The USC Macro Data-Flow Simulator*

Namhoon Yoo and Jean-Luc Gaudiot

Technical Report CENG-89-27

October 11, 1989

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, California 90089-0781

* This material is based upon work supported in part by the U.S. Department of Energy, Office of Energy Research Under Grant No. DE-FG03-87ER25043

Contents

List of Tables

List of Figures

1 Introduction

As device technology develops to the limit of speed of light, parallel processing becomes a major trend for obtaining high performance of computation. The conventional model of computation of von Neumann has shown some difficulties in this parallel processing model by its single threadness. Research has been pursued for long time. However the problem of intrinsic overheads in sequencing individual instructions and copying structured data have led to the study of hybrid models of data-flow and von Neumann. [4, 6, 8, 5, 7]

Many hybrid models have been proposed, in which a hierarchy of execution models, context switching mechanism and synchronization method are different. Those proposed models will be reviewed in the following sections and some critical features will be discussed. The discussion will lead to our macro data-flow model.

Our macro data-flow is a scheme having a multilevel of model of execution which higher model is a tagged data-flow and lower level is von Neumann. At the low level, we can exploit the simplicity of control loci of von Neumann model and at high level, the automatic parallel sequencing of data flow. However the size of macro data and macro actors is critical and the partitioning should be carefully done in order to utilize both advantages of the two different models. In order to obtain some ideas about partitioning, a simple simulator has been developed, executing a macro data-flow graph. Micro instructions within a macro actor can be defined, which can access and process those vector data gathered from higher data-flow level. The architectural description of this simulator and some special actors supporting this hybrid model will be discussed in subsequent sections.

Finally the details of instructions are explained as user reference manual including sample programs and description of statistic gathering methods.

In addition to the above hardware simulator, a graph simulator has been developed for simple execution of data-flow graph without resource limit of hardware details. This simulator uses the same program format of same instructions. However simulation time and parameter setting for simulation will be different, which are explained in detail in a separate chapter.

2 Review of Hybrid Models

Most hybrid data-flow models proposed to date use data-flow execution model at the highest level. However, in the late 60s IBM370 adapted a primitive form of data-flow for the scheduling functional units by register tagging at the lowest level of architecture. Except those micro architectures, most recent proposals of hybrid model use process model of execution at low-level sequencing and data-flow synchronization is applied at high level, invoking those processes.

Lubomir[4] termed sequential code segment(SCS) those communicating processes which run respectively by sequentially sequencing instructions. When a SCS meets an instruction slot for which the operand data is not ready, the SCS will be suspended. Another SCS will send a message to the waiting SCS providing the operand data. Then the waiting SCS will be invoked. It is very similar to the "process" in multi-tasking operating system. It is clear that it will reduce much of matching overhead using sequentiality of each SCS. However the efficiency of context switching of SCS has not been demonstrated. the ??? use the slot of SCS as a local storage within SCS itself or a meeting place for synchronization between SCS, but he does not exploit the SCS local storage as conventional state saver like register set for efficiency purpose reducing the context switching time.

Buehrer[5] presented a similar idea of communicating processes through shared memory of I-structure. He defines two memory hierarchy of local and global ones. One of the most important assumptions for viability of his model is again very fast context switching capability by hardware, which makes it possible to tolerate long memory latency. Further, he suggested a compiler's role of placing global latency memory read long before actual local read operation in order to efficiently tolerate the latency.

Iannucci[7] presents a more elaborate and detailed form of hybrid data-flow machine. He called scheduling quanta (SQ) process which runs serially but are blocked by an access to an absent frame of memory. He also definesa liveliness of SQ by dependency of dynamic cycle through I-structure between SQ's, which gives an idea of how to partition a given data-flow graph into SQ's without deadlock. He describes a parallel machine language which can express both implicit or explicit synchronization. Some parallel assembly instructions for accessing global I-structure can be suspended upon the non-availability of data at the SQ frame, while others will directly address local register or immediate data without suspension. Those predetermined instruction type on suspension will reduce much of the overhead of context switching preparation and scheduling.

Gaudiot and Najjar[6, 8] envisaged the overhead problem of data-flow execution especially for vector operations and suggested macro data-flow consuming structured data by a macro actor of which executions are assumed to be performed by the most efficient and expertised functional unit. The second level of execution may be another static graph execution within the locality of available data. Also, they suggested a partial execution model where partially matched data set upon predefined subset of incoming data will be allowed to run and suspended upon the non-availability of a datum. Eventually this model will converge to the same process model of hybrid data-flow stated above.

While many researchers described the architectural requirement of hybrid model, Sarkar [9] has investigated the issues of partitioning and scheduling of a data-flow graph in a general multiprocessing system. He termed those compiled-time partitioning and run-time scheduling as macro data-flow. By assuming convexity constraint (no partial execution model) he developed a suboptimal solution to the partitioning problem of data flow graphs. His results are suitable when we are given a data flow graph and a target system for executing the graph with detail informations of overhead parameters such as communication latency and execution time of each procedures, whether probabilistic or deterministic according to the application program. However, machine resources are dynamic and those parameters are hard to obtain in real applications. A macro data-flow simulator is needed in order to identify critical system resources various application programs. Together with compiler and simulator aids, we may evaluate important features of programs partitioning and obtain a design guideline for our future hybrid data-flow system.

With ths pupose in mind, a simple simulator of macro data-flow has been developed, which accepts the U-interpreter-like data-flow graph [1] while the macro actor could be executed at another level. The subsequent sections will explain the details of our proposed architectural features and simulator implementation.

3 A Macro Data-Flow Architecture

Arvind[2] has pointed out two critical issues of multiprocessing system, i.e. memory latency and synchronization. To overcome these problems, he suggested a tagged token data-flow system with I-structure which is a complex memory system with synchronization capability in each memory cell. Even though this approach addresses the problem in its basic level of architectural support, the individual hardware complexity and unnecessary overhead of unused parallelism between active tokens leads to overspecify the resource requirement. A natural approach for reducing such overheads is to have larger grains of data synchronization, vector token handling mechanisms throughout packet network and processing element , and larger grained data-flow instructions to represent macro actors.

Our macro data-flow simulator is based on the Arvind's tagged token data-flow as Arvind's. The system consists of identical Processing Elements (PE) connected by a packet network. I-structures which handle structured data sharing the same node address of n-cube network with PE. The details of each PE and I-structure pair is depicted in the attached figure 1.

Fundamentally, a four-stage pipelined data-flow architecture is the same as Arvind's: matching incoming tokens, fetching the instruction, executing in the ALU and forming output token. The basic difference is that macro data tokens can carry "fat" data such as vectors. The vector data of incoming tokens should be stored in the Data Store and only the data frame pointer or vector data are sent to ALU which can directly address the matched vector data. A von Neumann execution model is currently assumed in the ALU. The micro architecture of ALU is depicted in figure 2. The micro instruction within the ALU has three addressing mode such as immediate, direct and indirect addressing as in conventional assembly language. Each datum of vector token data can be referenced by its port number and its displacement from the vector head. After processing the macro data, the ALU should indicate the range of valid data for producing new tokens with a tag to the OUTPUT facility.

Another difference is the program store of instructions. As opposed to simple actor in pure data-flow model, FETCH should carry the segment of program which defines one macro actor composed by a set of micro-instructions whether totally or partially ordered.

The Match and the Network have some mechanism for handling fat data token. It is remarkable that in packet network transfer delay is not less proportional to the size of data packet where overhead of routing or forming a packet is uniform regardless of data size. Such overhead factor handling variable size data token among system facility in PE and network are detailedly simulated by setting processing time variously.

4 Some Special Actor Schemes

Now that we have defined the representation and addressing of vector data token is defined, the method of forming vector token should be discussed. Furthermore as we have two types of operation of data processing and tag manipulation, tag operating instructions cannot be mixed with micro instructions within a macro actor. Function call also involves tag resource management and requires a special system actor to keep track of invoked instances of context tag resource. The methods of handling these issues will be described in the following subsections.

4.1 How to form Vector Token

If we assume that array data are stored in I-structures distributed in the system, some portion of the structured may be accessed in aggregate form. In this case, it is better to form one vector token rather than accessing individual datum every time through complex I-structure synchronization. In order to accumulate a vector token, we defined a special instruction, ACC, gathering incoming scalar tokens which keep state information of currently gathered tokens. A sample program of matrix multiplication is shown in figure 3.

GST generates a stream of data with different iteration tag. It will replace the index generation loop. SEL is a I-structure select operator. Through this actor, arrays A and B will be read through I-structure with different iteration tags. A row of A vector and a column of B vector will be formed by disregarding the lower digit of iteration tag during matching at ACC accumulating actor. INI is defined as producing seed tokens with null vector data with special collecting mode and information of number of data to be collected. After the row and column vectors are formed, they will be copied to each row and column pair for every vector product. The macro actor of VECTOR PRODUCT can be easily programmed with collected vector data by a small iteration loop of micro instructions within the macro actor.

It is remarkable that access frequency of array A and B of N by N size is N times less than conventional methods. According to the data access pattern of application program, we can determine the granularity of vector data.

4.2 Tag Manipulating Actor with Function Call

Tag manipulating actors such as D or L in U-interpreter model cannot be mixed with other micro instructions within one actor. These instructions assume separate execution by ALU or OUTPUT without storing incoming token data. These instruction can only change the tag field of context or iteration tag. If we assume the fixed size of context field, we should have a context management scheme to keep the uniqueness of context instance throughout the system. A scheme of function calls is shown in figure 4.

The basic scheme is adapted from Arvind's work[3]. GXT will generate a new context by a manager. The caller's tag will be extracted by ETG, which will keep the return tag address. By CTG, the incoming argument token will be retagged with new one from GXT. This will pass the caller's data to the called context. After processing the data in the called context, it will be passed to the caller by the information of return address token through RXT instruction which also returns the used context to the manager implicitly.

4.3 Special Matching Schemes

Some tokens at prespecified actors are matched in special way in order to reduce token traffic or to get some non-determinacy of matchin in AI application.

4.3.1 Lock Mode Actor (Sticky Tokens)

Some constant tokens may not be copied every times by using the sticky token concept at special actor number as belows.

Lock Mode 0 In this mode, any token arriving at port 0 will reside in match store and will output matched token to the fetch store when there is token arrival to port 1. Another token arrival to port 0 will overwrite the existing token at match store. Actor of code block number 1000 will be matched in this way.

Lock Mode 1 In this mode, any token arriving at port 0 or port 1 will reside in match store and will output matched token to the fetch store when there is token arrival to other port whether port 1 or port 0. Another token arrival to the occupied will overwrite the existing token at match store. Actor of code block number 1001 will be matched in this way.

Lock Mode 2 In this mode, any token arriving at port 0 will reside in it. The next coming token at port 0 will be paired by the existing token as matched token to fetch facility. The late coming token will be port number 1 data and will be overwritten to the match store waiting another partner coming to the port 0. Actor of code block number 1002 will be matched in this way.

4.3.2 Matching within Limited Wait Time

Some AI applications may require a non-deterministic wait time. The limit of waiting time can be specified by an actor of which code block number is greater than 2000. For example, tokens arriving at code block number 2089 will be sent to the FETCH facility after waiting for 89 system time units from the arrival of the first token at the actor. The matched token will be the whole set of tokens which have arrived during the waiting time.

5 Instruction Description

The following notations are used throughout the remaining chapters.

Identifier	Description
opr0	First operand field
opr1	Second operand field
opr2	Third operand field
dxy	y-th digit of opr x
in$x$	input token at port x
out$y$	output token at port y
C	Code Block No.
S	Statement No.
I	Iteration Tag.
PORT	Port number of a macro actor node.
TKNNT	Number of token to fire.
PE	Processing Element Number
PC	Program Counter for Micro-instruction Execution.

5.1 Program Graph Format

Program graph files are read per line. Head character of each line except spaces and tab will specify the line type. But any lines preceded by "%" are interpreted as comments. Those intermediate characters except numeric ones of [0..9] are also interpreted as comments.

As the higher level of execution is data-flow, macro actors should be partially ordered set. Thus our macro actor set is a graph composed of node and edges, where nodes represent computation and edges the destinations for computed data to go. A node may be a set of micro instructions of the lower level execution model.

Lines preceded by ":" are interpreted as beginning a macro instruction node and the following three integers of the line are read as C, S and actor priority or mapping function respectively.

Lines preceded by "D" are interpreted as edges of current macro node and the following six integers of the line are read as edge number (Data Outport Number), C, S, I, TKNNT, and PORT of destination node.

Lines preceded by "I" are interpreted as instructions of current macro node and the following five integers of the line are read as micro instruction number, operation code, and operand 0 through 2. These instructions will be further detailed in the following sections.

Lines preceded by "T" are interpreted as initial token data at macro node and the following six integers of the line are read as PE, C, S, I, TKNNT, PORT and priority.

Lines preceded by ">" are interpreted as data element data of current token being read. Pair of Type and Data may be repeated according to the length of vector data. Currently two types are supported, i.e. integer and real. Integer is represented by 2 and Real by 3.

The overall view to this program style is presented in the attached figure 5.

5.2 Micro Instructions

Micro instructions should have an addressing capability to access its matched data set(a vector). The digit of hundred's of the operand field will specify the input port number of vector data to be addressed. The remaining digits (ten's and one's) specify the displacement of data element of the indicated vector data of that port. The displacement number initials 0.

```
Example: (112) will specify the data element of  12th data element
```

```
    of port 1 vector data.

Example: (400) will specify the data element of  0th data element
    of port 4 vector data.
```

Five ports are supported and the displacement is limited to 99 elements respectively.

In many cases we need indirect addressing. If the fourth digit of thousand's of operand is greater than one, it designates the indirect addressing mode.

```
Example : (1345) will indicate the data addressed by the content
  of  the 45th data of port 3 vector.
```

Generally the format of our micro instruction is similar to conventional assembly instructions. Operand 0 specifies the storing address and Operand 1 and 2 specify two source address of data for processing.

As for iteration tag operation other than special tag operating actors described in the next chapter, it can be read by the special address of [099] and also modified by writing into the address.

NOP - 0 No operations but occupies the instruction space.

LD - 1 Load immediate value to Micro Store.

Identifier	Description
opr0	Address to be set
opr1	Set value
opr2	Set value type (2: Integer, 3: Real)

ex) ld=1 233 34 2
will set Micro Store 233 by 34 of integer.

ex) ld=1 233 34 3
will set Micro Store 233 by 34.0 of real.

MOV - 2 Move data between Micro Store.

Identifier	Description
opr0	Destination address
opr1	Source address
opr2	Number of elements to move. If minus, then -opr2 to move. otherwise opr2 indicate the content specify the number.

ex) move=2 344 233 -1
will set the value of 344 by the value of 233.

ex) move=2 344 233 200
% block move if $store[200] = 10$, this instruction will move the contents of 344 through 353 to the locations of 233 through 232 respectively.

ADD - 10 Plus operation

Identifier	Description
opr0	Destination address
opr1	Source 1 address
opr2	Source 2 address

- Type overloading is automatic by the type of opr1.

ex) plus=10 233 122 233
$(233) \leftarrow (122) + (233)$

SUB - 11 Minus operation

- same as PLUS
ex) minus=11 233 122 233
$(233) \leftarrow (122) - (233)$

MUL - 12 Multiply operation

ex) mult=12 233 122 233
$(233) \leftarrow (122) * (233)$

DIV - 13 Divide operation

- Type overloading is automatic by the type of opr1.

ex) divide=10 233 122 233
$(233) \leftarrow (122)/(233)$

MOD - 14 Modulo operation

- Type overloading is automatic by the type of opr1.

ex) mod=10 233 122 233
$(233) \leftarrow (122) mod (233)$

INC - 20 Increment the data

Identifier	Description
opr0	The address to be incremented

ex) inc=20 123
will increase (123) by 1.

DEC - 21 Decrement the data

- same format as INC except the operation.

ex) dec=21 123
will decrease (123) by 1.

NOT - 22 Complement datum

ex) not=22 123 200
if (200) is 1, set (123) to 0.
otherwise set to 0.

AND - 23 Logical AND Operation

Identifier	Description
opr0	Destination address
opr1	Source 1 address
opr2	Source 2 address

- Type should be integer.
- if (opr1) or (opr2) is 0, then set (opr1) to 0. Otherwise set to 1.

ex) and=23 233 122 233
$(233) \leftarrow (122) AND (233)$

OR - 24 Logical OR Operation

- Same format as AND.
- Type should be integer.
- if (opr1) or (opr2) is 1, then set (opr1) to 1. Otherwise set to 0.

XOR - 25 Logical Exclusive OR Operation

- Same format as AND.
- Type should be integer.
- if (opr1=1 and opr2= 0) or (opr1=0 and opr1=1), then set (opr1) to 1. Otherwise set to 0.

ABS - 26 Absolute value

Identifier	Description
opr0	The destination address to write.
opr1	The source address to be taken for absolute value.

ex) abs=26 123 111
will set (123) by the absolute value of (111).

VCP - 27 Compare Vector Data

Identifier	Description
opr0	Vector 1 starting address
opr1	Vector 2 starting address
opr2	Number of elements to compare if minus, then -opr2 to move. otherwise opr2 indicates the content specify the number.

ex) vcp=27 344 233 -2
if (344) = (233) and (345) = (234), then set (346) = 1. Otherwise set (346) = 0.

ex) vcp=27 344 233 200
if *store*[200] = 2, this instruction has same effect as the above one.

JMP - 30 Set PC (program counter of micro instruction).

Identifier	Description
opr0	Jump address
opr1	Flag for interpreting jump address. if non-negative, immediate value of opr0. if negative, the content of the address opr0.

ex) jump=30 23 1
will set PC by 23. The instruction by instruction no 23 will be executed next.

ex) jump=30 123 -1
will set PC by the value of (123).

CBR - 31 conditional jump.

Identifier	Description
opr0	Condition check Micro Store address
opr1	Jump status type. 2 : Non negative 1 : Plus 0 : Zero 10 : Non zero -1 : Minus -2 : Non-negative
opr2	Jump instruction number

ex) cbranch=31 -1 001 23
will jump to instruction no 23 if the content of 001 is negative.

EXT - 99 Exit the current macro actor. When EXIT is executed, the execution of micro instructions stops and the output data will be formated as new token at OUTPUT.

SIG - 97 Signal the location of resulting data for OUTPUT to generate tokens to its destinations.

Identifier	Description
d02	Output port number of the macro actor.
d01	Micro Store high address to be output
opr1	Start address of Micro Store
opr2	Number of output data if positive, the content addressed by opr2 if negative, the absolute number.

ex) sig=97 12 34 000
Make output port 1 to be data list starting from Micro Store 234 by the number of content 000.

ex) sig=97 12 34 -4
Make output port 1 to be data list starting from Micro Store 234 by the number of 4.

RND - 96 Random Integer Generation:

Identifier	Description
opr0	Address of random number to be written.
opr1	Random number starting range
opr2	Random number ending range.

ex) rand=96 110 0 999
will set (110) by a random value x ranged [0, 999].

* This instruction may be used in AI application of non-deterministic programming style or programming resource manager.

PRT - 90 Print out the contents of Micro Store.

Identifier	Description
opr0	High address of Micro Store
opr1	Start low address of Micro Store
opr2	End low address of Micro Store

ex) out=90 2 23 45
will print out the contents of Micro Store from 223 to 245.

6 Tag Operations and other Miscellaneous Instructions

Other than micro instructions described above are those instructions regarding I-structure access, tag operation, some special vector actor for gathering or scattering. These operations are intended as atomic one which can not be mixed with any other micro instructions within a macro actor. Thus instruction number is numbered as 0 and operands have special interpretation other than vector data addressing. An opcode number greater than 100 indicates that it is different from the micro instruction set and the operation does not need to copy matched data into the Micro Store for random access and that it does need some special handling regarding tag and system I/O.

6.1 Tag Operation Instruction

The function calling actors are defined according to Arvind's as explained in the previous chapter. And tag manipulating actors are from U-interpreter model.

GXT - 100 Get context

Identifier	Description
in 0	PE number if ($opr0 < 0$). Otherwise input token is used triggering purpose.
opr 0	Policy of mapping new context to any PE 0 : Current PE 1 – 6 : The i-th bit reversal (adjacent PE) 9 : Least busy adjacent PE selection 10 : Adjacent random selection 11 : Rotational counter scheme 12 : Full random selection
opr 1	Code block no.
out 0	Context integer uniquely defined and code block no c. [PE number + context*1000 : Code Block No.]

ETG - 102 Extract tag

Identifier	Description
in 0	Any token of the caller's context
opr0	S of the destination
opr1	NT of the destination
opr2	PORT of the destination
out 0	Tag informations $[Context : C : S : NT : PORT : I]$

CTG - 103 Change tag

Identifier	Description
in 0	Context and Code Block number.
in 1	Data argument to be sent to the called code block.
opr0	S : The called's statement no.
opr1	NT : Number of token of the actor
opr2	PORT : Input port number to go

RXT - 101 Retag the computed data to caller and return context

Identifier	Description
in 0	Tag information of the caller
in 1	Data value to be returned
other actions	Return the caller's context to the manager for reuse

DML - 110 Multiply iteration tag with specified operand

Identifier	Description
in 0	Any tagged token
out 0	Same token except the iteration tag. If *opr*0 >= 0 , new tagi = old tagi * opr0 else new tagi = old tagi / opr0.

DAD - 111 Add iteration tag with specified operand

Identifier	Description
in 0	Any tagged token
out 0	Same token except the iteration tag. with new tagi = old tagi + opr0

TTG - 112 Retrieve iteration tag

Identifier	Description
in 0	Any data token
out 0	Token with data set by tag.

RTG - 113 Read ranged iteration tag.

Identifier	Description
in 0	Any data token
out 0	Token with data of specified iteration tag.
opr0	Starting field of iteration tag to read
opr1	Field ending of iteration tag to read

ex) data token with iteration tag 9876*543* 210

I : 0 read=iteration tag=113 3 5
will result integer data "543".

WTG - 114 Write ranged iteration tag.

Identifier	Description
in 0	Any data token
in 1	Any data token for iteration tag set, if used. (known by incoming token TKNNT).
out 0	Token with iteration tag set by incoming data or operand field.
opr0	Starting field of iteration tag to read
opr1	Ending field of iteration tag to read
opr2	If TKNNT= 1, this will be used for iteration tag.

ATG - 115 Add ranged iteration tag to existing iteration tag.

Identifier	Description
in 0	Any data token
out 0	Token with iteration tag set by incoming data or operand field.
opr0	Starting field of iteration tag to read
opr1	Ending field of iteration tag to read
opr2	Integer to be added the existing iteration field.

ex) data token with iteration tag 9876543210
I : 0 add=iteration tag 115 3 5 1
will result the same data token with iteration tag 9876*544* 210
I : 0 add=iteration tag=115 3 5 -1
will result the same data token with iteration tag 9876*542* 210

DDD - 120 Old version of iteration tag operator

Identifier	Description
in 0	Any tagged token
out 0	Same token except the iteration tag. new iteration tag = old iteration tag +1

LLL - 122 Old version of Context Changing operation

Identifier	Description
in 0	Any tagged token
out 0	Same token except the u tag new-u = new tag field tagged by old u

ILL - 123 Old version of Context Retrieving operation.

Identifier	Description
in 0	Any tagged token
out 0	Same token except the u tag. Context filed is replaced by previous context

6.2 I-Structure Instruction

I-structure operation cannot be mixed with those described in the previous chapter since these operations are split-phased (non-deterministic execution time). The ALU will reformat the read or write request token with setting I-access mode and destination list to tag after obtaining the data. The reformated token will be routed to the I-structure controller where the requested element of structure is stored.

CRE - 200 Create array

Identifier	Description
opr 0	Array type. 0 : no index type This zero dimensional array is allocated from the beginning. There is no need to create for this type. 1 : 1 dimensional 2 : 2 dimensional 3 : 3 dimensional
in 0	array name of integer
in 1	max index 1 range if used
in 2	max index 2 range if used
in 3	max index 3 range if used
out 0	The array name if used for synchronization purpose

APP - 201 Write the array

Identifier	Description
opr 0	Array type. if *opr* >= 10, overwrite mode is applied in case of second write access token. (User is responsible for single assignment requirement) Otherwise it appends at the end of the present data list if the first integer data is positive. If first integer data (n no of element) is negative, it will extract the incoming data set from existing vector data at the I-STRUCT cell by the following format. [(no of element)$\rightarrow$ *data*1 $\rightarrow$ *data*2... $\rightarrow$ *datan*] In any case (opr0 mod 10) indicates the array dimension.
in 0	Data to be written
in 1	Array name of integer
in 2	Index 1 range if used
in 3	Index 2 range if used
in 4	Index 3 range if used
out 0	The array name if used for synchronization purpose

APX - 211 Write the array in macro mode.
(Normally used when initializing array)

Identifier	Description
opr 0	Array dimension
in 0	Data to be written
in 1	Array name of integer
in 2	Maximum index 1 if used
in 3	Maximum index 2 range if used
in 4	Maximum index 3 range if used
out 0	The array name if used for synchronization purpose

SEL - 202 Read the array data

Identifier	Description
opr 0	Array dimension
in 0	Array name of integer
in 1	Maximum index 1 if used
in 2	Maximum index 2 range if used
in 3	Maximum index 3 range if used
out 0	The data read from structure

6.3 Vector Gathering Operator

GST - 220 Generate token stream with different iteration tag

Identifier	Description
in 0	Index range of integers $Istart > Iend > Jstart > Jend > Kstart > Kend$
in 1	Data range $data0 > IincData > JincData > KincData$
in 2	Iteration tag range $tag0 > IincTag > JincTag > KincTag$
opr 0	If 1, then copy mode. (the same data will be generated.)
out 0	Multiple tokens generated with data and iteration tag as below

```
Outputting tokens are defined by following loop struct.

 for (i= Istart ; i != Iend; i++){
   for (j= Jstart ; j != Jend; j++){
     for (k= Kstart ; k != Kend; k++){
       iteration tag = present_iteration tag + tag0 + i*IincTag + j*JincTag + k*KincTag;
       datai = data0 + i*IincData + j*JincData + k*KincData;
     }
   }
 }

    - but if incoming data type of in 1 is not integer,
        only iteration tag will be changed.

     ex) g_stream_2020
        with incoming tokens as
      in 0 : 0 1 1 3
             in 1 : 100 3 10
             in 2 : 0 1 20

        will generate token set with (data : iteration tag)
          (110 : 20) (120 : 40) (130: 60)
          (113 : 21) (123 : 41) (133: 61)
```

INI - **230** Initialize vector seed token

Identifier	Description
in 0	Number of data to be collected
out 0	Vector seed token

ACC - **231** Accumulate vector token by wild matching. In this accumulating mode, a wild matching is performed disregarding lower two digits of iteration tag of incoming token stream and sorted by these two digits.

Identifier	Description
in 0	Vector token. Initially generated from INI actor.
in 1	Set of vector elements coming individually.
out 0	If required number of element are collected, the complete vector token will be produced to the next destination actor. Otherwise the incomplete vector token will be fed back to input port 0 of this ACC actor.

6.4 Other Miscellaneous Instructions

TRU - **300** True gate

Identifier	Description
in 0	Any data token 0
in 1	Gate token of integer
out 0	If gate data is not 0, data token 0, otherwise no token is produced.

FAL - **301** False gate

Identifier	Description
in 0	Any data token 0
in 1	Gate token of integer
out 0	If gate data is 0, data token 0, otherwise no token is produced.

SWI - 310 Switch operation

Identifier	Description
in 0	Any data token 0
in 1	Gate token
out 0	If gate data is 0, data token 0, otherwise no token is produced.
out 1	If gate data is 1, data token 0, otherwise no token is produced.

MRG - 311 Merge operation

Identifier	Description
in 0	Any data token 0
in 1	Any data token 1
in 2	Gate token
out 0	If gate data is 0, data token 0 will be gated. If gate data is 1, data token 1 will be gated. Otherwise no token is outputted.

CON - 320 Constant token generation

Identifier	Description
in 0	Any token for triggering.
out 0	Token with data set by $opr0 \rightarrow opr1$. if $opr1 < 0$, opr0 only

IDN - 330 Identical tokens are produced for copy purpose.

Identifier	Description
in 0	Any token
out 0	The same token

TPR - 999 Token is printed out for debugging purpose

Identifier	Description
in 0	Any token
	No token is produced, but system will print out the token for debugging purpose.

7 Simulation Parameters and Error Handling

7.1 Simulation Parameters

Simulation is not graph level execution but assumes actual resources of facilities of PE or networks. Thus the simulation requires the setting of many system parameter such as number of PEs, token routing policy, array mapping policy, trace mode for debugging or histograming, trace break point, facility unit delay(execution time).

These parameters can be set by reading parameter files as shown below. The file name is specified in the command line as an argument with "–" prefix.

Some parameter can be read directly by arguments of command line preceded by a ":" character.

Sample Run Command:

$$dfm - para2prog1token1 : q = 16$$

will run simulator reading prog1 and token1 as program and token file and para2 as parameter files and setting number of simulating PE as 16.

Sample Parameter File:

```
%----------trace of token processing----------------------------------
% FORMAT = [time, number of active tokens, number of active function instance]
% trace_mode= -2 : only batch output
% trace_mode= -1 : time/token trace output (*)
% trace_mode = 0  NETWORK  facility
% trace_mode = 1  MATCH    facility
% trace_mode = 2  FETCH    facility
% trace_mode = 3  ALU      facility
% trace_mode = 4  OUTPUTTER facility
% trace_mode = 5  I-STRUCT facility
% ---------trace of queue length at each facility--------
% FORMAT =  [T= time,Number of tokens at defined facility of pe 0,1,.....n]
% trace_mode = 10 MATCH    facility
% trace_mode = 20 FETCH    facility
% trace_mode = 30 ALU      facility
% trace_mode = 40 OUTPUTTER facility
% trace_mode = 50 I-STRUCT facility
% trace_mode = 60 number of token waited in match
% trace_mode = 70 number of active instance
%-------------------------------------------------------
% code_block, statement no  (-1, -1) - trace all  (*)
% code_block, statement no  (0 , 6)
%-------------------------------------------------------
```

```
%                    EXECUTION TIME SET-UP
%                    ----------------------
%               NETWORK  MATCH FETCH ALU ROUTER I_STRUCT
% base time(*)  = ( 1     1     1    1    1     1)
% small time(*) = ( 0     0     0.1  0.1  0.1   0.1 ) /* micro time */
%-----------------------------------------------------------
% base time   = ( 1     1     1    1    1     1)
% small time  = ( 0     0     0.1  0.1  0.1   0.1 ) /* micro time */
%-----------------------------------------------------------
% network routing policy = 0      %  0 : first adjacent (*)
%-----------------------------------------------------------
% q_number of pe  = 64 (*)
% q_number of pe  = 32
% q_number of pe  = 16
% q_number of pe  =   8
% q_number of pe  = 4
% q_number of pe  = 2
% q_number of pe  = 1
%-----------------------------------------------------------
 delta_time          = 10     %  token trace interval (*)
% delta_time          = 20     %  token trace interval
% delta_time          = 1     %  token trace interval
%-----------------------------------------------------------
% mapping policy  = 10     by context (*)
% mapping policy  = 1    A=17 B=0 C=0 D=0 E=0 F=0
%
%    pe = (i1*A + i2*B + i3*C + c*D + s*E + context* F) % pe #
%    where tagi = 123456 i3= 12, i2= 34, i1= 56..
%          context
%
  mapping policy = 1  A= 0 B= 0 C=0 D= 0 E= 1 F= 0
%       initial token mapping is done automatically.
%-----------------------------------------------------------
% array_mapping policy          = 0     by array name. (*)
% pe = aname % no of pe.
%
 array mapping  = 1     (A= 1, B= 4, C= 0)   by array index.
%    pe = (index1*A + index2*B + index3*C) % number of pe.
%-----------------------------------------------------------
% priority policy         =  0     %  0 : FIFO (*)
%-----------------------------------------------------------
% sample token output
% >> context=0 c=100 s=21 i=200
% >>fpe= 2 ftype=4 tknnt= 2 port= 0 d= 3 count=-3 a= 0 ix=5 4 0
% >> data : > 5  > 5.000000  > 1.000000
%----------------------
% fpe : pe number of the facility, ftype = facility type
% tknnt = number of token required to match, port = token port to go
% d = token  type (0 : normal, 1 : i-structure access, 2 : accumulating )
% a= array name,  ix = index set, count = collecting status
%----------------------
% context = explicit context number, c = code block no,
% s = statement number, i = iteration tag
% data = linked list of data
%----------------------
```

7.2 Output of Trace and Final Statistic Report

The default trace of output will be Simulating Time, Number of Tokens in system and Number of Function Call Instances. This can be changed by setting trace parameter and delta time parameter differently as shown previous section.

The sample final statics are as below.

```
% A_TKN:2,  M_TKN:5,  G_TKN:0,  S_TIME: 17,  N_LOAD: 0
```

A_TKN is the average number of tokens, M_TKN is the maximum number of tokens, G_TKN is the number of garbage tokens unmatched after completion of simulation, S_TIME is the simulation time unit, and N_LOAD is the accumulated load on the communication network.

7.3 Simulation Error Handling

During simulation some error message will be generated by invalid type of data or instructions. All error messages are reported starting with "$" followed by the error number. The us can refer to the error message table attached and can handle it. Any fatal error will stop the simulation printing the current token at error. Otherwise error messages will be of the warning type.

Please contact author of this report in case that any error is reported other than those listed in the attached table.

8 Graph Simulator

A macro data-flow graph simulator has also been implemented in order to facilitate quick simulation for extracting program characteristics without limitation of hardware resources. The same instruction set and the same format of program are used as described for the hardware simulator in the previous chapter. The only difference of graph simulator is the facility description, which is characterized simply by four basic functional units and the number of each units can be set from 0 to infinity according to the simulating resource limit of each facility.

Those four facilities are Processing Elements for data and tag operation, Matching Unit for instruction scheduling, Structure Memory Unit and Communication Element. Variable number of servers for these units can be set up to infinity. When infinity of all units is assumed, pure graph simulation is performed showing program characteristics without hardware resource consideration.

The facility diagram of this graph simulator is depicted in the attached figure 6.

8.1 Parameter Setting

Many simulation parameters can be set in the command line preceding ":" as below.

```
Trace_Mode
         :t=-1          default, no tracing of token
         :t=1           token input tracing
         :t=2           ALU input tracing
         :t=3           CE input tracing
         :t=4           CE output tracing

Code Block no. for tracing
        :c=-1   defaults, all code block are traced.
        :c=38   code block 38 will be traced.

State No. for tracing
        :c=-1   defaults, all statements  are traced.
        :c=83   statement no. 83 will be traced.

Delta Time for printing token histogram
        :d=10   defaults, every 10 time unit, status will be printed.
        :d=20   every 20 time unit, status will be printed.
```

```
Ratio of Macro-Set up Time / Micro Execution Time
        :m=0    defaults, micro time =0
        :m=10         micro time/macro time = 10.

Communication Time set
         :x=10         10 unit time of transit for each token.
         :x=1    defaults.

Number of PE servers
        :P=0    infinity servers (defaults)
        :P=45   45 servers for PE

Number of CE servers
        :N=0    infinity servers (defaults)
        :N=45   45 servers for PE

Number of Structure Memory servers
       :S=0    infinity servers (defaults)
       :S=45   45 servers of Structure Memory

Number of Match
       :M=0    infinity servers (defaults)
       :M=45   45 servers of Match function
```

8.2 Output Trace and Final Statistic Report

The following statistics are printed during simulation at specified intervals.

Time: Current simulating time

Numtkns: Number of tokens issued at current time

No_of_Active_Inst: Number of busy servers of Processing Elements (effective parallelism)

Total_Inst: Number of total instruction executed.

In_Match: Number of tokens at MATCH facilities

In_SM: Number of tokens at Structure Memory

In_Transit: Number of tokens at Communication Element

F_Call: Number of active function calls

The followings statistics are reported after simulation.

A_TKN :	Average number of tokens
M_TKN :	Maximum number of tokens
G_TKN :	Number of garbage tokens
S_TIM :	Simulation time unit.

References

[1] Arvind and K.P. Gostelow. The U-Interpreter. *IEEE Computer*, pages 42–49, February 1982.

[2] Arvind and R.A. Iannucci. Two fundamental issues in multiprocessing: the dataflow solutions. Technical Report Computational Structure Group Memo 226-5, MIT Laboratory for Computer Science, July 1986.

[3] Arvind and R.S. Nikhil. Executing a program on the MIT tagged token dataflow architecture. In *Proc. Parallel Architecture and Language Europe, LNCS 259*, pages 1–29, June 1987.

[4] Lubomir Bic. A process-oriented model for efficient execution of dataflow programs. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 332–336, 1987.

[5] R. Buehrer and K. Ekanadham. Incorporating data flow ideas into von neumann processor for parallel execution. *IEEE Transactions on Computers*, pages 1515–1522, December 1987.

[6] J.L. Gaudiot and W.Najjar. Macro-actor execution on multilevel data-driven architecture. In *Proceedings of the Working Conference, Parallel Processing, IFiP Pisa, Italy*, April 1988.

[7] Robert A. Iannucci. Toward a dataflow / von neumann hybrid architecture. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 131–140, 1988.

[8] W. Najjar and J-L. Gaudiot. Multi-level execution in data-flow architectures. In S.K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987.

[9] V. Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. Technical Report CSL-TR-87-328, Stanford University, Computer Systems Laboratory, April 1987.

Mnemonic	Opcode No.	Description
		Arithmetic and Logical Instructions
NOP	0	No operations but occupies the instruction space.
ASN	1	Set some value into the store address.
MOV	2	Move data in store RAM.
ADD	1	Add operation
SUB	11	Subtract operation
MUL	12	Multiply operation
DIV	13	Divide operation
MOD	14	Modulo operation
INC	20	Increment the data
DEC	21	Decrement the data
NOT	22	Logical NOT operation
AND	23	Logical AND operation
OR	24	Logical OR operation
XOR	25	Logical exclusive OR operation
ABS	26	Absolute value
VCP	27	Vector Compare Operation
		Control Instruction
JMP	30	Set PC by operand (jump).
CBR	31	Conditional branch
		System Instruction
EXT	99	Indicate the end of micro instructions.
SIG	97	Interface actor from ALU to OUPUTTER.
RND	96	Random Integer Generation
PRT	90	Print out the contents of store RAM.
		Tag Operating Actor
GXT	100	Get context
ETG	102	Extract tag
CTG	103	Change tag
RXT	101	Return the computed data to caller and return context
DML	110	Multiply iteration tag with specified operand
DAD	111	Add iteration tag with specified operand
TTG	112	Retrieve iteration tag
RTG	113	Read ranged iteration tag.
WTG	114	Write ranged iteration tag.
ATG	115	Add ranged iteration tag to exeiting iteration tag.
DDD	120	Old version of iteration tag operator
LLL	122	Old version of U (context change) tag operator
ILL	123	Old version of inverse U (retrieving context) tag operator
		I-Structure Operation
CRE	200	Create array
APP	201	Append Array
APX	211	Extended Append
SEL	202	Select(READ) an element of array
		Vector Gathering Operator
GST	220	Generate stream tokens with different iteration tag
INI	230	Initialize vector seed token
ACC	231	Accumulate vector token by wild matching
		Other Miscellaneous Instructions
TRU	300	True gate
FAL	301	False gate
SWI	310	Switch operation
MRG	311	Merge operation
CON	320	Constant token generation
IDN	330	Ident token generations for copy purpose.
TPR	999	Token print for debugging purpose

Table 1: Instruction Summary Table

Error No.	Description	Action
100	Invalid Mode of Token at OUTPUT DEPARTURE.	Contact Distributer
101	Invalid Event Type !!	Consult Distributer
102	Invalid Index in stream token generation	Check the range of index or the depth of nestedness (3 is maximum) at GST instruction
103	Tknd Error in NETWORK	Consult Distributer
104	Invalid Token Port Number at Lock Matching	The valid port number for Lock Mode is 0 or 1. Check the lock mode instruction of which code block number is greater than 1999.
105	Instruction fetch NULL error at tokens below	Check whether the specified program code block is defined by the number of C and S.
200	Token Port Number is out of range.	Valid port number for all instruction is [0,1,2,3,4]. For some instructions the arity of actor is defined by operand. Check whether the range is valid.
201	Data Type is invalid.	Valid data type is Integer and Real of which number of type are 1 and 2 respectively. Check whether the prior actor is generating valid data and prpoerly inputted from Token Reading in the beginning.
202	Invalid Tag Range.	Iteration Tag should not be negative. Check whether your manipulation is producing proper integer for the iteration tag.
203	Invalid PE assigned at GCXT instruction.	PE number should be within [0.. User Specified number of PE]. The maximul PE number is 63. Check whether you are generating proper PE assigning dynamically.
204	Invalid Opcode of Atomic Type.	Atomic Type of Instruction of Tag Manipulating of I-structure access is numbered over 1000. Check whether you are assigning valid number for the opcode of the instruction.
300	Invalid micro Opcode.	Micro opcode is ranged between 0 and 999. Check whether you are assigning proper number for the micro instruction opcode.
301	Invalid Data Type at Mico-ALU	Valid data type is Integer and Real of which number of type are 1 and 2 respectively. Check whether the prior actor is generating valid data and prpoerly inputted from Token Reading in the beginning.
302	Invalid Addressing error at Micro ALU.	Valid addressing range to Micro Store is [000 ... 499] except [099].
303	Invalid Data Type at Outputting.	By the instruction of SIG the vector data is passed to OUTPUTTER for token generation. Check whether the specified range of micro store is well defined.
304	Invalid type in copying a data.	By the instruction of SIG the vector data is passed to OUTPUTTER for token generation. Check whether the specified range of micro store is well defined.
305	Invalid data type at reading token	The matched Token Data is read into Micro Store before processing. During reading invalid data type except real and integer will be reported.
306	Overflow of Data Length of input Token.	The size of Micro Store is Memory[4][99]. If matched Token Data exceeds 99, this error result. Check the source of the incoming tokens.
307	Max Micro Inst No is 99.	The maximum micro instruction number for a macro actor is 99.
400	Instance overflow.	By GCXT the new and unique instance number is assigned. If active function call is more than 10000 at one time, ths error will result.
401	Type error in prtdatas.	In printing vector data, invalid type occured. Check the token data's source.
402	Garbage tokens exist.	This warning are generated when not all tokens are collected. If you are using LOCK mode or your programming is indeterministic type, you may disregard.
500	Type Error in copy datas.	In copying vector data structure, invalid type occured. Check the token data's source.

Table 2: Error Message List and Action

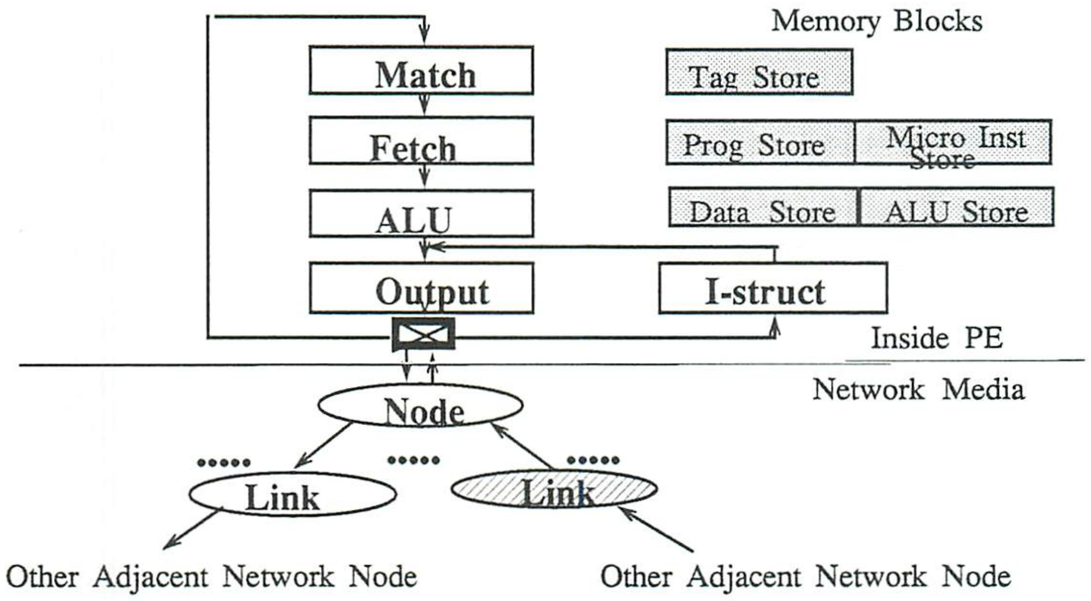


Figure 1: Macro Architecture Diagram

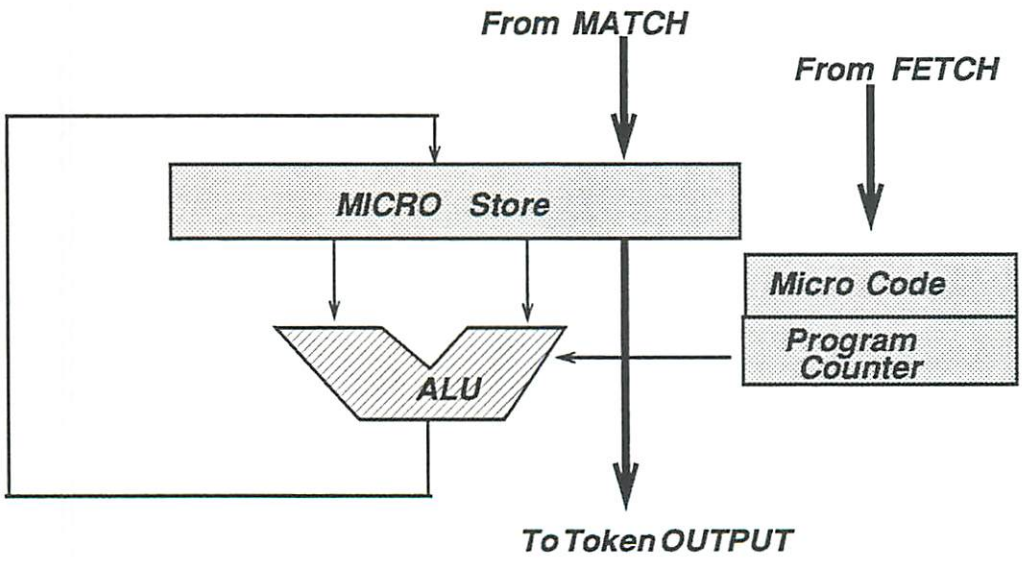


Figure 2: Micro Architecture Diagram

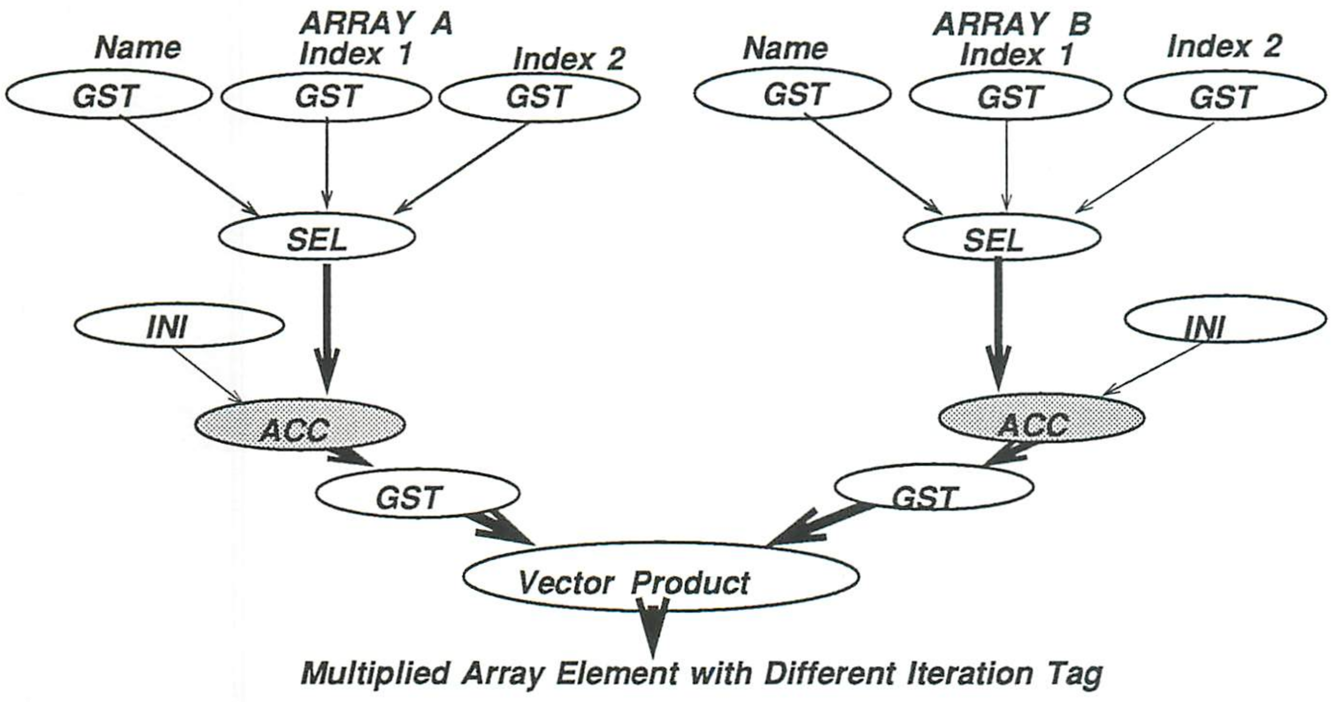


Figure 3: Sample Multiplication Graph

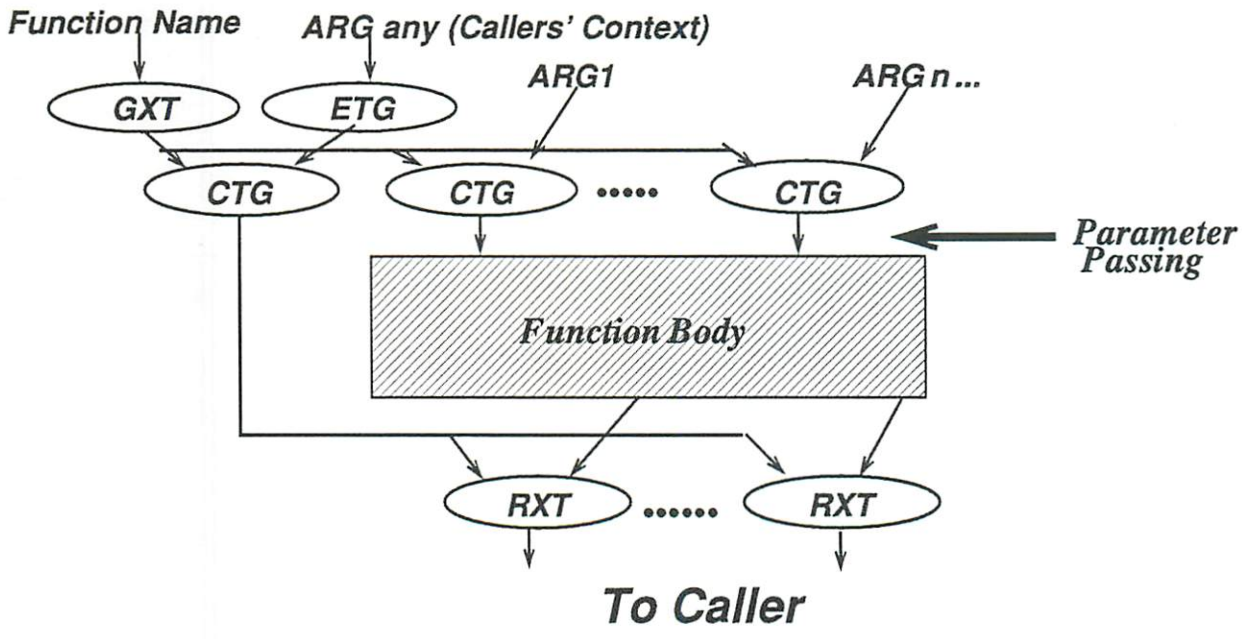
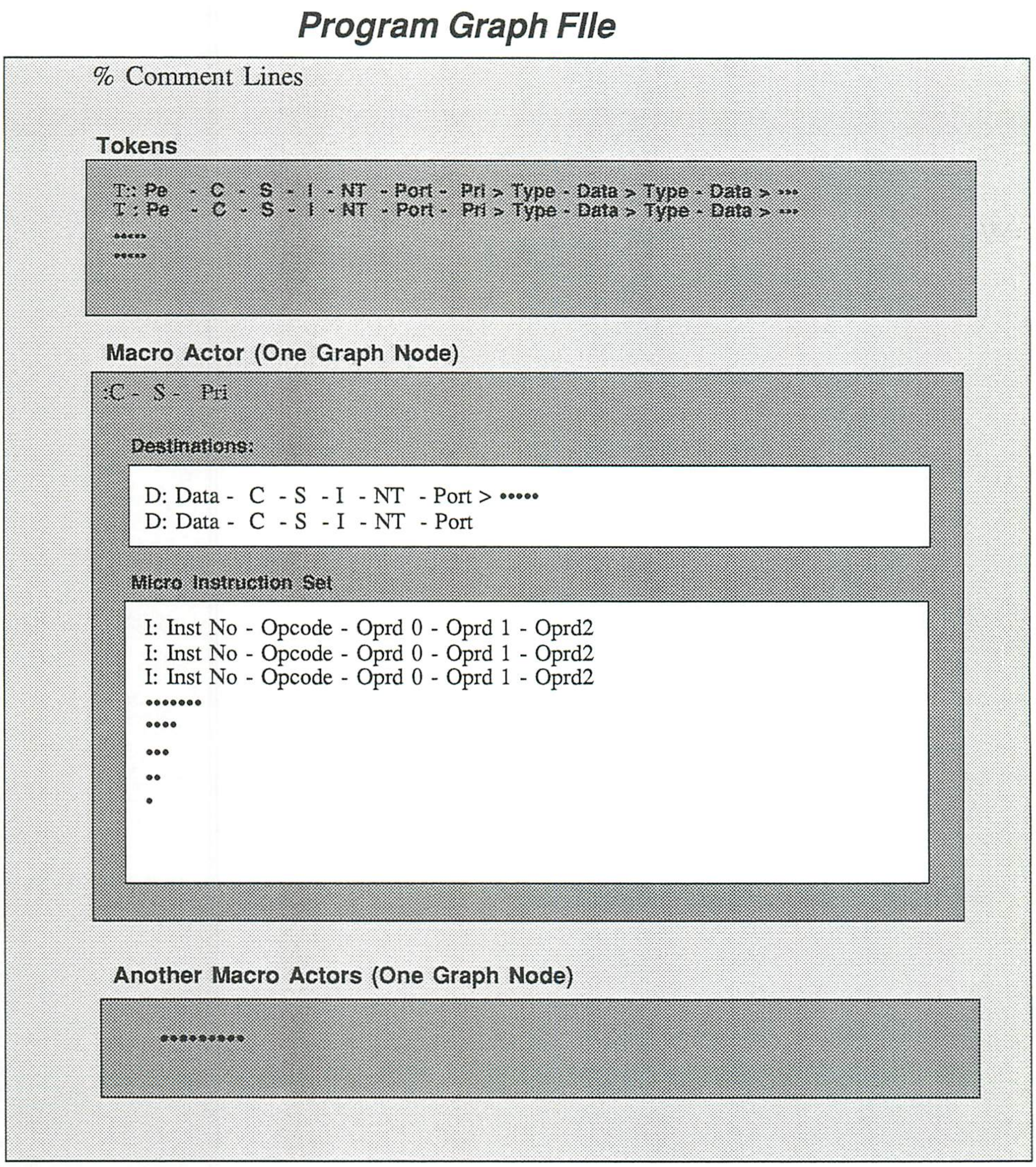


Figure 4: Function Call Graph


Program Graph FIle
Tokens
Macro Actor (One Graph Node)
Destinations:
Micro Instruction Set
Another Macro Actors (One Graph Node)

```
% Comment Lines

T:: Pe   - C - S - I - NT  - Port -  Pri > Type - Data > Type - Data > •••
T : Pe   - C - S - I - NT  - Port -  Pri > Type - Data > Type - Data > •••
•••••
•••••

:C - S -  Pri

D: Data -  C  - S  - I  - NT  - Port > •••••
D: Data -  C  - S  - I  - NT  - Port

I: Inst No - Opcode - Oprd 0 - Oprd 1 - Oprd2
I: Inst No - Opcode - Oprd 0 - Oprd 1 - Oprd2
I: Inst No - Opcode - Oprd 0 - Oprd 1 - Oprd2
•••••••
••••
•••
••
•

•••••••••
```

Figure 5: Typical Program Structure

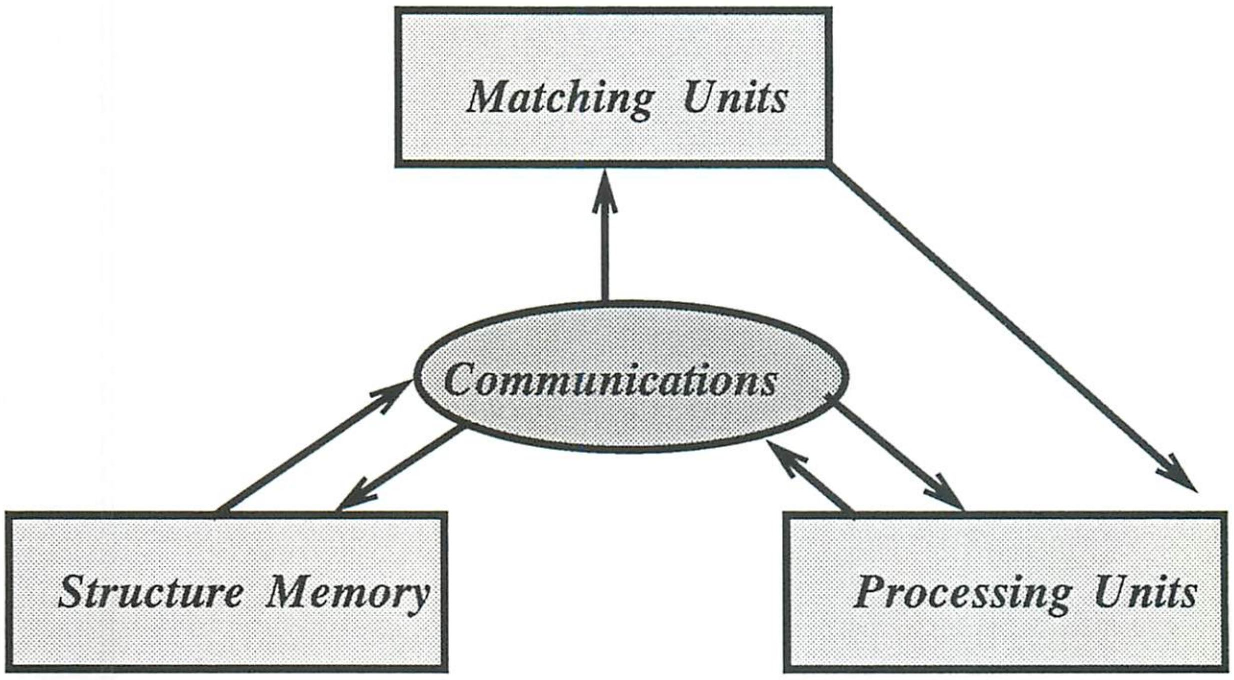


Figure 6: Facility Diagram for Graph Simulator