Weather with a Data Flow Supercomputer", IEEE Trans. Computers, Vol C-33, No 7, July 1984, pp 592-603.
- [10] G.K. Egan, "Data-flow: Its Application to Decentralised Control", Ph.D. Thesis, Department of Computer Science, University of Manchester, 1979.
- [11] G.K. Egan, C. Baharis, M. Rawling, and A. Young, "RMIT Data-flow Project", IREE. Proc 2nd Australian Computer Engineering Conference, Sydney, May 1986.
- [12] J. Gurd and I. Watson, "Data Driven Systems for High Speed Parallel Computing - part 2: Hardware Design", Computer Design, July 1980, pp 97-106.
- [13] A. Plas et al, "LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment", Proc. 1976 Int'l Conf. on Parallel Processing, pp 293-302.
- [14] V.P. Srin, "A Message-based Processor for a Data-flow System", Proc 1984 Int'l Workshop in High Level Computer Architecture, Los Angeles, May 1984, pp 1.10-1.19.

- [15] V.P.Srini, "An Architectural Comparison of Data Flow Systems", IEEE Computer. March 1986, pp 68-87.
- [16] G.K.Egan, and C.P. Richardson, "Object Recognition Using a Data-Flow Computing System", Microprocessing and Microprogramming 7, North Holland, 1981.
- [17] K.S. Weng, "Stream oriented Computation in Recursive Data-Flow Schemas", Technical Memo 68, Laboratory for Computer Science, Massachusetts Institute of Technology, Oct 1975.
- [18] M. Rawling and C.P. Richardson, "DL1 User's Manual", Department of Communication Engineering Technical Report, TR112059R, Royal Melbourne Institute of Technology, Melbourne, 1987.
- [19] T. Shimada, K. Hiraki, K. Nishida and S. Sekigucki, "Evaluation of a prototype data-flow processor of the SIGMA-1 for scientific computations", Proceedings of 13th Annual International Symposium on Computer Architecture, pp 226 - 234.
- [20] D. Abramson and G Egan. "The RMIT Dataflow Machine: A Hybrid Architecture", Department of Communication Engineering Technical Report, TR112057R, Royal Melbourne Institute of Technology, Melbourne, 1987.
- [21] Monolithic Memories, "PALASM 2 User Documentation", Monolithic Memories Inc, Santa Clara, CA 95054-1592.
- [22] D. Abramson and J. Rosenberg, "The Micro-Architecture of a Capability-Based Computer", Proc 19th Annual Workshop on Microprogramming, Oct 15-17 1986. pp138 - 145.

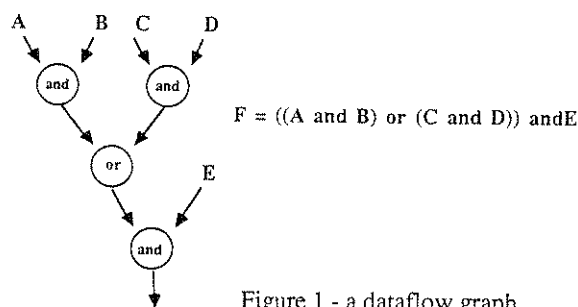


Figure 1 - a dataflow graph

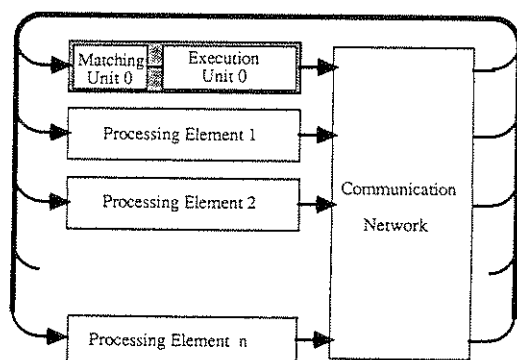


Figure 2 - hardware model for dataflow machine

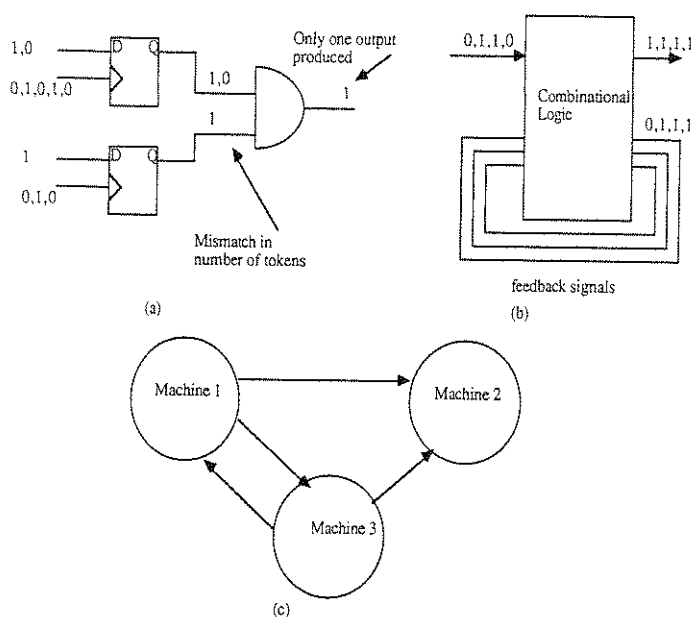


Figure 3 - examples of some asynchronous circuits

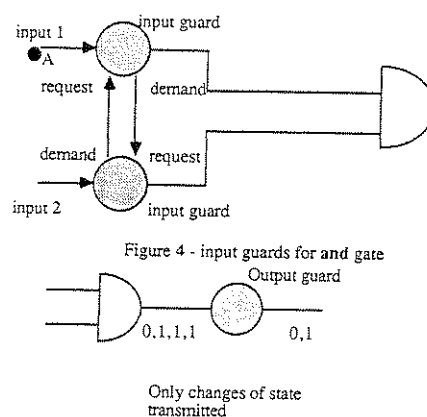


Figure 4 - input guards for and gate

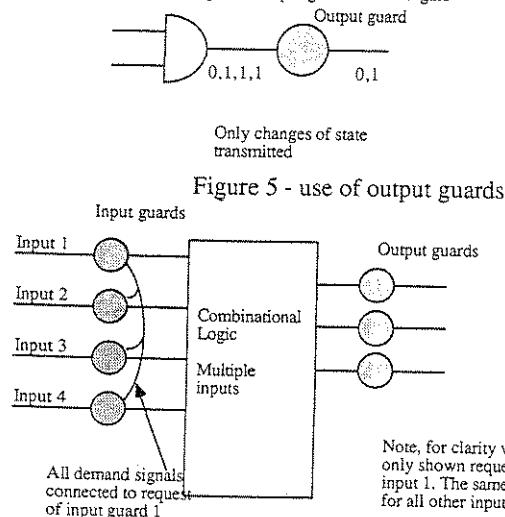


Figure 5 - use of output guards

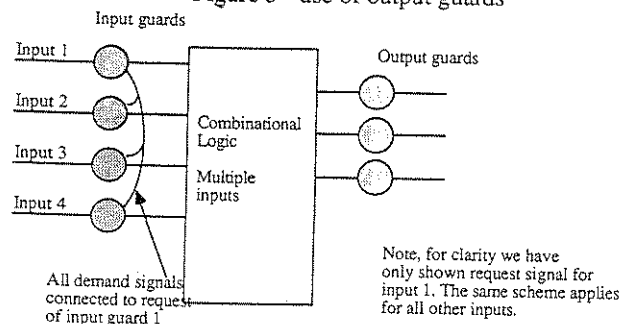


Figure 6 - guard connected to multiple inputs

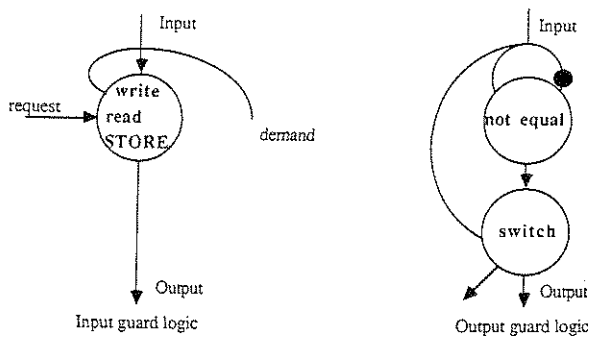


Figure 7 - logic for input and output guards

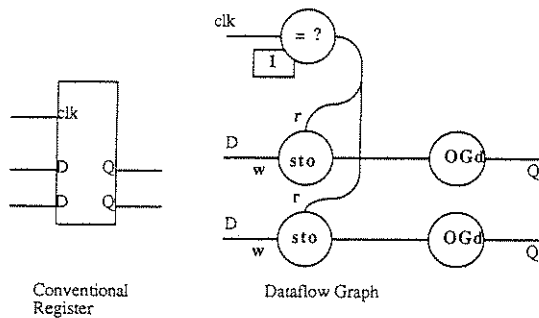


Figure 8 - implementation of register

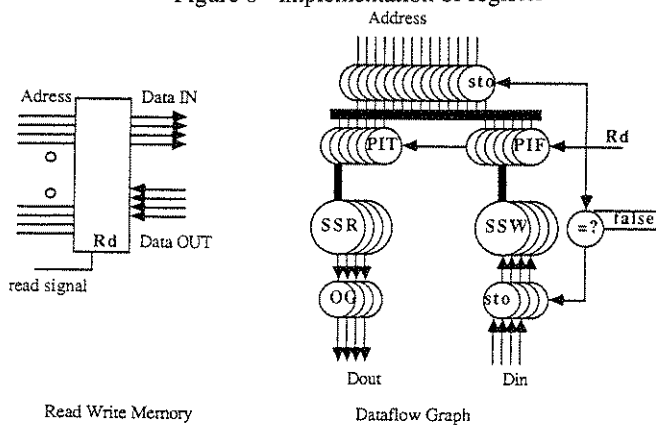


Figure 9 - Implementation of a Random Access Memory

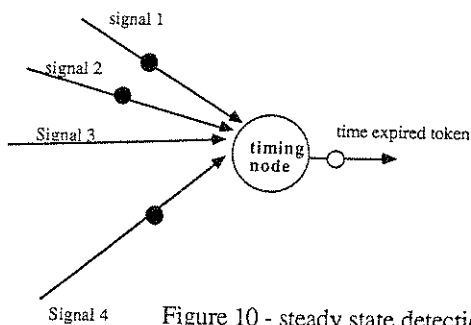


Figure 10 - steady state detection logic

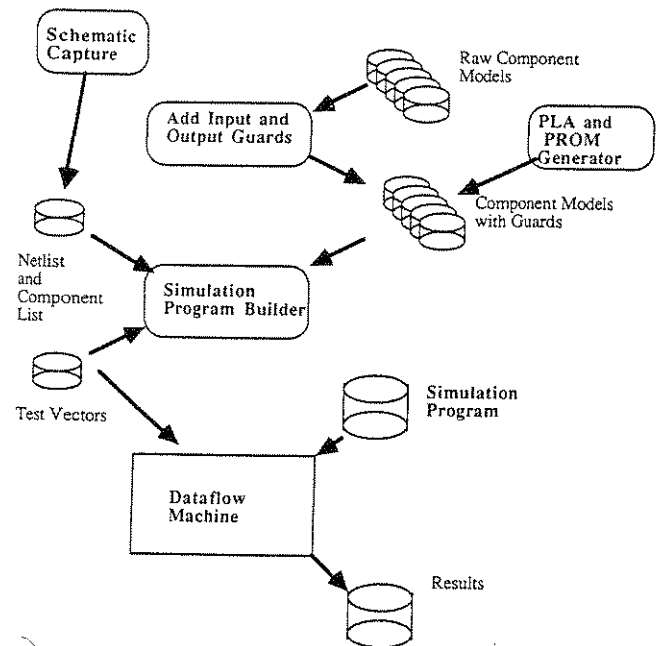


Figure 11 - simulation environment

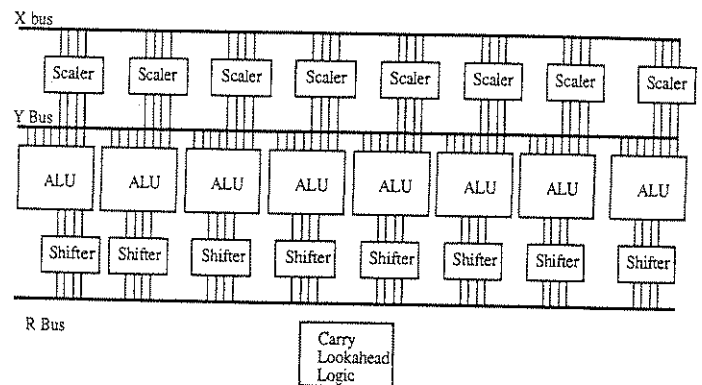


Figure 12 - ALU under investigation

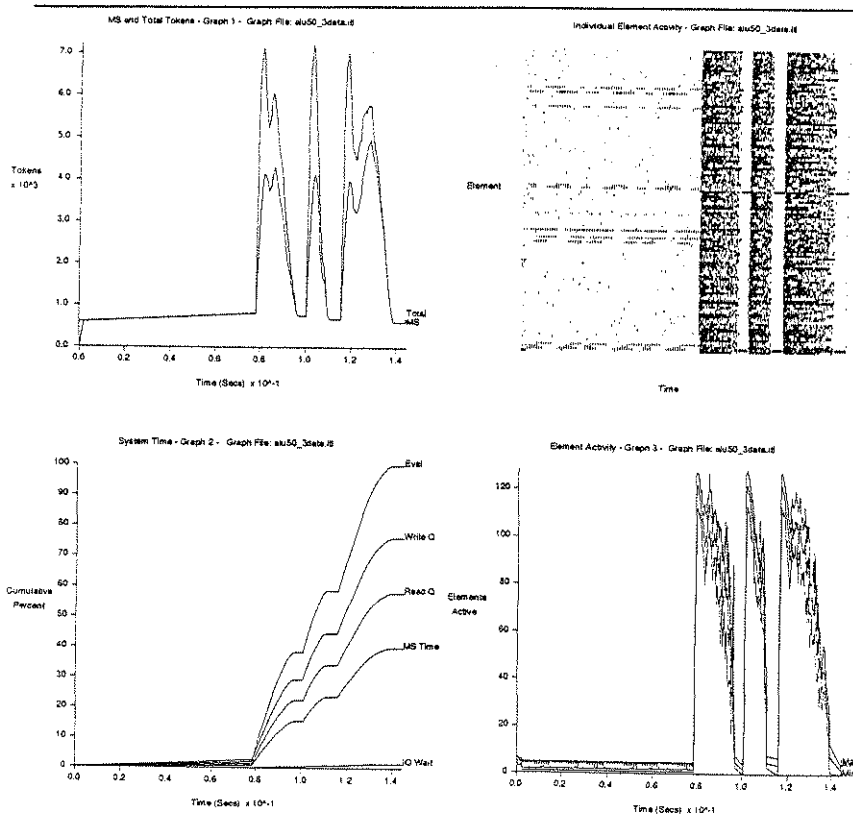


Figure 13 - Three data sets with TIMEOUT = 50

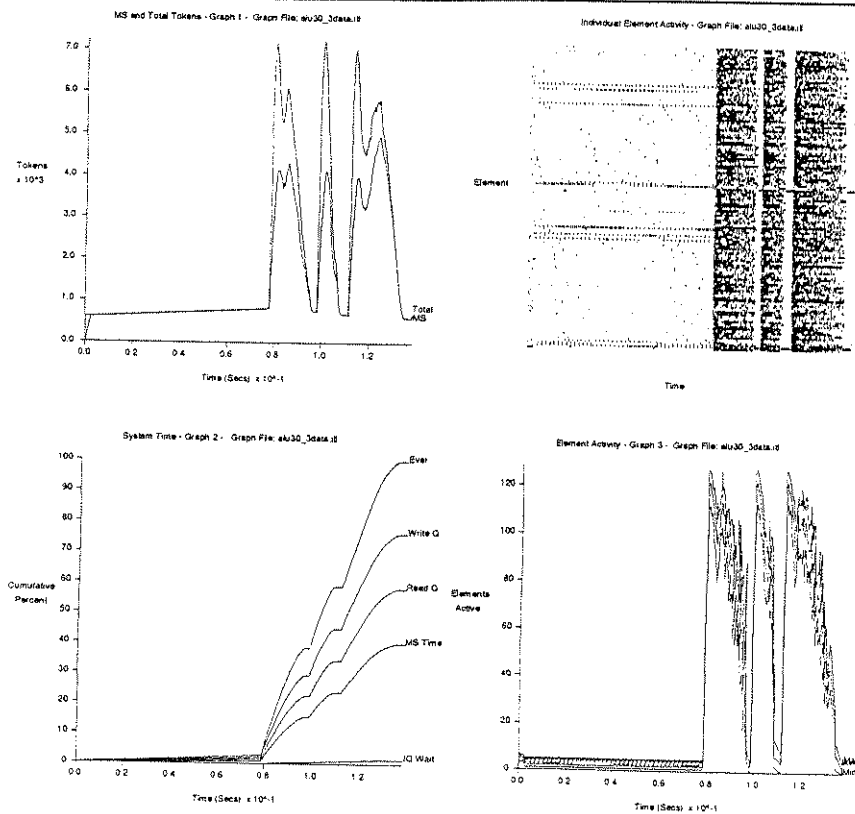


Figure 14 - Three data sets with TIMEOUT = 30

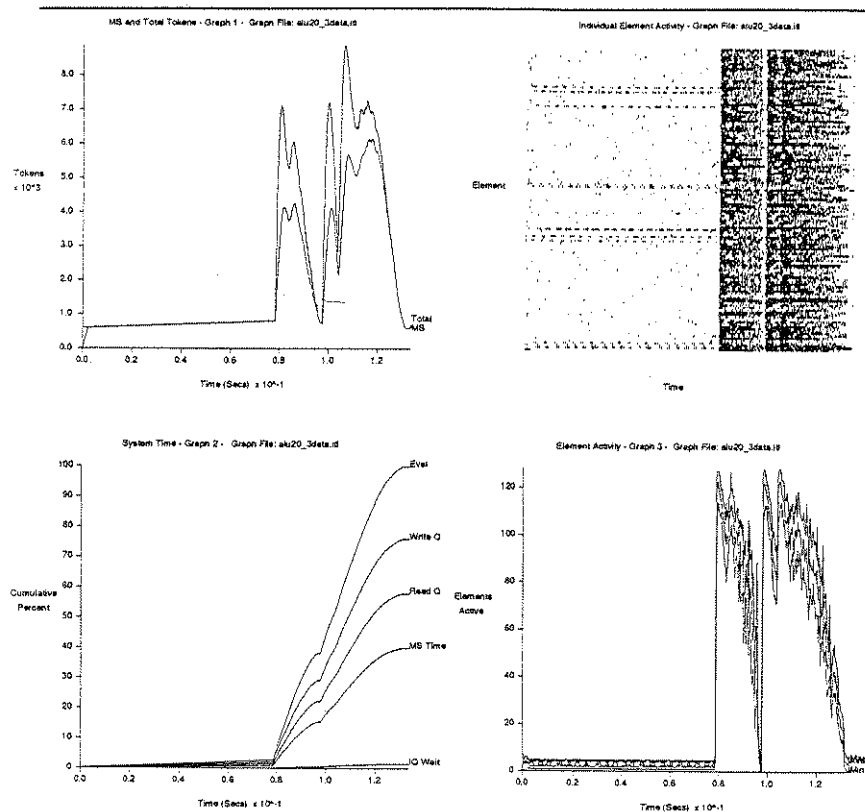


Figure 15 - Three data sets with TIMEOUT = 20

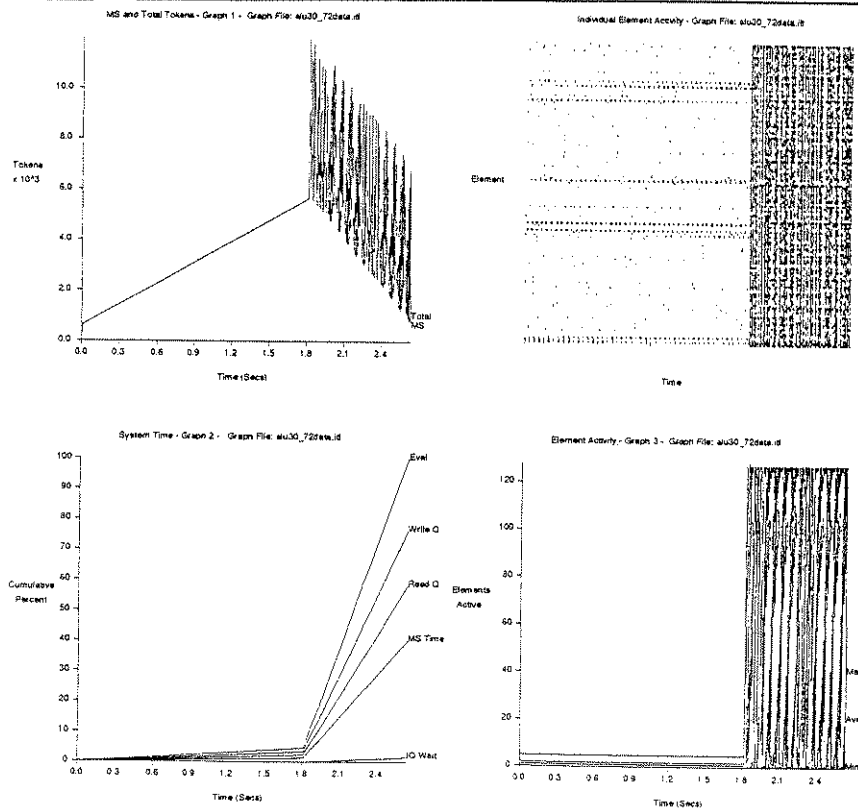


Figure 16- 72 data sets with TIMEOUT = 30

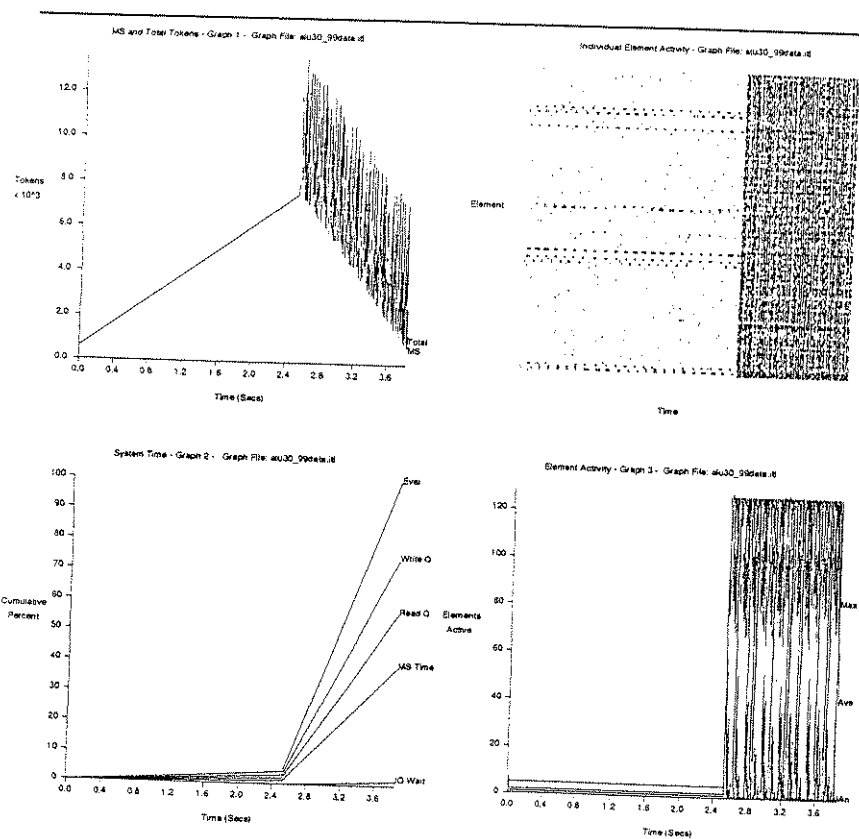


Figure 17 - 99 data sets with TIMEOUT = 30

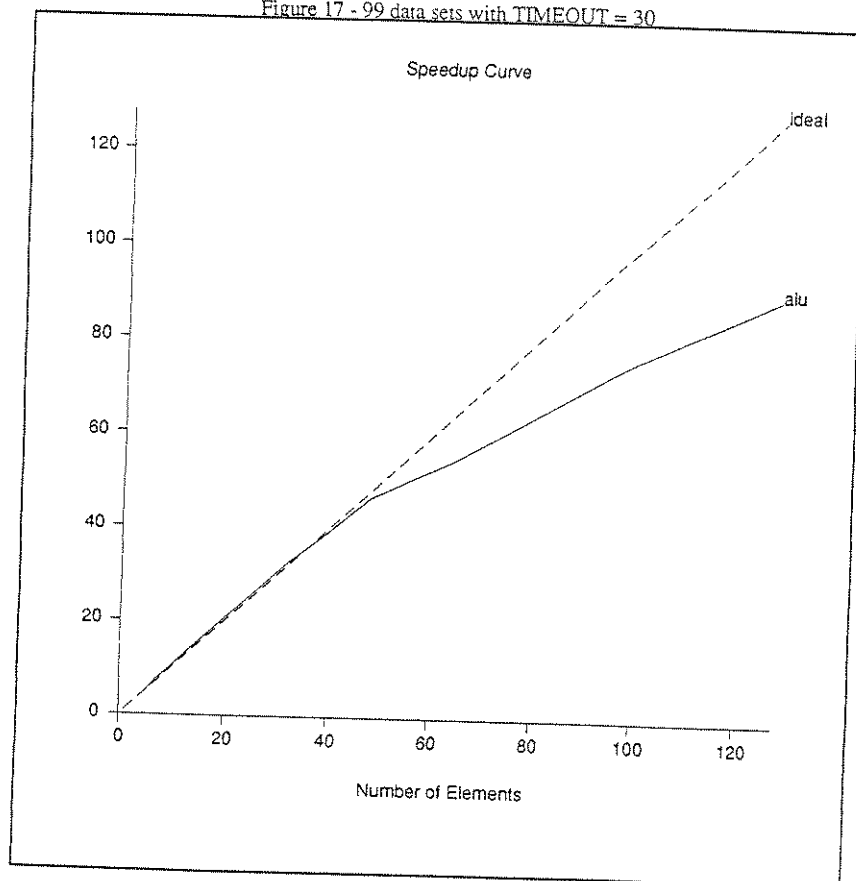


Figure 18 - Speedup Curve for ALU